XBee Python Library Documentation

Release 1.4.0

Digi International Inc.

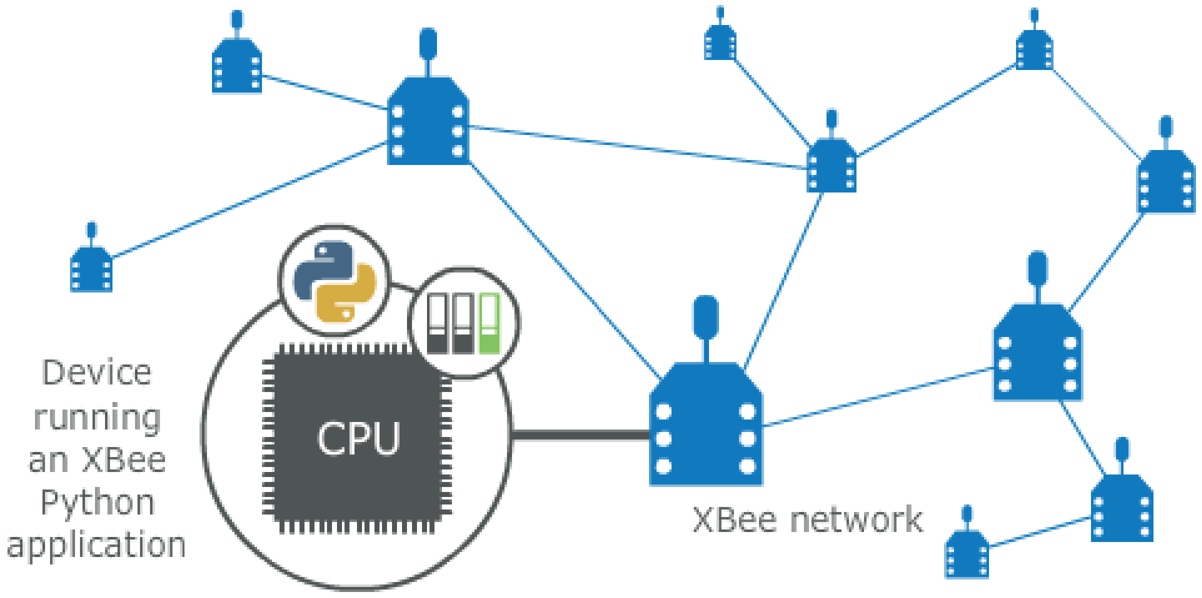
Mar 18, 2021

Getting Started

Release v1.4.0. (*Installation*)

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.

CHAPTER 1

Requirements

The XBee Python library requires the following components in order to work properly:

- **Python 3.6**. 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`).

CHAPTER 2

Contents

The XBee Python library documentation is split in different sections:

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

2.1 Getting Started

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

- *Get started with XBee Python library*

2.2 User Documentation

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

- *XBee terminology*
- *Work with XBee classes*
- *Configure the XBee device*
- *Discover the XBee network*
- *Communicate with XBee devices*

- *Handle analog and digital IO lines*
- *Update the XBee*
- *Log events*

2.3 Examples

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

- *XBee Python samples*

2.4 FAQ

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

- *Frequently Asked Questions (FAQs)*

2.5 Changelog

- *Changelog*

2.6 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*

2.6.1 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*

2.6.1.1 Install your software

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

- *Python 3*
- *PySerial 3*
- *SRP*

- *XBee Python library software*
- *XCTU*

Python 3

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

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

PySerial 3

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

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

SRP

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

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

XBee Python library software

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

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

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

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

Get the source code

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

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

XCTU

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

For instructions on downloading and using XCTU, go to:

http://www.digi.com/xctu

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

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

2.6.1.2 Configure your XBee modules

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

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

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

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

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

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

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

802.15.4 devices

1. Click **Load default firmware settings** in the **Radio Configuration** toolbar to load the default values for the device firmware.
2. Make sure API mode (API1 or API2) is enabled. To do so, set the **AP** parameter value to **1** (API mode without escapes) or **2** (API mode with escapes).
3. Configure **ID** (PAN ID) setting to **CAFE**.
4. Configure **CH** (Channel setting) to **C**.
5. Click **Write radio settings** in the **Radio Configuration** toolbar to apply the new values to the module.
6. Once you have configured both modules, check to make sure they can see each other. Click **Discover radio modules in the same network**, the second button of the device panel in the **Radio Modules** view. The other device must be listed in the **Discovering remote devices** dialog.

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

Zigbee devices

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

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

2. Click **Load default firmware settings** in the **Radio Configuration** toolbar to load the default values for the device firmware.
3. Do the following:
   - If the device has the **AP** parameter, set it to **1** (API mode without escapes) or **2** (API mode with escapes).
   - If the device has the **CE** parameter, set it to **Enabled** in the coordinator.
4. Configure **ID** (PAN ID) setting to **C001BEE**.
5. Configure **SC** (Scan Channels) setting to **FFF**.
6. Click **Write radio settings** in the **Radio Configuration** toolbar to apply the new values to the module.
7. Once you have configured both modules, check to make sure they can see each other. Click **Discover radio modules in the same network**, the second button of the device panel in the **Radio Modules** view. The other device must be listed in the **Discovering remote devices** dialog.

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

DigiMesh devices

1. Click **Load default firmware settings** in the **Radio Configuration** toolbar to load the default values for the device firmware.
2. Ensure the API mode (API1 or API2) is enabled. To do so, the **AP** parameter value must be **1** (API mode without escapes) or **2** (API mode with escapes).
3. Configure **ID** (PAN ID) setting to **CAFE**.
4. Configure **CH** (Operating Channel) to **C**.
5. Click **Write radio settings** in the **Radio Configuration** toolbar to apply the new values to the module.
6. Once you have configured both modules, check to make sure they can see each other. Click **Discover radio modules in the same network**, the second button of the device panel in the **Radio Modules** view. The other device must be listed in the **Discovering remote devices** dialog.

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

DigiPoint devices

1. Click **Load default firmware settings** in the **Radio Configuration** toolbar to load the default values for the device firmware.
2. Ensure the API mode (API1 or API2) is enabled. To do so, the **AP** parameter value must be **1** (API mode without escapes) or **2** (API mode with escapes).
3. Configure **ID** (PAN ID) setting to **CAFE**.
4. Configure **HP** (Hopping Channel) to **5**.
5. Click **Write radio settings** in the **Radio Configuration** toolbar to apply the new values to the module.
6. Once you have configured both modules, check to make sure they can see each other. Click **Discover radio modules in the same network**, the second button of the device panel in the **Radio Modules** view. The other device must be listed in the **Discovering remote devices** dialog.

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

Cellular devices

1. Click **Load default firmware** settings in the Radio Configuration toolbar to load the default values for the device firmware.
2. Ensure the API mode (API1 or API2) is enabled. To do so, the **AP** parameter value must be **1** (API mode without escapes) or **2** (API mode with escapes).
3. Click **Write radio settings** in the Radio Configuration toolbar to apply the new values to the module.
4. Verify the module is correctly registered and connected to the Internet. To do so check that the LED on the development board blinks. If it is solid or has a double-blink, registration has not occurred properly. Registration can take several minutes.

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

Wi-Fi devices

1. Click **Load default firmware** settings in the Radio Configuration toolbar to load the default values for the device firmware.
2. Ensure the API mode (API1 or API2) is enabled. To do so, the **AP** parameter value must be **1** (API mode without escapes) or **2** (API mode with escapes).
3. Connect to an access point:

   1. Click the **Active Scan** button.
   2. Select the desired access point from the list of the **Active Scan** result dialog.
   3. If the access point requires a password, type your password.
   4. Click the **Connect** button and wait for the module to connect to the access point.

4. Click **Write radio settings** in the Radio Configuration toolbar to apply the new values to the module.
5. Verify the module is correctly connected to the access point by checking the IP address assigned to the module by reading the **MY** parameter and verifying it has a value different than **0.0.0.0**.

2.6.1.3 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:

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

   2. Instantiate a generic XBee device:

      ```
      > 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:

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

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

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

5. Close the connection with the device:

```
> device.close()
```

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:

Start delimiter	7E
Length	Depends on the XBee protocol
Frame type	Depends on the XBee protocol
16/64-bit source address	XBee sender's 16/64-bit address
Options	02
RF data/Received data	48 65 6C 6C 6F 20 58 42 65 65 20 57 6F 72 6C 64 21

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:

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

   2. Instantiate a Wi-Fi XBee device:

```
> 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:

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

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

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

5. Close the connection with the device:

```
> 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:

Start delimiter	7E
Length	00 1C
Frame type	B0
IPv4 source address	XBee Wi-Fi sender's IP address
16-bit dest port	26 16
16-bit source port	26 16
Protocol	00
Status	00
RF data	48 65 6C 6C 6F 20 58 42 65 65 20 57 6F 72 6C 64 21

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:

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

   2. Instantiate a cellular XBee device:

   ```
   > 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:

   ```
   > 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.

   ```
   > 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*.

```
> 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:

```
> device.close()
```

2.6.2 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.

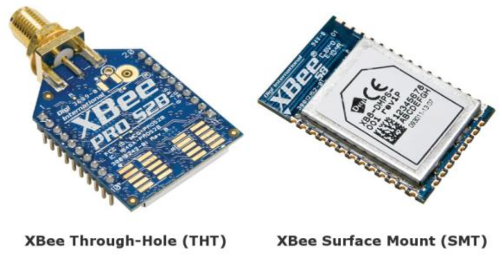
2.6.2.1 RF modules

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.

2.6.2.2 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 three form factors: Through-Hole, Surface Mount, and Micro, with different antenna options. Almost all modules are available in the Through-Hole form factor and share the same footprint.



2.6.2.3 Radio firmware

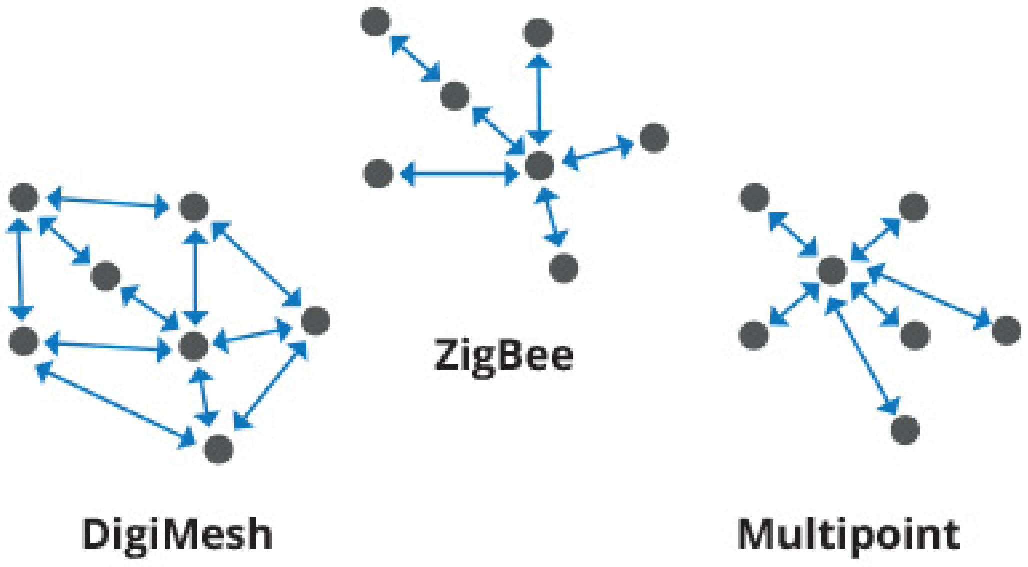
Radio firmware is the program code stored in the radio module's persistent memory that provides the control program for the device. From XCTU or the local web interface of the XBee Gateway, you can update or change the firmware of the local XBee module or any other module connected to the same network.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

2.6.2.4 Radio communication protocols

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

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

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



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

2.6.2.5 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.

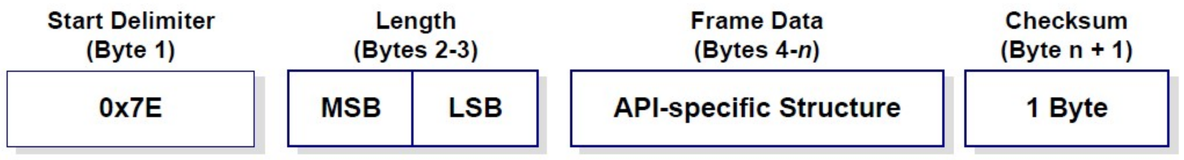
The following data bytes must be escaped:

- 0x7E: Frame delimiter
- 0x7D: Escape
- 0x11: XON
- 0x13: XOFF

2.6.2.6 API frames

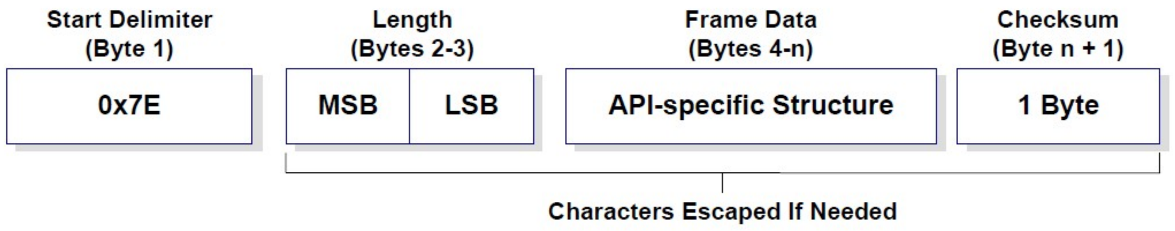
An API frame is the structured data sent and received through the serial interface of the radio module when it is configured in API or API escaped operating modes. API frames are used to communicate with the module or with other modules in the network.

An API frame has the following structure:



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.



2.6.2.7 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**.

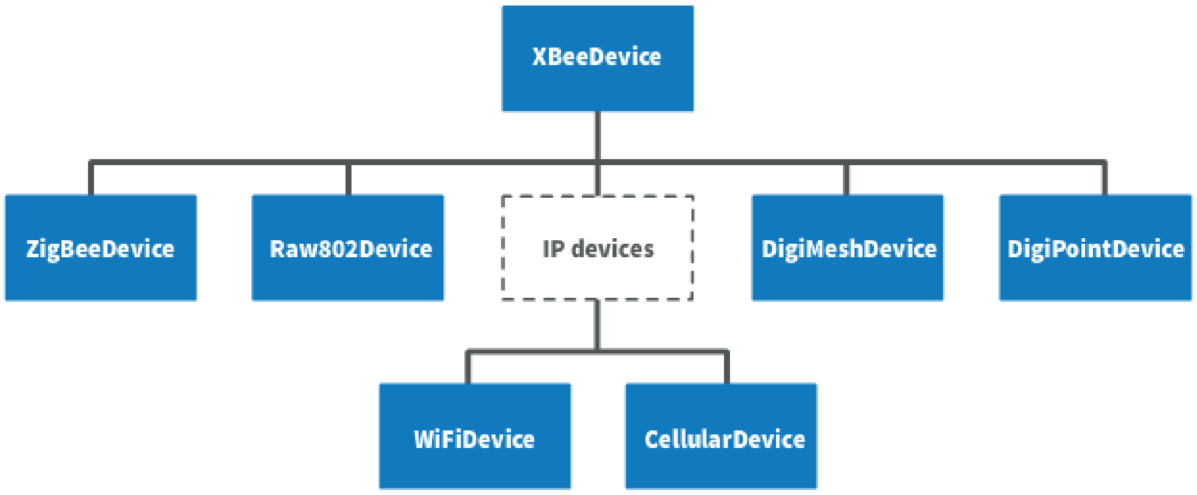
The configuration process of the AT settings varies depending on the operating mode of the XBee RF module.

- **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*.

2.6.3 Work with XBee classes

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

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



- XBee Zigbee device (`ZigBeeDevice`)
- XBee 802.15.4 device (`Raw802Device`)
- XBee DigiMesh device (`DigiMeshDevice`)
- XBee Point-to-multipoint device (`DigiPointDevice`)
- XBee IP devices (This is a non-instantiable class)

- XBee Cellular device (`CellularDevice`)
- XBee Wi-Fi device (`WiFiDevice`)

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

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

2.6.3.1 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:

Class	Description
XBeeDevice	Generic object, protocol-independent
ZigBeeDevice	Zigbee protocol
Raw802Device	802.15.4 protocol
DigiMeshDevice	DigiMesh protocol
DigiPointDevice	Point-to-multipoint protocol
CellularDevice	Cellular protocol
WiFiDevice	Wi-Fi protocol

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

```
[...]

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:

Class	Description
RemoteXBeeDevice	Generic object, protocol independent
RemoteZigBeeDevice	Zigbee protocol
RemoteRaw802Device	802.15.4 protocol
RemoteDigiMeshDevice	DigiMesh protocol
RemoteDigiPointDevice	Point-to-multipoint protocol

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

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

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

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

Instantiate a remote XBee device

```
[...]

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.

2.6.3.2 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.

```
[...]

# 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.

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

```
[...]

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

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

[...]
```

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

Read device information

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

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

- IMEI (only for cellular modules)

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

Initialize a remote XBee device

```
[...]

# 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"))

# 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

```
[...]

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

# Get the protocol of the device.
protocol = local_xbee.get_protocol()
```

Device operating mode

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

Get the operating mode

```
[...]

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

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

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

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

- API
- API with escaped characters

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

2.6.3.3 Close the XBee device connection

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

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

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

Close the connection

```
[...]

# 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.

2.6.4 Configure the XBee device

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

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

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*.

2.6.4.1 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:

Parameter	Method
64-bit address	**get_64bit_addr()**
16-bit address	**get_16bit_addr()**
Node identifier	**get_node_id()**
Firmware version	**get_firmware_version()**
Hardware version	**get_hardware_version()**
Role	**get_role()**

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.

Method	Description
read_device_info(init=False)	Updates cache parameters reading them from the XBee: If `init` is `True` it reads all values, else only those not initialized.

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:

Method	Description
set_node_id(String)	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.

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.

Method	Description
get_dest_address()	Returns the 64-bit address of the device that data will be reported to.
set_dest_address(XBee64BitAddress)	Specifies the 64-bit address of the device where the data will be reported.

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

Method	Description
get_pan_id()	Returns a byte array containing the ID of the Personal Area Network where the XBee device is operating.
set_pan_id(Bytearray)	Specifies the value in byte array format of the PAN ID where the XBee device should work.

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

Method	Description
get_power_level()	Returns a **PowerLevel** enumeration entry indicating the power level of the XBee device.
set_power_level(PowerLevel)	Specifies a **PowerLevel** enumeration entry containing the desired output level of the XBee device.

Configure non-cached parameters

```
[...]

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

# Set the destination address of the device.
dest_address = XBee64BitAddress.from_hex_string("0013A20040XXXXXX")
local_xbee.set_dest_address(dest_address)

# Read the operating PAN ID of the device.
dest_addr = local_xbee.get_dst_address()

# Read the operating PAN ID of the device.
pan_id = local_xbee.get_pan_id()

# Read the output power level.
p_level = local_xbee.get_power_level()

[...]
```

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

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

Example: Common parameters
The XBee Python Library includes a sample application that displays how to get and set common parameters. It can be located in the following path: **examples/configuration/ManageCommonParametersSample**

2.6.4.2 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.

Method	Description
get_parameter(String)	Specifies the AT command (string format) to retrieve its value. The method returns the value of the parameter in a byte array.

Get a parameter from the XBee device

```
[...]

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

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

[...]
```

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

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

Method	Description
set_parameter(String, Bytearray)	Specifies the AT command (String format) to be set in the device and a byte array containing the value of the parameter.

Set a parameter in the XBee device

```
[...]

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

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

[...]
```

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

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

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

Execute a command

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

Method	Description
execute_command(String)	Specifies the AT command (String format) to be run in the device.

Run a command in the XBee device

```
[...]

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

# Run the apply changes command.
```

(continues on next page)

(continued from previous page)

```
local_xbee.execute_command("AC")

[...]
```

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

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

2.6.4.3 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	Description	Notes
enable_apply_changes(Boolean)	Specifies whether the changes on settings and parameters are applied when set.	The apply configuration changes flag is enabled by default.
is_apply_changes_enabled()	Returns whether the XBee device is configured to apply parameter changes when they are set.	
apply_changes()	Applies the changes on parameters that were already set but are pending to be applied.	This method is useful when the XBee device is configured to not apply changes when they are set.

Apply configuration changes

```
[...]

# 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.
[...]
```

(continues on next page)

(continued from previous page)

```
# 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`.

2.6.4.4 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

```
[...]

# 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.

2.6.4.5 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

```
[...]

# 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**

2.6.4.6 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

Method	Description
set_ip_addressing_mode(IPAddressingMode)	Sets the IP addressing mode of the Wi-Fi module. Depending on the provided mode, network settings are configured differently: • **DHCP**: Network settings are assigned by a server. • **STATIC**: Network settings must be provided manually one by one.

Configure IP addressing mode

```
[...]

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

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

# Save the IP addressing mode.
local_xbee.write_changes()

[...]
```

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

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

Configure IP network settings

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

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

Parameter	Method
IP address	**get_ip_address()**
Subnet mask	**get_mask_address()**
Gateway IP	**get_gateway_address()**
DNS address	**get_dns_address()**

Read IP network settings

```
[...]

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

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

# Connect to access point with SSID 'My SSID' and password 'myPassword'
local_xbee.connect_by_ssid("My SSID", "myPassword")

# 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:

Parameter	Method
IP address	**set_ip_addr()**
Subnet mask	**set_mask_address()**
Gateway IP	**set_gateway_address()**
DNS address	**set_dns_address()**

2.6.4.7 Configure Bluetooth settings

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

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

Enable and disable Bluetooth

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

Method	Description
enable_bluetooth()	Enables the Bluetooth Low Energy interface of your XBee device.
disable_bluetooth()	Disables the Bluetooth Low Energy interface of your XBee device.

Enabling and disabling the Bluetooth interface

```
[...]

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

# Enable the Bluetooth interface.
local_xbee.enable_bluetooth()

[...]

# Disable the Bluetooth interface.
local_xbee.disable_bluetooth()

[...]
```

These methods may fail for the following reasons:

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

Configure the Bluetooth password

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

Method	Description
update_bluetooth_password(String)	Specifies the new Bluetooth password of the XBee device.

Configuring or changing the Bluetooth password

```
[...]

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

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

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

(continues on next page)

(continued from previous page)

```
[...]
```

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

```
[...]

# 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`.

2.6.5 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 nodes that form it.

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

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

- XBee Zigbee network (ZigBeeNetwork)
- XBee 802.15.4 network (Raw802Network)
- XBee DigiMesh network (DigiMeshNetwork)
- XBee DigiPoint network (DigiPointNetwork)

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.

The XBee network object can be retrieved from a local XBee after it has been opened with the method get_network().

Retrieve the XBee network

```
[...]

# Instantiate a local XBee 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 nodes that form the network and store them in a internal list. The XBeeNetwork object provides the following operations related to the XBee discovery feature:

- *Discovery types*
- *Deep discovery*
- *Standard discovery*
- *Discover the network*
- *Access discovered nodes*
- *Access connections between nodes*
- *Add and remove nodes manually*
- *Listen to network modification events*

2.6.5.1 Discovery types

There are two different types of discovery processes available in this API:

- *Deep discovery* finds network nodes and connections between them (including quality) even if they are sleeping. It also allows to establish a number of rounds to continually explore the network.
- *Standard discovery* only identifies network nodes. It may not discover sleeping nodes.

See *Discover the network* to know how to launch a deep or standard discovery process.

Note: In 802.15.4, both (deep and standard discovery) are the same and none discover the node connections nor their quality. The difference is the possibility of running more than one round using a deep discovery.

2.6.5.2 Deep discovery

This discovery process finds network nodes and their connections including the quality. It asks each node for its neighbors and retrieves information about the signal quality between them.

This mechanism also discovers sleeping nodes.

It is possible to configure the discovery process to run a specific number of times or even endlessly. Each discovery round is called a scan.

Deep discovery modes

This mode establishes the way the network deep discovery process is performed. Available modes are defined in the `NeighborDiscoveryMode` enumeration:

- **Cascade** (`NeighborDiscoveryMode.CASCADE`): The discovery of the neighbors of a node is requested once the previous request finishes. This means that just one discovery process is running at the same time. This mode is recommended for large networks, it might be a slower method but it generates less traffic than 'Flood'.
- **Flood** (`NeighborDiscoveryMode.FLOOD`): The discovery of the neighbors of a node is requested when the node is found in the network. This means that several discovery processes might be running at the same time. This might be a faster method, but it generates a lot of traffic and might saturate the network.

The default discovery mode is **Cascade**. You can configure the discovery mode with the method `set_deep_discovery_options(NeighborDiscoveryMode, Boolean).`

Configure the deep discovery process

Before discovering the nodes of a network, you can configure the settings of the process. The API provides two methods to configure the discovery timeout and discovery options.

Method	Description
set_deep_discovery_timeouts(Float, Float, Float)	Configures the deep discovery timeouts: • **node_timeout (Float, optional)**: Maximum duration in seconds of the discovery process used to find neighbors of a node. • **time_bw_requests (Float, optional)**: Time to wait between node neighbors requests (in seconds) – For cascade: Time to wait after completion of the a node neighbor discovery process and before next node request. – For flood: Minimum time to wait between each neighbor request. • **time_bw_scans (Float, optional)**: Time to wait before starting a new network scan (in seconds)
set_deep_discovery_options(NeighborDiscoveryMode, Boolean)	Configures the deep discovery options: • **deep_mode (NeighborDiscoveryMode, optional)**: Neighbor discovery mode, the way to perform the network discovery process. See **:ref:‘deepDiscoveryMode‘** • **del_not_discovered_nodes_in_last_scan (Boolean, optional)**: `True` to remove nodes from the network if they were not discovered in the last scan.

Configure deep discovery timeout and options

```
[...]

# Instantiate a local XBee object.
xbee = XBeeDevice(...)

[...]

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

# Configure the discovery options.
xnet.set_deep_discovery_options(deep_mode=NeighborDiscoveryMode.CASCADE,
                                del_not_discovered_nodes_in_last_scan=False)

# Configure the discovery timeout, in SECONDS.
xnet.set_deep_discovery_timeout(node_timeout=30, time_bw_requests=10,
                                time_bw_scans=20)

[...]
```

2.6.5.3 Standard discovery

This type of discovery process only finds network nodes, it does not include information about the quality of the connections between them.

XBee nodes sleeping may not respond to this request, this means, it may not be found using this discovery process type.

The discovery process runs until the configured timeout expires or, in case of 802.15.4, until the 'end' packet is received (see *Configure the standard discovery process*)

Configure the standard discovery process

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

Method	Description
set_discovery_timeout(Float)	Configures the discovery timeout (`NT` parameter) with the given value in seconds.
set_discovery_options(Set<DiscoveryOptions>)	Configures the discovery options (`NO` parameter) with the set of options. The set of discovery options contains the different `DiscoveryOptions` configuration values that are applied to the local XBee module when performing the discovery process. These options are the following: • **DiscoveryOptions.APPEND_DD**: Appends the device type identifier (`DD`) to the information retrieved when a node is discovered. This option is valid for DigiMesh, Point-to-multipoint (Digi Point) and Zigbee protocols. • **DiscoveryOptions.DISCOVER_MYSELF**: The local XBee is returned as a discovered node. This option is valid for all protocols. • **DiscoveryOptions.APPEND_RSSI**: Appends the RSSI value of the last hop to the information retrieved when a node is discovered. This option is valid for DigiMesh and Point-to-multipoint (Digi Point) protocols.

Configure discovery timeout and options

```
[...]

# Instantiate a local XBee 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)

(continued from previous page)

```
[...]
```

2.6.5.4 Discover the network

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

Method	Description
start_discovery_process(Boolean, Integer)	Starts the discovery process, saving the remote XBee found inside the `XBeeNetwork` object. • **deep (Boolean, optional)**: `True` for a deep network scan, `False` otherwise. See *Discovery types*. • **n_deep_scans (Integer, optional)**: Number of discovery scans to perform. Only for deep discovery.

When a discovery process has started, you can monitor and manage it using the following methods provided by the `XBeeNetwork` object:

Method	Description
is_discovery_running()	Returns whether or not the discovery process is running.
stop_discovery_process()	Stops the discovery process that is taking place.

Warning: For a standard discovery and depending on your hardware and firmware version, although you call the `stop_discovery_process` method, DigiMesh and DigiPoint modules are blocked until the configured discovery time has elapsed. This means, if you try to get or set any parameter during that time, a `TimeoutException` may be thrown. This does not occur for:

- XBee 3 modules running DigiMesh firmware 300B or higher.
- XBee SX modules running firmware A008 or higher, 9008 or higher.

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

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

Discover the network (deep)

```
[...]

# Instantiate a local XBee object.
xbee = XBeeDevice(...)

# Get the XBee network object from the local XBee.
xnet = xbee.get_network()

# Start the discovery process and wait for it to be over.
```

(continues on next page)

(continued from previous page)

```
xnet.start_discovery_process(deep=True, n_deep_scans=1)
while xnet.is_discovery_running():
    time.sleep(0.5)

# Get the list of the nodes in the network.
nodes = xnet.get_devices()

[...]
```

Discover the network (standard)

```
[...]

# Instantiate a local XBee object.
xbee = XBeeDevice(...)

# Get the XBee network object from the local XBee.
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 the list of the nodes in the network.
nodes = xnet.get_devices()

[...]
```

Discover the network with an event notification

The API also allows you to add a discovery event listener to notify when:

- New nodes are discovered.
- The process finishes.
- An error occurs during the process.

Notify new discovered nodes

To get notifications when nodes are discovered, you must provide a callback before starting the discovery process using the `add_device_discovered_callback()` method.

Add a callback to device discovered event

```
[...]

# Instantiate a local XBee object.
xbee = XBeeDevice(...)

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

(continues on next page)

(continued from previous page)

```
# Get the XBee network object from the local XBee.
xnet = xbee.get_network()

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

# Start the discovery process.
xnet.start_discovery_process(deep=True)

[...]
```

Every time a new remote XBee node is discovered all registered device discovered callbacks are executed, even if the discovered node is already in the node list of the network. Each callback receives a `RemoteXBeeDevice` as argument, with all the available information. Unknown parameters of this remote node are `None`.

Notify discovery finishes

To get notifications when a discovery process finishes, you must provide a callback before starting the discovery process using the `add_discovery_process_finished_callback()` method.

Add a callback to discovery process finished event

```
[...]

# Instantiate a local XBee 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)

[...]
```

When a discovery process finishes (either successfully or with an error), all registered discovery finished callbacks 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 network discovery using a callback. It can be located in the following path: **examples/network/DiscoverDevicesSample/DiscoverDevicesSample.py**

Discover specific nodes

The `XBeeNetwork` object also provides methods to discover specific nodes within a network. This may be useful, for example, if you only need to work with a particular remote node.

Method	Description
discover_device(String)	Specify the node identifier of the XBee to find. Returns the remote XBee whose node identifier equals the one provided or `None` if the node was not found. In the case of more than one coincidences, it returns the first one.
discover_devices([String])	Specify the node identifiers of the XBee nodes to find. Returns a list with the remote XBee nodes whose node identifiers equal those provided.

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

Note: These methods may not discover sleeping nodes.

Discover specific nodes

```
[...]

# Instantiate a local XBee object.
xbee = XBeeDevice(...)

[...]

# Get the XBee network object from the local XBee.
xnet = xbee.get_network()

# Discover the remote node whose node ID is 'SOME NODE ID'.
remote = xnet.discover_device("SOME NODE ID")

# Discover the remote nodes whose node IDs are 'ID 2' and 'ID 3'.
remote_list = xnet.discover_devices(["ID 2", "ID 3"])

[...]
```

2.6.5.5 Access discovered nodes

Once a discovery process finishes, the discovered nodes are saved inside the `XBeeNetwork` object. You can get a list of discovered nodes at any time using the `get_devices()`.

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

Method	Description
get_devices()	Returns a copy of the list of remote XBee nodes. If any node is added to the network after calling this method, the returned list is not updated.
get_device_by_64(XBee64BitAddress)	Returns the remote node already in the network whose 64-bit address matches the given one or `None` if the node is not in the network.
get_device_by_16(XBee16BitAddress)	Returns the remote node already in the network whose 16-bit address matches the given one or `None` if the node is not in the network.
get_device_by_node_id(String)	Returns the remote node already in the network whose node identifier matches the given one or `None` if the node is not in the network.

Access discovered nodes

```
[...]

# Instantiate a local XBee object.
xbee = XBeeDevice(...)

# Get the XBee network object from the local XBee.
xnet = xbee.get_network()

[...]

x64addr = XBee64BitAddress(...)
node_id = "SOME_XBEE"

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

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

[...]
```

2.6.5.6 Access connections between nodes

A deep discovery process stores the connections between found nodes inside the `XBeeNetwork` object. You can get these connections using the `get_connections()` method.

This is the list of methods provided by the `XBeeNetwork` object that allow you to retrieve the connections between nodes:

Method	Description
get_connections()	Returns a copy of the network connections. If any connection is added after the execution of this method, returned list is not updated.
get_node_connections(AbstractXBeeDevice)	Returns the connections with the provided node in one of its ends. If any connection is added after the execution of this method, returned list is not updated.

Warning: A deep discovery process must be performed to have network connections available.

Each `Connection` object contains:

- The two nodes between this connection is established.
- The link quality of the connection in both directions (`LinkQuality`):
  - From node A to node B
  - From node B to node A
- The connection status in both directions (`RouteStatus`), active, inactive, etc:
  - From node A to node B
  - From node B to node A

Access network connections

```
[...]

# Instantiate a local XBee object.
xbee = XBeeDevice(...)

# Get the XBee network object from the local XBee.
xnet = xbee.get_network()

[...]

# Start the discovery process and wait for it to be over.
xnet.start_discovery_process(deep=True, n_deep_scans=1)
while xnet.is_discovery_running():
    time.sleep(0.5)

print("%s" % '\n'.join(map(str, xnet.get_connections())))

[...]
```

2.6.5.7 Add and remove nodes manually

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

Note: These methods modifies the list of nodes inside the `XBeeNetwork` object, but do not change the real XBee network. They do not trigger a node join event, a disassociation, or a network reset.

Manually add nodes to the XBee network

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

Method	Description
add_remote(RemoteXBeeDevice)	Specifies the remote XBee to add to the list of remote nodes of the `XBeeNetwork` object. **Notice** that this operation does not join the remote XBee to the network; it just adds that node to the list. The node is added to the node list, but may not be physically in the same network. **Note** that if the given node already exists in the network, it will not be added, but the node in the current network will be updated with the known parameters of the given node. This method returns the same node with its information updated. If the node was not in the list yet, this method returns it without changes.
add_remotes([RemoteXBeeDevice])	Specifies the remote XBee nodes to add to the list of remote nodes of the `XBeeNetwork` object. **Notice** that this operation does not join the remote XBee nodes to the network; it just adds those nodes to the list. Nodes are added to the node list but may not be physically in the same network.

Add a remote node manually to the network

```
[...]

# Instantiate a local XBee object.
xbee = XBeeDevice(...)
```

(continues on next page)

(continued from previous page)

```
[...]

# Get the XBee network object from the local XBee.
xnet = xbee.get_network()

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

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

[...]
```

Remove an existing node from the XBee network

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

Method	Description
remove_device(RemoteXBeeDevice)	Specifies the remote XBee to remove from the list of remote nodes of the XBeeNetwork object. If the node was not contained in the list, the method will raise a `ValueError`. **Notice** that this operation does not disassociates the remote XBee from the actual XBee network; it just deletes the node from the network object list. However, next time you perform a discovery, it could be added again automatically.

Remove a remote node from the network

```
[...]

# Instantiate a local XBee object.
xbee = XBeeDevice(...)

[...]

# Get the XBee network object from the local XBee.
xnet = xbee.get_network()

# Get the remote XBee and add it to the network.
remote = xnet.get_remote(...)
xnet.add_remote(remote)

# Remove the remote node from the network.
xnet.remove_device(remote)

[...]
```

Clear the list of remote XBee nodes from the XBee network

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

Method	Description
clear()	Removes all the devices from the list of remote nodes of the network. **Notice** that this does not imply dismantling the XBee the actual XBee network; it just clears the list of nodes in the `XBeeNetwork` object. Next time you perform a discovery, the list could be filled with the found remote XBee nodes.

Clear the list of remote nodes

```
[...]

# Instantiate a local XBee object.
xbee = XBeeDevice(...)

[...]

# Get the XBee network object from the local XBee.
xnet = xbee.get_network()

# Discover XBee devices in the network and add them to the list of nodes.
[...]

# Clear the list of nodes.
xnet.clear()

[...]
```

2.6.5.8 Listen to network modification events

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

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

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

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

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

  Args:
    event_type (:class:`.NetworkEventType`): The type of modification.
    reason (:class:`.NetworkEventReason`): The cause of the modification.
    node (:class:`.AbstractXBeeDevice`): The node involved in the
      modification (``None`` for ``NetworkEventType.CLEAR`` events)
  """
  [...]
```

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

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

Register a network modifications callback

```
[...]

# Define the network modified callback.
def cb_network_modified(event_type, reason, node):
  print("  >>>> Network event:")
  print("         Type: %s (%d)" % (event_type.description, event_type.code))
  print("         Reason: %s (%d)" % (reason.description, reason.code))

  if not node:
    return

  print("         Node:")
  print("            %s" % node)

xnet = xbee.get_network()

# Add the network modified callback.
xnet.add_network_modified_callback(cb_network_modified)

[...]
```

Network events

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

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

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

- `NetworkEventReason.DISCOVERED`: The node was added/removed/updated during a standard discovery process.
- `NetworkEventReason.NEIGHBOR`: The node was added/removed/updated during a deep discovery process.
- `NetworkEventReason.RECEIVED_MSG`: The node was added/updated after receiving a message from it.
- `NetworkEventReason.ROUTE`: The node was added/updated as a hop of a received network route.
- `NetworkEventReason.READ_INFO`: The node was updated after refreshing its information.

- `NetworkEventReason.FIRMWARE_UPDATE`: The node was updated/removed, or the network cleared after a firmware update.
- `NetworkEventReason.MANUAL`: The node was manually added/updated/removed, or the network cleared.

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

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

Example: Network modifications
The XBee Python Library includes a sample application that displays how to receive network modification events. It can be located in the following path: **examples/network/NetworkModificationsSample/NetworkModificationsSample.py**

2.6.6 Communicate with XBee devices

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

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

2.6.6.1 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:

Method	Description
send_data(RemoteXBeeDevice, String or Bytearray, Integer)	Specifies the remote XBee destination object, the data to send and optionally the transmit options.

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

XBee class	Method	Description
ZigBeeDevice	**send_data_64_16(XBee64BitAddress, XBee16BitAddress, String or Bytearray, Integer)**	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`.
Raw802Device	**send_data_16(XBee16BitAddress, String or Bytearray, Integer)**	Specifies the 16-bit destination address, the data to send and optionally the transmit options.
	send_data_64(XBee64BitAddress, String or Bytearray, Integer)	Specifies the 64-bit destination address, the data to send and optionally the transmit options.
DigiMeshDevice	**send_data_64(XBee64BitAddress, String or Bytearray, Integer)**	Specifies the 64-bit destination address, the data to send and optionally the transmit options.
DigiPointDevice	**send_data_64_16(XBee64BitAddress, XBee16BitAddress, String or Bytearray, Integer)**	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`.

Send data synchronously

```
[...]

# 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

```
[...]

NEW_TIMEOUT_FOR_SYNC_OPERATIONS = 5 # 5 seconds

device = [...]

# Retrieving the configured timeout for synchronous operations.
print("Current timeout: %d seconds" % device.get_sync_ops_timeout())

[...]

# Configuring the new timeout (in seconds) for synchronous operations.
device.set_sync_ops_timeout(NEW_TIMEOUT_FOR_SYNC_OPERATIONS)

[...]
```

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:

Method	Description
send_data_async(RemoteXBeeDevice, String or Bytearray, Integer)	Specifies the remote XBee destination object, the data to send and optionally the transmit options.

Protocol-specific classes offer some other asynchronous unicast transmission methods in addition to the one provided by the XBeeDevice object:

XBee class	Method	Description
ZigBeeDevice	**send_data_async_64_16(XBee64BitAddress, XBee16BitAddress, String or Bytearray, Integer)**	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`.
Raw802Device	**send_data_async_16(XBee16BitAddress, String or Bytearray, Integer)**	Specifies the 16-bit destination address, the data to send and optionally the transmit options.
	send_data_async_64(XBee64BitAddress, String or Bytearray, Integer)	Specifies the 64-bit destination address, the data to send and optionally the transmit options.
DigiMeshDevice	**send_data_async_64(XBee64BitAddress, String or Bytearray, Integer)**	Specifies the 64-bit destination address, the data to send and optionally the transmit options.
DigiPointDevice	**send_data_async_64_16(XBee64BitAddress, XBee16BitAddress, String or Bytearray, Integer)**	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`.

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:

Method	Description
send_data_broadcast(String or Bytearray, Integer)	Specifies the data to send and optionally the transmit options.

Send broadcast data

```
[...]

# 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:

Method	Description
read_data(Integer)	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.

Reading data from any remote XBee device (polling)

```
[...]

# 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

```
[...]

xbee_message = device.read_data()

remote_device = xbee_message.remote_device
data = xbee_message.data
is_broadcast = xbee_message.is_broadcast
timestamp = xbee_message.timestamp

[...]
```

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

Method	Description
read_data_from(RemoteXBeeDevice, Integer)	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.

Read data from a specific remote XBee device (polling)

```
[...]

# Instantiate an XBee device object.
```

(continues on next page)

(continued from previous page)

```
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

```
[...]

# 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)

[...]
```

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

```
[...]

def my_data_received_callback(xbee_message):
    [...]

device.add_data_received_callback(my_data_received_callback)

[...]

# Delete the callback
device.del_data_received_callback(my_data_received_callback)

[...]
```

Example: Register for data reception

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

2.6.6.2 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 on 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:

Method	Description
send_expl_data(RemoteXBeeDevice, Integer, Integer, Integer, Integer, String or Bytearray, Integer)	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.

Send unicast explicit data synchronously

```
[...]

# 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:

Method	Description
send_expl_data_async(RemoteXBeeDevice, Integer, Integer, Integer, Integer, String or Bytearray, Integer)	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.

Send unicast explicit data asynchronously

```
[...]

# 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:

Method	Description
send_expl_data_broadcast(Integer, Integer, Integer, Integer, String or Bytearray, Integer)	Specifies the four application layer fields (source endpoint, destination endpoint, cluster ID, and profile ID), the data to send and optionally the transmit options.

Send broadcast data

```
[...]

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

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

[...]
```

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

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

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

Receive explicit data

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

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

Method	Description
get_api_output_mode_value()	Returns the API output mode of the data received by the XBee device.
set_api_output_mode_value(Integer)	Specifies the API output mode of the data received by the XBee device. Calculate the mode with the method *calculate_api_output_mode_value* with a set of *APIOutputModeBit*.

Set API output mode

```
[...]

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

# Set explicit output mode
mode = APIOutputModeBit.calculate_api_output_mode_value(device.get_protocol(),
  {APIOutputModeBit.EXPLICIT})
device.set_api_output_mode_value(mode)
```

(continues on next page)

(continued from previous page)

```
# Set native output mode
mode = 0
device.set_api_output_mode_value(mode)

# Set explicit plus unsupported ZDO request pass-through
mode = APIOutputModeBit.calculate_api_output_mode_value(device.get_protocol(),
  {APIOutputModeBit.EXPLICIT, APIOutputModeBit.UNSUPPORTED_ZDO_PASSTHRU})
device.set_api_output_mode_value(mode)

[...]
```

Once you have configured the device to receive data in explicit format, you can read it using one of the following mechanisms provided by the XBee device object.

Polling for explicit data

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

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

Read explicit data from any remote XBee device (polling)

```
[...]

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

# Read data.
xbee_message = device.read_expl_data()

[...]
```

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

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

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

Get the ExplicitXBeeMessage information

```
[...]

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:

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

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

```
[...]

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

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

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

[...]
```

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

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

Example: Receive explicit data with polling
The XBee Python Library includes a sample application that demonstrates how to receive explicit data using the polling mechanism. It can be located in the following path: **examples/communication/explicit/ReceiveExplicitDataPollingSample**

Explicit data reception callback

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

Explicit data reception registration

```
[...]

# 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 unsubscribe the already-registered callback.

Explicit data reception deregistration

```
[...]

def my_expl_data_received_callback(xbee_message):
    [...]

device.add_expl_data_received_callback(my_expl_data_received_callback)

```

(continues on next page)

(continued from previous page)

```
[...]

# 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 (API output mode greater than 0) and another device sends non-explicit data or a IO sample, you receive an explicit message whose application layer field values are:

- For remote data:
  - Source endpoint: 0xE8
  - Destination endpoint: 0xE8
  - Cluster ID: 0x0011
  - Profile ID: 0xC105
- For remote IO sample:
  - Source endpoint: 0xE8
  - Destination endpoint: 0xE8
  - Cluster ID: 0x0092
  - Profile ID: 0xC105

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

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

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

2.6.6.3 Send and receive IP data

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

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

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

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

Send IP data

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

Synchronous operation

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

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

Method	Description
send_ip_data(IPv4Address, Integer, IPProtocol, String or Bytearray, Boolean)	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).

Send network data synchronously

```
[...]

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

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

xbee.send_ip_data(dest_addr, dest_port, protocol, data)

[...]
```

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

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

Example: Transmit IP data synchronously
The XBee Python Library includes a sample application that demonstrates how to send IP data. You can locate the example in the following path: **examples/communication/ip/SendIPDataSample**

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

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

Asynchronous operation

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

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

Method	Description
send_ip_data_async(IPv4Address, Integer, IPProtocol, String or Bytearray, Boolean)	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).

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:

Method	Description
start_listening(Integer)	Starts listening for incoming IP transmissions in the provided port.

Listen for incoming transmissions

```
[...]


# 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:

Method	Description
stop_listening()	Stops listening for incoming IP transmissions.

Stop listening for incoming transmissions

```
[...]

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

(continues on next page)

(continued from previous page)

```
# 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.

Method	Description
read_ip_data(Integer)	Specifies the time to wait in seconds for IP data reception (method blocks during that time or until IP data is received). If you don't specify a timeout, the method uses the default receive timeout configured in **XBeeDevice**.

Read IP data (polling)

```
[...]

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

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

[...]
```

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

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

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

Get the IPMessage information

```
[...]

# Instantiate a cellular device object.
```

(continues on next page)

(continued from previous page)

```
device = CellularDevice("COM1", 9600)
device.open()

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


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

[...]
```

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

Read IP data from a specific IP address (polling)

```
[...]

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

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

[...]
```

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

Example: Receive IP data with polling
The XBee Python Library includes a sample application that demonstrates how to receive IP data using the polling mechanism. You can locate the example in the following path: **examples/communication/ip/ConnectToEchoServerSample**

IP data reception callback

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

IP data reception registration

```
[...]

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


# Define the callback.
```

(continues on next page)

(continued from previous page)

```
def my_ip_data_received_callback(ip_message):
    print("Received IP data from %s: %s" % (ip_message.ip_addr, ip_message.data))

# Add the callback.
device.add_ip_data_received_callback(my_ip_data_received_callback)

[...]
```

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

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

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

Data reception deregistration

```
[...]

device = [...]

def my_ip_data_received_callback(ip_message):
    [...]

device.add_ip_data_received_callback(my_ip_data_received_callback)

[...]

# Delete the IP data callback.
device.del_ip_data_received_callback(my_ip_data_received_callback)

[...]
```

Example: Receive IP data with listener
The XBee Python Library includes a sample application that demonstrates how to receive IP data using the listener. You can locate the example in the following path: **examples/communication/ip/ReceiveIPDataSample**

2.6.6.4 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:

Method	Description
send_sms(String, String)	Specifies the the phone number to send the SMS to and the data to send as the body of the SMS message.

Send SMS message synchronously

```
[...]

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

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

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

[...]
```

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

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

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

Asynchronous operation

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

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

Method	Description
send_sms_async(String, String)	Specifies the the phone number to send the SMS to and the data to send as the body of the SMS message.

Send SMS message asynchronously

```
[...]

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

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

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

[...]
```

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

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

Receive SMS messages

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

SMS reception callback

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

SMS reception registration

```
[...]

# 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 `SMSMessage` object as paramater. 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

```
[...]

device = [...]

def my_sms_callback(sms_message):
    [...]

device.add_sms_callback(my_sms_callback)

[...]

# 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**

2.6.6.5 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:

Method	Description
send_bluetooth_data(Bytearray)	Specifies the data to send to the Bluetooth Low Energy interface.

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

Send data to Bluetooth

```
[...]

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

data = "Bluetooth, are you there?"

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

[...]
```

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

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

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

Receive Bluetooth data

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

Bluetooth data reception registration

```
[...]

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

(continues on next page)

(continued from previous page)

```
device.open()

# Define the callback.
def my_bluetooth_data_callback(data):
    print("Data received from the Bluetooth interface >> '%s'" % 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.

Deregister Bluetooth data reception

```
[...]

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**

2.6.6.6 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:

Method	Description
send_micropython_data(Bytearray)	Specifies the data to send to the MicroPython interface.

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

Send data to MicroPython

```
[...]

# 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

```
[...]

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

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

(continues on next page)

(continued from previous page)

```
# 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

```
[...]

device = [...]

def my_micropython_data_callback(data):
    [...]

device.add_micropython_data_received_callback(my_micropython_data_callback)

[...]

# 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**

2.6.6.7 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

```
[...]

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

(continues on next page)

(continued from previous page)

```
# 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

```
[...]

device = [...]

def my_modem_status_callback(status):
    [...]

device.add_modem_status_received_callback(my_modem_status_callback)

[...]

# 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**

2.6.6.8 Communicate using XBee sockets

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

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

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

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

Create an XBee socket

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

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

Create an XBee socket

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

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

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

Work with an XBee socket

Once the XBee socket is created, you can work with it to behave as a client or a server. The API offers the following methods:

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

Client sockets

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

Work with an XBee socket as client

```
[...]

HOST = "numbersapi.com"
PORT = "80"
```

(continues on next page)

(continued from previous page)

```
REQUEST = "GET /random/trivia HTTP/1.1\r\nHost: numbersapi.com\r\n\r\n"

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

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

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

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

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

Server sockets

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

Work with an XBee socket as server

```
[...]

PORT = "1234"

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

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

    # Listen for new connections.
    sock.listen()

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

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

(continues on next page)

(continued from previous page)

```
            if data:
                print(data.decode("utf8"))
```

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

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

2.6.7 Handle analog and digital IO lines

All the XBee modules, regardless of the protocol they run, have a set of IO 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.

2.6.7.1 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

```
[...]

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

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

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

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

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

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

[...]
```

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

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

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

Get IO configuration

```
[...]

# 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

```
[...]

# 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

```
[...]

# 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

```
[...]

# 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`.

The `get_pwm_duty_cycle(IOLine)` method of a PWM line returns a double value with the current duty cycle percentage of the PWM.

Get the duty cycle of an IO line configured as PWM

```
[...]

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

[...]
```

(continues on next page)

(continued from previous page)

```
# 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.

2.6.7.2 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

```
[...]

# 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

```
[...]

# 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

```
[...]

# Instantiate an XBee device object.
```

(continues on next page)

(continued from previous page)

```
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`.

2.6.7.3 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

```
[...]

# 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

```
[...]

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

[...]

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

[...]
```

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

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

Register an IO sample listener

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

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

Add an IO sample callback

```
[...]

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

[...]

# Define the IO sample receive callback.
```

(continues on next page)

(continued from previous page)

```
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.
local_xbee.add_io_sample_received_callback(io_sample_callback)

[...]
```

This callback function will receive three parameters when an IO sample receive event is raised:

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

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

Remove an IO sample callback

```
[...]

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

[...]

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

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

[...]

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

[...]
```

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

Example: Receive IO samples
The XBee Python Library includes a sample application that demonstrates how to configure a remote device to monitor IO lines and receive the IO samples in the local device. The example is located in the following path: **examples/io/IOSamplingSample/IOSamplingSample.py**

2.6.8 Update the XBee

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

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

Warning:

At the moment, update features are only supported in:

- **XBee 3:**
  - Local and remote firmware updates
  - Local and remote file system updates
  - Local and remote profile updates
- **XBee SX 868/900 MHz**
  - Local and remote firmware updates
  - Local and remote profile updates
- **XBee S2C**
  - Remote firmware updates
  - Remote profile updates

2.6.8.1 Update the XBee firmware

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

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

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

Warning:

At the moment, firmware update is only supported in:

- **XBee 3**: Local and remote firmware updates
- **XBee SX 868/900 MHz**: Local and remote firmware updates
- **XBee S2C**: Remote firmware updates

Update the firmware of a local XBee

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

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

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

Warning: At the moment, local firmware update is only supported in **XBee 3** and **XBee SX 868/900 MHz** devices.

Example: Local Firmware Update
The XBee Python Library includes a sample application that displays how to perform a local firmware update. It can be located in the following path: **examples/firmware/LocalFirmwareUpdateSample/LocalFirmwareUpdateSample.py**

Update the local firmware using an XBee device object

If you have an object instance of your local XBee device, you have to call the `update_firmware` method of the `XBeeDevice` class providing the required parameters:

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

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

- The device does not support the firmware update operation, throwing a `OperationNotSupportedException`.
- There is an error during the firmware update operation, throwing a `FirmwareUpdateException`.
- Other errors caught as `XBeeException`:

- The device is not open, throwing a generic `XBeeException`.
- The operating mode of the local XBee device is not `API_MODE` or `ESCAPED_API_MODE`, throwing an `InvalidOperatingModeException`.

Update local XBee device firmware using an XBee device object

```
[...]

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

[...]

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

[...]

# Update the XBee device firmware.
device.update_firmware(XML_FIRMWARE_FILE,
                       xbee_firmware_file=XBEE_FIRMWARE_FILE,
                       bootloader_firmware_file=BOOTLOADER_FIRMWARE_FILE,
                       progress_callback=progress_callback,)

[...]
```

Update the local firmware using a serial port

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

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

Method	Description
update_local_firmware(String or XBeeDevice, String, String, String, Integer, Function)	Performs a local firmware update operation in the given target. • **target (String or :class:'.XBeeDevice')**: target of the firmware upload operation. * **String**: serial port identifier. * **:class:'.AbstractXBeeDevice'**: the XBee device to upload its firmware. • **xml_firmware_file (String)**: path of the XML file that describes the firmware to upload. • **xbee_firmware_file (String, optional)**: location of the XBee binary firmware file (*.gbl). • **bootloader_firmware_file (String, optional)**: location of the bootloader binary firmware file. • **timeout (Integer, optional)**: the maximum amount of seconds to wait for target read operations during the update process. • **progress_callback (Function, optional)**: function to execute to receive progress information. Receives two arguments: – The current update task as a String – The current update task percentage as an Integer

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

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

Update local XBee device firmware using a serial port

```
import digi.xbee.firmware

[...]

SERIAL_PORT = "COM1"

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

[...]

# Update the XBee device firmware using the serial port name.
firmware.update_local_firmware(SERIAL_PORT,
                               XML_FIRMWARE_FILE,
                               xbee_firmware_file=XBEE_FIRMWARE_FILE,
                               bootloader_firmware_file=BOOTLOADER_FIRMWARE_FILE,
                               progress_callback=progress_callback,)

[...]
```

Update the firmware of a remote XBee

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

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

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

Warning: At the moment, remote firmware update is only supported in **XBee 3**, **XBee SX 868/900 MHz**, and **XBee S2C** devices.

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

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

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

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

Update remote XBee device firmware

```
[...]

XML_FIRMWARE_FILE = "my_path/my_firmware.xml"
OTA_FIRMWARE_FILE = "my_path/my_firmware.ota"
OTB_FIRMWARE_FILE = "my_path/my_firmware.otb"

REMOTE_DEVICE_NAME = "REMOTE"

[...]

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

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

# Get the remote device.
remote = xnet.discover_device(REMOTE_DEVICE_NAME)

# Update the remote XBee device firmware.
remote.update_firmware(SERIAL_PORT,
                       XML_FIRMWARE_FILE,
                       xbee_firmware_file=OTA_FIRMWARE_FILE,
                       bootloader_firmware_file=OTB_FIRMWARE_FILE,
                       progress_callback=progress_callback,)

[...]
```

Example: Remote Firmware Update
The XBee Python Library includes a sample application that displays how to perform a remote firmware update. It can be located in the following path: **examples/firmware/RemoteFirmwareUpdateSample/RemoteFirmwareUpdateSample.py**

2.6.8.2 Update the XBee file system

XBee 3 devices feature file system capabilities, meaning that they are able to persistently store files and folders in flash. The XBee Python Library provides classes and methods to manage these files.

- *Create file system manager*
- *File system operations*

Warning: At the moment file system capabilities are only supported in **XBee 3** devices.

Create file system manager

A `LocalXBeeFileSystemManager` object is required to work with local devices file system. You can instantiate this class by providing the local XBee device object. Once you have the object instance, you must call the `connect` method to open the file system connection and leave it ready to work.

Warning: File system operations take ownership of the serial port, meaning that you will stop receiving messages from the device until file system connection is closed. For this reason it is highly recommended to call the `disconnect` method of the file system manager as soon as you finish working with it.

Method	Description
connect()	Connects the file system manager.
disconnect()	Disconnects the file system manager and restores the device connection.

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

- The device does not support the file system capabilities, throwing a `FileSystemNotSupportedException`.
- There is an error during the connect operation, throwing a `FileSystemException`.

Create a local file system manager

```
from digi.xbee.filesystem import LocalXBeeFileSystemManager

[...]

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

[...]

# Create the file system manager and connect it.
filesystem_manager = LocalXBeeFileSystemManager(xbee)
filesystem_manager.connect()

[...]

filesystem_manager.disconnect()

[...]
```

File system operations

The file system manager provides several methods to navigate through the device file system and operate with the different files and folders:

Method	Description
get_current_directory()	Returns the current device directory.
change_directory(String)	Changes the current device working directory to the given one. • **directory (String)**: the new directory to change to.
make_directory(String)	Creates the provided directory. • **directory (String)**: the new directory to create.
list_directory(String)	Lists the contents of the given directory. • **directory (String, optional)**: the directory to list its contents. Optional. If not provided, the current directory contents are listed.
remove_element(String)	Removes the given file system element path. • **element_path (String)**: path of the file system element to remove.
move_element(String, String)	Moves the given source element to the given destination path. • **source_path (String)**: source path of the element to move. • **dest_path (String)**: destination path of the element to move.
put_file(String, String, Boolean, Function)	Transfers the given file in the specified destination path of the XBee device. • **source_path (String)**: the path of the file to transfer. • **dest_path (String)**: the destination path to put the file in. • **secure (Boolean, optional)**: `True` if the file should be stored securely, `False` otherwise. Defaults to `False`. • **progress_callback (Function, optional)**: function to execute to receive progress information. Takes the following arguments: – The progress percentage as integer.
put_dir(String, String, Function)	Uploads the given source directory contents into the given destination directory in the device. • **source_dir (String)**: the local directory to upload its contents. • **dest_dir (String, optional)**: the remote directory to upload the contents to. Defaults to current directory. • **progress_callback (Function, optional)**: function to execute to receive progress information. Takes the following arguments: – The file being uploaded as string. – The progress percentage as integer.
get_file(String, String, Function)	Downloads the given XBee device file in the specified destination path. • **source_path (String)**: the path of the XBee device file to download. • **dest_path (String)**: the destination path to store the file in.

The methods above may fail for the following reasons:

- There is an error executing the requested operation, throwing a `FileSystemException`.

Example: Format file system
The XBee Python Library includes a sample application that displays how to format the device file system. It can be located in the following path: **examples/filesystem/FormatFilesystemSample/FormatFilesystemSample.py**

Example: List directory
The XBee Python Library includes a sample application that displays how to list the contents of a device directory. It can be located in the following path: **examples/filesystem/ListDirectorySample/ListDirectorySample.py**

Example: Upload/download file
The XBee Python Library includes a sample application that displays how to upload/download a file from the device. It can be located in the following path: **examples/filesystem/UploadDownloadFileSample/UploadDownloadFileSample.py**

2.6.8.3 Apply an XBee profile

An XBee profile is a snapshot of a specific XBee configuration, including firmware, settings, and file system contents. The XBee Python API includes a set of classes and methods to work with XBee profiles and apply them to local and remote devices.

- *Read an XBee profile*
- *Apply an XBee profile to a local device*
- *Apply an XBee profile to a remote device*

To configure individual settings see *Configure the XBee device*.

Note: Use XCTU to create configuration profiles.

Warning:

At the moment, firmware update is only supported in:

- **XBee 3**: Local and remote profile updates
- **XBee SX 868/900 MHz**: Local and remote profile updates
- **XBee S2C**: Remote profile updates

Read an XBee profile

The library provides a class called `XBeeProfile` that is used to read and extract information of an existing XBee profile file.

To create an `XBeeProfile` object, provide the location of the profile file in the class constructor.

Instantiate a profile

```
from digi.xbee.profile import XBeeProfile

[...]

PROFILE_PATH = "/home/user/my_profile.xpro"

[...]

# Create the XBee profile object.
xbee_profile = XBeeProfile(PROFILE_PATH)

[...]
```

The creation of the XBee profile object may fail for the following reasons:

- The provided profile file is not valid, throwing a `ValueError`.
- There is any error reading the profile file, throwing a `ProfileReadException`.

Once the XBee profile object is created, you can extract some profile information by accessing each of the exposed properties:

Property	Description
profile_file	Returns the profile file.
version	Returns the profile version.
flash_firmware_option	Returns the profile flash firmware option.
description	Returns the profile description.
reset_settings	Returns whether the settings of the XBee device are reset before applying the profile ones.
has_firmware_files	Returns whether the profile has firmware binaries (local or remote)
has_local_firmware_files	Returns whether the profile has local firmware binaries.
has_remote_firmware_files	Returns whether the profile has remote firmware binaries.
has_filesystem	Returns whether the profile has filesystem information (local or remote)
has_local_filesystem	Returns whether the profile has local filesystem information.
has_remote_filesystem	Returns whether the profile has remote filesystem information.
profile_settings	Returns all the firmware settings that the profile configures.
firmware_version	Returns the compatible firmware version of the profile.
hardware_version	Returns the compatible hardware version of the profile.
compatibility_number	Returns the compatibility number of the profile.
region_lock	Returns the region lock of the profile.

To access to the files inside, use `open` method. Once done with it, use `close` method.

Open/close a profile

```
xbee_profile = XBeeProfile(PROFILE_PATH)

xbee_profile.open()

[...]

xbee_profile.close()

[...]
```

An opened profile also offers the following properties:

Property	Description
profile_description_file	Returns the path of the profile description file.
firmware_description_file	Returns the path of the profile firmware description file.
file_system_path	Returns the profile file system path.
remote_file_system_image	Returns the path of the remote OTA file system image.
bootloader_file	Returns the profile bootloader file path.

Read a profile

```
from digi.xbee.profile import XBeeProfile

[...]

PROFILE_PATH = "/home/user/my_profile.xpro"

[...]

# Create the XBee profile object.
xbee_profile = XBeeProfile(PROFILE_PATH)

# Print profile compatible hardware and software versions
print("  - Firmware version: %s" % xbee_profile.firmware_version)
print("  - Hardware version: %s" % xbee_profile.hardware_version)

[...]
```

Example: Read an XBee profile
The XBee Python Library includes a sample application that displays how to read an XBee profile. It can be located in the following path: **examples/profile/ReadXBeeProfileSample/ReadXBeeProfileSample.py**

Apply an XBee profile to a local device

Applying a profile to a local XBee device requires the following components:

- The local XBee device object instance.
- The profile file to apply (*.xpro).

Note: Use XCTU to create configuration profiles.

Warning: At the moment, local profile update is only supported in **XBee 3** and **XBee SX 868/900 MHz** devices.

To apply the XBee profile to a local XBee, you have to call the `apply_profile` method of the `XBeeDevice` class providing the required parameters:

Method	Description
apply_profile(String, timeout, Function)	Applies the given XBee profile to the XBee device. • **profile_path (String)**: path of the XBee profile file to apply. • **timeout (Integer, optional)**: maximum time to wait for read operations during the apply profile. • **progress_callback (Function, optional)**: function to execute to receive progress information. Receives two arguments: – The current apply profile task as a String – The current apply profile task percentage as an Integer

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

- The local device does not support the apply profile operation, throwing a `OperationNotSupportedException`.
- There is an error while applying the XBee profile, throwing a `UpdateProfileException`.
- Other errors caught as `XBeeException`:
  - The local device is not open, throwing a generic `XBeeException`.
  - The operating mode of the local device is not `API_MODE` or `ESCAPED_API_MODE`, throwing an `InvalidOperatingModeException`.

Apply an XBee profile to a local device

```
[...]

PROFILE_PATH = "/home/user/my_profile.xpro"

[...]

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

[...]

# Apply the XBee device profile.
device.apply_profile(PROFILE_PATH, progress_callback=progress_callback)

[...]
```

Example: Apply local XBee profile
The XBee Python Library includes a sample application that displays how to apply an XBee profile to a local device. It can be located in the following path: **examples/profile/ApplyXBeeProfileSample/ApplyXBeeProfileSample.py**

Apply an XBee profile to a remote device

Applying a profile to a remote XBee requires the following components:

- The remote XBee device object instance.

- The profile file to apply (*.xpro).

Note: Use XCTU to create configuration profiles.

Warning: At the moment, remote profile update is only supported in **XBee 3**, **XBee SX 868/900 MHz**, and **XBee S2C** devices.

To apply the XBee profile to a remote XBee device, you have to call the `apply_profile` method of the `RemoteXBeeDevice` class providing the required parameters:

Method	Description
apply_profile(String, timeout, Function)	Applies the given XBee profile to the remote XBee device. • **profile_path (String)**: path of the XBee profile file to apply. • **timeout (Integer, optional)**: maximum time to wait for read operations during the apply profile. • **progress_callback (Function, optional)**: function to execute to receive progress information. Receives two arguments: – The current apply profile task as a String – The current apply profile task percentage as an Integer

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

- The remote device does not support the apply profile operation, throwing a `OperationNotSupportedException`.
- There is an error while applying the XBee profile, throwing a `UpdateProfileException`.
- Other errors caught as `XBeeException`:
  - The local device is not open, throwing a generic `XBeeException`.
  - The operating mode of the local device is not `API_MODE` or `ESCAPED_API_MODE`, throwing an `InvalidOperatingModeException`.

Apply an XBee profile to a remote device

```
[...]

PROFILE_PATH = "/home/user/my_profile.xpro"
REMOTE_DEVICE_NAME = "REMOTE"

[...]

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

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

(continues on next page)

(continued from previous page)

```
# Get the remote device.
remote = xnet.discover_device(REMOTE_DEVICE_NAME)

[...]

# Apply the XBee device profile.
remote.apply_profile(PROFILE_PATH, progress_callback=progress_callback)

[...]
```

Example: Apply remote XBee profile
The XBee Python Library includes a sample application that displays how to apply an XBee profile to a remote device. It can be located in the following path: **examples/profile/ApplyXBeeProfileRemoteSample/ApplyXBeeProfileRemoteSample.py**

2.6.9 Log events

Logging is a fundamental part of applications, and every application includes this feature. A well-designed logging system is a useful utility for system administrators, developers, and the support team and can save valuable time in sorting through the cause of issues. As users execute programs on the front end, the system invisibly builds a vault of event information (log entries).

The XBee Python Library uses the Python standard logging module for registering logging events. The logger works at module level; that is, each module has a logger with a unique name.

The modules that have logging integrated are `devices` and `reader`. By default, all loggers are disabled so you will not see any logging message in the console if you do not activate them.

In the XBee Python Library, you need three things to enable the logger:

1. The logger itself.
2. A handler. This will determine if the messages will be displayed in the console, written in a file, sent through a socket, etc.
3. A formatter. This will determine the message format. For example, a format could be:
   - *Timestamp with the current date - logger name - level (debug, info, warning. . .) - data.*

To retrieve the logger, use the `get_logger()` method of the logging module, providing the name of the logger that you want to get as parameter. In the XBee Python Library all loggers have the name of the module they belong to. For example, the name of the logger of the `devices` module is `digi.xbee.devices`. You can get a module name with the special attribute `\_\_name\_\_`.

Retrieve a module name and its logger

```
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

```
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

```
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

```
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')
```

(continues on next page)

(continued from previous page)

```
handler.setFormatter(formatter)

# Add the handler to the logger.
dev_logger.addHandler(handler)

[...]
```

2.6.9.1 Logging level

The XBee Python Library also provides a method in the `utils` module, `enable_logger()`, to enable the logger with the default settings. These settings are:

- Handler: `StreamHandler`
- Format: *timestamp - logger name - level - message*

Method	Description
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

```
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.

2.6.10 XBee Python samples

The XBee Python Library includes several samples to demonstrate how to do the following:

- Communicate with your modules
- Configure your modules
- Read the IO lines
- Update device's firmware
- Work with device's file system
- Apply XBee profiles
- Perform other common operations

All of the sample applications are contained in the examples folder, organized by category. Every sample includes the source code and a **readme.txt** file to clarify the purpose and the required setup to launch the application.

Examples are split by categories:

- *Configuration samples*
- *Network samples*
- *Communication samples*
- *IO samples*
- *Firmware samples*
- *File system samples*
- *Profile samples*

2.6.10.1 Configuration samples

Manage common parameters

This sample application shows how to get and set common parameters of the XBee device. Common parameters are split in cached and non-cached parameters. For that reason, the application refreshes the cached parameters before reading and displaying them. The application then configures, reads, and displays the value of non-cached parameters.

The application uses the specific setters and getters provided by the XBee device object to configure and read the different parameters.

You can locate the example in the following path: **examples/configuration/ManageCommonParametersSample**

Note: For more information about how to manage common parameters, see *Read and set common parameters*.

Set and get parameters

This sample application shows how to set and get parameters of a local or remote XBee device. Use this method when you need to set or get the value of a parameter that does not have its own getter and setter within the XBee device object.

The application sets the value of four parameters with different value types:

- String
- Byte
- Array

- Integer

The application then reads the parameters from the device to verify that the read values are the same as the values that were set.

You can locate the example in the following path: **examples/configuration/SetAndGetParametersSample**

Note: For more information about how to get and set other parameters, see *Read, set and execute other parameters*.

Reset module

This sample application shows how to perform a software reset on the local XBee module.

You can locate the example in the following path: **examples/configuration/ResetModuleSample**

Note: For more information about how to reset a module, see *Reset the device*.

Recover XBee serial connection

This sample application shows how to recover the serial settings of a local XBee.

You can locate the example at the following path: **examples/configuration/RecoverSerialConnection**

Note: For more information about this, see *Open the XBee device connection*.

Connect to access point (Wi-Fi)

This sample application shows how to configure a Wi-Fi module to connect to a specific access point and read its addressing settings.

You can locate the example at the following path: **examples/configuration/ConnectToAccessPoint**

Note: For more information about connecting to an access point, see *Configure Wi-Fi settings*.

2.6.10.2 Network samples

Discover devices

This sample application demonstrates how to obtain the XBee network object from a local XBee device and discover the remote XBee devices that compose the network. The example adds a discovery listener, so the callbacks provided by the listener object receive the events.

The remote XBee devices are printed out as soon as they are found during discovery.

You can locate the example in the following path: **examples/network/DiscoverDevicesSample**

Note: For more information about how to perform a network discovery, see *Discover the network*.

Network modifications sample

This sample application demonstrates how to listen to network modification events. The example adds a modifications network callback, so modifications events are received and printed out.

A network is modified when:

- a new node is added by discovering, manually, or because data is received from it
- an existing node is removed from the network
- an existing node is updated with new information
- it is fully cleared

You can locate the example in the following path: **examples/network/NetworkModificationsSample**

Note: For more information about how to listen to network modifications, see *Listen to network modification events*.

2.6.10.3 Communication samples

Send data

This sample application shows how to send data from the XBee device to another remote device on the same network using the XBee Python Library. In this example, the application sends data using a reliable transmission method. The application blocks during the transmission request, but you are notified if there is any error during the process.

The application sends data to a remote XBee device on the network with a specific node identifier (name).

You can locate the example in the following path: **examples/communication/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 successfully sent.

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 the application layer (explicit) format to a remote Zigbee device 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 the application layer (explicit) format asynchronously to a remote Zigbee device 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 successfully sent.

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 the application layer (explicit) format to all remote devices on the 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/explicit/SendBroadcastExplicitDataSample**

Note: For more information about how to send broadcast explicit data, see *Send explicit data to all devices in the network*.

Send IP data (IP devices)

This sample application shows how to send IP data to another device specified by its IP address and port number.

You can find the example at the following path: **examples/communication/ip/SendIPDataSample**

Note: For more information about how to send IP data, see *Send IP data*.

Send SMS (cellular devices)

This sample application shows how to send an SMS to a phone or cellular device.

You can find the example at the following path: **examples/communication/cellular/SendSMSSample**

Note: For more information about how to send SMS messages, see *Send SMS messages*.

Send UDP data (IP devices)

This sample application shows how to send UDP data to another device specified by its IP address and port number.

You can find the example at the following path: **examples/communication/ip/SendUDPDataSample**

Note: For more information about how to send IP data, see *Send IP data*.

Send Bluetooth Data

This sample application shows how to send data to the XBee Bluetooth Low Energy interface.

You can find the example at the following path: **examples/communication/bluetooth/SendBluetoothDataSample**

Note: For more information about sending Bluetooth data, see *Send Bluetooth data*.

Send MicroPython Data

This sample application shows how to send data to the XBee MicroPython interface.

You can find the example at the following path: **examples/communication/micropython/SendMicroPythonDataSample**

Note: For more information about sending MicroPython data, see *Send MicroPython data*.

Send User Data Relay

This sample application shows how to send data to other XBee interface.

You can find the example at the following path: **examples/communication/relay/SendUserDataRelaySample**

Note: For more information about sending User Data Relay messages, see *Send Bluetooth data* or *Send MicroPython data*.

Receive data

This sample application shows how data packets are received from another XBee device on the same network.

The application prints the received data to the standard output in ASCII and hexadecimal formats after the sender address.

You can locate the example in the following path: **examples/communication/ReceiveDataSample**

Note: For more information about how to receive data using a callback, see *Data reception callback*.

Receive data polling

This sample application shows how data packets are received from another XBee device on the same network using a polling mechanism.

The application prints the data that was received to the standard output in ASCII and hexadecimal formats after the sender address.

You can locate the example in the following path: **examples/communication/ReceiveDataPollingSample**

Note: For more information about how to receive data using a polling mechanism, see *Polling for data*.

Receive explicit data

This sample application shows how a Zigbee device receives data in the 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 the application layer (explicit) format using a polling mechanism. Before receiving data in explicit format, the API output mode of the Zigbee device is configured in explicit mode.

You can locate the example in the following path: **examples/communication/explicit/ReceiveExplicitDataPollingSample**

Note: For more information about how to receive explicit data using a polling mechanism, see *Polling for explicit data*.

Receive IP data (IP devices)

This sample application shows how an IP device receives IP data using a callback executed every time it receives new IP data.

You can find the example at the following path: **examples/communication/ip/ReceiveIPDataSample**

Note: For more information about how to receive IP data using a polling mechanism, see *Receive IP data*.

Receive SMS (cellular devices)

This sample application shows how to receive SMS messages configuring a callback executed when new SMS is received.

You can find the example at the following path: **examples/communication/cellular/ReceiveSMSSample**

Note: For more information about how to receive SMS messages, see *Receive SMS messages*.

Receive Bluetooth data

This sample application shows how to receive data from the XBee Bluetooth Low Energy interface.

You can find the example at the following path: **examples/communication/bluetooth/ReceiveBluetoothDataSample**

Note: For more information about receiving Bluetooth data, see *Receive Bluetooth data*.

Receive Bluetooth file

This sample application shows how to receive a file from the XBee Bluetooth Low Energy interface.

You can find the example at the following path: **examples/communication/bluetooth/ReceiveBluetoothFileSample**

Note: For more information about receiving Bluetooth data, see *Receive Bluetooth data*.

Receive MicroPython data

This sample application shows how to receive data from the XBee MicroPython interface.

You can find the example at the following path: **examples/communication/micropython/ReceiveMicroPythonDataSample**

Note: For more information about receiving MicroPython data, see *Receive MicroPython data*.

Receive User Data Relay

This sample application shows how to receive data from other XBee interface.

You can find the example at the following path: **examples/communication/relay/ReceiveUserDataRelaySample**

Note: For more information about receiving User Data Relay messages, see *Receive Bluetooth data* or *Receive MicroPython data*.

Receive modem status

This sample application shows how modem status packets (events related to the device and the network) are handled using the API.

The application prints the modem status events to the standard output when received.

You can locate the example in the following path: **examples/communication/ReceiveModemStatusSample**

Note: For more information about how to receive modem status events, see *Receive modem status events*.

Connect to echo server (IP devices)

This sample application shows how IP devices can connect to an echo server, send data to it and reads the echoed data.

You can find the example at the following path: **examples/communication/ip/ConnectToEchoServerSample**

Note: For more information about how to send and receive IP data, see *Send IP data* and *Receive IP data*.

Create a TCP client socket (cellular devices)

This sample application shows how to create a TCP client socket to send HTTP requests.

You can find the example at the following path: **examples/communication/socket/SocketTCPClientSample**

Note: For more information about how to use the XBee socket API, see *Communicate using XBee sockets*.

Create a TCP server socket (cellular devices)

This sample application shows how to create a TCP server socket to receive data from incoming sockets.

You can find the example at the following path: **examples/communication/socket/SocketTCPServerSample**

Note: For more information about how to use the XBee socket API, see *Communicate using XBee sockets*.

Create a UDP server/client socket (cellular devices)

This sample application shows how to create a UDP socket to deliver messages to a server and listen for data coming from multiple peers.

You can find the example at the following path: **examples/communication/socket/SocketUDPServerClientSample**

Note: For more information about how to use the XBee socket API, see *Communicate using XBee sockets*.

2.6.10.4 IO samples

Local DIO

This sample application shows how to set and read XBee digital lines of the device attached to the serial/USB port of your PC.

The application configures two IO lines of the XBee device: one as a digital input (button) and the other as a digital output (LED). The application reads the status of the input line periodically and updates the output to follow the input.

The LED lights up while you press the button.

You can locate the example in the following path: **examples/io/LocalDIOSample**

Note: For more information about how to set and read digital lines, see *Digital Input/Output*.

Local ADC

This sample application shows how to read XBee analog inputs of the device attached to the serial/USB port of your PC.

The application configures an IO line of the XBee device as ADC. It periodically reads its value and prints it in the output console.

You can locate the example in the following path: **examples/io/LocalADCSample**

Note: For more information about how to read analog lines, see *ADC*.

Remote DIO

This sample application shows how to set and read XBee digital lines of remote devices.

The application configures two IO lines of the XBee devices: one in the remote device as a digital input (button) and the other in the local device as a digital output (LED). The application reads the status of the input line periodically and updates the output to follow the input.

The LED lights up while you press the button.

You can locate the example in the following path: **examples/io/RemoteDIOSample**

Note: For more information about how to set and read digital lines, see *Digital Input/Output*.

Remote ADC

This sample application shows how to read XBee analog inputs of remote XBee devices.

The application configures an IO line of the remote XBee device as ADC. It periodically reads its value and prints it in the output console.

You can locate the example in the following path: **examples/io/RemoteADCSample**

Note: For more information about how to read analog lines, see *ADC*.

IO sampling

This sample application shows how to configure a remote device to send automatic IO samples and how to read them from the local module.

The application configures two IO lines of the remote XBee device: one as digital input (button) and the other as ADC, and enables periodic sampling and change detection. The device sends a sample every five seconds containing the values of the two monitored lines. The device sends another sample every time the button is pressed or released, which only contains the value of this digital line.

The application registers a listener in the local device to receive and handle all IO samples sent by the remote XBee module.

You can locate the example in the following path: **examples/io/IOSamplingSample**

Note: For more information about how to read IO samples, see *Read IO samples*.

2.6.10.5 Firmware samples

Update local firmware

This sample Python application shows how to update the firmware of a local XBee device.

The application provides the required hardware files to the update method as well as a callback function to be notified of progress.

You can locate the example in the following path: **examples/firmware/LocalFirmwareUpdateSample**

Update remote firmware

This sample Python application shows how to update the firmware of a remote XBee device.

The application provides the required hardware files to the update method as well as a callback function to be notified of progress.

You can locate the example in the following path: **examples/firmware/RemotelFirmwareUpdateSample**

2.6.10.6 File system samples

Format file system

This sample Python application shows how to format the filesystem of a local XBee device and retrieve usage information.

The application uses the LocalXBeeFileSystemManager to access the device filesystem and execute the required actions.

You can locate the example in the following path: **examples/filesystem/FormatFilesystemSample**

List directory contents

This sample Python application shows how to list the contents of an XBee device filesystem directory.

The application uses the LocalXBeeFileSystemManager to access the device filesystem and executes the required actions.

You can locate the example in the following path: **examples/filesystem/ListDirectorySample**

Upload/download file

This sample Python application shows how to upload and download a file from a local XBee device filesystem.

The application uses the LocalXBeeFileSystemManager to access the device filesystem and provides the local file and the necessary paths to the upload/download methods as well as callback functions to be notified of progress.

You can locate the example in the following path: **examples/filesystem/UploadDownloadFileSample**

2.6.10.7 Profile samples

Apply local profile

This sample Python application shows how to apply an existing XBee profile to a XBee device.

The application provides the profile file to the update method as well as a callback function to be notified of progress.

You can locate the example in the following path: **examples/profile/ApplyXBeeProfileSample**

Apply remote profile

This sample Python application shows how to apply an existing XBee profile to a remote XBee device.

The application provides the profile file to the update method as well as a callback function to be notified of progress.

You can locate the example in the following path: **examples/profile/ApplyXBeeProfileRemoteSample**

Read profile

This sample Python application shows how to read an existing XBee profile and extract its properties.

The application creates an XBee profile object from an existing XBee profile file and prints all the accessible settings and properties.

You can locate the example in the following path: **examples/profile/ReadXBeeProfileSample**

2.6.11 Frequently Asked Questions (FAQs)

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

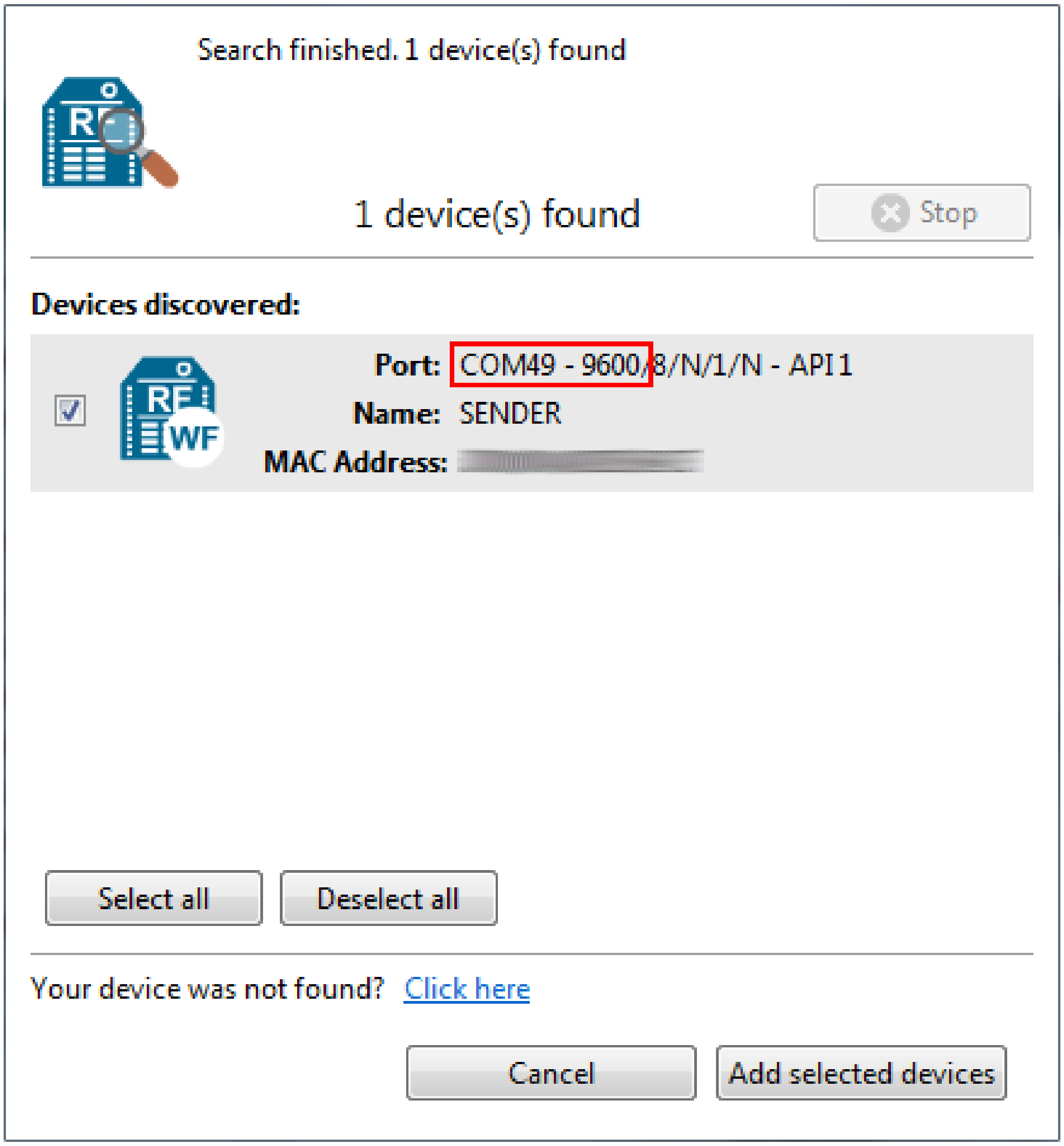
2.6.11.1 What is XCTU and how do I download it?

XCTU is a free multi-platform application designed to enable developers to interact with Digi RF modules through a simple-to-use graphical interface. You can download it at www.digi.com/xctu.

2.6.11.2 How do I find the serial port and baud rate of my module?

Open the XCTU application, and click the **Discover radio modules connected to your machine** button.

Select all ports to be scanned, click **Next** and then **Finish**. Once the discovery process has finished, a new window notifies you how many devices have been found and their details. The serial port and the baud rate are shown in the **Port** label.


Search finished. 1 device(s) found

1 device(s) found

Devices discovered:

Port: COM49 - 9600/8/N/1/N - API1
Name: SENDER
MAC Address: [redacted]

Your device was not found? Click here

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

2.6.11.3 Can I use the XBee Python Library with modules in AT operating mode?

No, the XBee Python Library only supports **API** and **API Escaped** operating modes.

2.6.11.4 I get the Python error `ImportError: No module named 'serial'`

This error means that Python cannot find the `serial` module, which is used by the library for the serial communication with the XBee devices.

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.

2.6.11.5 I get the Python error `ImportError: No module named 'srp'`

This error means that Python cannot find the `srp` module, which is used by the library to authenticate with XBee devices over Bluetooth Low Energy.

You can install SRP running this command in your terminal application:

```
$ pip install srp
```

2.6.12 Changelog

2.6.12.1 v1.4.0 - 03/18/2021

- Deep node discovery for Zigbee, DigiMesh, and 802.15.4.
- Get route from local XBee to a remote XBee:
  - New method to register a callback to listen for new received routes (`add_route_received_callback()`)
  - New blocking method to ask for the route to the remote node (`get_route_to_node()`)
- Allow to recover a local node from a profile not only from firmware.
- Support to be notified when new frames are received from a specific node (`add_packet_received_from_callback()`).
- Update network information from sent/received AT Command frames.
- New optional argument for parameter value in `execute_command()`.
- New optional argument to apply pending settings in `get_parameter()`, `set_parameter()`, and `execute_command()`.
- XBee 3:
  - Support to update remote file system OTA images.
- XBee SX 900/868:
  - Firmware update for local and remote XBee devices.
  - Profile update for local and remote XBee devices.
- XBee S2C:
  - OTA firmware/profile update support for remote nodes.
- Zigbee:
  - Methods to get nodes routing and neighbor tables: `get_routes()` and `get_neighbors()`.
  - Methods to get/set many-to-one broadcasting time: `get_many_to_one_broadcasting_time()` and `set_many_to_one_broadcasting_time()`.
  - Support for source route creation: `create_source_route()`.

  - New frames: * 'Route Record Indicator' (0xA1) * 'Create Source Route Packet' (0x21)
- DigiMesh:
  - Method to get node neighbors: `get_neighbors()`.
  - Method to build aggregate route: `build_aggregate_routes()`.
  - New frames: * 'Route Information Packet' (0x8D)
- Documentation update
- Bug fixing:
  - Captured possible exception while determining the XBee role (#103)
  - Memory leak: empty list of last discovered nodes using ND (#172)
  - Fix Python 3.9 syntax error (#204)
  - Use least significant nibble of status field in local/remote AT Command Responses (XCTUNG-376)
  - Do not lose already registered socket callbacks when closing a local XBee.
  - Reload node information after firmware/profile update (XBPL-348)
  - OTA firmware update:
    - Fix sequence number in ZCL responses during fw update (XCTUNG-1975)
    - Immediate update after transferring the OTA file (XBPL-350)
    - Use requested file offset and size instead of fixed chunks (XBPL-344)
    - Mechanism to calculate the proper block size based on the maximum size received by the client and the maximum payload size (XBPL-346)
    - For asyncronous sleeping nodes (Zigbee, DigiMesh, 802.15.4) and synchronous sleeping networks (DigiMesh), configure a minimum sleep time before update and restore settings at the end. For DigiMesh synchronous sleeping network, the local XBee must be a non-sleeping node but synchronized with the network (SM=7)
  - Profile application:
    - Do not uncompress profile when reading its information. This change avoids extra processing time and required space when retrieving profile info.
    - Remove profile extracted files. A profile is opened to access to its contents, and must be closed when done with it.
    - Fixed the application of XBee profiles with 'AP' setting changes (XBPL-340)
    - Fixed bootloader update from profile due to bootloader image path mismatch (XBPL-338)
    - Fix bootloader update operation by waiting some time until the new bootloader is running (XBPL-339)
    - Fixed application of profile with filesystem from Windows(XBPL-341)
    - Read firmware version as an hexadecimal value (#177)
  - Several minor bug fixes.

2.6.12.2 v1.3.0 - 11/05/2019

- Zigbee: Support to register joining devices to a trust center.
- Cellular: XBee TCP/UDP socket support.
- XBee 3:
  - Firmware update for local and remote XBee devices.
  - Profile update for local and remote XBee devices.
  - File system management for local XBee devices.
- New recover serial connection functionality to force the XBee serial connection settings.
- Support for notification of network cache modifications events (new node added, removed of existing node, network clear, . . .)
- Deprecate `get_api_output_mode` and `set_api_output_mode` methods to use new `get_api_output_mode_value` and `set_api_output_mode_value` with `APIOutputModeBit` enumeration.
- Role as one of the cached parameters.
- Report an error on 'finished discovery' callback if node discovery fails.
- Several minor bug fixes.

2.6.12.3 v1.2.0 - 04/05/2019

- Add new methods to send and receive data from other XBee interfaces through User Data Relay frames.
- Add new methods to manage the Bluetooth interface.
- Add support to set AT parameters without applying them with the AT Command Queue packet.
- Improve the callbacks mechanism:
  - Callbacks are now executed in parallel.
  - Internal callbacks are now defined when needed to avoid issues when more than one callback of the same type is defined.
- Add missing 'Transmit Status', 'Modem Status' and 'Cellular Association Indication Status' values to cover all XBee Cellular/XBee3 Cellular features.
- Bug Fixing:
  - Fix some bugs related to package spec data.
  - Log an error when processing a wrong frame instead of stopping the reader.
  - Fix an issue parsing Explicit RX Indicator packets.
  - Fix a couple of leaks with StreamHandlers.

2.6.12.4 v1.1.1 - 04/25/2018

- Add support for DigiMesh and 802.15.4 protocols on XBee3 modules.
- Return an unknown XBee packet when the received packet is not supported by the library instead of raising an exception.

- Change logging handler to log messages in the console.
- Bug Fixing:
  - Fix a problem when closing the device connection in the reader.
  - Fix how is determined whether the module has entered in AT command mode or not.
  - Fix the string encoding and decoding in some API packets.
  - Fix the message displayed when the XBee device protocol is not correct one.

2.6.12.5 v1.1.0 - 01/19/2018

- Add support for new hardware variants:
  - XB8X
- Add missing 'Modem Status' values for Remote Manager connect and disconnect events.
- Bug Fixing:
  - Fix timeouts on Unix platforms.
  - Fix the return source endpoint method from the 'ExplicitRXIndicatorPacket' class.
  - Perform general bug fixing when working in API escaped mode.

2.6.12.6 v1.0.0 - 10/02/2017

Initial release of XBee Python library. The main features of the library include:

- Support for ZigBee, 802.15.4, DigiMesh, Point-to-Multipoint, Wi-Fi, Cellular and NB-IoT devices.
- Support for API and API escaped operating modes.
- Management of local (attached to the PC) and remote XBee device objects.
- Discovery of remote XBee devices associated with the same network as the local device.
- Configuration of local and remote XBee devices:
  - Configure common parameters with specific setters and getters.
  - Configure any other parameter with generic methods.
  - Execute AT commands.
  - Apply configuration changes.
  - Write configuration changes.
  - Reset the device.
- Transmission of data to all the XBee devices on the network or to a specific device.
- Reception of data from remote XBee devices:
  - Data polling.
  - Data reception callback.
- Transmission and reception of IP and SMS messages.
- Reception of network status changes related to the local XBee device.
- IO lines management:

  - Configure IO lines.
  - Set IO line value.
  - Read IO line value.
  - Receive IO data samples from any remote XBee device on the network.
- Support for explicit frames and application layer fields (Source endpoint, Destination endpoint, Profile ID, and Cluster ID).
- Multiple examples that show how to use the available APIs.

2.6.13 API reference

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

2.6.13.1 digi package

Subpackages

digi.xbee package

Subpackages

digi.xbee.models package

Submodules

digi.xbee.models.accesspoint module

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.

- **signal_quality** (*Integer, optional*) – signal quality with the access point in %.

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
Returns the SSID of the access point.

Returns the SSID of the access point.

Return type String

encryption_type
Returns the encryption type of the access point.

Returns the encryption type of the access point.

Return type *WiFiEncryptionType*

See also:

WiFiEncryptionType

channel
Returns the channel of the access point.

Returns the channel of the access point.

Return type Integer

See also:

`AccessPoint.set_channel()`

signal_quality
Returns the signal quality with the access point in %.

Returns the signal quality with the access point in %.

Return type Integer

See also:

`AccessPoint.__set_signal_quality()`

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`

Returns the code of the WiFiEncryptionType element.

Returns the code of the WiFiEncryptionType element.

Return type Integer

`description`

Returns the description of the WiFiEncryptionType element.

Returns the description of the WiFiEncryptionType element.

Return type String

digi.xbee.models.atcomm module

class `digi.xbee.models.atcomm.`**`ATStringCommand`**(*command*, *description*)

Bases: `enum.Enum`

This class represents basic AT commands.

Inherited properties:

name (String): name (ID) of this ATStringCommand.
value (String): value of this ATStringCommand.

Values:

ATStringCommand.AC = ('AC', 'Apply changes')
ATStringCommand.AG = ('AG', 'Aggregator support')
ATStringCommand.AI = ('AI', 'Association indication')
ATStringCommand.AO = ('AO', 'API options')
ATStringCommand.AP = ('AP', 'API enable')
ATStringCommand.AR = ('AR', 'Many-to-one route broadcast time')
ATStringCommand.AS = ('AS', 'Active scan')
ATStringCommand.BD = ('BD', 'UART baudrate')
ATStringCommand.BI = ('BI', 'Bluetooth identifier')
ATStringCommand.BL = ('BL', 'Bluetooth address')
ATStringCommand.BP = ('BP', 'Bluetooth advertisement power')
ATStringCommand.BT = ('BT', 'Bluetooth enable')

ATStringCommand.BR = ('BR', 'RF data rate')
ATStringCommand.C0 = ('C0', 'Source port')
ATStringCommand.C8 = ('C8', 'Compatibility mode')
ATStringCommand.CC = ('CC', 'Command sequence character')
ATStringCommand.CE = ('CE', 'Device role')
ATStringCommand.CH = ('CH', 'Channel')
ATStringCommand.CK = ('CK', 'Configuration checksum')
ATStringCommand.CM = ('CM', 'Channel mask')
ATStringCommand.CN = ('CN', 'Exit command mode')
ATStringCommand.DA = ('DA', 'Force Disassociation')
ATStringCommand.DB = ('DB', 'RSSI')
ATStringCommand.DD = ('DD', 'Device type')
ATStringCommand.DH = ('DH', 'Destination address high')
ATStringCommand.DJ = ('DJ', 'Disable joining')
ATStringCommand.DL = ('DL', 'Destination address low')
ATStringCommand.DM = ('DM', 'Disable device functionality')
ATStringCommand.DO = ('DO', 'Device options')
ATStringCommand.D0 = ('D0', 'DIO0 configuration')
ATStringCommand.D1 = ('D1', 'DIO1 configuration')
ATStringCommand.D2 = ('D2', 'DIO2 configuration')
ATStringCommand.D3 = ('D3', 'DIO3 configuration')
ATStringCommand.D4 = ('D4', 'DIO4 configuration')
ATStringCommand.D5 = ('D5', 'DIO5 configuration')
ATStringCommand.D6 = ('D6', 'RTS configuration')
ATStringCommand.D7 = ('D7', 'CTS configuration')
ATStringCommand.D8 = ('D8', 'DIO8 configuration')
ATStringCommand.D9 = ('D9', 'DIO9 configuration')
ATStringCommand.EE = ('EE', 'Encryption enable')
ATStringCommand.EO = ('EO', 'Encryption options')
ATStringCommand.FN = ('FN', 'Find neighbors')
ATStringCommand.FR = ('FR', 'Software reset')
ATStringCommand.FS = ('FS', 'File system')
ATStringCommand.GW = ('GW', 'Gateway address')
ATStringCommand.GT = ('GT', 'Guard times')
ATStringCommand.HV = ('HV', 'Hardware version')
ATStringCommand.HP = ('HP', 'Preamble ID')
ATStringCommand.IC = ('IC', 'Digital change detection')
ATStringCommand.ID = ('ID', 'Network PAN ID/Network ID/SSID')
ATStringCommand.IR = ('IR', 'I/O sample rate')
ATStringCommand.IS = ('IS', 'Force sample')
ATStringCommand.JN = ('JN', 'Join notification')
ATStringCommand.JV = ('JV', 'Join verification')
ATStringCommand.KY = ('KY', 'Link/Encryption key')
ATStringCommand.MA = ('MA', 'IP addressing mode')
ATStringCommand.MK = ('MK', 'IP address mask')
ATStringCommand.MP = ('MP', '16-bit parent address')
ATStringCommand.MY = ('MY', '16-bit address/IP address')

ATStringCommand.M0 = ('M0', 'PWM0 configuration')
ATStringCommand.M1 = ('M1', 'PWM1 configuration')
ATStringCommand.NB = ('NB', 'Parity')
ATStringCommand.NI = ('NI', 'Node identifier')
ATStringCommand.ND = ('ND', 'Node discover')
ATStringCommand.NJ = ('NJ', 'Join time')
ATStringCommand.NK = ('NK', 'Trust Center network key')
ATStringCommand.NO = ('NO', 'Node discover options')
ATStringCommand.NR = ('NR', 'Network reset')
ATStringCommand.NS = ('NS', 'DNS address')
ATStringCommand.NP = ('NP', 'Maximum number of transmission bytes')
ATStringCommand.NT = ('NT', 'Node discover back-off')
ATStringCommand.N_QUESTION = ('N?', 'Network discovery timeout')
ATStringCommand.OP = ('OP', 'Operating extended PAN ID')
ATStringCommand.OS = ('OS', 'Operating sleep time')
ATStringCommand.OW = ('OW', 'Operating wake time')
ATStringCommand.PK = ('PK', 'Passphrase')
ATStringCommand.PL = ('PL', 'TX power level')
ATStringCommand.PP = ('PP', 'Output power')
ATStringCommand.PS = ('PS', 'MicroPython auto start')
ATStringCommand.P0 = ('P0', 'DIO10 configuration')
ATStringCommand.P1 = ('P1', 'DIO11 configuration')
ATStringCommand.P2 = ('P2', 'DIO12 configuration')
ATStringCommand.P3 = ('P3', 'UART DOUT configuration')
ATStringCommand.P4 = ('P4', 'UART DIN configuration')
ATStringCommand.P5 = ('P5', 'DIO15 configuration')
ATStringCommand.P6 = ('P6', 'DIO16 configuration')
ATStringCommand.P7 = ('P7', 'DIO17 configuration')
ATStringCommand.P8 = ('P8', 'DIO18 configuration')
ATStringCommand.P9 = ('P9', 'DIO19 configuration')
ATStringCommand.RE = ('RE', 'Restore defaults')
ATStringCommand.RR = ('RR', 'XBee retries')
ATStringCommand.R_QUESTION = ('R?', 'Region lock')
ATStringCommand.SB = ('SB', 'Stop bits')
ATStringCommand.SC = ('SC', 'Scan channels')
ATStringCommand.SD = ('SD', 'Scan duration')
ATStringCommand.SH = ('SH', 'Serial number high')
ATStringCommand.SI = ('SI', 'Socket info')
ATStringCommand.SL = ('SL', 'Serial number low')
ATStringCommand.SM = ('SM', 'Sleep mode')
ATStringCommand.SN = ('SN', 'Sleep count')
ATStringCommand.SO = ('SO', 'Sleep options')
ATStringCommand.SP = ('SP', 'Sleep time')
ATStringCommand.SS = ('SS', 'Sleep status')
ATStringCommand.ST = ('ST', 'Wake time')
ATStringCommand.TP = ('TP', 'Temperature')
ATStringCommand.VH = ('VH', 'Bootloader version')

ATStringCommand.VR = ('VR', 'Firmware version')
ATStringCommand.WR = ('WR', 'Write')
ATStringCommand.DOLLAR_S = ('$S', 'SRP salt')
ATStringCommand.DOLLAR_V = ('$V', 'SRP salt verifier')
ATStringCommand.DOLLAR_W = ('$W', 'SRP salt verifier')
ATStringCommand.DOLLAR_X = ('$X', 'SRP salt verifier')
ATStringCommand.DOLLAR_Y = ('$Y', 'SRP salt verifier')
ATStringCommand.PERCENT_C = ('%C', 'Hardware/software compatibility')
ATStringCommand.PERCENT_P = ('%P', 'Invoke bootloader')
ATStringCommand.PERCENT_U = ('%U', 'Recover')
ATStringCommand.PERCENT_V = ('%V', 'Supply voltage')

`command`
AT command alias

Returns The AT command alias.

Return type String

`description`
AT command description.

Returns The AT command description.

Return type String

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`
Returns the code of the SpecialByte element.

Returns the code of the SpecialByte element.

Return type Integer

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.

command

Returns the AT command.

Returns the AT command.

Return type String

get_parameter_string()

Returns this ATCommand parameter as a String.

Returns this ATCommand parameter. *None* if there is no parameter.

Return type String

parameter

Returns the AT command parameter.

Returns

the AT command parameter. *None* if there is no parameter.

Return type Bytearray

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

Returns the AT command.

Returns the AT command.

Return type *ATCommand*

response
Returns the AT command response.

Returns the AT command response.

Return type Bytearray

status
Returns the AT command response status.

Returns The AT command response status.

Return type *ATCommandStatus*

digi.xbee.models.filesystem module

class digi.xbee.models.filesystem.**FSCmdType**(*code*, *description*)
Bases: enum.Enum

This enumeration lists all the available file system commands.

Inherited properties:
name (String): Name (id) of this FSCmdType.
value (String): Value of this FSCmdType.

Values:
Open/create file (1) = (1, 'Open/create file')
Close file (2) = (2, 'Close file')
Read file (3) = (3, 'Read file')
Write file (4) = (4, 'Write file')
File hash (8) = (8, 'File hash')
Create directory (16) = (16, 'Create directory')
Open directory (17) = (17, 'Open directory')
Close directory (18) = (18, 'Close directory')
Read directory (19) = (19, 'Read directory')
Get directory path ID (28) = (28, 'Get directory path ID')
Rename (33) = (33, 'Rename')
Delete (47) = (47, 'Delete')
Stat filesystem (64) = (64, 'Stat filesystem')
Format filesystem (79) = (79, 'Format filesystem')

code
Returns the code of the file system command element.

Returns Code of the file system command element.

Return type Integer

description
Returns the description of the file system command element.

Returns Description of the file system command element.

Return type Integer

class `digi.xbee.models.filesystem.`**`FSCmd`**(*cmd_type*, *direction=0*, *status=None*)

Bases: `object`

This class represents a file system command.

Class constructor. Instantiates a new *`FSCmd`* object with the provided parameters.

Parameters

- **`cmd_type`** (*`FSCmdType`* or Integer) – The command type.
- **`direction`** (*`Integer, optional, default=0`*) – If this command is a request (0) or a response (1).
- **`status`** (*`FSCommandStatus`* or Integer) – Status of the file system command execution. Only for response commands.

Raises

- `ValueError` – If *cmd_type* is not an integer or a *`FSCmdType`*.
- `ValueError` – If *cmd_type* is invalid.

See also:

`FSCmdType`

`type`

Returns the command type.

Returns The command type.

Return type *`FSCmdType`*

`direction`

Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

`status`

Returns the file system command response status.

Returns File system command response status.

Return type *`FSCommandStatus`*

See also:

`FSCommandStatus`

`FSCmd.status_value()`

`status_value`

Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

`FSCmd.status()`

output()
Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

to_dict()
Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

classmethod create_cmd(*raw*, *direction=0*)
Creates a file system command with the given parameters. This method ensures that the FSCmd returned is valid and is well built (if not exceptions are raised).

Parameters

- **raw** (*Bytearray*) – Bytearray to create the command.
- **direction** (*Integer, optional, default=0*) – If this command is a request (0) or a response (1).

Returns The file system command created.

Return type *FSCmd*

Raises `InvalidPacketException` – If something is wrong with *raw* and the command cannot be built.

class digi.xbee.models.filesystem.**UnknownFSCmd**(*raw*, *direction=0*)
Bases: *digi.xbee.models.filesystem.FSCmd*

This class represents an unknown file system command.

Class constructor. Instantiates a new *UnknownFSCmd* object with the provided parameters.

Parameters

- **raw** (*Bytearray*) – Data of the unknown command.
- **direction** (*Integer, optional, default=0*) – If this command is a request (0) or a response (1).

Raises `ValueError` – If *data* is not a bytearray, its length is less than 3, or the command type is a known one.

See also:

FSCmd

type
Returns the command type.

Returns The command type.

Return type Integer

classmethod create_cmd(*raw*, *direction=0*)
Override method.

Returns *UnknownFSCmd*.

Raises

- InvalidPacketException – If *raw* is not a bytearray.
- InvalidPacketException – If *raw* length is less than 3, or the command type is a known one.

See also:

FSCmd.create_cmd()

output()
Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

to_dict()
Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

direction
Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

status
Returns the file system command response status.

Returns File system command response status.

Return type *FSCommandStatus*

See also:

FSCommandStatus
FSCmd.status_value()

status_value
Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

`FSCmd.status()`

class `digi.xbee.models.filesystem.`**`FileIdCmd`**(*cmd_type*, *fid*, *direction=0*, *status=None*)

Bases: *`digi.xbee.models.filesystem.FSCmd`*

This class represents a file system command request or response that includes a file or path id.

Class constructor. Instantiates a new *`FileIdCmd`* object with the provided parameters.

Parameters

- **`cmd_type`** (*`FSCmdType`* or Integer) – The command type.
- **`fid`** (*`Integer`*) – Id of the file/path to operate with. A file id expires and becomes invalid if not referenced for over 2 minutes. Set to 0x0000 for the root directory (/).
- **`direction`** (*`Integer, optional, default=0`*) – If this command is a request (0) or a response (1).
- **`status`** (*`FSCommandStatus`* or Integer) – Status of the file system command execution. Only for response commands.

Raises `ValueError` – If *fid* is invalid.

See also:

`FSCmd`
`FSCommandStatus`

`fs_id`

Returns the file/path identifier.

Returns The file/path id value.

Return type Integer

classmethod `create_cmd`(*raw*, *direction=0*)

Override method.

Returns *`FileIdCmd`*.

Raises `InvalidPacketException` – If the bytearray length is less than the minimum required.

See also:

`FSCmd.create_cmd()`

`direction`

Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

output()

Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

status

Returns the file system command response status.

Returns File system command response status.

Return type *FSCommandStatus*

See also:

FSCommandStatus

FSCmd.status_value()

status_value

Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

FSCmd.status()

to_dict()

Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

type

Returns the command type.

Returns The command type.

Return type *FSCmdType*

class digi.xbee.models.filesystem.**FileIdNameCmd**(*cmd_type*, *fid*, *name*, *direction=0*, *status=None*)

Bases: *digi.xbee.models.filesystem.FileIdCmd*

This class represents a file system command request or response that includes a file or path id and a name.

The file/path id is the next byte after the command type in the frame, and name are the following bytes until the end of the frame.

Class constructor. Instantiates a new *FileIdNameCmd* object with the provided parameters.

Parameters

- **cmd_type** (*FSCmdType* or Integer) – The command type.

- **fid** (*Integer*) – Id of the file/path to operate with. Set to 0x0000 for the root directory (/).
- **name** (*String or bytearray*) – The path name of the file to operate with. Its maximum length is 252 characters.
- **direction** (*Integer, optional, default=0*) – If this command is a request (0) or a response (1).
- **status** (*FSCommandStatus* or Integer) – Status of the file system command execution. Only for response commands.

Raises `ValueError` – If *fid* or *name* are invalid.

See also:

FSCmd

name

Returns the path name of the file.

Returns The file path name.

Return type String

classmethod create_cmd(*raw*, *direction=0*)

Override method. Direction must be 0.

Returns *FileIdNameCmd*.

Raises `InvalidPacketException` – If the bytearray length is less than the minimum required.

See also:

FSCmd.create_cmd()

direction

Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

fs_id

Returns the file/path identifier.

Returns The file/path id value.

Return type Integer

output()

Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

status

Returns the file system command response status.

Returns File system command response status.

Return type *FSCommandStatus*

See also:

FSCommandStatus
`FSCmd.status_value()`

status_value
Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

`FSCmd.status()`

to_dict()
Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

type
Returns the command type.

Returns The command type.

Return type *FSCmdType*

class digi.xbee.models.filesystem.**OpenFileCmdRequest**(*path_id*, *name*, *flags*)
Bases: *digi.xbee.models.filesystem.FileIdNameCmd*

This class represents a file open/create file system command request. Open a file for reading and/or writing. Use *FileOpenRequestOption.SECURE* bitmask to upload a write-only file (one that cannot be downloaded or viewed), useful for protecting MicroPython source code on the device.

Command response is received as a *OpenFileCmdResponse*.

Class constructor. Instantiates a new *OpenFileCmdRequest* object with the provided parameters.

Parameters

- **path_id** (*Integer*) – Directory path id. Set to 0x0000 for the root directory (/).
- **name** (*String or bytearray*) – The path name of the file to open/create, relative to *path_id*. Its maximum length is 251 chars.
- **flags** (*FileOpenRequestOption*) – Bitfield of supported flags. Use *FileOpenRequestOption* to compose its value.

Raises `ValueError` – If any of the parameters is invalid.

See also:

FileIdNameCmd
FileOpenRequestOption

options
Returns the options to open the file.

Returns The options to open the file.

Return type FileOpenRequestOption

classmethod create_cmd(*raw*, *direction=0*)
Override method. Direction must be 0.

Returns OpenFileCmdRequest.

Raises

- InvalidPacketException – If the bytearray length is less than 5. (cmd id + path id (2 bytes) + flags (1 byte) + name (at least 1 byte) = 5 bytes).
- InvalidPacketException – If the command type is not FSCmdType or direction is not 0.

See also:

FileIdNameCmd.create_cmd()

direction
Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

fs_id
Returns the file/path identifier.

Returns The file/path id value.

Return type Integer

name
Returns the path name of the file.

Returns The file path name.

Return type String

output()
Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

status
Returns the file system command response status.

Returns File system command response status.

Return type FSCommandStatus

See also:

FSCommandStatus

`FSCmd.status_value()`

`status_value`

Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

`FSCmd.status()`

`to_dict``()`

Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

`type`

Returns the command type.

Returns The command type.

Return type *FSCmdType*

`class` `digi.xbee.models.filesystem.`**`OpenFileCmdResponse`**(*status*, *fid=None*, *size=None*)

Bases: *digi.xbee.models.filesystem.FileIdCmd*

This class represents a file open/create file system command response.

This is received in response of an *OpenFileCmdRequest*.

Class constructor. Instantiates a new *OpenFileCmdResponse* object with the provided parameters.

Parameters

- **`status`** (*FSCommandStatus* or Integer) – Status of the file system command execution.
- **`fid`** (*Integer, optional, default=`None`*) – Id of the file that has been opened. It expires and becomes invalid if not referenced for over 2 minutes.
- **`size`** (*Integer, optional, default=`None`*) – Size in bytes of the file. 0xFFFFFFFF if unknown.

Raises `ValueError` – If any of the parameters is invalid.

See also:

FileIdCmd

`size`

Returns the size of the opened file. 0xFFFFFFFF if unknown.

Returns Size in bytes of the opened file.

Return type Integer

classmethod create_cmd(*raw*, *direction=1*)

Override method. Direction must be 1.

Returns *OpenFileCmdResponse*.

Raises

- `InvalidPacketException` – If the bytearray length is less than 8. (cmd id + status + file id (2 bytes) + size (4 bytes) = 8).
- `InvalidPacketException` – If the command type is not *FSCmdType* or direction is not 1.

See also:

FileIdCmd.create_cmd()

direction

Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

fs_id

Returns the file/path identifier.

Returns The file/path id value.

Return type Integer

output()

Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

status

Returns the file system command response status.

Returns File system command response status.

Return type *FSCommandStatus*

See also:

FSCommandStatus

`FSCmd.status_value()`

status_value

Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

`FSCmd.status()`

to_dict()
Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

type
Returns the command type.

Returns The command type.

Return type *FSCmdType*

class digi.xbee.models.filesystem.**CloseFileCmdRequest**(*fid*)
Bases: *digi.xbee.models.filesystem.FileIdCmd*

This class represents a file close file system command request. Close an open file and release its File Handle.

Command response is received as a *CloseFileCmdResponse*.

Class constructor. Instantiates a new *CloseFileCmdRequest* object with the provided parameters.

Parameters **fid** (*Integer*) – Id of the file to close returned in Open File Response. It expires and becomes invalid if not referenced for over 2 minutes.

Raises `ValueError` – If any of the parameters is invalid.

See also:

FileIdCmd

classmethod create_cmd(*raw*, *direction=0*)
Override method. Direction must be 0.

Returns *CloseFileCmdRequest*.

Raises

- `InvalidPacketException` – If the bytearray length is less than 3. (cmd id + file_id (2 bytes) = 3 bytes).
- `InvalidPacketException` – If the command type is not *FSCmdType* or direction is not 0.

See also:

FileIdCmd.create_cmd()

direction
Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

fs_id
Returns the file/path identifier.

Returns The file/path id value.

Return type Integer

output()
Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

status
Returns the file system command response status.

Returns File system command response status.

Return type *FSCommandStatus*

See also:

FSCommandStatus
`FSCmd.status_value()`

status_value
Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

`FSCmd.status()`

to_dict()
Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

type
Returns the command type.

Returns The command type.

Return type *FSCmdType*

class digi.xbee.models.filesystem.**CloseFileCmdResponse**(*status*)
Bases: *digi.xbee.models.filesystem.FSCmd*

This class represents a file close file system command response.

Command response is received as a *CloseFileCmdRequest*.

Class constructor. Instantiates a new *CloseFileCmdResponse* object with the provided parameters.

Parameters **`status`** (*`FSCommandStatus`* or Integer) – Status of the file system command execution.

See also:

`FSCmd`

`classmethod create_cmd`(*raw*, *direction=1*)

Override method. Direction must be 1.

Returns *`OpenFileCmdResponse`*.

Raises

- `InvalidPacketException` – If the bytearray length is less than 1. (cmd id = 1 byte).
- `InvalidPacketException` – If the command type is not *`FSCmdType`* or direction is not 1.

See also:

`FSCmd.create_cmd()`

`direction`

Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

`output`()

Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

`status`

Returns the file system command response status.

Returns File system command response status.

Return type *`FSCommandStatus`*

See also:

`FSCommandStatus`

`FSCmd.status_value()`

`status_value`

Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

```
FSCmd.status()
```

to_dict()

Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

type

Returns the command type.

Returns The command type.

Return type *FSCmdType*

class digi.xbee.models.filesystem.**ReadFileCmdRequest**(*fid*, *offset*, *size*)

Bases: *digi.xbee.models.filesystem.FileIdCmd*

This class represents a read file system command request.

Command response is received as a *ReadFileCmdResponse*.

Class constructor. Instantiates a new *ReadFileCmdRequest* object with the provided parameters.

Parameters

- **fid** (*Integer*) – Id of the file to read returned in Open File Response. It expires and becomes invalid if not referenced for over 2 minutes.
- **offset** (*Integer*) – The file offset to start reading. 0xFFFFFFFF to use current position (*ReadFileCmdRequest.USE_CURRENT_OFFSET*)
- **size** (*Integer*) – The number of bytes to read. 0xFFFF (*ReadFileCmdRequest.READ_AS_MANY*) to read as many as possible (limited by file size or maximum response frame size)

Raises ValueError – If any of the parameters is invalid.

See also:

FileIdCmd

USE_CURRENT_OFFSET = 4294967295

Use current file position to start reading.

READ_AS_MANY = 65535

Read as many bytes as possible (limited by file size or maximum response frame size)

offset

Returns the file offset to start reading. 0xFFFFFFFF to use current position (*ReadFileCmdRequest.0xFFFFFFFF*)

Returns The file offset.

Return type Integer

size

Returns the number of bytes to read. 0xFFFF (*ReadFileCmdRequest.READ_AS_MANY*) to read as many as possible (limited by file size or maximum response frame size)

Returns The number of bytes to read.

Return type Integer

classmethod create_cmd(*raw*, *direction=0*)

Override method. Direction must be 0.

Returns *ReadFileCmdRequest*.

Raises

- `InvalidPacketException` – If the bytearray length is less than 9. (cmd id + file_id (2 bytes) + offset (4 bytes) + size (2 bytes) = 9 bytes).
- `InvalidPacketException` – If the command type is not *FSCmdType* or direction is not 0.

See also:

FileIdCmd.create_cmd()

direction

Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

fs_id

Returns the file/path identifier.

Returns The file/path id value.

Return type Integer

output()

Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

status

Returns the file system command response status.

Returns File system command response status.

Return type *FSCommandStatus*

See also:

FSCommandStatus

`FSCmd.status_value()`

status_value

Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

`FSCmd.status()`

`to_dict()`
Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

`type`
Returns the command type.

Returns The command type.

Return type *`FSCmdType`*

`class` `digi.xbee.models.filesystem.`**`ReadFileCmdResponse`**(*status*, *fid=None*, *offset=None*, *data=None*)
Bases: *`digi.xbee.models.filesystem.FileIdCmd`*

This class represents a read file system command response.

Command response is received as a *`ReadFileCmdRequest`*.

Class constructor. Instantiates a new *`ReadFileCmdResponse`* object with the provided parameters.

Parameters

- **`status`** (*`FSCommandStatus`* or Integer) – Status of the file system command execution.
- **`fid`** (*`Integer, optional, default=`None``*) – Id of the read file.
- **`offset`** (*`Integer, optional, default=`None``*) – The offset of the read data.
- **`data`** (*`Bytearray, optional, default=`None``*) – The file read data.

Raises `ValueError` – If any of the parameters is invalid.

See also:

`FileIdCmd`

`offset`
Returns the offset of the read data.

Returns The data offset.

Return type Integer

`data`
Returns the read data from the file.

Returns Read data.

Return type Bytearray

`classmethod create_cmd`(*raw*, *direction=1*)
Override method. Direction must be 1.

Returns *`ReadFileCmdResponse`*.

Raises

- `InvalidPacketException` – If the bytearray length is less than 8. (cmd id + status + file_id (2 bytes) + offset (4 bytes) + data = 8)
- `InvalidPacketException` – If the command type is not *`FSCmdType`* or direction is not 1.

See also:

`FileIdCmd.create_cmd()`

`direction`

Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

`fs_id`

Returns the file/path identifier.

Returns The file/path id value.

Return type Integer

`output``()`

Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

`status`

Returns the file system command response status.

Returns File system command response status.

Return type *`FSCommandStatus`*

See also:

`FSCommandStatus`
`FSCmd.status_value()`

`status_value`

Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

`FSCmd.status()`

to_dict()
Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

type
Returns the command type.

Returns The command type.

Return type *FSCmdType*

class digi.xbee.models.filesystem.**WriteFileCmdRequest**(*fid*, *offset*, *data=None*)
Bases: *digi.xbee.models.filesystem.FileIdCmd*

This class represents a write file system command request.

Command response is received as a *WriteFileCmdResponse*.

Class constructor. Instantiates a new *WriteFileCmdRequest* object with the provided parameters.

Parameters

- **fid** (*Integer*) – Id of the file to write returned in Open File Response. It expires and becomes invalid if not referenced for over 2 minutes.
- **offset** (*Integer*) – The file offset to start writing. 0xFFFFFFFF to use current position (*ReadFileCmdRequest.USE_CURRENT_OFFSET*)
- **data** (*Bytearray, optional, default=`None`*) – The data to write. If empty, frame just refreshes the File Handle timeout to keep the file open.

Raises ValueError – If any of the parameters is invalid.

See also:

FileIdCmd

USE_CURRENT_OFFSET = 4294967295
Use current file position to start writing.

offset
Returns the file offset to start writing.

Returns The file offset.

Return type Integer

data
Returns the data to write. If empty, frame just refreshes the File Handle timeout to keep the file open.

Returns The data to write.

Return type Bytearray

classmethod create_cmd(*raw*, *direction=0*)
Override method. Direction must be 0.

Returns *WriteFileCmdRequest*.

Raises

- `InvalidPacketException` – If the bytearray length is less than 7. (cmd id + file_id (2 bytes) + offset (4 bytes) = 7 bytes).
- `InvalidPacketException` – If the command type is not *FSCmdType* or direction is not 0.

See also:

FileIdCmd.create_cmd()

direction
Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

fs_id
Returns the file/path identifier.

Returns The file/path id value.

Return type Integer

output()
Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

status
Returns the file system command response status.

Returns File system command response status.

Return type *FSCommandStatus*

See also:

FSCommandStatus
`FSCmd.status_value()`

status_value
Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

`FSCmd.status()`

to_dict()
Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

type
Returns the command type.

Returns The command type.

Return type *FSCmdType*

class digi.xbee.models.filesystem.**WriteFileCmdResponse**(*status*, *fid=None*, *actual_offset=None*)
Bases: *digi.xbee.models.filesystem.FileIdCmd*

This class represents a write file system command response.

Command response is received as a *WriteFileCmdRequest*.

Class constructor. Instantiates a new *WriteFileCmdResponse* object with the provided parameters.

Parameters

- **status** (*FSCommandStatus* or Integer) – Status of the file system command execution.
- **fid** (*Integer, optional, default=`None`*) – Id of the written file.
- **actual_offset** (*Integer, optional, default=`None`*) – The current file offset after writing.

Raises ValueError – If any of the parameters is invalid.

See also:

FileIdCmd

actual_offset
Returns the file offset after writing.

Returns The file offset.

Return type Integer

classmethod create_cmd(*raw*, *direction=1*)
Override method. Direction must be 1.

Returns *WriteFileCmdResponse*.

Raises

- InvalidPacketException – If the bytearray length is less than 8. (cmd id + status + file_id (2 bytes) + offset (4 bytes) = 8)
- InvalidPacketException – If the command type is not *FSCmdType* or direction is not 1.

See also:

FileIdCmd.create_cmd()

direction
Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

fs_id
Returns the file/path identifier.

Returns The file/path id value.

Return type Integer

output()
Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

status
Returns the file system command response status.

Returns File system command response status.

Return type *FSCommandStatus*

See also:

FSCommandStatus
`FSCmd.status_value()`

status_value
Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

`FSCmd.status()`

to_dict()
Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

type
Returns the command type.

Returns The command type.

Return type *FSCmdType*

class `digi.xbee.models.filesystem.`**`HashFileCmdRequest`**(*path_id*, *name*)
Bases: *`digi.xbee.models.filesystem.FileIdNameCmd`*

This class represents a file hash command request. Use this command to get a sha256 hash to verify a file's contents without downloading the entire file (something not even possible for secure files). On XBee Cellular modules, there is a response delay in order to calculate the hash of a non-secure file. Secure files on XBee Cellular and all files on XBee 3 802.15.4, DigiMesh, and Zigbee have a cached hash.

Command response is received as a *`HashFileCmdResponse`*.

Class constructor. Instantiates a new *`HashFileCmdRequest`* object with the provided parameters.

Parameters

- **`path_id`** (*`Integer`*) – Directory path id. Set to 0x0000 for the root directory (/).
- **`name`** (*`String or bytearray`*) – The path name of the file to hash, relative to *path_id*. Its maximum length is 252 chars.

Raises `ValueError` – If any of the parameters is invalid.

See also:

`FileIdNameCmd`

classmethod create_cmd(*raw*, *direction=0*)
Override method. Direction must be 0.

Returns *`HashFileCmdRequest`*.

Raises

- `InvalidPacketException` – If the bytearray length is less than 4. (cmd id + path id (2 bytes) + name (at least 1 byte) = 4 bytes).
- `InvalidPacketException` – If the command type is not *`FSCmdType`* or direction is not 0.

See also:

`FileIdNameCmd.create_cmd()`

direction
Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

fs_id
Returns the file/path identifier.

Returns The file/path id value.

Return type Integer

name
Returns the path name of the file.

Returns The file path name.

Return type String

output()

Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

status

Returns the file system command response status.

Returns File system command response status.

Return type *FSCommandStatus*

See also:

FSCommandStatus

`FSCmd.status_value()`

status_value

Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

`FSCmd.status()`

to_dict()

Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

type

Returns the command type.

Returns The command type.

Return type *FSCmdType*

class digi.xbee.models.filesystem.**HashFileCmdResponse**(*status*, *file_hash=None*)

Bases: *digi.xbee.models.filesystem.FSCmd*

This class represents a file hash command response.

This is received in response of an *HashFileCmdRequest*.

Class constructor. Instantiates a new *HashFileCmdResponse* object with the provided parameters.

Parameters

- **status** (*FSCommandStatus* or Integer) – Status of the file system command execution.
- **file_hash** (*Bytearray, optional, default=`None`*) – The hash value.

Raises `ValueError` – If any of the parameters is invalid.

See also:

FSCmd

file_hash
Returns the hash of the file.

Returns The hash of the file.

Return type Bytearray

classmethod create_cmd(*raw*, *direction=1*)
Override method. Direction must be 1.

Returns *HashFileCmdResponse*.

Raises

- `InvalidPacketException` – If the bytearray length is less than 34. (cmd id + status + hash (32 bytes) = 34).
- `InvalidPacketException` – If the command type is not *FSCmdType* or direction is not 1.

See also:

FSCmd.create_cmd()

direction
Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

output()
Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

status
Returns the file system command response status.

Returns File system command response status.

Return type *FSCommandStatus*

See also:

FSCommandStatus

`FSCmd.status_value()`

status_value

Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

`FSCmd.status()`

to_dict()

Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

type

Returns the command type.

Returns The command type.

Return type *FSCmdType*

class digi.xbee.models.filesystem.**CreateDirCmdRequest**(*path_id*, *name*)

Bases: *digi.xbee.models.filesystem.FileIdNameCmd*

This class represents a create directory file system command request. Parent directories of the one to be created must exist. Separate request must be dane to make intermediate directories.

Command response is received as a *CreateDirCmdResponse*.

Class constructor. Instantiates a new *CreateDirCmdRequest* object with the provided parameters.

Parameters

- **path_id** (*Integer*) – Directory path id. Set to 0x0000 for the root directory (/).
- **name** (*String or bytearray*) – The path name of the directory to create, relative to *path_id*. Its maximum length is 252 chars.

Raises `ValueError` – If any of the parameters is invalid.

See also:

FileIdNameCmd

classmethod create_cmd(*raw*, *direction=0*)

Override method. Direction must be 0.

Returns *CreateDirCmdRequest*.

Raises

- `InvalidPacketException` – If the bytearray length is less than 4. (cmd id + path id (2 bytes) + name (at least 1 byte) = 4 bytes).
- `InvalidPacketException` – If the command type is not *FSCmdType* or direction is not 0.

See also:

FileIdNameCmd.create_cmd()

direction
Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

fs_id
Returns the file/path identifier.

Returns The file/path id value.

Return type Integer

name
Returns the path name of the file.

Returns The file path name.

Return type String

output()
Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

status
Returns the file system command response status.

Returns File system command response status.

Return type *FSCommandStatus*

See also:

FSCommandStatus
FSCmd.status_value()

status_value
Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

FSCmd.status()

to_dict()
Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

type
Returns the command type.

Returns The command type.

Return type *FSCmdType*

class digi.xbee.models.filesystem.**CreateDirCmdResponse**(*status*)
Bases: *digi.xbee.models.filesystem.FSCmd*

This class represents a create directory file system command response.

Command response is received as a *CreateDirCmdRequest*.

Class constructor. Instantiates a new *CreateDirCmdResponse* object with the provided parameters.

Parameters **status** (*FSCommandStatus* or Integer) – Status of the file system command execution.

See also:

FSCmd

classmethod create_cmd(*raw*, *direction=1*)
Override method. Direction must be 1.

Returns *CreateDirCmdResponse*.

Raises

- InvalidPacketException – If the bytearray length is less than 2. (cmd id + status = 2).
- InvalidPacketException – If the command type is not *FSCmdType* or direction is not 1.

See also:

FSCmd.create_cmd()

direction
Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

output()
Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

status
Returns the file system command response status.

Returns File system command response status.

Return type *FSCommandStatus*

See also:

FSCommandStatus
`FSCmd.status_value()`

status_value
Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

`FSCmd.status()`

to_dict()
Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

type
Returns the command type.

Returns The command type.

Return type *FSCmdType*

class digi.xbee.models.filesystem.**OpenDirCmdRequest**(*path_id*, *name*)
Bases: *digi.xbee.models.filesystem.FileIdNameCmd*

This class represents an open directory file system command request.

Command response is received as a *OpenDirCmdResponse*.

Class constructor. Instantiates a new *OpenDirCmdRequest* object with the provided parameters.

Parameters

- **path_id** (*Integer*) – Directory path id. Set to 0x0000 for the root directory (/).
- **name** (*String or bytearray*) – Path name of the directory to open, relative to *path_id*. An empty name is equivalent to '.', both refer to the current directory path id. Its maximum length is 252 chars.

Raises `ValueError` – If any of the parameters is invalid.

See also:

FileIdNameCmd

classmethod create_cmd(*raw*, *direction=0*)
Override method. Direction must be 0.

Returns `OpenDirCmdRequest`.

Raises

- `InvalidPacketException` – If the bytearray length is less than 4. (cmd id + path id (2 bytes) + name (at least 1 byte) = 4 bytes).
- `InvalidPacketException` – If the command type is not `FSCmdType` or direction is not 0.

See also:

`FileIdNameCmd.create_cmd()`

direction
Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

fs_id
Returns the file/path identifier.

Returns The file/path id value.

Return type Integer

name
Returns the path name of the file.

Returns The file path name.

Return type String

output()
Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

status
Returns the file system command response status.

Returns File system command response status.

Return type `FSCommandStatus`

See also:

`FSCommandStatus`
`FSCmd.status_value()`

status_value
Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

`FSCmd.status()`

to_dict()

Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

type

Returns the command type.

Returns The command type.

Return type *FSCmdType*

class digi.xbee.models.filesystem.**OpenDirCmdResponse**(*status*, *did=None*, *fs_entries=None*)

Bases: *digi.xbee.models.filesystem.FileIdCmd*

This class represents an open directory file system command response. If the final file system element does not have *DirResponseFlag.ENTRY_IS_LAST* set, send a Directory Read Request to get additional entries. A response ending with an *DirResponseFlag.ENTRY_IS_LAST* flag automatically closes the Directory Handle. An empty directory returns a single entry with just the *DirResponseFlag.ENTRY_IS_LAST* flag set, and a 0-byte name.

This is received in response of an *OpenDirCmdRequest*.

Class constructor. Instantiates a new *OpenFileCmdResponse* object with the provided parameters.

Parameters

- **status** (*FSCommandStatus* or Integer) – Status of the file system command execution.
- **did** (*Integer, optional, default=`None`*) – Id of the directory that has been opened. It expires and becomes invalid if not referenced for over 2 minutes.
- **fs_entries** (*List, optional, default=`None`*) – List of bytearrays with the info and name of the entries inside the opened directory.

Raises `ValueError` – If any of the parameters is invalid.

See also:

FileIdCmd

is_last

Returns whether there are more elements not included in this response.

Returns

True* if there are no more elements to list, *False otherwise.

Return type Boolean

fs_entries
Returns the list of entries inside the opened directory.

Returns List of :class: .'FileSystemElement' inside the directory.

Return type List

classmethod create_cmd(*raw*, *direction=1*)
Override method. Direction must be 1.

Returns *OpenDirCmdResponse*.

Raises

- `InvalidPacketException` – If the bytearray length is less than 8. (cmd id + status + dir id (2 bytes) + filesize_and_flags (4 bytes) = 8).
- `InvalidPacketException` – If the command type is not *FSCmdType* or direction is not 1.

See also:

FileIdCmd.create_cmd()

direction
Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

fs_id
Returns the file/path identifier.

Returns The file/path id value.

Return type Integer

output()
Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

status
Returns the file system command response status.

Returns File system command response status.

Return type *FSCommandStatus*

See also:

FSCommandStatus
`FSCmd.status_value()`

status_value
Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

`FSCmd.status()`

`to_dict()`

Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

`type`

Returns the command type.

Returns The command type.

Return type *`FSCmdType`*

class `digi.xbee.models.filesystem.`**`CloseDirCmdRequest`**(*did*)

Bases: *`digi.xbee.models.filesystem.FileIdCmd`*

This class represents a directory close file system command request.

Command response is received as a *`CloseDirCmdResponse`*.

Class constructor. Instantiates a new *`CloseDirCmdRequest`* object with the provided parameters.

Parameters **`did`** (*`Integer`*) – Id of the directory to close. It expires and becomes invalid if not referenced for over 2 minutes.

Raises `ValueError` – If any of the parameters is invalid.

See also:

`FileIdCmd`

classmethod **`create_cmd`**(*raw*, *direction=0*)

Override method. Direction must be 0.

Returns *`CloseDirCmdRequest`*.

Raises

- `InvalidPacketException` – If the bytearray length is less than 3. (cmd id + dir_id (2 bytes) = 3 bytes).
- `InvalidPacketException` – If the command type is not *`FSCmdType`* or direction is not 0.

See also:

`FileIdCmd.create_cmd()`

direction
Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

fs_id
Returns the file/path identifier.

Returns The file/path id value.

Return type Integer

output()
Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

status
Returns the file system command response status.

Returns File system command response status.

Return type *FSCommandStatus*

See also:

FSCommandStatus
`FSCmd.status_value()`

status_value
Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

`FSCmd.status()`

to_dict()
Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

type
Returns the command type.

Returns The command type.

Return type *FSCmdType*

class `digi.xbee.models.filesystem.`**`CloseDirCmdResponse`**`(`*status*`)`

Bases: *`digi.xbee.models.filesystem.FSCmd`*

This class represents a directory close file system command response. Send this command to indicate that it is done reading the directory and no longer needs the Directory Handle. Typical usage scenario is to use a Directory Open Request and additional Directory Read Requests until the Response includes an entry with the *DirResponseFlag.ENTRY_IS_LAST* flag set.

Command response is received as a *`CloseDirCmdRequest`*.

Class constructor. Instantiates a new *`CloseDirCmdResponse`* object with the provided parameters.

Parameters **`status`** (*`FSCommandStatus`* or Integer) – Status of the file system command execution.

See also:

`FSCmd`

`classmethod create_cmd``(`*raw*, *direction=1*`)`

Override method. Direction must be 1.

Returns *`CloseDirCmdResponse`*.

Raises

- `InvalidPacketException` – If the bytearray length is less than 2. (cmd id + status = 2).
- `InvalidPacketException` – If the command type is not *`FSCmdType`* or direction is not 1.

See also:

`FSCmd.create_cmd()`

`direction`

Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

`output``()`

Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

`status`

Returns the file system command response status.

Returns File system command response status.

Return type *`FSCommandStatus`*

See also:

FSCommandStatus

`FSCmd.status_value()`

status_value

Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

`FSCmd.status()`

to_dict()

Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

type

Returns the command type.

Returns The command type.

Return type *FSCmdType*

class digi.xbee.models.filesystem.**ReadDirCmdRequest**(*did*)

Bases: *digi.xbee.models.filesystem.FileIdCmd*

This class represents a directory read file system command request.

Command response is received as a *ReadDirCmdResponse*.

Class constructor. Instantiates a new *ReadDirCmdRequest* object with the provided parameters.

Parameters did (*Integer*) – Id of the directory to close. It expires and becomes invalid if not referenced for over 2 minutes.

Raises `ValueError` – If any of the parameters is invalid.

See also:

FileIdCmd

classmethod create_cmd(*raw*, *direction=0*)

Override method. Direction must be 0.

Returns *ReadDirCmdRequest*.

Raises

- `InvalidPacketException` – If the bytearray length is less than 3. (cmd id + dir_id (2 bytes) = 3 bytes).
- `InvalidPacketException` – If the command type is not *FSCmdType* or direction is not 0.

See also:

FileIdCmd.create_cmd()

direction
Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

fs_id
Returns the file/path identifier.

Returns The file/path id value.

Return type Integer

output()
Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

status
Returns the file system command response status.

Returns File system command response status.

Return type *FSCommandStatus*

See also:

FSCommandStatus
FSCmd.status_value()

status_value
Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

FSCmd.status()

to_dict()
Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

type
Returns the command type.

Returns The command type.

Return type *FSCmdType*

class digi.xbee.models.filesystem.**ReadDirCmdResponse**(*status*, *did=None*, *fs_entries=None*)

Bases: *digi.xbee.models.filesystem.OpenDirCmdResponse*

This class represents a read directory file system command response. If the final file system element does not have *DirResponseFlag.ENTRY_IS_LAST* set, send another Directory Read Request to get additional entries. A response ending with an *DirResponseFlag.ENTRY_IS_LAST* flag automatically closes the Directory Handle.

This is received in response of an *ReadDirCmdRequest*.

Class constructor. Instantiates a new *ReadDirCmdResponse* object with the provided parameters.

Parameters

- **status** (*FSCommandStatus* or Integer) – Status of the file system command execution.
- **did** (*Integer, optional, default=`None`*) – Id of the directory that has been read.
- **fs_entries** (*List, optional, default=`None`*) – List of bytearrays with the info and name of the entries inside the directory.

Raises ValueError – If any of the parameters is invalid.

See also:

FileIdCmd
DirResponseFlag

classmethod create_cmd(*raw*, *direction=1*)

Override method. Direction must be 1.

Returns *ReadDirCmdResponse*.

Raises

- InvalidPacketException – If the bytearray length is less than 4. (cmd id + status + dir id (2 bytes) = 4).
- InvalidPacketException – If the command type is not *FSCmdType* or direction is not 1.

See also:

FileIdCmd.create_cmd()

direction

Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

fs_entries

Returns the list of entries inside the opened directory.

Returns List of :class: .‘FileSystemElement‘ inside the directory.

Return type List

fs_id
Returns the file/path identifier.

Returns The file/path id value.

Return type Integer

is_last
Returns whether there are more elements not included in this response.

Returns

True* if there are no more elements to list, *False otherwise.

Return type Boolean

output()
Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

status
Returns the file system command response status.

Returns File system command response status.

Return type *FSCommandStatus*

See also:

FSCommandStatus
`FSCmd.status_value()`

status_value
Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

`FSCmd.status()`

to_dict()
Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

type
Returns the command type.

Returns The command type.

Return type *FSCmdType*

class digi.xbee.models.filesystem.**GetPathIdCmdRequest**(*path_id*, *name*)

Bases: *digi.xbee.models.filesystem.FileIdNameCmd*

This class represents a get path id file system command request. A directory path id (path_id) of 0x0000 in any command, means path names are relative to the root directory of the filesystem (/).

- '/' as path separator
- '..' to refer to the parent directory
- '.' to refer to the current path directory

Use this command to get a shortcut to a subdirectory of the file system to allow the use of shorter path names in the frame:

- If the PATH ID field of this command is 0x0000, the XBee allocates a new PATH ID for use in later requests.
- If the PATH ID field of this command is non-zero, the XBee updates the directory path of that ID.

To release a PATH ID when no longer needed:

- Send a request with that ID and a single slash ("/") as the pathname. Any Change Directory Request that resolves to the root directory releases the PATH ID and return a 0x0000 ID.
- Wait for a timeout (2 minutes)

Any file system id expires after 2 minutes if not referenced. Refresh this timeout by sending a Change Directory request with an empty or a single period ('.') as the pathname.

Command response is received as a *GetPathIdCmdResponse*.

Class constructor. Instantiates a new *GetPathIdCmdRequest* object with the provided parameters.

Parameters

- **path_id** (*Integer*) – Directory path id. Set to 0x0000 for the root directory (/).
- **name** (*String or bytearray*) – The path name of the directory to change, relative to *path_id*. An empty name is equivalent to '.', both refer to the current directory path id. Its maximum length is 252 chars.

Raises ValueError – If any of the parameters is invalid.

See also:

FileIdNameCmd

classmethod create_cmd(*raw*, *direction=0*)

Override method. Direction must be 0.

Returns *GetPathIdCmdRequest*.

Raises

- InvalidPacketException – If the bytearray length is less than 4. (cmd id + path id (2 bytes) + name (at least 1 byte) = 4 bytes).
- InvalidPacketException – If the command type is not *FSCmdType* or direction is not 0.

See also:

FileIdNameCmd.create_cmd()

direction
Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

fs_id
Returns the file/path identifier.

Returns The file/path id value.

Return type Integer

name
Returns the path name of the file.

Returns The file path name.

Return type String

output()
Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

status
Returns the file system command response status.

Returns File system command response status.

Return type *FSCommandStatus*

See also:

FSCommandStatus
FSCmd.status_value()

status_value
Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

FSCmd.status()

to_dict()
Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

type
Returns the command type.

Returns The command type.

Return type *FSCmdType*

class digi.xbee.models.filesystem.**GetPathIdCmdResponse**(*status*, *path_id=None*, *full_path=None*)
Bases: *digi.xbee.models.filesystem.FileIdCmd*

This class represents a get path id file system command response. The full path of the new current directory is included if can fit.

This is received in response of an *GetPathIdCmdRequest*.

Class constructor. Instantiates a new *GetPathIdCmdResponse* object with the provided parameters.

Parameters

- **status** (*FSCommandStatus* or Integer) – Status of the file system command execution.
- **path_id** (*Integer, optional, default=`None`*) – New directory path id.
- **full_path** (*String or bytearray, optional, default=`None`*) – If short enough, the full path of the current directory , relative to *path_id*. Deep subdirectories may return an empty field instead of their full path name. The maximum full path length is 255 characters.

Raises ValueError – If any of the parameters is invalid.

See also:

FileIdCmd

full_path
Returns the full path of the current directory.

Returns The directory full path.

Return type String

classmethod create_cmd(*raw*, *direction=1*)
Override method. Direction must be 1.

Returns *GetPathIdCmdResponse*.

Raises

- InvalidPacketException – If the bytearray length is less than 4. (cmd id + status + path id (2 bytes) = 4).
- InvalidPacketException – If the command type is not *FSCmdType* or direction is not 1.

See also:

FileIdNameCmd.create_cmd()

direction
Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

fs_id
Returns the file/path identifier.

Returns The file/path id value.

Return type Integer

output()
Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

status
Returns the file system command response status.

Returns File system command response status.

Return type *FSCommandStatus*

See also:

FSCommandStatus
`FSCmd.status_value()`

status_value
Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

`FSCmd.status()`

to_dict()
Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

type
Returns the command type.

Returns The command type.

Return type *FSCmdType*

class digi.xbee.models.filesystem.**RenameCmdRequest**(*path_id*, *name*, *new_name*)

Bases: *digi.xbee.models.filesystem.FileIdNameCmd*

This class represents a file/directory rename file system command request. Current firmware for XBee 3 802.15.4, DigiMesh, and Zigbee do not support renaming files. Contact Digi International to request it as a feature in a future release.

Command response is received as a *RenameCmdResponse*.

Class constructor. Instantiates a new *RenameCmdRequest* object with the provided parameters.

Parameters

- **path_id** (*Integer*) – Directory path id. Set to 0x0000 for the root directory (/).
- **name** (*String or bytearray*) – The current path name of the file/directory to rename relative to *path_id*. Its maximum length is 255 chars.
- **new_name** (*String or bytearray*) – The new name of the file/directory relative to *path_id*. Its maximum length is 255 chars.

Raises ValueError – If any of the parameters is invalid.

See also:

FileIdNameCmd

new_name

Returns the new name of the file or directory.

Returns The new name.

Return type String

classmethod create_cmd(*raw*, *direction=0*)

Override method. Direction must be 0.

Returns *RenameCmdRequest*.

Raises

- InvalidPacketException – If the bytearray length is less than 6. (cmd id + path id (2 bytes) + name (1 byte at least) + ‘,’ + new name (at least 1 byte) = 6 bytes).
- InvalidPacketException – If the command type is not *FSCmdType* or direction is not 0.

See also:

FileIdNameCmd.create_cmd()

direction

Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

fs_id

Returns the file/path identifier.

Returns The file/path id value.

Return type Integer

name

Returns the path name of the file.

Returns The file path name.

Return type String

output()

Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

status

Returns the file system command response status.

Returns File system command response status.

Return type *FSCommandStatus*

See also:

FSCommandStatus

`FSCmd.status_value()`

status_value

Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

`FSCmd.status()`

to_dict()

Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

type

Returns the command type.

Returns The command type.

Return type *FSCmdType*

class digi.xbee.models.filesystem.**RenameCmdResponse**(*status*)

Bases: *digi.xbee.models.filesystem.FSCmd*

This class represents a rename file system command response.

Command response is received as a *RenameCmdRequest*.

Class constructor. Instantiates a new `RenameCmdResponse` object with the provided parameters.

Parameters **status** (`FSCommandStatus` or Integer) – Status of the file system command execution.

See also:

`FSCmd`

classmethod create_cmd(*raw*, *direction=1*)

Override method. Direction must be 1.

Returns `RenameCmdResponse`.

Raises

- `InvalidPacketException` – If the bytearray length is less than 2. (cmd id + status = 2).
- `InvalidPacketException` – If the command type is not `FSCmdType` or direction is not 1.

See also:

`FSCmd.create_cmd()`

direction

Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

output()

Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

status

Returns the file system command response status.

Returns File system command response status.

Return type `FSCommandStatus`

See also:

`FSCommandStatus`
`FSCmd.status_value()`

status_value

Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

`FSCmd.status()`

to_dict()

Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

type

Returns the command type.

Returns The command type.

Return type *FSCmdType*

class digi.xbee.models.filesystem.**DeleteCmdRequest**(*path_id*, *name*)

Bases: *digi.xbee.models.filesystem.FileIdNameCmd*

This class represents a delete file system command request. All files in a directory must be deleted before removing the directory. On XBee 3 802.15.4, DigiMesh, and Zigbee, deleted files are marked as as unusable space unless they are at the "end" of the file system (most-recently created). On these products, deleting a file triggers recovery of any deleted file space at the end of the file system, and can lead to a delayed response.

Command response is received as a *DeleteCmdResponse*.

Class constructor. Instantiates a new *DeleteCmdRequest* object with the provided parameters.

Parameters

- **path_id** (*Integer*) – Directory path id. Set to 0x0000 for the root directory (/).
- **name** (*String or bytearray*) – The name of the file/directory to delete relative to *path_id*. Its maximum length is 252 chars.

Raises `ValueError` – If any of the parameters is invalid.

See also:

FileIdNameCmd

classmethod create_cmd(*raw*, *direction=0*)

Override method. Direction must be 0.

Returns *DeleteCmdRequest*.

Raises

- `InvalidPacketException` – If the bytearray length is less than 4. (cmd id + path id (2 bytes) + name (at least 1 byte) = 4 bytes).
- `InvalidPacketException` – If the command type is not *FSCmdType* or direction is not 0.

See also:

FileIdNameCmd.create_cmd()

direction
Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

fs_id
Returns the file/path identifier.

Returns The file/path id value.

Return type Integer

name
Returns the path name of the file.

Returns The file path name.

Return type String

output()
Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

status
Returns the file system command response status.

Returns File system command response status.

Return type *FSCommandStatus*

See also:

FSCommandStatus
`FSCmd.status_value()`

status_value
Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

`FSCmd.status()`

to_dict()
Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

type
Returns the command type.

Returns The command type.

Return type *FSCmdType*

class digi.xbee.models.filesystem.**DeleteCmdResponse**(*status*)
Bases: *digi.xbee.models.filesystem.FSCmd*

This class represents a delete file system command response.

Command response is received as a *DeleteCmdRequest*.

Class constructor. Instantiates a new *DeleteCmdResponse* object with the provided parameters.

Parameters **status** (*FSCommandStatus* or Integer) – Status of the file system command execution.

See also:

FSCmd

classmethod create_cmd(*raw*, *direction=1*)
Override method. Direction must be 1.

Returns *DeleteCmdResponse*.

Raises

- `InvalidPacketException` – If the bytearray length is less than 2. (cmd id + status = 2).
- `InvalidPacketException` – If the command type is not *FSCmdType* or direction is not 1.

See also:

FSCmd.create_cmd()

direction
Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

output()
Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

status
Returns the file system command response status.

Returns File system command response status.

Return type *FSCommandStatus*

See also:

FSCommandStatus

FSCmd.status_value()

status_value

Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

FSCmd.status()

to_dict()

Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

type

Returns the command type.

Returns The command type.

Return type *FSCmdType*

class digi.xbee.models.filesystem.**VolStatCmdRequest**(*name*)

Bases: *digi.xbee.models.filesystem.FSCmd*

This class represents a volume stat file system command request. Formatting the file system takes time, and any other requests fails until it completes and sends a response.

Command response is received as a *VolStatCmdResponse*.

Class constructor. Instantiates a new *VolStatCmdRequest* object with the provided parameters.

Parameters **name** (*String or bytearray*) – The name of the volume. Its maximum length is 254 characters.

Raises ValueError – If *name* is invalid.

See also:

FSCmd

name

Returns the name of the volume.

Returns The volume name.

Return type String

classmethod create_cmd(*raw*, *direction=0*)

Override method. Direction must be 0.

Returns *VolStatCmdRequest*.

Raises

- `InvalidPacketException` – If the bytearray length is less than 2. (cmd id + name (at least 1 byte) = 2 bytes).
- `InvalidPacketException` – If the command type is not *FSCmdType* or direction is not 0.

See also:

FSCmd.create_cmd()

direction

Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

output()

Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

status

Returns the file system command response status.

Returns File system command response status.

Return type *FSCommandStatus*

See also:

FSCommandStatus

`FSCmd.status_value()`

status_value

Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

`FSCmd.status()`

to_dict()

Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

type
Returns the command type.

Returns The command type.

Return type *FSCmdType*

class digi.xbee.models.filesystem.**VolStatCmdResponse**(*status*, *bytes_used=None*, *bytes_free=None*, *bytes_bad=None*)

Bases: *digi.xbee.models.filesystem.FSCmd*

This class represents a stat file system command response.

Command response is received as a *VolStatCmdRequest*.

Class constructor. Instantiates a new *VolStatCmdResponse* object with the provided parameters.

Parameters

- **status** (*FSCommandStatus* or Integer) – Status of the file system command execution.
- **bytes_used** (*Integer, optional, default=`None`*) – Number of used bytes.
- **bytes_free** (*Integer, optional, default=`None`*) – Number of free bytes.
- **bytes_bad** (*Integer, optional, default=`None`*) – Number of bad bytes. For XBee 3 802.15.4, DigiMesh, and Zigbee, this represents space used by deleted files.

Raises ValueError – If any of the parameters is invalid.

See also:

FSCmd

bytes_used
Returns the used space on volume.

Returns Number of used bytes.

Return type Integer

bytes_free
Returns the available space on volume.

Returns Number of free bytes.

Return type Integer

bytes_bad
Returns “bad” bytes on volume. For XBee 3 802.15.4, DigiMesh, and Zigbee, this represents space used by deleted files.

Returns Number of bad bytes.

Return type Integer

classmethod create_cmd(*raw*, *direction=1*)
Override method. Direction must be 1.

Returns *VolStatCmdResponse*.

Raises

- `InvalidPacketException` – If the bytearray length is less than 14. (cmd id + status + used (4 bytes) + free (4 bytes) + bad (4 bytes) = 14)
- `InvalidPacketException` – If the command type is not *`FSCmdType`* or direction is not 1.

See also:

`FileIdCmd.create_cmd()`

`direction`
Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

`output``()`
Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

`status`
Returns the file system command response status.

Returns File system command response status.

Return type *`FSCommandStatus`*

See also:

`FSCommandStatus`
`FSCmd.status_value()`

`status_value`
Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

`FSCmd.status()`

`to_dict``()`
Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

type
Returns the command type.

Returns The command type.

Return type *FSCmdType*

class digi.xbee.models.filesystem.**VolFormatCmdRequest**(*name*)
Bases: *digi.xbee.models.filesystem.VolStatCmdRequest*

This class represents a volume format file system command request.

Command response is received as a *VolFormatCmdResponse*.

Class constructor. Instantiates a new *VolFormatCmdRequest* object with the provided parameters.

Parameters **name** (*String or bytearray*) – The name of the volume. Its maximum length is 254 chars.

Raises ValueError – If *name* is invalid.

See also:

FSCmd

direction
Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

name
Returns the name of the volume.

Returns The volume name.

Return type String

output()
Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

status
Returns the file system command response status.

Returns File system command response status.

Return type *FSCommandStatus*

See also:

FSCommandStatus

FSCmd.status_value()

status_value
Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

FSCmd.status()

to_dict()
Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

type
Returns the command type.

Returns The command type.

Return type *FSCmdType*

classmethod create_cmd(*raw*, *direction=0*)
Override method. Direction must be 0.

Returns *VolFormatCmdRequest*.

Raises

- InvalidPacketException – If the bytearray length is less than 2. (cmd id + name (at least 1 byte) = 2 bytes).
- InvalidPacketException – If the command type is not *FSCmdType* or direction is not 0.

See also:

FSCmd.create_cmd()

class digi.xbee.models.filesystem.**VolFormatCmdResponse**(*status*, *bytes_used=None*, *bytes_free=None*, *bytes_bad=None*)
Bases: *digi.xbee.models.filesystem.VolStatCmdResponse*

This class represents a format file system command response.

Command response is received as a *VolStatCmdRequest*.

Class constructor. Instantiates a new *VolFormatCmdResponse* object with the provided parameters.

Parameters

- **status** (*FSCommandStatus* or Integer) – Status of the file system command execution.
- **bytes_used** (*Integer, optional, default=`None`*) – Number of used bytes.
- **bytes_free** (*Integer, optional, default=`None`*) – Number of free bytes.
- **bytes_bad** (*Integer, optional, default=`None`*) – Number of bad bytes.

Raises ValueError – If any of the parameters is invalid.

See also:

FSCmd

`bytes_bad`

Returns "bad" bytes on volume. For XBee 3 802.15.4, DigiMesh, and Zigbee, this represents space used by deleted files.

Returns Number of bad bytes.

Return type Integer

`bytes_free`

Returns the available space on volume.

Returns Number of free bytes.

Return type Integer

`bytes_used`

Returns the used space on volume.

Returns Number of used bytes.

Return type Integer

`direction`

Returns the command direction.

Returns 0 for request, 1 for response.

Return type Integer

`output``()`

Returns the raw bytearray of this command.

Returns Raw bytearray of the command.

Return type Bytearray

`status`

Returns the file system command response status.

Returns File system command response status.

Return type *FSCommandStatus*

See also:

FSCommandStatus

`FSCmd.status_value()`

`status_value`

Returns the file system command response status of the packet.

Returns File system command response status.

Return type Integer

See also:

FSCmd.status()

to_dict()
Returns a dictionary with all information of the command fields.

Returns Dictionary with all info of the command fields.

Return type Dictionary

type
Returns the command type.

Returns The command type.

Return type *FSCmdType*

classmethod create_cmd(*raw*, *direction=1*)
Override method. Direction must be 1.

Returns *VolFormatCmdResponse*.

Raises

- InvalidPacketException – If the bytearray length is less than 14. (cmd id + status + used (4 bytes) + free (4 bytes) + bad (4 bytes) = 14)
- InvalidPacketException – If the command type is not *FSCmdType* or direction is not 1.

See also:

FileIdCmd.create_cmd()

digi.xbee.models.hw module

class digi.xbee.models.hw.**HardwareVersion**(*code*, *description*)
Bases: enum.Enum

This class lists all hardware versions.

Inherited properties:
name (String): The name of this HardwareVersion.
value (Integer): The ID of this HardwareVersion.

Values:
HardwareVersion.X09_009 = (1, 'X09-009')
HardwareVersion.X09_019 = (2, 'X09-019')
HardwareVersion.XH9_009 = (3, 'XH9-009')
HardwareVersion.XH9_019 = (4, 'XH9-019')
HardwareVersion.X24_009 = (5, 'X24-009')
HardwareVersion.X24_019 = (6, 'X24-019')
HardwareVersion.X09_001 = (7, 'X09-001')
HardwareVersion.XH9_001 = (8, 'XH9-001')

HardwareVersion.X08_004 = (9, 'X08-004')
HardwareVersion.XC09_009 = (10, 'XC09-009')
HardwareVersion.XC09_038 = (11, 'XC09-038')
HardwareVersion.X24_038 = (12, 'X24-038')
HardwareVersion.X09_009_TX = (13, 'X09-009-TX')
HardwareVersion.X09_019_TX = (14, 'X09-019-TX')
HardwareVersion.XH9_009_TX = (15, 'XH9-009-TX')
HardwareVersion.XH9_019_TX = (16, 'XH9-019-TX')
HardwareVersion.X09_001_TX = (17, 'X09-001-TX')
HardwareVersion.XH9_001_TX = (18, 'XH9-001-TX')
HardwareVersion.XT09B_XXX = (19, 'XT09B-xxx (Attenuator version)')
HardwareVersion.XT09_XXX = (20, 'XT09-xxx')
HardwareVersion.XC08_009 = (21, 'XC08-009')
HardwareVersion.XC08_038 = (22, 'XC08-038')
HardwareVersion.XB24_AXX_XX = (23, 'XB24-Axx-xx')
HardwareVersion.XBP24_AXX_XX = (24, 'XBP24-Axx-xx')
HardwareVersion.XB24_BXIX_XXX = (25, 'XB24-BxIx-xxx and XB24-Z7xx-xxx')
HardwareVersion.XBP24_BXIX_XXX = (26, 'XBP24-BxIx-xxx and XBP24-Z7xx-xxx')
HardwareVersion.XBP09_DXIX_XXX = (27, 'XBP09-DxIx-xxx Digi Mesh')
HardwareVersion.XBP09_XCXX_XXX = (28, 'XBP09-XCxx-xxx: S3 XSC Compatibility')
HardwareVersion.XBP08_DXXX_XXX = (29, 'XBP08-Dxx-xxx 868MHz')
HardwareVersion.XBP24B = (30, 'XBP24B: Low cost ZB PRO and PLUS S2B')
HardwareVersion.XB24_WF = (31, 'XB24-WF: XBee 802.11 (Redpine module)')
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, 'Abandonated')
HardwareVersion.SMT_900LP = (38, "900LP (SMT): 900LP on 'S8 HW'")
HardwareVersion.WIFI_ATHEROS = (39, 'WiFi Atheros (TH-DIP) XB2S-WF')
HardwareVersion.SMT_WIFI_ATHEROS = (40, 'WiFi Atheros (SMT) XB2B-WF')
HardwareVersion.SMT_475LP = (41, '475LP (SMT): Beta 475MHz')
HardwareVersion.XBEE_CELL_TH = (42, 'XBee-Cell (TH): XBee Cellular')
HardwareVersion.XLR_MODULE = (43, 'XLR Module')
HardwareVersion.XB900HP_NZ = (44, 'XB900HP (New Zealand): XB9 NZ HW/SW')
HardwareVersion.XBP24C_TH_DIP = (45, 'XBP24C (TH-DIP): XBee PRO DIP')
HardwareVersion.XB24C_TH_DIP = (46, 'XB24C (TH-DIP): XBee DIP')
HardwareVersion.XLR_BASEBOARD = (47, 'XLR Baseboard')
HardwareVersion.XBP24C_S2C_SMT = (48, 'XBee PRO SMT')
HardwareVersion.SX_PRO = (49, 'SX Pro')
HardwareVersion.S2D_SMT_PRO = (50, 'XBP24D: S2D SMT PRO')
HardwareVersion.S2D_SMT_REG = (51, 'XB24D: S2D SMT Reg')
HardwareVersion.S2D_TH_PRO = (52, 'XBP24D: S2D TH PRO')
HardwareVersion.S2D_TH_REG = (53, 'XB24D: S2D TH Reg')
HardwareVersion.SX = (62, 'SX')
HardwareVersion.XTR = (63, 'XTR')

HardwareVersion.CELLULAR_CAT1_LTE_VERIZON = (64, 'XBee Cellular Cat 1 LTE Verizon')
HardwareVersion.XBEE3_SMT = (65, 'XBee 3 Micro and SMT')
HardwareVersion.XBEE3_TH = (66, 'XBee 3 TH')
HardwareVersion.XBEE3 = (67, 'XBee 3 Reserved')
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_NBIOT_EUROPE = (72, 'XBee Cellular NBIoT 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')
HardwareVersion.CELLULAR_3_CAT1_LTE_VERIZON = (77, 'XBee Cellular 3 Cat 1 LTE Verizon')

code

Returns the code of the HardwareVersion element.

Returns the code of the HardwareVersion element.

Return type Integer

description

Returns the description of the HardwareVersion element.

Returns the description of the HardwareVersion element.

Return type String

class digi.xbee.models.hw.**LegacyHardwareVersion**(*code*, *letter*)

Bases: `enum.Enum`

This class lists all legacy hardware versions.

Inherited properties:
name (String): The name of this LegacyHardwareVersion.
value (Integer): The ID of this LegacyHardwareVersion.

Values:
LegacyHardwareVersion.A = (1, 'A')
LegacyHardwareVersion.B = (2, 'B')
LegacyHardwareVersion.C = (3, 'C')
LegacyHardwareVersion.D = (4, 'D')
LegacyHardwareVersion.E = (5, 'E')
LegacyHardwareVersion.F = (6, 'F')
LegacyHardwareVersion.G = (7, 'G')
LegacyHardwareVersion.H = (8, 'H')
LegacyHardwareVersion.I = (9, 'I')
LegacyHardwareVersion.J = (10, 'J')

LegacyHardwareVersion.K = (11, 'K')
LegacyHardwareVersion.L = (12, 'L')
LegacyHardwareVersion.M = (13, 'M')
LegacyHardwareVersion.N = (14, 'N')
LegacyHardwareVersion.O = (15, 'O')
LegacyHardwareVersion.P = (16, 'P')
LegacyHardwareVersion.Q = (17, 'Q')
LegacyHardwareVersion.R = (18, 'R')
LegacyHardwareVersion.S = (19, 'S')
LegacyHardwareVersion.T = (20, 'T')
LegacyHardwareVersion.U = (21, 'U')
LegacyHardwareVersion.V = (22, 'V')
LegacyHardwareVersion.W = (23, 'W')
LegacyHardwareVersion.X = (24, 'X')
LegacyHardwareVersion.Y = (25, 'Y')
LegacyHardwareVersion.Z = (26, 'Z')

`code`
Returns the code of the LegacyHardwareVersion element.

Returns the code of the LegacyHardwareVersion element.

Return type Integer

`letter`
Returns the letter of the LegacyHardwareVersion element.

Returns the letter of the LegacyHardwareVersion element.

Return type String

digi.xbee.models.info module

class `digi.xbee.models.info.`**`SocketInfo`**(*socket_id*, *state*, *protocol*, *local_port*, *remote_port*, *remote_address*)

Bases: `object`

This class represents the information of an XBee socket:

- Socket ID.
- State.
- Protocol.
- Local port.
- Remote port.
- Remote address.

Class constructor. Instantiates a *SocketInfo* object with the given parameters.

Parameters

- **`socket_id`** (*`Integer`*) – The ID of the socket.

- **state** (*SocketInfoState*) – The state of the socket.
- **protocol** (*IPProtocol*) – The protocol of the socket.
- **local_port** (*Integer*) – The local port of the socket.
- **remote_port** (*Integer*) – The remote port of the socket.
- **remote_address** (*String*) – The remote IPv4 address of the socket.

static create_socket_info(*raw*)
Parses the given bytearray data and returns a *SocketInfo* object.

Parameters raw (*Bytearray*) – received data from the *SI* command with a socket ID as argument.

Returns

The socket information, or *None* if the provided data is invalid.

Return type *SocketInfo*

static parse_socket_list(*raw*)
Parses the given bytearray data and returns a list with the active socket IDs.

Parameters raw (*Bytearray*) – received data from the *SI* command.

Returns

list with the IDs of all active (open) sockets, or empty list if there is not any active socket.

Return type List

socket_id
Returns the ID of the socket.

Returns the ID of the socket.

Return type Integer

state
Returns the state of the socket.

Returns the state of the socket.

Return type *SocketInfoState*

protocol
Returns the protocol of the socket.

Returns the protocol of the socket.

Return type *IPProtocol*

local_port
Returns the local port of the socket. This is 0 unless the socket is explicitly bound to a port.

Returns the local port of the socket.

Return type Integer

remote_port
Returns the remote port of the socket.

Returns the remote port of the socket.

Return type Integer

remote_address

Returns the remote IPv4 address of the socket. This is *0.0.0.0* for an unconnected socket.

Returns the remote IPv4 address of the socket.

Return type String

digi.xbee.models.mode module

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

Bases: `enum.Enum`

This class represents all operating modes available.

Inherited properties:

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

Values:

OperatingMode.AT_MODE = (0, 'AT mode')
OperatingMode.API_MODE = (1, 'API mode')
OperatingMode.ESCAPED_API_MODE = (2, 'API mode with escaped characters')
OperatingMode.MICROPYTHON_MODE = (4, 'MicroPython REPL')
OperatingMode.BYPASS_MODE = (5, 'Bypass mode')
OperatingMode.UNKNOWN = (99, 'Unknown')

code

Returns the code of the OperatingMode element.

Returns the code of the OperatingMode element.

Return type String

description

Returns the description of the OperatingMode element.

Returns the description of the OperatingMode element.

Return type String

class digi.xbee.models.mode.**APIOutputMode**(*code*, *description*)

Bases: `enum.Enum`

Enumerates the different API output modes. The API output mode establishes the way data will be output through the serial interface of an XBee device.

Inherited properties:

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

Values:

APIOutputMode.NATIVE = (0, 'Native')

APIOutputMode.EXPLICIT = (1, 'Explicit')

APIOutputMode.EXPLICIT_ZDO_PASSTHRU = (3, 'Explicit with ZDO Passthru')

code

Returns the code of the APIOutputMode element.

Returns the code of the APIOutputMode element.

Return type String

description

Returns the description of the APIOutputMode element.

Returns the description of the APIOutputMode element.

Return type String

class digi.xbee.models.mode.**APIOutputModeBit**(*code*, *description*)

Bases: enum.Enum

Enumerates the different API output mode bit options. The API output mode establishes the way data will be output through the serial interface of an XBee.

Inherited properties:

name (String): the name (id) of this APIOutputModeBit.

value (String): the value of this APIOutputModeBit.

Values:

APIOutputModeBit.EXPLICIT = (1, 'Output in Native/Explicit API format')

APIOutputModeBit.SUPPORTED_ZDO_PASSTHRU = (2, 'Zigbee: Supported ZDO request pass-throughn802.15.4/DigiMesh: Legacy API Indicator')

APIOutputModeBit.UNSUPPORTED_ZDO_PASSTHRU = (4, 'Unsupported ZDO request pass-through. Only Zigbee')

APIOutputModeBit.BINDING_PASSTHRU = (8, 'Binding request pass-through. Only Zigbee')

APIOutputModeBit.ECHO_RCV_SUPPORTED_ZDO = (16, 'Echo received supported ZDO requests out the serial port. Only Zigbee')

APIOutputModeBit.SUPPRESS_ALL_ZDO_MSG = (32, 'Suppress all ZDO messages from being sent out the serial port and disable pass-through. Only Zigbee')

code

Returns the code of the APIOutputModeBit element.

Returns the code of the APIOutputModeBit element.

Return type Integer

description

Returns the description of the APIOutputModeBit element.

Returns the description of the APIOutputModeBit element.

Return type String

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

Bases: enum.Enum

Enumerates the different IP addressing modes.

Values:

IPAddressingMode.DHCP = (0, 'DHCP')

IPAddressingMode.STATIC = (1, 'Static')

code

Returns the code of the IPAddressingMode element.

Returns the code of the IPAddressingMode element.

Return type String

description

Returns the description of the IPAddressingMode element.

Returns the description of the IPAddressingMode element.

Return type String

class digi.xbee.models.mode.**NeighborDiscoveryMode**(*code*, *description*)

Bases: enum.Enum

Enumerates the different neighbor discovery modes. This mode establishes the way the network discovery process is performed.

Inherited properties:

name (String): the name (id) of this OperatingMode.

value (String): the value of this OperatingMode.

Values:

NeighborDiscoveryMode.CASCADE = (0, 'Cascade')

NeighborDiscoveryMode.FLOOD = (1, 'Flood')

CASCADE = (0, 'Cascade')

The discovery of a node neighbors is requested once the previous request finishes. This means that just one discovery process is running at the same time.

This mode is recommended for large networks, it might be a slower method but it generates less traffic than 'Flood'.

FLOOD = (1, 'Flood')

The discovery of a node neighbors is requested when the node is found in the network. This means that several discovery processes might be running at the same time.

code
Returns the code of the NeighborDiscoveryMode element.

Returns the code of the NeighborDiscoveryMode element.

Return type String

description
Returns the description of the NeighborDiscoveryMode element.

Returns the description of the NeighborDiscoveryMode element.

Return type String

digi.xbee.models.address module

class `digi.xbee.models.address.`**`XBee16BitAddress`**(*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* is *None* or has less than 1 byte or more than 2.

PATTERN = '^(0[xX])?[0-9a-fA-F]{1,4}$'
16-bit address string pattern.

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

Type 16-bit address reserved for the coordinator (value

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.

classmethod is_valid(*address*)
Checks if the provided hex string is a valid 16-bit address.

Parameters **address** (String or Bytearray, or *XBee16BitAddress*) – String: String with the address only with hex digits without blanks. Minimum 1 character, maximum 4 (16-bit). Bytearray: Address as byte array. Must be 1-2 digits.

Returns *True* for a valid 16-bit address, *False* otherwise.

Return type Boolean

classmethod is_known_node_addr(*address*)
Checks if a provided address is a known value. That is, if it is a valid 16-bit address and it is not the unknown or the broadcast address.

Parameters **address** (String, Bytearray, or *XBee16BitAddress*) – The 16-bit address to check as a string, bytearray or *XBee16BitAddress*.

Returns *True* for a known node 16-bit address, *False* otherwise.

Return type Boolean

get_hsb()
Returns the high part of the bytearray (component 0).

Returns high part of the bytearray.

Return type Integer

get_lsb()
Returns the low part of the bytearray (component 1).

Returns low part of the bytearray.

Return type Integer

address
Returns a bytearray representation of this XBee16BitAddress.

Returns bytearray representation of this XBee16BitAddress.

Return type Bytearray

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 *address* is *None* or its length less than 1 or greater than 8.

PATTERN = '^(0[xX])?[0-9a-fA-F]{1,16}$'
64-bit address string pattern.

COORDINATOR_ADDRESS = <digi.xbee.models.address.XBee64BitAddress object>
0000000000000000).

Type 64-bit address reserved for the coordinator (value

BROADCAST_ADDRESS = <digi.xbee.models.address.XBee64BitAddress object>
000000000000FFFF).

Type 64-bit broadcast address (value

UNKNOWN_ADDRESS = <digi.xbee.models.address.XBee64BitAddress object>
FFFFFFFFFFFFFFFF).

Type 64-bit unknown address (value

classmethod from_hex_string(*address*)
Class constructor. Instantiates a new *XBee64BitAddress* object from the provided hex string.

Parameters **address** (*String*) – The 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 *XBee64BitAddress* object from the provided bytes.

Parameters **args** (*8 Integers*) – 8 integers that represent the bytes 1 to 8 of this XBee64BitAddress.

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

classmethod is_valid(*address*)
Checks if the provided hex string is a valid 64-bit address.

Parameters **address** (String, Bytearray, or *XBee64BitAddress*) – String: String with the address only with hex digits without blanks. Minimum 1 character, maximum 16 (64-bit). Bytearray: Address as byte array. Must be 1-8 digits.

Returns Boolean: *True* for a valid 64-bit address, *False* otherwise.

classmethod is_known_node_addr(*address*)

Checks if a provided address is a known value. That is, if it is a valid 64-bit address and it is not the unknown or the broadcast address.

Parameters **address** (String, Bytearray, or *XBee64BitAddress*) – The 64-bit address to check as a string, bytearray or *XBee64BitAddress*.

Returns *True* for a known node 64-bit address, *False* otherwise.

Return type Boolean

address

Returns a bytearray representation of this XBee64BitAddress.

Returns bytearray representation of this XBee64BitAddress.

Return type Bytearray

class digi.xbee.models.address.**XBeeIMEIAddress**(*address*)

Bases: `object`

This class represents an IMEI address used by cellular devices.

This address is only applicable for Cellular protocol.

Class constructor. Instantiates a new :.*XBeeIMEIAddress* object with the provided parameters.

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

Raises

- `ValueError` – if *address* is *None*.
- `ValueError` – if length of *address* greater than 8.

PATTERN = '^\\d{0,15}$'

IMEI address string pattern.

classmethod from_string(*address*)

Class constructor. Instantiates a new :.*XBeeIMEIAddress* object from the provided string.

Parameters **address** (*String*) – The XBee IMEI address as a string.

Raises

- `ValueError` – if *address* is *None*.
- `ValueError` – if *address* does not match the pattern: *^d{0,15}$*.

classmethod is_valid(*address*)

Checks if the provided hex string is a valid IMEI.

Parameters **address** (*String or Bytearray*) – The XBee IMEI address as a string or bytearray.

Returns *True* for a valid IMEI, *False* otherwise.

Return type Boolean

address
Returns a string representation of this XBeeIMEIAddress.

Returns the IMEI address in string format.

Return type String

digi.xbee.models.message module

class `digi.xbee.models.message.`**XBeeMessage**(*data*, *remote_node*, *timestamp*, *broadcast=False*)

Bases: `object`

This class represents a XBee message, which is formed by a *RemoteXBeeDevice* (the sender) and some data (the data sent) as a bytearray.

Class constructor.

Parameters

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

data
Returns a bytearray containing the data of the message.

Returns the data of the message.

Return type Bytearray

remote_device
Returns the device which has sent the message.

Returns the device which has sent the message.

Return type *RemoteXBeeDevice*

is_broadcast
Returns whether the message is broadcast or not.

Returns *True* if the message is broadcast, *False* otherwise.

Return type Boolean

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

to_dict()
Returns the message information as a dictionary.

class digi.xbee.models.message.**ExplicitXBeeMessage**(*data*, *remote_node*, *timestamp*, *src_endpoint*, *dest_endpoint*, *cluster_id*, *profile_id*, *broadcast=False*)

Bases: *digi.xbee.models.message.XBeeMessage*

This class represents an Explicit XBee message, which is formed by all parameters of a common XBee message and: Source endpoint, destination endpoint, cluster ID, profile ID.

Class constructor.

Parameters

- **data** (*Bytearray*) – the data sent.
- **remote_node** (*RemoteXBeeDevice*) – the sender device.
- **timestamp** – instant of time when the message was received.
- **src_endpoint** (*Integer*) – source endpoint of the message. 1 byte.
- **dest_endpoint** (*Integer*) – destination endpoint of the message. 1 byte.
- **cluster_id** (*Integer*) – cluster id of the message. 2 bytes.
- **profile_id** (*Integer*) – profile id of the message. 2 bytes.
- **broadcast** (*Boolean, optional, default=`False`*) – flag indicating whether the message is broadcast (*True*) or not (*False*). Optional.

source_endpoint

Returns the source endpoint of the message.

Returns the source endpoint of the message. 1 byte.

Return type Integer

dest_endpoint

Returns the destination endpoint of the message.

Returns the destination endpoint of the message. 1 byte.

Return type Integer

cluster_id

Returns the cluster ID of the message.

Returns the cluster ID of the message. 2 bytes.

Return type Integer

profile_id

Returns the profile ID of the message.

Returns the profile ID of the message. 2 bytes.

Return type Integer

to_dict()

Returns the message information as a dictionary.

data

Returns a bytearray containing 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*, *src_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.
- **src_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.

ip_addr
Returns the IPv4 address this message is associated to.

Returns The IPv4 address this message is associated to.

Return type `ipaddress.IPv4Address`

source_port
Returns the source port of the transmission.

Returns The source port of the transmission.

Return type Integer

dest_port
Returns the destination port of the transmission.

Returns The destination port of the transmission.

Return type Integer

protocol
Returns the protocol used in the transmission.

Returns The protocol used in the transmission.

Return type *IPProtocol*

data
Returns a bytearray containing the data of the message.

Returns the data of the message.

Return type Bytearray

to_dict()
Returns the message information as a dictionary.

class digi.xbee.models.message.**SMSMessage**(*phone_number*, *data*)
Bases: object

This class represents an SMS message containing the phone number that sent the message and the content (data) of the message.

This class is used within the library to read SMS sent to Cellular devices.

Class constructor. Instantiates a new *SMSMessage* object with the provided parameters.

Parameters

- **phone_number** (*String*) – The phone number that sent the message.
- **data** (*String*) – The message text.

Raises

- ValueError – if *phone_number* is *None*.
- ValueError – if *data* is *None*.
- ValueError – if *phone_number* is not a valid phone number.

phone_number
Returns the phone number that sent the message.

Returns The phone number that sent the message.

Return type String

data
Returns the data of the message.

Returns The data of the message.

Return type String

to_dict()
Returns the message information as a dictionary.

class digi.xbee.models.message.**UserDataRelayMessage**(*local_iface*, *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_iface** (*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

local_interface

Returns the source interface that sent the message.

Returns The source interface that sent the message.

Return type *XBeeLocalInterface*

data

Returns the data of the message.

Returns The data of the message.

Return type Bytearray

to_dict()

Returns the message information as a dictionary.

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.BROADCAST_PANS_PACKET = 4
ReceiveOptions.SECURE_SESSION_ENC = 16

ReceiveOptions.APS_ENCRYPTED = 32
ReceiveOptions.SENT_FROM_END_DEVICE = 64
ReceiveOptions.REPEATER_MODE = 128
ReceiveOptions.DIGIMESH_MODE = 192

NONE = 0
No special receive options.

PACKET_ACKNOWLEDGED = 1
Packet was acknowledged.

Not valid for WiFi protocol.

BROADCAST_PACKET = 2
Packet was sent as a broadcast.

Not valid for WiFi protocol.

BROADCAST_PANS_PACKET = 4
Packet was broadcast accros all PANs.

Only for 802.15.4 protocol.

SECURE_SESSION_ENC = 16
Packet sent across a Secure Session.

Only for XBee 3.

APS_ENCRYPTED = 32
Packet encrypted with APS encryption.

Only valid for Zigbee protocol.

SENT_FROM_END_DEVICE = 64
Packet was sent from an end device (if known).

Only valid for Zigbee protocol.

POINT_MULTIPOINT_MODE = 64
Transmission is performed using point-to-Multipoint mode.

Only valid for DigiMesh 868/900 and Point-to-Multipoint 868/900 protocols.

REPEATER_MODE = 128
Transmission is performed using repeater mode.

Only valid for DigiMesh 868/900 and Point-to-Multipoint 868/900 protocols.

DIGIMESH_MODE = 192
Transmission is performed using DigiMesh mode.

Only valid for DigiMesh 868/900 and Point-to-Multipoint 868/900 protocols.

class digi.xbee.models.options.**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.SECURE_SESSION_ENC = 16
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 Zigbee, 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.

`BROADCAST_PAN = 2`

Sends packet with broadcast {@code PAN ID}. Packet will be sent to all PANs.

Only valid for 802.15.4 XBee 3 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, and XBee 3 DigiMesh.

`ENABLE_UNICAST_TRACE_ROUTE = 4`

Enables unicast trace route messages.

Trace route is enabled on the packets.

Only valid for DigiMesh 868/900 protocol.

INDIRECT_TRANSMISSION = 4
Used for binding transmissions.

Only valid for Zigbee protocol.

ENABLE_MULTICAST = 8
Enables multicast transmission request.

Only valid for Zigbee XBee protocol.

ENABLE_TRACE_ROUTE = 8
Enable a unicast Trace Route on DigiMesh transmissions When set, the transmission will generate a Route Information - 0x8D frame.

Only valid for DigiMesh XBee protocol.

SECURE_SESSION_ENC = 16
Encrypt payload for transmission across a Secure Session. Reduces maximum payload size by 4 bytes.

Only for XBee 3.

ENABLE_APS_ENCRYPTION = 32
Enables APS encryption, only if {@code EE=1}.

Enabling APS encryption decreases the maximum number of RF payload bytes by 4 (below the value reported by {@code NP}).

Only valid for Zigbee XBee protocol.

USE_EXTENDED_TIMEOUT = 64
Uses the extended transmission timeout.

Setting the extended timeout bit causes the stack to set the extended transmission timeout for the destination address.

Only valid for Zigbee XBee protocol.

POINT_MULTIPOINT_MODE = 64
Transmission is performed using point-to-Multipoint mode.

Only valid for DigiMesh 868/900 and Point-to-Multipoint 868/900 protocols.

REPEATER_MODE = 128
Transmission is performed using repeater mode.

Only valid for DigiMesh 868/900 and Point-to-Multipoint 868/900 protocols.

DIGIMESH_MODE = 192
Transmission is performed using DigiMesh mode.

Only valid for DigiMesh 868/900 and Point-to-Multipoint 868/900 protocols.

class `digi.xbee.models.options.`**RemoteATCmdOptions**
Bases: `enum.Enum`

This class lists all the possible options that can be set while transmitting a remote AT Command.

These options are usually set as a bitfield meaning that the options can be combined using the '|' operand.

Values:

RemoteATCmdOptions.NONE = 0
RemoteATCmdOptions.DISABLE_ACK = 1
RemoteATCmdOptions.APPLY_CHANGES = 2
RemoteATCmdOptions.SECURE_SESSION_ENC = 16
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.

`SECURE_SESSION_ENC = 16`
Send the remote command securely. Requires a Secure Session be established with the destination.

Only for XBee 3.

`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`
Returns the code of the SendDataRequestOptions element.

Returns the code of the SendDataRequestOptions element.

Return type Integer

`description`
Returns the description of the SendDataRequestOptions element.

Returns the description of the SendDataRequestOptions element.

Return type String

class digi.xbee.models.options.**DiscoveryOptions**(*code*, *description*)
Bases: `enum.Enum`

Enumerates the different options used in the discovery process.

Values:

DiscoveryOptions.APPEND_DD = (1, 'Append device type identifier (DD)')
DiscoveryOptions.DISCOVER_MYSELF = (2, 'Local device sends response frame')
DiscoveryOptions.APPEND_RSSI = (4, 'Append RSSI (of the last hop)')

APPEND_DD = (1, 'Append device type identifier (DD)')
Append device type identifier (DD) to the discovery response.

Valid for the following protocols:

- DigiMesh
- Point-to-multipoint (Digi Point)
- Zigbee

DISCOVER_MYSELF = (2, 'Local device sends response frame')
Local device sends response frame when discovery is issued.

Valid for the following protocols:

- DigiMesh
- Point-to-multipoint (Digi Point)
- Zigbee
- 802.15.4

APPEND_RSSI = (4, 'Append RSSI (of the last hop)')
Append RSSI of the last hop to the discovery response.

Valid for the following protocols:

- DigiMesh
- Point-to-multipoint (Digi Point)

code
Returns the code of the *DiscoveryOptions* element.

Returns the code of the *DiscoveryOptions* element.

Return type Integer

description
Returns the description of the *DiscoveryOptions* element.

Returns the description of the *DiscoveryOptions* element.

Return type String

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`

Returns the code of the *XBeeLocalInterface* element.

Returns the code of the *XBeeLocalInterface* element.

Return type Integer

`description`

Returns the description of the *XBeeLocalInterface* element.

Returns the description of the *XBeeLocalInterface* element.

Return type String

`class` `digi.xbee.models.options.`**`RegisterKeyOptions`**(*code*, *description*)

Bases: `enum.Enum`

This class lists all the possible options that have been set while receiving an XBee packet.

The receive options are usually set as a bitfield meaning that the options can be combined using the '|' operand.

Values:

RegisterKeyOptions.LINK_KEY = (0, 'Key is a Link Key (KY on joining node)')

RegisterKeyOptions.INSTALL_CODE = (1, 'Key is an Install Code (I? on joining node,DC must be set to 1 on joiner)')

RegisterKeyOptions.UNKNOWN = (255, 'Unknown key option')

`code`

Returns the code of the *RegisterKeyOptions* element.

Returns the code of the *RegisterKeyOptions* element.

Return type Integer

`description`

Returns the description of the *RegisterKeyOptions* element.

Returns the description of the *RegisterKeyOptions* element.

Return type String

class `digi.xbee.models.options.`**`SocketOption`**(*code*, *description*)

Bases: `enum.Enum`

Enumerates the different Socket Options.

Values:

SocketOption.TLS_PROFILE = (0, 'TLS Profile')
SocketOption.UNKNOWN = (255, 'Unknown')

`code`

Returns the code of the *SocketOption* element.

Returns the code of the *SocketOption* element.

Return type Integer

`description`

Returns the description of the *SocketOption* element.

Returns the description of the *SocketOption* element.

Return type String

class `digi.xbee.models.options.`**`FileOpenRequestOption`**

Bases: `enum.IntFlag`

This enumeration lists all the available options for *FSCmdType.FILE_OPEN* command requests.

Inherited properties:

name (String): Name (id) of this FileOpenRequestOption.
value (String): Value of this FileOpenRequestOption.

Values:

FileOpenRequestOption.CREATE = 1
FileOpenRequestOption.EXCLUSIVE = 2
FileOpenRequestOption.READ = 4
FileOpenRequestOption.WRITE = 8
FileOpenRequestOption.TRUNCATE = 16
FileOpenRequestOption.APPEND = 32
FileOpenRequestOption.SECURE = 128

`CREATE = 1`

Create if file does not exist.

`EXCLUSIVE = 2`

Error out if file exists.

`READ = 4`

Open file for reading.

WRITE = 8
Open file for writing.

TRUNCATE = 16
Truncate file to 0 bytes.

APPEND = 32
Append to end of file.

SECURE = 128
Create a secure write-only file.

class `digi.xbee.models.options.`**DirResponseFlag**
Bases: `enum.IntFlag`

This enumeration lists all the available flags for *FSCmdType.DIR_OPEN* and *FSCmdType.DIR_READ* command responses.

Inherited properties:
name (String): Name (id) of this DirResponseFlag.
value (String): Value of this DirResponseFlag.

Values:
DirResponseFlag.IS_DIR = 128
DirResponseFlag.IS_SECURE = 64
DirResponseFlag.IS_LAST = 1

IS_DIR = 128
Entry is a directory.

IS_SECURE = 64
Entry is stored securely.

IS_LAST = 1
Entry is the last.

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`
Returns the code of the XBeeProtocol element.

Returns the code of the XBeeProtocol element.

Return type Integer

`description`
Returns the description of the XBeeProtocol element.

Returns the description of the XBeeProtocol element.

Return type String

class `digi.xbee.models.protocol.`**`IPProtocol`**(*code*, *description*)
Bases: `enum.Enum`

Enumerates the available network protocols.

Inherited properties:
name (String): the name (id) of this IPProtocol.
value (String): the value of this IPProtocol.

Values:
IPProtocol.UDP = (0, 'UDP')
IPProtocol.TCP = (1, 'TCP')
IPProtocol.TCP_SSL = (4, 'TLS')

`code`
Returns the code of the IP protocol.

Returns code of the IP protocol.

Return type Integer

`description`
Returns the description of the IP protocol.

Returns description of the IP protocol.

Return type String

class `digi.xbee.models.protocol.`**`Role`**(*identifier*, *description*)
Bases: `enum.Enum`

Enumerates the available roles for an XBee.

Inherited properties:
name (String): the name (id) of this Role.
value (String): the value of this Role.

Values:
Role.COORDINATOR = (0, 'Coordinator')
Role.ROUTER = (1, 'Router')
Role.END_DEVICE = (2, 'End device')
Role.UNKNOWN = (3, 'Unknown')

`id`
Gets the identifier of the role.

Returns the role identifier.

Return type Integer

`description`
Gets the description of the role.

Returns the role description.

Return type String

digi.xbee.models.status module

class `digi.xbee.models.status.`**`ATCommandStatus`**(*code*, *description*)
Bases: `enum.Enum`

This class lists all the possible states of an AT command after execution.

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.NO_SECURE_SESSION = (11, 'No secure session: Remote command access requires a secure session be established first')

ATCommandStatus.ENC_ERROR = (12, 'Encryption error')

ATCommandStatus.CMD_SENT_INSECURELY = (13, 'Command sent insecurely: A secure session exists, but the request needs to have the appropriate command option set (bit 4)')

ATCommandStatus.UNKNOWN = (255, 'Unknown status')

`code`

Returns the code of the ATCommandStatus element.

Returns the code of the ATCommandStatus element.

Return type Integer

`description`

Returns the description of the ATCommandStatus element.

Returns the description of the ATCommandStatus element.

Return type String

`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`

Returns the code of the DiscoveryStatus element.

Returns the code of the DiscoveryStatus element.

Return type Integer

description

Returns the description of the DiscoveryStatus element.

Returns The description of the DiscoveryStatus element.

Return type String

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 ths TransmitStatus.

value (String): the value of ths 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, 'Transceiver was unable to complete the transmission')

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.NO_SECURE_SESSION = (52, 'No Secure session connection')

TransmitStatus.ENC_FAILURE = (53, 'Encryption failure')

TransmitStatus.PAYLOAD_TOO_LARGE = (116, 'Data payload too large')

TransmitStatus.INDIRECT_MESSAGE_UNREQUESTED = (117, 'Indirect message unrequested')

TransmitStatus.SOCKET_CREATION_FAILED = (118, 'Attempt to create a client socket failed')

TransmitStatus.IP_PORT_NOT_EXIST = (119, 'TCP connection to given IP address and port does not exist. Source port is non-zero, so a new connection is not attempted')

TransmitStatus.UDP_SRC_PORT_NOT_MATCH_LISTENING_PORT = (120, 'Source port on a UDP transmission does not match a listening port on the transmitting module')

TransmitStatus.TCP_SRC_PORT_NOT_MATCH_LISTENING_PORT = (121, 'Source port on a TCP transmission does not match a listening port on the transmitting module')

TransmitStatus.INVALID_IP_ADDRESS = (122, 'Destination IPv4 address is invalid')

TransmitStatus.INVALID_IP_PROTOCOL = (123, 'Protocol on an IPv4 transmission is invalid')

TransmitStatus.RELAY_INTERFACE_INVALID = (124, 'Destination interface on a User Data Relay Frame does not exist')

TransmitStatus.RELAY_INTERFACE_REJECTED = (125, 'Destination interface on a User Data Relay Frame exists, but the interface is not accepting data')

TransmitStatus.MODEM_UPDATE_IN_PROGRESS = (126, 'Modem update in progress. Try again after update completion.')

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, 'No server')

TransmitStatus.SOCKET_ERROR_CLOSED = (131, 'The existing connection was closed')

TransmitStatus.SOCKET_ERROR_UNKNOWN_SERVER = (132, 'The server could not be found')

TransmitStatus.SOCKET_ERROR_UNKNOWN_ERROR = (133, 'An unknown error occurred')

TransmitStatus.INVALID_TLS_CONFIGURATION = (134, 'TLS Profile on a 0x23 API request does not exist, or one or more certificates is invalid')

TransmitStatus.SOCKET_NOT_CONNECTED = (135, 'Socket not connected')

TransmitStatus.SOCKET_NOT_BOUND = (136, 'Socket not bound')

TransmitStatus.KEY_NOT_AUTHORIZED = (187, 'Key not authorized')

TransmitStatus.UNKNOWN = (255, 'Unknown')

`code`

Returns the code of the TransmitStatus element.

Returns the code of the TransmitStatus element.

Return type Integer

`description`

Returns the description of the TransmitStatus element.

Returns the description of the TransmitStatus element.

Return type String

`class` `digi.xbee.models.status.`**`ModemStatus`**(*code*, *description*)

Bases: `enum.Enum`

Enumerates the different modem status events. This enumeration list is intended to be used within the *`ModemStatusPacket`* packet.

Values:

ModemStatus.HARDWARE_RESET = (0, 'Device was reset')

ModemStatus.WATCHDOG_TIMER_RESET = (1, 'Watchdog timer was reset')

ModemStatus.JOINED_NETWORK = (2, 'Device joined to network')

ModemStatus.DISASSOCIATED = (3, 'Device disassociated')

ModemStatus.ERROR_SYNCHRONIZATION_LOST = (4, 'Configuration error/synchronization lost')

ModemStatus.COORDINATOR_REALIGNMENT = (5, 'Coordinator realignment')

ModemStatus.COORDINATOR_STARTED = (6, 'The coordinator started')

ModemStatus.NETWORK_SECURITY_KEY_UPDATED = (7, 'Network security key was updated')

ModemStatus.NETWORK_WOKE_UP = (11, 'Network woke up')

ModemStatus.NETWORK_WENT_TO_SLEEP = (12, 'Network went to sleep')

ModemStatus.VOLTAGE_SUPPLY_LIMIT_EXCEEDED = (13, 'Voltage supply limit exceeded')

ModemStatus.REMOTE_MANAGER_CONNECTED = (14, 'Remote Manager connected')

ModemStatus.REMOTE_MANAGER_DISCONNECTED = (15, 'Remote Manager disconnected')

ModemStatus.MODEM_CONFIG_CHANGED_WHILE_JOINING = (17, 'Modem configuration changed while joining')

ModemStatus.ACCESS_FAULT = (18, 'Access fault')

ModemStatus.FATAL_ERROR = (19, 'Fatal error')

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.CELLULAR_UPDATE_START = (53, 'Cellular component update started')

ModemStatus.CELLULAR_UPDATE_FAILED = (54, 'Cellular component update failed')

ModemStatus.CELLULAR_UPDATE_SUCCESS = (55, 'Cellular component update completed')

ModemStatus.FIRMWARE_UPDATE_START = (56, 'XBee firmware update started')

ModemStatus.FIRMWARE_UPDATE_FAILED = (57, 'XBee firmware update failed')

ModemStatus.FIRMWARE_UPDATE_APPLYING = (58, 'XBee firmware update applying')

ModemStatus.SEC_SESSION_ESTABLISHED = (59, 'Secure session successfully established')

ModemStatus.SEC_SESSION_END = (60, 'Secure session ended')

ModemStatus.SEC_SESSION_AUTH_FAILED = (61, 'Secure session authentication failed')

ModemStatus.COORD_PAN_ID_CONFLICT = (62, 'Coordinator detected a PAN ID conflict but took no action because CR=0')

ModemStatus.COORD_CHANGE_PAN_ID = (63, 'Coordinator changed PAN ID due to a conflict')

ModemStatus.ROUTER_PAN_ID_CHANGED = (64, 'Router PAN ID was changed by coordinator due to a conflict')

ModemStatus.NET_WATCHDOG_EXPIRED = (66, 'Network watchdog timeout expired')

ModemStatus.ERROR_STACK = (128, 'Stack error')

ModemStatus.ERROR_AP_NOT_CONNECTED = (130, 'Send/join command issued without connecting from AP')

ModemStatus.ERROR_AP_NOT_FOUND = (131, 'Access point not found')

ModemStatus.ERROR_PSK_NOT_CONFIGURED = (132, 'PSK not configured')

ModemStatus.ERROR_SSID_NOT_FOUND = (135, 'SSID not found')

ModemStatus.ERROR_FAILED_JOIN_SECURITY = (136, 'Failed to join with security enabled')

ModemStatus.ERROR_INVALID_CHANNEL = (138, 'Invalid channel')

ModemStatus.ERROR_FAILED_JOIN_AP = (142, 'Failed to join access point')

ModemStatus.UNKNOWN = (255, 'UNKNOWN')

code

Returns the code of the ModemStatus element.

Returns the code of the ModemStatus element.

Return type Integer

description
Returns the description of the ModemStatus element.

Returns the description of the ModemStatus element.

Return type String

class digi.xbee.models.status.**PowerLevel**(*code*, *description*)
Bases: enum.Enum

Enumerates the different power levels. The power level indicates the output power value of a radio when transmitting data.

Values:

PowerLevel.LEVEL_LOWEST = (0, 'Lowest')
PowerLevel.LEVEL_LOW = (1, 'Low')
PowerLevel.LEVEL_MEDIUM = (2, 'Medium')
PowerLevel.LEVEL_HIGH = (3, 'High')
PowerLevel.LEVEL_HIGHEST = (4, 'Highest')
PowerLevel.LEVEL_UNKNOWN = (255, 'Unknown')

code
Returns the code of the PowerLevel element.

Returns the code of the PowerLevel element.

Return type Integer

description
Returns the description of the PowerLevel element.

Returns the description of the PowerLevel element.

Return type String

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 onfigured to support beacons')

AssociationIndicationStatus.AS_ID_DOESNT_MATCH = (5, 'Active Scan found PAN, but the Coordinator ID parameter does not match the ID parameter of the End Device')

AssociationIndicationStatus.AS_CHANNEL_DOESNT_MATCH = (6, 'Active Scan found PAN, but the Coordinator CH parameter does not match the CH parameter of the End Device')

AssociationIndicationStatus.ENERGY_SCAN_TIMEOUT = (7, 'Energy Scan Timeout')

AssociationIndicationStatus.COORDINATOR_START_REQUEST_FAILED = (8, 'Coordinator start request failed')

AssociationIndicationStatus.COORDINATOR_INVALID_PARAMETER = (9, 'Coordinator could not start due to invalid parameter')

AssociationIndicationStatus.COORDINATOR_REALIGNMENT = (10, 'Coordinator Realignment is in progress')

AssociationIndicationStatus.AR_NOT_SENT = (11, 'Association Request not sent')

AssociationIndicationStatus.AR_TIMED_OUT = (12, 'Association Request timed out - no reply was received')

AssociationIndicationStatus.AR_INVALID_PARAMETER = (13, 'Association Request had an Invalid Parameter')

AssociationIndicationStatus.AR_CHANNEL_ACCESS_FAILURE = (14, 'Association Request Channel Access Failure. Request was not transmitted - CCA failure')

AssociationIndicationStatus.AR_COORDINATOR_ACK_WASNT_RECEIVED = (15, 'Remote Coordinator did not send an ACK after Association Request was sent')

AssociationIndicationStatus.AR_COORDINATOR_DIDNT_REPLY = (16, 'Remote Coordinator did not reply to the Association Request, but an ACK was received after sending the request')

AssociationIndicationStatus.SYNCHRONIZATION_LOST = (18, 'Sync-Loss - Lost synchronization with a Beaconing Coordinator')

AssociationIndicationStatus.DISASSOCIATED = (19, ' Disassociated - No longer associated to Coordinator')

AssociationIndicationStatus.NO_PANS_FOUND = (33, 'Scan found no PANs.')

AssociationIndicationStatus.NO_PANS_WITH_ID_FOUND = (34, 'Scan found no valid PANs based on current SC and ID settings')

AssociationIndicationStatus.NJ_EXPIRED = (35, 'Valid Coordinator or Routers found, but they are not allowing joining (NJ expired)')

AssociationIndicationStatus.NO_JOINABLE_BEACONS_FOUND = (36, 'No joinable beacons were found')

AssociationIndicationStatus.UNEXPECTED_STATE = (37, 'Unexpected state, node should not be attempting to join at this time')

AssociationIndicationStatus.JOIN_FAILED = (39, 'Node Joining attempt failed (typically due to incompatible security settings)')

AssociationIndicationStatus.COORDINATOR_START_FAILED = (42, 'Coordinator Start attempt failed')

AssociationIndicationStatus.CHECKING_FOR_COORDINATOR = (43, 'Checking for an existing coordinator')

AssociationIndicationStatus.NETWORK_LEAVE_FAILED = (44, 'Attempt to leave the network failed')

AssociationIndicationStatus.DEVICE_DIDNT_RESPOND = (171, 'Attempted to join a device that did not respond')

AssociationIndicationStatus.UNSECURED_KEY_RECEIVED = (172, 'Secure join error - network security key received unsecured')

AssociationIndicationStatus.KEY_NOT_RECEIVED = (173, 'Secure join error - network security key not received')

AssociationIndicationStatus.INVALID_SECURITY_KEY = (175, 'Secure join error - joining device does not have the right preconfigured link key')

AssociationIndicationStatus.SCANNING_NETWORK = (255, 'Scanning for a network/Attempting to associate')

code

Returns the code of the *AssociationIndicationStatus* element.

Returns the code of the *AssociationIndicationStatus* element.

Return type Integer

description

Returns the description of the *AssociationIndicationStatus* element.

Returns

the description of the *AssociationIndicationStatus* element.

Return type String

class digi.xbee.models.status.**CellularAssociationIndicationStatus**(*code*, *description*)

Bases: enum.Enum

Enumerates the different association indication statuses for the Cellular protocol.

Values:

CellularAssociationIndicationStatus.SUCCESSFULLY_CONNECTED = (0, 'Connected to the Internet')

CellularAssociationIndicationStatus.REGISTERING_CELLULAR_NETWORK = (34, 'Registering to cellular network')

CellularAssociationIndicationStatus.CONNECTING_INTERNET = (35, 'Connecting to the Internet')

CellularAssociationIndicationStatus.MODEM_FIRMWARE_CORRUPT = (36, 'The cellular component requires a new firmware image')

CellularAssociationIndicationStatus.REGISTRATION_DENIED = (37, 'Cellular network registration was denied')

CellularAssociationIndicationStatus.AIRPLANE_MODE = (42, 'Airplane mode is active')

CellularAssociationIndicationStatus.USB_DIRECT = (43, 'USB Direct mode is active')

CellularAssociationIndicationStatus.PSM_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

Returns the code of the *CellularAssociationIndicationStatus* element.

Returns

the code of the *CellularAssociationIndicationStatus* element.

Return type Integer

description

Returns the description of the *CellularAssociationIndicationStatus* element.

Returns

the description of the *CellularAssociationIndicationStatus* element.

Return type String

class digi.xbee.models.status.**DeviceCloudStatus**(*code*, *description*)

Bases: `enum.Enum`

Enumerates the different Device Cloud statuses.

Values:

DeviceCloudStatus.SUCCESS = (0, 'Success')

DeviceCloudStatus.BAD_REQUEST = (1, 'Bad request')

DeviceCloudStatus.RESPONSE_UNAVAILABLE = (2, 'Response unavailable')

DeviceCloudStatus.DEVICE_CLOUD_ERROR = (3, 'Device Cloud error')

DeviceCloudStatus.CANCELED = (32, 'Device Request canceled by user')

DeviceCloudStatus.TIME_OUT = (33, 'Session timed out')

DeviceCloudStatus.UNKNOWN_ERROR = (64, 'Unknown error')

code

Returns the code of the *DeviceCloudStatus* element.

Returns the code of the *DeviceCloudStatus* element.

Return type Integer

description

Returns the description of the *DeviceCloudStatus* element.

Returns the description of the *DeviceCloudStatus* element.

Return type String

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 do not match the number in the request')

code
Returns the code of the *FrameError* element.

Returns the code of the *FrameError* element.

Return type Integer

description
Returns the description of the *FrameError* element.

Returns the description of the *FrameError* element.

Return type String

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
Returns the code of the *WiFiAssociationIndicationStatus* element.

Returns the code of the *WiFiAssociationIndicationStatus* element.

Return type Integer

description
Returns the description of the *WiFiAssociationIndicationStatus* element.

Returns the description of the *WiFiAssociationIndicationStatus* element.

Return type String

class digi.xbee.models.status.**NetworkDiscoveryStatus**(*code*, *description*)
Bases: enum.Enum

Enumerates the different statuses of the network discovery process.

Values:

NetworkDiscoveryStatus.SUCCESS = (0, 'Success')
NetworkDiscoveryStatus.ERROR_READ_TIMEOUT = (1, 'Read timeout error')
NetworkDiscoveryStatus.ERROR_NET_DISCOVER = (2, 'Error executing node discovery')
NetworkDiscoveryStatus.ERROR_GENERAL = (3, 'Error while discovering network')
NetworkDiscoveryStatus.CANCEL = (4, 'Discovery process cancelled')

code

Returns the code of the *NetworkDiscoveryStatus* element.

Returns the code of the *NetworkDiscoveryStatus* element.

Return type Integer

description

Returns the description of the *NetworkDiscoveryStatus* element.

Returns the description of the *NetworkDiscoveryStatus* element.

Return type String

class digi.xbee.models.status.**ZigbeeRegisterStatus**(*code*, *description*)

Bases: enum.Enum

Enumerates the different statuses of the Zigbee Device Register process.

Values:

ZigbeeRegisterStatus.SUCCESS = (0, 'Success')
ZigbeeRegisterStatus.KEY_TOO_LONG = (1, 'Key too long')
ZigbeeRegisterStatus.ADDRESS_NOT_FOUND = (177, 'Address not found in the key table')
ZigbeeRegisterStatus.INVALID_KEY = (178, 'Key is invalid (00 and FF are reserved)')
ZigbeeRegisterStatus.INVALID_ADDRESS = (179, 'Invalid address')
ZigbeeRegisterStatus.KEY_TABLE_FULL = (180, 'Key table is full')
ZigbeeRegisterStatus.KEY_NOT_FOUND = (255, 'Key not found')
ZigbeeRegisterStatus.UNKNOWN = (238, 'Unknown')

code

Returns the code of the *ZigbeeRegisterStatus* element.

Returns the code of the *ZigbeeRegisterStatus* element.

Return type Integer

description

Returns the description of the *ZigbeeRegisterStatus* element.

Returns the description of the *ZigbeeRegisterStatus* element.

Return type String

class digi.xbee.models.status.**EmberBootloaderMessageType**(*code*, *description*)

Bases: `enum.Enum`

Enumerates the different types of the Ember bootloader messages.

Values:

EmberBootloaderMessageType.ACK = (6, 'ACK message')
EmberBootloaderMessageType.NACK = (21, 'NACK message')
EmberBootloaderMessageType.NO_MAC_ACK = (64, 'No MAC ACK message')
EmberBootloaderMessageType.QUERY = (81, 'Query message')
EmberBootloaderMessageType.QUERY_RESPONSE = (82, 'Query response message')
EmberBootloaderMessageType.UNKNOWN = (255, 'Unknown')

code

Returns the code of the *EmberBootloaderMessageType* element.

Returns the code of the *EmberBootloaderMessageType* element.

Return type Integer

description

Returns the description of the *EmberBootloaderMessageType* element.

Returns the description of the *EmberBootloaderMessageType* element.

Return type String

class digi.xbee.models.status.**SocketStatus**(*code*, *description*)

Bases: `enum.Enum`

Enumerates the different Socket statuses.

Values:

SocketStatus.SUCCESS = (0, 'Operation successful')
SocketStatus.INVALID_PARAM = (1, 'Invalid parameters')
SocketStatus.FAILED_TO_READ = (2, 'Failed to retrieve option value')
SocketStatus.CONNECTION_IN_PROGRESS = (3, 'Connection already in progress')
SocketStatus.ALREADY_CONNECTED = (4, 'Already connected/bound/listening')
SocketStatus.UNKNOWN_ERROR = (5, 'Unknown error')
SocketStatus.BAD_SOCKET = (32, 'Bad socket ID')
SocketStatus.NOT_REGISTERED = (34, 'Not registered to cell network')
SocketStatus.INTERNAL_ERROR = (49, 'Internal error')
SocketStatus.RESOURCE_ERROR = (50, 'Resource error: retry the operation later')
SocketStatus.INVALID_PROTOCOL = (123, 'Invalid protocol')
SocketStatus.UNKNOWN = (255, 'Unknown')

code

Returns the code of the *SocketStatus* element.

Returns the code of the *SocketStatus* element.

Return type Integer

`description`
Returns the description of the *SocketStatus* element.

Returns the description of the *SocketStatus* element.

Return type String

class `digi.xbee.models.status.`**`SocketState`**(*code*, *description*)
Bases: `enum.Enum`

Enumerates the different Socket states.

Values:

SocketState.CONNECTED = (0, 'Connected')
SocketState.FAILED_DNS = (1, 'Failed DNS lookup')
SocketState.CONNECTION_REFUSED = (2, 'Connection refused')
SocketState.TRANSPORT_CLOSED = (3, 'Transport closed')
SocketState.TIMED_OUT = (4, 'Timed out')
SocketState.INTERNAL_ERROR = (5, 'Internal error')
SocketState.HOST_UNREACHABLE = (6, 'Host unreachable')
SocketState.CONNECTION_LOST = (7, 'Connection lost')
SocketState.UNKNOWN_ERROR = (8, 'Unknown error')
SocketState.UNKNOWN_SERVER = (9, 'Unknown server')
SocketState.RESOURCE_ERROR = (10, 'Resource error')
SocketState.LISTENER_CLOSED = (11, 'Listener closed')
SocketState.UNKNOWN = (255, 'Unknown')

`code`
Returns the code of the *SocketState* element.

Returns the code of the *SocketState* element.

Return type Integer

`description`
Returns the description of the *SocketState* element.

Returns the description of the *SocketState* element.

Return type String

class `digi.xbee.models.status.`**`SocketInfoState`**(*code*, *description*)
Bases: `enum.Enum`

Enumerates the different Socket info states.

Values:

SocketInfoState.ALLOCATED = (0, 'Allocated')
SocketInfoState.CONNECTING = (1, 'Connecting')
SocketInfoState.CONNECTED = (2, 'Connected')

SocketInfoState.LISTENING = (3, 'Listening')
SocketInfoState.BOUND = (4, 'Bound')
SocketInfoState.CLOSING = (5, 'Closing')
SocketInfoState.UNKNOWN = (255, 'Unknown')

`code`
Returns the code of the *SocketInfoState* element.

Returns the code of the *SocketInfoState* element.

Return type Integer

`description`
Returns the description of the *SocketInfoState* element.

Returns the description of the *SocketInfoState* element.

Return type String

`class` `digi.xbee.models.status.`**`FSCommandStatus`**(*code*, *description*)
Bases: `enum.Enum`

This class lists all the possible states of an file system command after execution.

Inherited properties:
name (String): Name (id) of the FSCommandStatus.
value (String): Value of the FSCommandStatus.

Values:
Success (0x00) = (0, 'Success')
Error (0x01) = (1, 'Error')
Invalid file system command (0x02) = (2, 'Invalid file system command')
Invalid command parameter (0x03) = (3, 'Invalid command parameter')
Access denied (0x50) = (80, 'Access denied')
File or directory already exists (0x51) = (81, 'File or directory already exists')
File or directory does not exist (0x52) = (82, 'File or directory does not exist')
Invalid file or directory name (0x53) = (83, 'Invalid file or directory name')
File operation on directory (0x54) = (84, 'File operation on directory')
Directory is not empty (0x55) = (85, 'Directory is not empty')
Attempt to read past EOF (end of file) (0x56) = (86, 'Attempt to read past EOF (end of file)')
Hardware failure (0x57) = (87, 'Hardware failure')
Volume offline / format required (0x58) = (88, 'Volume offline / format required')
Volume full (0x59) = (89, 'Volume full')
Operation timed out (0x5A) = (90, 'Operation timed out')
Busy with prior operation (0x5B) = (91, 'Busy with prior operation')
Resource failure (memory allocation failed, try again) (0x5C) = (92, 'Resource failure (memory allocation failed, try again)')

code
Returns the code of the FSCommandStatus element.

Returns Code of the FSCommandStatus element.

Return type Integer

description
Returns the description of the FSCommandStatus element.

Returns Description of the FSCommandStatus element.

Return type String

digi.xbee.models.zdo package

class `digi.xbee.models.zdo.`**`NodeDescriptorReader`**(*xbee*, *configure_ao=True*, *timeout=20*)
Bases: `digi.xbee.models.zdo._ZDOCommand`

This class performs a node descriptor read of the given XBee using a ZDO command.

The node descriptor read works only with Zigbee devices in API mode.

Class constructor. Instantiates a new `NodeDescriptorReader` object with the provided parameters.

Parameters

- **(class** (*xbee*) – *.XBeeDevice* or class:*.RemoteXBeeDevice*): XBee to send the command.
- **configure_ao** (*Boolean, optional, default=`True`*) – *True* to set AO value before and after executing, *False* otherwise.
- **timeout** (*Float, optional, default=`.__DEFAULT_TIMEOUT`*) – The ZDO command timeout in seconds.

Raises

- `ValueError` – If *xbee* is *None*.
- `ValueError` – If *cluster_id*, *receive_cluster_id*, or *timeout* are less than 0.
- `TypeError` – If the *xbee* is not a *.XBeeDevice* or a *RemoteXBeeDevice*.

get_node_descriptor()
Returns the descriptor of the node.

Returns The node descriptor.

Return type `NodeDescriptor`

error
Returns the error string if any.

Returns The error string.

Return type String

running
Returns if this ZDO command is running.

Returns *True* if it is running, *False* otherwise.

Return type Boolean

stop()

Stops the ZDO command process if it is running.

class digi.xbee.models.zdo.**NodeDescriptor**(*role*, *complex_desc_supported*, *user_desc_supported*, *freq_band*, *mac_capabilities*, *manufacturer_code*, *max_buffer_size*, *max_in_transfer_size*, *max_out_transfer_size*, *desc_capabilities*)

Bases: object

This class represents a node descriptor of an XBee.

Class constructor. Instantiates a new *NodeDescriptor* object with the provided parameters.

Parameters

- **role** (*Role*) – The device role.
- **complex_desc_supported** (*Boolean*) – *True* if the complex descriptor is supported.
- **user_desc_supported** (*Boolean*) – *True* if the user descriptor is supported.
- **freq_band** (*List*) – Byte array with the frequency bands.
- **mac_capabilities** (*List*) – Byte array with MAC capabilities.
- **manufacturer_code** (*Integer*) – The manufacturer's code assigned by the Zigbee Alliance.
- **max_buffer_size** (*Integer*) – Maximum size in bytes of a data transmission.
- **max_in_transfer_size** (*Integer*) – Maximum number of bytes that can be received by the node.
- **max_out_transfer_size** (*Integer*) – Maximum number of bytes that can be transmitted by the node.
- **desc_capabilities** (*List*) – Byte array with descriptor capabilities.

role

Gets the role in this node descriptor.

Returns The role of the node descriptor.

Return type *Role*

See also:

Role

complex_desc_supported

Gets if the complex descriptor is supported.

Returns *True* if supported, *False* otherwise.

Return type Boolean

user_desc_supported

Gets if the user descriptor is supported.

Returns *True* if supported, *False* otherwise.

Return type Boolean

freq_band
868 MHz * Bit1: Reserved * Bit2: 900 MHz * Bit3: 2.4 GHz * Bit4: Reserved

Returns List of integers with the frequency bands bits.

Return type List

Type Gets the frequency bands (LSB - bit0- index 0, MSB - bit4 - index 4)

Type

- Bit0

mac_capabilities
Alternate PAN coordinator * Bit1: Device Type * Bit2: Power source * Bit3: Receiver on when idle * Bit4-5: Reserved * Bit6: Security capability * Bit7: Allocate address

Returns List of integers with MAC capabilities bits.

Return type List

Type Gets the MAC capabilities (LSB - bit0- index 0, MSB - bit7 - index 7)

Type

- Bit0

manufacturer_code
Gets the manufacturer's code assigned by the Zigbee Alliance.

Returns The manufacturer's code.

Return type Integer

max_buffer_size
Gets the maximum size in bytes of a data transmission (including APS bytes).

Returns Maximum size in bytes.

Return type Integer

max_in_transfer_size
Gets the maximum number of bytes that can be received by the node.

Returns Maximum number of bytes that can be received by the node.

Return type Integer

max_out_transfer_size
Gets the maximum number of bytes that can be transmitted by the node, including fragmentation.

Returns Maximum number of bytes that can be transmitted by the node.

Return type Integer

desc_capabilities
Extended active endpoint list available * Bit1: Extended simple descriptor list available

Returns List of integers with descriptor capabilities bits.

Return type List

Type Gets the descriptor capabilities (LSB - bit0- index 0, MSB - bit1 - index 1)

Type

- Bit0

class digi.xbee.models.zdo.**RouteTableReader**(*xbee*, *configure_ao=True*, *timeout=20*)

Bases: `digi.xbee.models.zdo._ZDOCommand`

This class performs a route table read of the given XBee using a ZDO command.

The node descriptor read works only with Zigbee devices in API mode.

Class constructor. Instantiates a new `RouteTableReader` object with the provided parameters.

Parameters

- **(class** (*xbee*) – *.XBeeDevice* or class:*.RemoteXBeeDevice*): XBee to send the command.
- **configure_ao** (*Boolean, optional, default=`True`*) – *True* to set AO value before and after executing, *False* otherwise.
- **timeout** (*Float, optional, default=`.DEFAULT_TIMEOUT`*) – The ZDO command timeout in seconds.

Raises

- `ValueError` – If *xbee* is *None*.
- `ValueError` – If *cluster_id*, *receive_cluster_id*, or *timeout* are less than 0.
- `TypeError` – If the *xbee* is not a *.XBeeDevice* or a *.RemoteXBeeDevice*.

get_route_table(*route_cb=None*, *finished_cb=None*)

Returns the routes of the XBee. If *route_cb* is not defined, the process blocks until the complete routing table is read.

Parameters

- **route_cb** (*Function, optional, default=`None`*) – Method called when a new route is received. Receives two arguments:
  - The XBee that owns this new route.
  - The new route.
- **finished_cb** (*Function, optional, default=`None`*) – Method to execute when the process finishes. Receives three arguments:
  - The XBee that executed the ZDO command.
  - A list with the discovered routes.
  - An error message if something went wrong.

Returns

List of `Route` when *route_cb* is not defined, *None* otherwise (in this case routes are received in the callback).

Return type List

See also:

`Route`

error

Returns the error string if any.

Returns The error string.

Return type String

running
Returns if this ZDO command is running.

Returns *True* if it is running, *False* otherwise.

Return type Boolean

stop()
Stops the ZDO command process if it is running.

class digi.xbee.models.zdo.**RouteStatus**(*identifier*, *name*)
Bases: enum.Enum

Enumerates the available route status.

id
Returns the identifier of the RouteStatus.

Returns RouteStatus identifier.

Return type Integer

class digi.xbee.models.zdo.**Route**(*destination*, *next_hop*, *status*, *is_low_memory*, *is_many_to_one*, *is_route_record_required*)
Bases: object

This class represents a Zigbee route read from the route table of an XBee.

Class constructor. Instantiates a new *Route* object with the provided parameters.

Parameters

- **destination** (*XBee16BitAddress*) – 16-bit destination address of the route.
- **next_hop** (*XBee16BitAddress*) – 16-bit address of the next hop.
- **status** (*RouteStatus*) – Status of the route.
- **is_low_memory** (*Boolean*) – *True* to indicate if the device is a low-memory concentrator.
- **is_many_to_one** (*Boolean*) – *True* to indicate the destination is a concentrator.
- **is_route_record_required** (*Boolean*) – *True* to indicate a route record message should be sent prior to the next data transmission.

See also:

RouteStatus
XBee16BitAddress

destination
Gets the 16-bit address of this route destination.

Returns 16-bit address of the destination.

Return type *XBee16BitAddress*

See also:

XBee16BitAddress

next_hop

Gets the 16-bit address of this route next hop.

Returns 16-bit address of the next hop.

Return type *XBee16BitAddress*

See also:

XBee16BitAddress

status

Gets this route status.

Returns The route status.

Return type *RouteStatus*

See also:

RouteStatus

is_low_memory

Gets whether the device is a low-memory concentrator.

Returns *True* if the device is a low-memory concentrator, *False* otherwise.

Return type Boolean

is_many_to_one

Gets whether the destination is a concentrator.

Returns *True* if destination is a concentrator, *False* otherwise.

Return type Boolean

is_route_record_required

Gets whether a route record message should be sent prior the next data transmission.

Returns *True* if a route record message should be sent, *False* otherwise.

Return type Boolean

class digi.xbee.models.zdo.**NeighborTableReader**(*xbee*, *configure_ao=True*, *timeout=20*)

Bases: digi.xbee.models.zdo._ZDOCommand

This class performs a neighbor table read of the given XBee using a ZDO command.

The node descriptor read works only with Zigbee devices in API mode.

Class constructor. Instantiates a new *NeighborTableReader* object with the provided parameters.

Parameters

- **(class** (*xbee*) – *.XBeeDevice* or class:*.RemoteXBeeDevice*): XBee to send the command.

- **configure_ao** (*Boolean, optional, default=`True`*) – *True* to set AO value before and after executing, *False* otherwise.
- **timeout** (*Float, optional, default=`.DEFAULT_TIMEOUT`*) – The ZDO command timeout in seconds.

Raises

- ValueError – If *xbee* is *None*.
- ValueError – If *cluster_id*, *receive_cluster_id*, or *timeout* are less than 0.
- TypeError – If the *xbee* is not a *.XBeeDevice* or a *.RemoteXBeeDevice*.

get_neighbor_table(*neighbor_cb=None, finished_cb=None*)

Returns the neighbors of the XBee. If *neighbor_cb* is not defined, the process blocks until the complete neighbor table is read.

Parameters

- **neighbor_cb** (*Function, optional, default=`None`*) – Method called when a new neighbor is received. Receives two arguments:
  - The XBee that owns this new neighbor.
  - The new neighbor.
- **finished_cb** (*Function, optional, default=`None`*) – Method to execute when the process finishes. Receives three arguments:
  - The XBee that executed the ZDO command.
  - A list with the discovered neighbors.
  - An error message if something went wrong.

Returns

List of `Neighbor` when *neighbor_cb* is not defined, *None* otherwise (in this case neighbors are received in the callback)

Return type List

See also:

Neighbor

error

Returns the error string if any.

Returns The error string.

Return type String

running

Returns if this ZDO command is running.

Returns *True* if it is running, *False* otherwise.

Return type Boolean

stop()

Stops the ZDO command process if it is running.

class digi.xbee.models.zdo.**NeighborRelationship**(*identifier*, *name*)
Bases: enum.Enum

Enumerates the available relationships between two nodes of the same network.

id
Returns the identifier of the NeighborRelationship.

Returns NeighborRelationship identifier.

Return type Integer

class digi.xbee.models.zdo.**Neighbor**(*node*, *relationship*, *depth*, *lq*)
Bases: object

This class represents a Zigbee or DigiMesh neighbor.

This information is read from the neighbor table of a Zigbee XBee, or provided by the 'FN' command in a Digimesh XBee.

Class constructor. Instantiates a new *Neighbor* object with the provided parameters.

Parameters

- **node** (*RemoteXBeeDevice*) – The neighbor node.
- **relationship** (*NeighborRelationship*) – The relationship of this neighbor with the node.
- **depth** (*Integer*) – The tree depth of the neighbor. A value of 0 indicates the device is a Zigbee coordinator for the network. -1 means this is unknown.
- **lq** (*Integer*) – The estimated link quality (LQI or RSSI) of data transmission from this neighbor.

See also:

NeighborRelationship
RemoteXBeeDevice

node
Gets the neighbor node.

Returns The node itself.

Return type *RemoteXBeeDevice*

See also:

RemoteXBeeDevice

relationship
Gets the neighbor node.

Returns The neighbor relationship.

Return type *NeighborRelationship*

See also:

NeighborRelationship

depth
Gets the tree depth of the neighbor.

Returns The tree depth of the neighbor.

Return type Integer

lq
Gets the estimated link quality (LQI or RSSI) of data transmission from this neighbor.

Returns The estimated link quality of data transmission from this neighbor.

Return type Integer

class digi.xbee.models.zdo.**NeighborFinder**(*xbee*, *timeout=20*)
Bases: object

This class performs a find neighbors (FN) of an XBee. This action requires an XBee and optionally a find timeout.

The process works only in DigiMesh.

Class constructor. Instantiates a new *NeighborFinder* object with the provided parameters.

Parameters

- **(class** (*xbee*) – *.XBeeDevice* or class:*.RemoteXBeeDevice*): The XBee to get neighbors from.
- **timeout** (*Float*) – The timeout for the process in seconds.

Raises

- OperationNotSupportedException – If the process is not supported in the XBee.
- TypeError – If the *xbee* is not a *.AbstractXBeeDevice*.
- ValueError – If *xbee* is *None*.
- ValueError – If *timeout* is less than 0.

running
Returns whether this find neighbors process is running.

Returns *True* if it is running, *False* otherwise.

Return type Boolean

error
Returns the error string if any.

Returns The error string.

Return type String

stop()
Stops the find neighbors process if it is running.

get_neighbors(*neighbor_cb=None*, *finished_cb=None*)
Returns the neighbors of the XBee. If *neighbor_cb* is not defined, the process blocks until the complete neighbor table is read.

Parameters

- **neighbor_cb** (*Function, optional, default=`None`*) – Method called when a new neighbor is received. Receives two arguments:
  - The XBee that owns this new neighbor.
  - The new neighbor.
- **finished_cb** (*Function, optional, default=`None`*) – Method to execute when the process finishes. Receives three arguments:
  - The XBee that executed the FN command.
  - A list with the discovered neighbors.
  - An error message if something went wrong.

Returns

List of `Neighbor` when *neighbor_cb* is not defined, *None* otherwise (in this case neighbors are received in the callback)

Return type List

See also:

Neighbor

digi.xbee.packets package

Submodules

digi.xbee.packets.aft module

class digi.xbee.packets.aft.**ApiFrameType**(*code*, *description*)

Bases: `enum.Enum`

This enumeration lists all the available frame types used in any XBee protocol.

Inherited properties:

name (String): the name (id) of this ApiFrameType.

value (String): the value of this ApiFrameType.

Values:

ApiFrameType.TX_64 = (0, 'TX (Transmit) Request 64-bit address')
ApiFrameType.TX_16 = (1, 'TX (Transmit) Request 16-bit address')
ApiFrameType.REMOTE_AT_COMMAND_REQUEST_WIFI = (7, 'Remote AT Command Request (Wi-Fi)')
ApiFrameType.AT_COMMAND = (8, 'AT Command')
ApiFrameType.AT_COMMAND_QUEUE = (9, 'AT Command Queue')
ApiFrameType.TRANSMIT_REQUEST = (16, 'Transmit Request')
ApiFrameType.EXPLICIT_ADDRESSING = (17, 'Explicit Addressing Command Frame')
ApiFrameType.REMOTE_AT_COMMAND_REQUEST = (23, 'Remote AT Command Request')
ApiFrameType.TX_SMS = (31, 'TX SMS')

ApiFrameType.TX_IPV4 = (32, 'TX IPv4')

ApiFrameType.CREATE_SOURCE_ROUTE = (33, 'Create Source Route')

ApiFrameType.REGISTER_JOINING_DEVICE = (36, 'Register Joining Device')

ApiFrameType.SEND_DATA_REQUEST = (40, 'Send Data Request')

ApiFrameType.DEVICE_RESPONSE = (42, 'Device Response')

ApiFrameType.USER_DATA_RELAY_REQUEST = (45, 'User Data Relay Request')

ApiFrameType.FILE_SYSTEM_REQUEST = (59, 'File System Request')

ApiFrameType.REMOTE_FILE_SYSTEM_REQUEST = (60, 'Remote File System Request')

ApiFrameType.SOCKET_CREATE = (64, 'Socket Create')

ApiFrameType.SOCKET_OPTION_REQUEST = (65, 'Socket Option Request')

ApiFrameType.SOCKET_CONNECT = (66, 'Socket Connect')

ApiFrameType.SOCKET_CLOSE = (67, 'Socket Close')

ApiFrameType.SOCKET_SEND = (68, 'Socket Send (Transmit)')

ApiFrameType.SOCKET_SENDTO = (69, 'Socket SendTo (Transmit Explicit Data): IPv4')

ApiFrameType.SOCKET_BIND = (70, 'Socket Bind/Listen')

ApiFrameType.RX_64 = (128, 'RX (Receive) Packet 64-bit Address')

ApiFrameType.RX_16 = (129, 'RX (Receive) Packet 16-bit Address')

ApiFrameType.RX_IO_64 = (130, 'IO Data Sample RX 64-bit Address Indicator')

ApiFrameType.RX_IO_16 = (131, 'IO Data Sample RX 16-bit Address Indicator')

ApiFrameType.REMOTE_AT_COMMAND_RESPONSE_WIFI = (135, 'Remote AT Command Response (Wi-Fi)')

ApiFrameType.AT_COMMAND_RESPONSE = (136, 'AT Command Response')

ApiFrameType.TX_STATUS = (137, 'TX (Transmit) Status')

ApiFrameType.MODEM_STATUS = (138, 'Modem Status')

ApiFrameType.TRANSMIT_STATUS = (139, 'Transmit Status')

ApiFrameType.DIGIMESH_ROUTE_INFORMATION = (141, 'Route Information')

ApiFrameType.IO_DATA_SAMPLE_RX_INDICATOR_WIFI = (143, 'IO Data Sample RX Indicator (Wi-Fi)')

ApiFrameType.RECEIVE_PACKET = (144, 'Receive Packet')

ApiFrameType.EXPLICIT_RX_INDICATOR = (145, 'Explicit RX Indicator')

ApiFrameType.IO_DATA_SAMPLE_RX_INDICATOR = (146, 'IO Data Sample RX Indicator')

ApiFrameType.REMOTE_AT_COMMAND_RESPONSE = (151, 'Remote Command Response')

ApiFrameType.RX_SMS = (159, 'RX SMS')

ApiFrameType.OTA_FIRMWARE_UPDATE_STATUS = (160, 'OTA Firmware Update Status')

ApiFrameType.ROUTE_RECORD_INDICATOR = (161, 'Route Record Indicator')

ApiFrameType.REGISTER_JOINING_DEVICE_STATUS = (164, 'Register Joining Device Status')

ApiFrameType.USER_DATA_RELAY_OUTPUT = (173, 'User Data Relay Output')

ApiFrameType.RX_IPV4 = (176, 'RX IPv4')

ApiFrameType.SEND_DATA_RESPONSE = (184, 'Send Data Response')

ApiFrameType.DEVICE_REQUEST = (185, 'Device Request')

ApiFrameType.DEVICE_RESPONSE_STATUS = (186, 'Device Response Status')

ApiFrameType.FILE_SYSTEM_RESPONSE = (187, 'File System Response')

ApiFrameType.REMOTE_FILE_SYSTEM_RESPONSE = (188, 'Remote File System Response')

ApiFrameType.SOCKET_CREATE_RESPONSE = (192, 'Socket Create Response')

ApiFrameType.SOCKET_OPTION_RESPONSE = (193, 'Socket Option Response')

ApiFrameType.SOCKET_CONNECT_RESPONSE = (194, 'Socket Connect Response')

ApiFrameType.SOCKET_CLOSE_RESPONSE = (195, 'Socket Close Response')

ApiFrameType.SOCKET_LISTEN_RESPONSE = (198, 'Socket Listen Response')
ApiFrameType.SOCKET_NEW_IPV4_CLIENT = (204, 'Socket New IPv4 Client')
ApiFrameType.SOCKET_RECEIVE = (205, 'Socket Receive')
ApiFrameType.SOCKET_RECEIVE_FROM = (206, 'Socket Receive From')
ApiFrameType.SOCKET_STATE = (207, 'Socket State')
ApiFrameType.FRAME_ERROR = (254, 'Frame Error')
ApiFrameType.GENERIC = (255, 'Generic')
ApiFrameType.UNKNOWN = (-1, 'Unknown Packet')

code
Returns the code of the ApiFrameType element.

Returns the code of the ApiFrameType element.

Return type Integer

description
Returns the description of the ApiFrameType element.

Returns the description of the ApiFrameType element.

Return type Integer

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**(*op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)
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.

Parameters **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

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.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

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

frame_id
Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

needs_id()
Returns whether the packet requires frame ID or not.

Returns *True* if the packet needs frame ID, *False* otherwise.

Return type Boolean

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_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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type `OperatingMode`

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 must be escaped.

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 info of the XBeePacket fields.

Return type Dictionary

static unescape_data(*data*)
Un-escapes the provided bytearray data.

Parameters **data** (*Bytearray*) – the bytearray to unescape.

Returns *data* unescaped.

Return type Bytearray

class digi.xbee.packets.base.**GenericXBeePacket**(*data*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)
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

- **data** (*bytearray*) – the frame specific data without frame type and frame ID.
- **op_mode** (`OperatingMode`, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

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 from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is different from `ApiFrameType.GENERIC`.
- `InvalidOperatingModeException` – if *operating_mode* is not supported.

See also:

XBeePacket.create_packet()

`XBeeAPIPacket._check_api_packet()`

needs_id()

Override method.

See also:

XBeeAPIPacket.needs_id()

frame_id

Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

get_checksum()

Returns the checksum value of this XBeePacket. The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer

See also:

factory

get_frame_spec_data()

Override method.

See also:

XBeePacket.get_frame_spec_data()

get_frame_type()
Returns the frame type of this packet.

Returns the frame type of this packet.

Return type *ApiFrameType*

See also:

ApiFrameType

get_frame_type_value()
Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer

See also:

ApiFrameType

is_broadcast()
Returns whether this packet is broadcast or not.

Returns *True* if this packet is broadcast, *False* otherwise.

Return type Boolean

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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*, *data*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents an unknown XBee packet.

See also:

XBeeAPIPacket

Class constructor. Instantiates a *UnknownXBeePacket* object with the provided parameters.

Parameters

- **api_frame** (*Integer*) – the API frame integer value of this packet.
- **data** (*bytearray*) – the frame specific data without frame type and frame ID.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

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 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 from the checksum field value (last byte).
- InvalidOperatingModeException – if *operating_mode* is not supported.

See also:

XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()

frame_id
Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

get_checksum()
Returns the checksum value of this XBeePacket. The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer

See also:

factory

get_frame_spec_data()
Override method.

See also:

XBeePacket.get_frame_spec_data()

get_frame_type()
Returns the frame type of this packet.

Returns the frame type of this packet.

Return type *ApiFrameType*

See also:

ApiFrameType

get_frame_type_value()
Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer

See also:

ApiFrameType

is_broadcast()
Returns whether this packet is broadcast or not.

Returns *True* if this packet is broadcast, *False* otherwise.

Return type Boolean

`op_mode`
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *`OperatingMode`*

`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 must be escaped.

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 info of the XBeePacket fields.

Return type Dictionary

`static unescape_data`(*data*)
Un-escapes the provided bytearray data.

Parameters **`data`** (*`Bytearray`*) – the bytearray to unescape.

Returns *data* unescaped.

Return type Bytearray

`needs_id`()
Override method.

See also:

`XBeeAPIPacket.needs_id()`

digi.xbee.packets.cellular module

`digi.xbee.packets.cellular.`**`PATTERN_PHONE_NUMBER = '^\\+?\\d+$'`**
Pattern used to validate the phone number parameter of SMS packets.

`class` `digi.xbee.packets.cellular.`**`RXSMSPacket`**(*phone_number*, *data*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)
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 or bytearray*) – Packet data (text of the SMS).
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

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 from 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 from the checksum field value (last byte).
- InvalidPacketException – if the frame type is different than ApiFrameType.RX_SMS.
- InvalidOperatingModeException – if *operating_mode* is not supported.

See also:

XBeePacket.create_packet()

needs_id()

Override method.

See also:

XBeeAPIPacket.needs_id()

get_phone_number_byte_array()

Returns the phone number byte array.

Returns phone number of the device that sent the SMS.

Return type Bytearray

phone_number
Returns the phone number of the device that sent the SMS.

Returns phone number of the device that sent the SMS.

Return type String

data
Returns the data of the packet (SMS text).

Returns the data of the packet.

Return type String

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)
Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents a TX (Transmit) SMS packet. Packet is built using the parameters of the constructor or providing a valid byte array.

See also:

RXSMSPacket
XBeeAPIPacket

Class constructor. Instantiates a new *TXSMSPacket* object with the provided parameters.

Parameters

- **frame_id** (*Integer*) – the frame ID. Must be between 0 and 255.
- **phone_number** (*String*) – the phone number.
- **data** (*String or bytearray*) – this packet's data.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

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 from 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 from the checksum field value (last byte).
- InvalidPacketException – if the frame type is different than ApiFrameType.TX_SMS.
- InvalidOperatingModeException – if *operating_mode* is not supported.

See also:

XBeePacket.create_packet()

needs_id()
Override method.

See also:

XBeeAPIPacket.needs_id()

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

get_phone_number_byte_array()
Returns the phone number byte array.

Returns phone number of the device that sent the SMS.

Return type Bytearray

is_broadcast()
Returns whether this packet is broadcast or not.

Returns *True* if this packet is broadcast, *False* otherwise.

Return type Boolean

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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
Returns the phone number of the transmitter device.

Returns the phone number of the transmitter device.

Return type String

data
Returns the data of the packet (SMS text).

Returns packet's data.

Return type Bytearray

digi.xbee.packets.common module

class digi.xbee.packets.common.**ATCommPacket**(*frame_id*, *command*, *parameter=None*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)
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 or bytearray`*) – AT command of the packet.
- **`parameter`** (*`Bytearray, optional`*) – the AT command parameter.
- **`op_mode`** (`OperatingMode`, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

Raises

- `ValueError` – if *frame_id* is less than 0 or greater than 255.
- `ValueError` – if length of *command* is different from 2.

See also:

`XBeeAPIPacket`

`static create_packet`(*raw*, *operating_mode*)

Override method.

Returns `ATCommPacket`

Raises

- `InvalidPacketException` – if the bytearray length is less than 8. (start delim. + length (2 bytes) + frame type + frame id + command (2 bytes) + checksum = 8 bytes).
- `InvalidPacketException` – if the length field of 'raw' is different from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is different from `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
Returns the AT command of the packet.

Returns the AT command of the packet.

Return type String

parameter
Returns the parameter of the packet.

Returns the parameter of the packet.

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

`op_mode`
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type `OperatingMode`

`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 must be escaped.

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 info 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*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

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 or bytearray`*) – the AT command of the packet.
- **`parameter`** (*`Bytearray, optional`*) – the AT command parameter. Optional.
- **`op_mode`** (*`OperatingMode`*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

Raises

- `ValueError` – if *frame_id* is less than 0 or greater than 255.
- `ValueError` – if length of *command* is different from 2.

See also:

`XBeeAPIPacket`

`static create_packet`(*raw*, *operating_mode*)
Override method.

Returns `ATCommQueuePacket`

Raises

- `InvalidPacketException` – if the bytearray length is less than 8. (start delim. + length (2 bytes) + frame type + frame id + command + checksum = 8 bytes).
- `InvalidPacketException` – if the length field of 'raw' is different from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is different from `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
Returns the AT command of the packet.

Returns the AT command of the packet.

Return type String

parameter
Returns the parameter of the packet.

Returns the parameter of the packet.

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

op_mode

Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

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 or bytearray*) – the AT command of the packet.
- **response_status** (*ATCommandStatus* or Integer) – the status of the AT command.
- **comm_value** (*Bytearray, optional*) – the AT command response value.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

Raises

- ValueError – if *frame_id* is less than 0 or greater than 255.
- ValueError – if length of *command* is different from 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 from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is different from `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
Returns the AT command of the packet.

Returns the AT command of the packet.

Return type String

command_value
Returns the AT command response value.

Returns the AT command response value.

Return type Bytearray

real_status
Returns the AT command response status of the packet.

Returns the AT command response status of the packet.

Return type Integer

status
Returns the AT command response status of the packet.

Returns the AT command response status of the packet.

Return type *ATCommandStatus*

See also:

ATCommandStatus

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

op_mode

Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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*, *rx_options*, *rf_data=None*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

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.
- **rx_options** (*Integer*) – bitfield indicating the receive options.
- **rf_data** (*Bytearray, optional*) – received RF data.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

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 + 64bit addr. + 16bit addr. + Receive options + checksum = 16 bytes).
- InvalidPacketException – if the length field of 'raw' is different from 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 from the checksum field value (last byte).
- InvalidPacketException – if the frame type is not ApiFrameType.RECEIVE_PACKET.
- InvalidOperatingModeException – if *operating_mode* is not supported.

See also:

XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()

needs_id()
Override method.

See also:

XBeeAPIPacket.needs_id()

is_broadcast()
Override method.

See also:

XBeeAPIPacket.is_broadcast()

x64bit_source_addr
Returns the 64-bit source address.

Returns the 64-bit source address.

Return type *XBee64BitAddress*

See also:

XBee64BitAddress

x16bit_source_addr
Returns the 16-bit source address.

Returns the 16-bit source address.

Return type *XBee16BitAddress*

See also:

XBee16BitAddress

receive_options
Returns the receive options bitfield.

Returns the receive options bitfield.

Return type Integer

See also:

ReceiveOptions

rf_data
Returns the received RF data.

Returns the received RF data.

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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*, *tx_options*, *command*, *parameter=None*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

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.
- **tx_options** (*Integer*) – bitfield of supported transmission options.
- **command** (*String or bytearray*) – AT command to send.
- **parameter** (*Bytearray, optional*) – AT command parameter.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

Raises

- `ValueError` – if *frame_id* is less than 0 or greater than 255.
- `ValueError` – if length of *command* is different from 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 from 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 from 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

Returns the 64-bit destination address.

Returns the 64-bit destination address.

Return type *XBee64BitAddress*

See also:

XBee64BitAddress

x16bit_dest_addr

Returns the 16-bit destination address.

Returns the 16-bit destination address.

Return type *XBee16BitAddress*

See also:

XBee16BitAddress

transmit_options

Returns the transmit options bitfield.

Returns the transmit options bitfield.

Return type Integer

See also:

RemoteATCmdOptions

parameter

Returns the AT command parameter.

Returns the AT command parameter.

Return type Bytearray

command

Returns the AT command.

Returns the AT command.

Return type String

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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*, *resp_status*, *comm_value=None*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

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 or bytearray*) – the AT command of the packet.
- **resp_status** (*ATCommandStatus* or Integer) – the status of the AT command.
- **comm_value** (*Bytearray, optional*) – the AT command response value. Optional.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

Raises

- `ValueError` – if *frame_id* is less than 0 or greater than 255.
- `ValueError` – if length of *command* is different from 2.

See also:

ATCommandStatus
XBee16BitAddress
XBee64BitAddress
XBeeAPIPacket

static create_packet(*raw*, *operating_mode*)
Override method.

Returns *RemoteATCommandResponsePacket*.

Raises

- `InvalidPacketException` – if the bytearray length is less than 19. (start delim. + length (2 bytes) + frame type + frame id + 64bit addr. + 16bit addr. + receive options + command (2 bytes) + checksum = 19 bytes).
- `InvalidPacketException` – if the length field of 'raw' is different from 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 from 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()

command
Returns the AT command of the packet.

Returns the AT command of the packet.

Return type String

command_value
Returns the AT command response value.

Returns the AT command response value.

Return type Bytearray

real_status
Returns the AT command response status of the packet.

Returns the AT command response status of the packet.

Return type Integer

status
Returns the AT command response status of the packet.

Returns the AT command response status of the packet.

Return type *ATCommandStatus*

See also:

ATCommandStatus

x64bit_source_addr
Returns the 64-bit source address.

Returns the 64-bit source address.

Return type *XBee64BitAddress*

See also:

XBee64BitAddress

x16bit_source_addr
Returns the 16-bit source address.

Returns the 16-bit source address.

Return type *XBee16BitAddress*

See also:

XBee16BitAddress

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

op_mode

Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type `OperatingMode`

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 must be escaped.

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 info 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*, *tx_options*, *rf_data=None*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

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 *0x0000000000000000* 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`

XBee16BitAddress.COORDINATOR_ADDRESS
XBee16BitAddress.UNKNOWN_ADDRESS
XBee64BitAddress.BROADCAST_ADDRESS
XBee64BitAddress.COORDINATOR_ADDRESS
XBeeAPIPacket

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.
- **tx_options** (*Integer*) – bitfield of supported transmission options.
- **rf_data** (*Bytearray, optional*) – RF data that is sent to the destination device.
- **op_mode** (*OperatingMode*, optional, default=‘OperatingMode.API_MODE‘) – The mode in which the frame was captured.

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. + broadcast radius + Transmit options + checksum = 18 bytes).
- `InvalidPacketException` – if the length field of ‘raw’ is different from 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 from 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()

rf_data
Returns the RF data to send.

Returns the RF data to send.

Return type Bytearray

transmit_options
Returns the transmit options bitfield.

Returns the transmit options bitfield.

Return type Integer

See also:

TransmitOptions

broadcast_radius
Returns the broadcast radius. Broadcast radius is the maximum number of hops a broadcast transmission.

Returns the broadcast radius.

Return type Integer

x64bit_dest_addr
Returns the 64-bit destination address.

Returns the 64-bit destination address.

Return type *XBee64BitAddress*

See also:

XBee64BitAddress

x16bit_dest_addr
Returns the 16-bit destination address.

Returns the 16-bit destination address.

Return type `XBee16BitAddress`

See also:

`XBee16BitAddress`

`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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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.**TransmitStatusPacket**(*frame_id, x16bit_addr, tx_retry_count, transmit_status=<TransmitStatus.SUCCESS: (0, 'Success')>, discovery_status=<DiscoveryStatus.NO_DISCOVERY_OVERH (0, 'No discovery overhead')>, op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

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.
- **tx_retry_count** (*Integer*) – the number of application transmission retries that took place.
- **transmit_status** (*TransmitStatus*, optional) – transmit status. Default: SUCCESS.
- **discovery_status** (DiscoveryStatus, optional) – discovery status. Default: NO_DISCOVERY_OVERHEAD.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

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 from 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 from the checksum field value (last byte).
- InvalidPacketException – if the frame type is not ApiFrameType.TRANSMIT_STATUS.
- InvalidOperatingModeException – if *operating_mode* is not supported.

See also:

XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()

`needs_id``()`
Override method.

See also:

`XBeeAPIPacket.needs_id()`

`x16bit_dest_addr`
Returns the 16-bit destination address.

Returns the 16-bit destination address.

Return type *`XBee16BitAddress`*

See also:

`XBee16BitAddress`

`transmit_status`
Returns the transmit status.

Returns the transmit status.

Return type *`TransmitStatus`*

See also:

`TransmitStatus`

`transmit_retry_count`
Returns the transmit retry count.

Returns the transmit retry count.

Return type Integer

`discovery_status`
Returns the discovery status.

Returns the discovery status.

Return type *`DiscoveryStatus`*

See also:

`DiscoveryStatus`

`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

op_mode

Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type `OperatingMode`

`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 must be escaped.

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 info 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*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)
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.
- **`op_mode`** (`OperatingMode`, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

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 from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is not `ApiFrameType.MODEM_STATUS`.
- `InvalidOperatingModeException` – if *operating_mode* is not supported.

See also:

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

`needs_id``()`
Override method.

See also:

`XBeeAPIPacket.needs_id()`

`modem_status`
Returns the modem status event.

Returns The modem status event.

Return type *`ModemStatus`*

See also:

`ModemStatus`

`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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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*, *rx_options*, *rf_data=None*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

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.
- **rx_options** (*Integer*) – bitfield indicating the receive options.
- **rf_data** (*Bytearray, optional*) – received RF data.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

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 from 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 from 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
Returns the 64-bit source address.

Returns the 64-bit source address.

Return type *XBee64BitAddress*

See also:

XBee64BitAddress

x16bit_source_addr

Returns the 16-bit source address.

Returns the 16-bit source address.

Return type *XBee16BitAddress*

See also:

XBee16BitAddress

receive_options

Returns the receive options bitfield.

Returns the receive options bitfield.

Return type Integer

See also:

ReceiveOptions

rf_data

Returns the received RF data.

Returns the received RF data.

Return type Bytearray

io_sample

Returns the IO sample corresponding to the data contained in the packet.

Returns

the IO sample of the packet, *None* if the packet has not any data or if the sample could not be generated correctly.

Return type *IOSample*

See also:

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

`op_mode`
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

`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 must be escaped.

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 info 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*, *src_endpoint*, *dest_endpoint*, *cluster_id*, *profile_id*, *broadcast_radius=0*, *transmit_options=0*, *rf_data=None*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

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 *0x000000000000FFF* for a broadcast transmission (to all devices).

The coordinator can be addressed by either setting the 64-bit address to *0x000000000000000* 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 right 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.
- **src_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.
- **op_mode** (*OperatingMode*, optional, default=‘OperatingMode.API_MODE‘) – The mode in which the frame was captured.

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 from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is different from `ApiFrameType.EXPLICIT_ADDRESSING`.
- `InvalidOperatingModeException` – if *operating_mode* is not supported.

See also:

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

`frame_id`
Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

`get_checksum``()`
Returns the checksum value of this XBeePacket. The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer

See also:

`factory`

`get_frame_spec_data``()`
Override method.

See also:

`XBeePacket.get_frame_spec_data()`

`get_frame_type``()`
Returns the frame type of this packet.

Returns the frame type of this packet.

Return type *ApiFrameType*

See also:

ApiFrameType

get_frame_type_value()

Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer

See also:

ApiFrameType

is_broadcast()

Returns whether this packet is broadcast or not.

Returns *True* if this packet is broadcast, *False* otherwise.

Return type Boolean

needs_id()

Override method.

See also:

XBeeAPIPacket.needs_id()

op_mode

Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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_endpoint
Returns the source endpoint of the transmission.

Returns the source endpoint of the transmission.

Return type Integer

dest_endpoint
Returns the destination endpoint of the transmission.

Returns the destination endpoint of the transmission.

Return type Integer

cluster_id
Returns the cluster ID of the transmission.

Returns the cluster ID of the transmission.

Return type Integer

profile_id
Returns the profile ID of the transmission.

Returns Integer: the profile ID of the transmission.

rf_data
Returns the RF data to send.

Returns the RF data to send.

Return type Bytearray

transmit_options
Returns the transmit options bitfield.

Returns the transmit options bitfield.

Return type Integer

See also:

TransmitOptions

broadcast_radius
Returns the broadcast radius. Broadcast radius is the maximum number of hops a broadcast transmission.

Returns the broadcast radius.

Return type Integer

x64bit_dest_addr
Returns the 64-bit destination address.

Returns the 64-bit destination address.

Return type *XBee64BitAddress*

See also:

XBee64BitAddress

x16bit_dest_addr
Returns the 16-bit destination address.

Returns the 16-bit destination address.

Return type XBee16BitAddress

See also:

XBee16BitAddress

class digi.xbee.packets.common.**ExplicitRXIndicatorPacket**(*x64bit_addr*, *x16bit_addr*, *src_endpoint*, *dest_endpoint*, *cluster_id*, *profile_id*, *rx_options*, *rf_data=None*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

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:

ReceiveOptions
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.
- **src_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.
- **rx_options** (*Integer*) – bitfield indicating the receive options.

- **rf_data** (*Bytearray, optional*) – received RF data.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

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 from 2.

See also:

XBee16BitAddress
XBee64BitAddress
ReceiveOptions
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

op_mode

Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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 from 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 from the checksum field value (last byte).

- `InvalidPacketException` – if the frame type is different from `ApiFrameType.EXPLICIT_RX_INDICATOR`.
- `InvalidOperatingModeException` – if *operating_mode* is not supported.

See also:

XBeePacket.create_packet()

`XBeeAPIPacket._check_api_packet()`

needs_id()

Override method.

See also:

XBeeAPIPacket.needs_id()

is_broadcast()

Override method.

See also:

`XBeeAPIPacket.is_broadcast()`

x64bit_source_addr

Returns the 64-bit source address.

Returns the 64-bit source address.

Return type *XBee64BitAddress*

See also:

XBee64BitAddress

x16bit_source_addr

Returns the 16-bit source address.

Returns the 16-bit source address.

Return type *XBee16BitAddress*

See also:

XBee16BitAddress

source_endpoint

Returns the source endpoint of the transmission.

Returns the source endpoint of the transmission.

Return type Integer

dest_endpoint
Returns the destination endpoint of the transmission.

Returns the destination endpoint of the transmission.

Return type Integer

cluster_id
Returns the cluster ID of the transmission.

Returns the cluster ID of the transmission.

Return type Integer

profile_id
Returns the profile ID of the transmission.

Returns Integer: the profile ID of the transmission.

receive_options
Returns the receive options bitfield.

Returns the receive options bitfield.

Return type Integer

See also:

ReceiveOptions

rf_data
Returns the received RF data.

Returns the received RF data.

Return type Bytearray

digi.xbee.packets.devicecloud module

class digi.xbee.packets.devicecloud.**DeviceRequestPacket**(*request_id*, *target=None*, *request_data=None*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

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.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

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 from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is different from `ApiFrameType.DEVICE_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()

request_id
Returns the request ID of the packet.

Returns the request ID of the packet.

Return type Integer

transport
Returns the transport of the packet.

Returns the transport of the packet.

Return type Integer

flags
Returns the flags of the packet.

Returns the flags of the packet.

Return type Integer

target
Returns the device request target of the packet.

Returns the device request target of the packet.

Return type String

request_data
Returns the data of the device request.

Returns the data of the device request.

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

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.
- **op_mode** (`OperatingMode`, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

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 from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is different from `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`

Returns the request ID of the packet.

Returns the request ID of the packet.

Return type Integer

`request_data`

Returns the data of the device response.

Returns the data of the device response.

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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*, *op_mode=<OperatingMode.API_MODE (1, 'API mode')>*)

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 from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is different from `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`
Returns the status of the device response packet.

Returns the status of the device response packet.

Return type *`DeviceCloudStatus`*

See also:

`DeviceCloudStatus`

`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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)
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.
- **op_mode** (`OperatingMode`, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

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 from 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 from the checksum field value (last byte).

- `InvalidPacketException` – if the frame type is different from `ApiFrameType.FRAME_ERROR`.
- `InvalidOperatingModeException` – if *operating_mode* is not supported.

See also:

XBeePacket.create_packet()

`XBeeAPIPacket._check_api_packet()`

needs_id()

Override method.

See also:

XBeeAPIPacket.needs_id()

error

Returns the frame error of the packet.

Returns the frame error of the packet.

Return type *FrameError*

See also:

FrameError

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

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.
- **op_mode** (`OperatingMode`, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

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 from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is different from `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
Returns the path of the file to upload to Device Cloud.

Returns the path of the file to upload to Device Cloud.

Return type String

content_type
Returns the content type of the file to upload.

Returns the content type of the file to upload.

Return type String

options
Returns the file upload operation options.

Returns the file upload operation options.

Return type *`SendDataRequestOptions`*

See also:

`SendDataRequestOptions`

file_data
Returns the data of the file to upload.

Returns the data of the file to upload.

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type `OperatingMode`

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 must be escaped.

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 info 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*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

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.
- **op_mode** (`OperatingMode`, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

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_broadcast()
Returns whether this packet is broadcast or not.

Returns *True* if this packet is broadcast, *False* otherwise.

Return type Boolean

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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 from 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 from the checksum field value (last byte).
- InvalidPacketException – if the frame type is different from 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

Returns the file upload status.

Returns the file upload status.

Return type *DeviceCloudStatus*

See also:

DeviceCloudStatus

digi.xbee.packets.digimesh module

class digi.xbee.packets.digimesh.**RouteInformationPacket**(*src_event*, *timestamp*, *ack_timeout_count*, *tx_block_count*, *dst_addr*, *src_addr*, *responder_addr*, *successor_addr*, *additional_data=None*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents a DigiMesh Route Information packet. Packet is built using the parameters of the constructor or providing a valid API payload.

A Route Information Packet can be output for DigiMesh unicast transmissions on which the NACK enable or the Trace Route enable TX option is enabled.

See also:

XBeeAPIPacket

Class constructor. Instantiates a new *RouteInformationPacket* object with the provided parameters.

Parameters

- **src_event** (*Integer*) – Source event identifier. 0x11=NACK, 0x12=Trace route
- **timestamp** (*Integer*) – System timer value on the node generating the this packet. The timestamp is in microseconds.
- **ack_timeout_count** (*Integer*) – The number of MAC ACK timeouts.

- **tx_block_count** (*Integer*) – The number of times the transmission was blocked due to reception in progress.
- **dst_addr** (*XBee64BitAddress*) – The 64-bit address of the final destination node of this network-level transmission.
- **src_addr** (*XBee64BitAddress*) – The 64-bit address of the source node of this network-level transmission.
- **responder_addr** (*XBee64BitAddress*) – The 64-bit address of the node that generates this packet after it sends (or attempts to send) the packet to the next hop (successor node).
- **successor_addr** (*XBee64BitAddress*) – The 64-bit address of the next node after the responder in the route towards the destination, whether or not the packet arrived successfully at the successor node.
- **additional_data** (*Bytearray, optional, default=`None`*) – Additional data of the packet.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

Raises

- `ValueError` – if *src_event* is not 0x11 or 0x12.
- `ValueError` – if *timestamp* is not between 0 and 0xFFFFFFFF.
- `ValueError` – if *ack_timeout_count* or *tx_block_count* are not between 0 and 255.

See also:

XBee64BitAddress
XBeeAPIPacket

static create_packet(*raw*, *operating_mode*)
Override method.

Returns *RouteInformationPacket*.

Raises

- `InvalidPacketException` – If the bytearray length is less than 46. (start delim. + length (2 bytes) + frame type + src_event + length + timestamp (4 bytes) + ack timeout count + tx blocked count + reserved + dest addr (8 bytes) + src addr (8 bytes) + responder addr (8 bytes) + successor addr (8 bytes) + checksum = 46 bytes).
- `InvalidPacketException` – If the length field of *raw* is different from its real length. (length field: bytes 1 and 3)
- `InvalidPacketException` – If the first byte of 'raw' is not the header byte. See *SpecialByte*.
- `InvalidPacketException` – If the calculated checksum is different from the checksum field value (last byte).
- `InvalidPacketException` – If the frame type is not `ApiFrameType.DIGIMESH_ROUTE_INFORMATION`.

- `InvalidPacketException` – If the internal length byte of the rest of the frame (without the checksum) is different from its real length.
- `InvalidOperatingModeException` – If *operating_mode* is not supported.

See also:

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

`needs_id``()`
Override method.

See also:

`XBeeAPIPacket.needs_id()`

`src_event`
Returns the source event.

Returns The source event.

Return type Integer

`length`
Returns the number of bytes that follow, excluding the checksum.

Returns Data length.

Return type Integer

`timestamp`
Returns the system timer value on the node generating this package. The timestamp is in microseconds.

Returns The system timer value in microseconds.

Return type Integer

`ack_timeout_count`
Returns the number of MAC ACK timeouts that occur.

Returns The number of MAC ACK timeouts that occur.

Return type Integer

`tx_block_count`
Returns the number of times the transmission was blocked due to reception in progress.

Returns

The number of times the transmission was blocked due to reception in progress.

Return type Integer

`frame_id`
Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

get_checksum()

Returns the checksum value of this XBeePacket. The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer

See also:

factory

get_frame_spec_data()

Override method.

See also:

XBeePacket.get_frame_spec_data()

get_frame_type()

Returns the frame type of this packet.

Returns the frame type of this packet.

Return type *ApiFrameType*

See also:

ApiFrameType

get_frame_type_value()

Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer

See also:

ApiFrameType

is_broadcast()

Returns whether this packet is broadcast or not.

Returns *True* if this packet is broadcast, *False* otherwise.

Return type Boolean

op_mode

Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info of the XBeePacket fields.

Return type Dictionary

static unescape_data (*data*)
Un-escapes the provided bytearray data.

Parameters **data** (*Bytearray*) – the bytearray to unescape.

Returns *data* unescaped.

Return type Bytearray

dst_addr
Returns the 64-bit source address.

Returns

The 64-bit address of the final destination node.

Return type *XBee64BitAddress*

See also:

XBee64BitAddress

src_addr
Returns the 64-bit address of the source node of this network-level transmission.

Returns The 64-bit address of the source node.

Return type *XBee64BitAddress*

See also:

XBee64BitAddress

responder_addr
Returns the 64-bit address of the node that generates this packet after it sends (or attempts to send) the packet to the next hop (successor node).

Returns The 64-bit address of the responder node.

Return type *XBee64BitAddress*

See also:

XBee64BitAddress

successor_addr

Returns the 64-bit address of the next node after the responder in the route towards the destination, whether or not the packet arrived successfully at the successor node.

Returns The 64-bit address of the successor node.

Return type *XBee64BitAddress*

See also:

XBee64BitAddress

digi.xbee.packets.filesystem module

class digi.xbee.packets.filesystem.**FSRequestPacket**(*frame_id*, *command*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents a File System Request. Packet is built using the parameters of the constructor or providing a valid API payload.

A File System Request allows to access the filesystem and perform different operations.

Command response is received as an *FSResponsePacket*.

See also:

XBeeAPIPacket

Class constructor. Instantiates a new *FSRequestPacket* object with the provided parameters.

Parameters

- **frame_id** (*Integer*) – Frame ID of the packet.
- **command** (*FSCmd* or bytearray) – File system command to execute.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

Raises

- ValueError – If *frame_id* is less than 0 or greater than 255.
- TypeError – If *command* is not a *FSCmd* or a bytearray.

See also:

FSCmd
XBeeAPIPacket

static create_packet(*raw*, *operating_mode*)
Override method.

Returns *FSRequestPacket*

Raises

- `InvalidPacketException` – If the bytearray length is less than 7 + the minimum length of the command. (start delim. + length (2 bytes) + frame type + frame id + fs cmd id + checksum + cmd data = 7 bytes + cmd data).
- `InvalidPacketException` – If the length field of 'raw' is different from 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 from the checksum field value (last byte).
- `InvalidPacketException` – If the frame type is different from `ApiFrameType.FILE_SYSTEM_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()

command
Returns the file system command of the packet.

Returns File system command of the packet.

Return type String

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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.filesystem.**FSResponsePacket**(*frame_id*, *command*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents a File System Response. Packet is built using the parameters of the constructor or providing a valid API payload.

This packet is received in response of an *FSRequestPacket*.

See also:

XBeeAPIPacket

Class constructor. Instantiates a new *FSResponsePacket* object with the provided parameters.

Parameters

- **frame_id** (*Integer*) – The frame ID of the packet.
- **command** (*FSCmd* or bytearray) – File system command to execute.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

Raises

- ValueError – If *frame_id* is less than 0 or greater than 255.
- TypeError – If *command* is not a *FSCmd* or a bytearray.

See also:

FSCmd
XBeeAPIPacket

static create_packet(*raw*, *operating_mode*)
Override method.

Returns *FSResponsePacket*

Raises

- `InvalidPacketException` – If the bytearray length is less than 8 + the minimum length of the command. (start delim. + length (2 bytes) + frame type + frame id + fs cmd id + status + checksum + cmd data = 8 bytes + cmd data).
- `InvalidPacketException` – If the length field of 'raw' is different from 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 from the checksum field value (last byte).
- `InvalidPacketException` – If the frame type is different from `ApiFrameType.FILE_SYSTEM_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()`

`command`
Returns the file system command of the packet.

Returns File system command of the packet.

Return type String

`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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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.filesystem.**RemoteFSRequestPacket**(*frame_id*, *x64bit_addr*, *command*, *transmit_options=0*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents a remote File System Request. Packet is built using the parameters of the constructor or providing a valid API payload.

Used to access the filesystem on a remote device and perform different operations.

Remote command options are set as a bitfield.

If configured, command response is received as a *RemoteFSResponsePacket*.

See also:

RemoteFSResponsePacket
XBeeAPIPacket

Class constructor. Instantiates a new *RemoteFSRequestPacket* object with the provided parameters.

Parameters

- **frame_id** (*Integer*) – Frame ID of the packet.
- **x64bit_addr** (*XBee64BitAddress*) – 64-bit destination address.
- **command** (*FSCmd* or bytearray) – File system command to execute.
- **transmit_options** (*Integer, optional, default=`TransmitOptions.NONE.value`*) – Bitfield of supported transmission options.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

Raises

- ValueError – If *frame_id* is less than 0 or greater than 255.
- TypeError – If *command* is not a *FSCmd* or a bytearray.

See also:

FSCmd
TransmitOptions
XBee64BitAddress
XBeeAPIPacket

static create_packet(*raw*, *operating_mode*)
Override method.

Returns *RemoteFSRequestPacket*

Raises

- `InvalidPacketException` – If the bytearray length is less than 7 + the minimum length of the command. (start delim. + length (2 bytes) + frame type + frame id + 64bit addr. + transmit options + fs cmd id + checksum + cmd data = 16 bytes + cmd data).
- `InvalidPacketException` – If the length field of 'raw' is different from 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 from the checksum field value (last byte).
- `InvalidPacketException` – If the frame type is different from `ApiFrameType.REMOTE_FILE_SYSTEM_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
Returns the 64-bit destination address.

Returns 64-bit destination address.

Return type *XBee64BitAddress*

See also:

XBee64BitAddress

command
Returns the file system command of the packet.

Returns File system command of the packet.

Return type String

transmit_options
Returns the transmit options bitfield.

Returns Transmit options bitfield.

Return type Integer

See also:

TransmitOptions

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

`op_mode`
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

`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 must be escaped.

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 info 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.filesystem.**RemoteFSResponsePacket**(*frame_id*, *x64bit_addr*, *command*, *rx_options*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents a Remote File System Response. Packet is built using the parameters of the constructor or providing a valid API payload.

This packet is received in response of an *RemoteFSRequestPacket*.

See also:

RemoteFSRequestPacket
XBeeAPIPacket

Class constructor. Instantiates a new *RemoteFSResponsePacket* object with the provided parameters.

Parameters

- **frame_id** (*Integer*) – The frame ID of the packet.
- **x64bit_addr** (*XBee64BitAddress*) – 64-bit source address.
- **command** (*FSCmd* or bytearray) – File system command to execute.
- **rx_options** (*Integer*) – Bitfield indicating the receive options.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

Raises

- ValueError – If *frame_id* is less than 0 or greater than 255.
- TypeError – If *command* is not a *FSCmd* or a bytearray.

See also:

FSCmd
ReceiveOptions
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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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 *RemoteFSResponsePacket*

Raises

- `InvalidPacketException` – If the bytearray length is less than 8 + the minimum length of the command. (start delim. + length (2 bytes) + frame type + frame id + 64bit addr. + receive options + fs cmd id + status + checksum + cmd data = 17 bytes + cmd data).
- `InvalidPacketException` – If the length field of 'raw' is different from 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 from the checksum field value (last byte).
- `InvalidPacketException` – If the frame type is different from `ApiFrameType.REMOTE_FILE_SYSTEM_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`
Returns the 64-bit source address.

Returns 64-bit source address.

Return type *`XBee64BitAddress`*

See also:

`XBee64BitAddress`

`command`
Returns the file system command of the packet.

Returns File system command of the packet.

Return type String

`receive_options`
Returns the receive options bitfield.

Returns Receive options bitfield.

Return type Integer

See also:

ReceiveOptions

digi.xbee.packets.filesystem.**build_fs_command**(*cmd_bytearray*, *direction=0*)

Creates a file system command from raw data.

Parameters

- **cmd_bytearray** (*Bytearray*) – Raw data of the packet to build.
- **direction** (*Integer, optional, default=0*) – If this command is a request (0) or a response (1).

Raises InvalidPacketException – If *cmd_bytearray* is not a bytearray or its length is less than 1 for requests 2 for responses.

See also:

FSCmd

digi.xbee.packets.network module

class digi.xbee.packets.network.**RXIPv4Packet**(*src_address*, *dest_port*, *src_port*, *ip_protocol*, *data=None*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

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

- **src_address** (IPv4Address) – IPv4 address of the source device.
- **dest_port** (*Integer*) – destination port number.
- **src_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.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

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 class: *.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 from 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 from 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
Returns the IPv4 address of the source device.

Returns the IPv4 address of the source device.

Return type `ipaddress.IPv4Address`

dest_port
Returns the destination port.

Returns the destination port.

Return type Integer

source_port
Returns the source port.

Returns the source port.

Return type Integer

ip_protocol
Returns the IP protocol used for transmitted data.

Returns the IP protocol used for transmitted data.

Return type *IPProtocol*

data
Returns the data of the packet.

Returns the data of the packet.

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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*, *src_port*, *ip_protocol*, *tx_opts*, *data=None*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)
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.
- **src_port** (*Integer*) – source port number.
- **ip_protocol** (*IPProtocol*) – IP protocol used for transmitted data.
- **tx_opts** (*Integer*) – the transmit options of the packet.
- **data** (*Bytearray, optional*) – data that is sent to the destination device.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

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 from 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 from 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()`

`dest_address`
Returns the IPv4 address of the destination device.

Returns the IPv4 address of the destination device.

Return type `ipaddress.IPv4Address`

`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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info of the XBeePacket fields.

Return type Dictionary

static unescape_data(*data*)
Un-escapes the provided bytearray data.

Parameters **data** (*Bytearray*) – the bytearray to unescape.

Returns *data* unescaped.

Return type Bytearray

dest_port
Returns the destination port.

Returns the destination port.

Return type Integer

source_port
Returns the source port.

Returns the source port.

Return type Integer

ip_protocol
Returns the IP protocol used for transmitted data.

Returns the IP protocol used for transmitted data.

Return type *IPProtocol*

transmit_options
Returns the transmit options of the packet.

Returns the transmit options of the packet.

Return type Integer

data
Returns the data of the packet.

Returns the data of the packet.

Return type Bytearray

digi.xbee.packets.raw module

class digi.xbee.packets.raw.**TX64Packet** (*frame_id*, *x64bit_addr*, *tx_opts*, *rf_data=None*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)
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.
- **tx_opts** (*Integer*) – bitfield of supported transmission options.
- **rf_data** (*Bytearray, optional*) – RF data that is sent to the destination device.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

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 from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is different from `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
Returns the 64-bit destination address.

Returns the 64-bit destination address.

Return type *XBee64BitAddress*

See also:

XBee64BitAddress

transmit_options
Returns the transmit options bitfield.

Returns the transmit options bitfield.

Return type Integer

See also:

TransmitOptions

rf_data
Returns the RF data to send.

Returns the RF data to send.

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

op_mode

Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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*, *tx_opts*, *rf_data=None*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

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.
- **tx_opts** (*Integer*) – bitfield of supported transmission options.
- **rf_data** (*Bytearray, optional*) – RF data that is sent to the destination device.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

See also:

TransmitOptions
XBee16BitAddress
XBeeAPIPacket

Raises `ValueError` – if *frame_id* is less than 0 or greater than 255.

static create_packet(*raw*, *operating_mode*)
Override method.

Returns *TX16Packet*.

Raises

- `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 from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is different from `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
Returns the 16-bit destination address.

Returns the 16-bit destination address.

Return type *XBee16BitAddress*

See also:

XBee16BitAddress

transmit_options
Returns the transmit options bitfield.

Returns the transmit options bitfield.

Return type Integer

See also:

TransmitOptions

rf_data
Returns the RF data to send.

Returns the RF data to send.

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

op_mode

Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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*, *tx_status*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

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.
- **`tx_status`** (*`TransmitStatus`*) – transmit status.
- **`op_mode`** (*`OperatingMode`*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

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 from 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 from the checksum field value (last byte).

- `InvalidPacketException` – if the frame type is different from `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

Returns the transmit status.

Returns the transmit status.

Return type *TransmitStatus*

See also:

TransmitStatus

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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*, *rx_opts*, *rf_data=None*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

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.
- **rx_opts** (`Integer`) – bitfield indicating the receive options.
- **rf_data** (`Bytearray, optional`) – received RF data.
- **op_mode** (`OperatingMode`, optional, default=‘OperatingMode.API_MODE‘) – The mode in which the frame was captured.

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 from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is different from `ApiFrameType.RX_64`.
- `InvalidOperatingModeException` – if *operating_mode* is not supported.

See also:

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

needs_id()
Override method.

See also:

`XBeeAPIPacket.needs_id()`

is_broadcast()
Override method.

See also:

`XBeeAPIPacket.is_broadcast()`

x64bit_source_addr
Returns the 64-bit source address.

Returns the 64-bit source address.

Return type `XBee64BitAddress`

See also:

`XBee64BitAddress`

rssi
Returns the received Signal Strength Indicator (RSSI).

Returns the received Signal Strength Indicator (RSSI).

Return type Integer

receive_options
Returns the receive options bitfield.

Returns the receive options bitfield.

Return type Integer

See also:

ReceiveOptions

rf_data
Returns the received RF data.

Returns the received RF data.

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

op_mode

Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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, rx_opts, rf_data=None, op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

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.
- **rx_opts** (*Integer*) – bitfield indicating the receive options.
- **rf_data** (*Bytearray, optional*) – received RF data.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

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 from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is different from `ApiFrameType.RX_16`.
- `InvalidOperatingModeException` – if *operating_mode* is not supported.

See also:

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

needs_id()
Override method.

See also:

XBeeAPIPacket.needs_id()

is_broadcast()
Override method.

See also:

`XBeeAPIPacket.is_broadcast()`

x16bit_source_addr
Returns the 16-bit source address.

Returns the 16-bit source address.

Return type *XBee16BitAddress*

See also:

XBee16BitAddress

rssi
Returns the received Signal Strength Indicator (RSSI).

Returns the received Signal Strength Indicator (RSSI).

Return type Integer

receive_options
Returns the receive options bitfield.

Returns the receive options bitfield.

Return type Integer

See also:

ReceiveOptions

rf_data
Returns the received RF data.

Returns the received RF data.

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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*, *rx_opts*, *data*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

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.
- **rx_opts** (*Integer*) – bitfield indicating the receive options.
- **data** (*Bytearray*) – received RF data.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

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 from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is different from `ApiFrameType.RX_IO_64`.
- `InvalidOperatingModeException` – if *operating_mode* is not supported.

See also:

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

`needs_id``()`
Override method.

See also:

`XBeeAPIPacket.needs_id()`

`is_broadcast``()`
Override method.

See also:

`XBeeAPIPacket.is_broadcast()`

`x64bit_source_addr`
Returns the 64-bit source address.

Returns the 64-bit source address.

Return type `XBee64BitAddress`

See also:

`XBee64BitAddress`

`rssi`
Returns the received Signal Strength Indicator (RSSI).

Returns the received Signal Strength Indicator (RSSI).

Return type Integer

`receive_options`
Returns the receive options bitfield.

Returns the receive options bitfield.

Return type Integer

See also:

ReceiveOptions

rf_data
Returns the received RF data.

Returns the received RF data.

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

op_mode

Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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

io_sample

Returns the IO sample corresponding to the data contained in the packet.

Returns

the IO sample of the packet, *None* if the packet has not any data or if the sample could not be generated correctly.

Return type *IOSample*

See also:

IOSample

class digi.xbee.packets.raw.**RX16IOPacket** (*x16bit_addr*, *rssi*, *rx_opts*, *data*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents an RX16 address IO packet. Packet is built using the parameters of the constructor or providing a valid byte array.

I/O data is sent out the UART using an API frame.

See also:

XBeeAPIPacket

Class constructor. Instantiates an *RX16IOPacket* object with the provided parameters.

Parameters

- **x16bit_addr** (*XBee16BitAddress*) – the 16-bit source address.
- **rssi** (*Integer*) – received signal strength indicator.
- **rx_opts** (*Integer*) – bitfield indicating the receive options.
- **data** (*Bytearray*) – received RF data.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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 from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is different from `ApiFrameType.RX_IO_16`.
- `InvalidOperatingModeException` – if *operating_mode* is not supported.

See also:

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

needs_id()
Override method.

See also:

XBeeAPIPacket.needs_id()

is_broadcast()
Override method.

See also:

`XBeeAPIPacket.is_broadcast()`

x16bit_source_addr
Returns the 16-bit source address.

Returns the 16-bit source address.

Return type *XBee16BitAddress*

See also:

XBee16BitAddress

rssi
Returns the received Signal Strength Indicator (RSSI).

Returns the received Signal Strength Indicator (RSSI).

Return type Integer

receive_options

Returns the receive options bitfield.

Returns the receive options bitfield.

Return type Integer

See also:

ReceiveOptions

rf_data

Returns the received RF data.

Returns the received RF data.

Return type Bytearray

io_sample

Returns the IO sample corresponding to the data contained in the packet.

Returns

the IO sample of the packet, *None* if the packet has not any data or if the sample could not be generated correctly.

Return type *IOSample*

See also:

IOSample

digi.xbee.packets.relay module

class digi.xbee.packets.relay.**UserDataRelayPacket**(*frame_id*, *local_iface*, *data=None*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

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_iface** (*XBeeLocalInterface*) – the destination interface.
- **data** (*Bytearray, optional*) – Data to send to the destination interface.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

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 from 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 from 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()

data

Returns the data to send.

Returns the data to send.

Return type Bytearray

dest_interface

Returns the the destination interface.

Returns the destination interface.

Return type *XBeeLocalInterface*

See also:

XBeeLocalInterface

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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_iface*, *data=None*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

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_iface** (*XBeeLocalInterface*) – the source interface.
- **data** (*Bytearray, optional*) – Data received from the source interface.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

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 from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is not `ApiFrameType.USER_DATA_RELAY_OUTPUT`.
- `InvalidOperatingModeException` – if *operating_mode* is not supported.

See also:

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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

data
Returns the received data.

Returns the received data.

Return type Bytearray

src_interface
Returns the the source interface.

Returns the source interface.

Return type *XBeeLocalInterface*

See also:

XBeeLocalInterface

digi.xbee.packets.socket module

class digi.xbee.packets.socket.**SocketCreatePacket**(*frame_id*, *protocol*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents a Socket Create packet. Packet is built using the parameters of the constructor.

Use this frame to create a new socket with the following protocols: TCP, UDP, or TLS.

See also:

SocketCreateResponsePacket
XBeeAPIPacket

Class constructor. Instantiates a new *SocketCreatePacket* object with the provided parameters.

Parameters

- **frame_id** (*Integer*) – the frame ID of the packet.
- **protocol** (*IPProtocol*) – the protocol used to create the socket.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

See also:

XBeeAPIPacket
IPProtocol

Raises `ValueError` – if *frame_id* is less than 0 or greater than 255.

static create_packet(*raw*, *operating_mode*)
Override method.

Returns *SocketCreatePacket*.

Raises

- `InvalidPacketException` – if the bytearray length is less than 7. (start delim. + length (2 bytes) + frame type + frame id + protocol + checksum = 7 bytes).
- `InvalidPacketException` – if the length field of 'raw' is different from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is not `ApiFrameType.SOCKET_CREATE`.
- `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()

protocol

Returns the communication protocol.

Returns the communication protocol.

Return type *IPProtocol*

See also:

IPProtocol

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info of the XBeePacket fields.

Return type Dictionary

static unescape_data(*data*)
Un-escapes the provided bytearray data.

Parameters **data** (*Bytearray*) – the bytearray to unescape.

Returns *data* unescaped.

Return type Bytearray

class digi.xbee.packets.socket.**SocketCreateResponsePacket**(*frame_id*, *socket_id*, *status*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents a Socket Create Response packet. Packet is built using the parameters of the constructor.

The device sends this frame in response to a Socket Create (0x40) frame. It contains a socket ID that should be used for future transactions with the socket and a status field.

If the status field is non-zero, which indicates an error, the socket ID will be set to 0xFF and the socket will not be opened.

See also:

SocketCreatePacket
XBeeAPIPacket

Class constructor. Instantiates a new *SocketCreateResponsePacket* object with the provided parameters.

Parameters

- **frame_id** (*Integer*) – the frame ID of the packet.
- **socket_id** (*Integer*) – the unique socket ID to address the socket.
- **status** (*SocketStatus*) – the socket create status.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

See also:

XBeeAPIPacket
SocketStatus

Raises

- ValueError – if *frame_id* is less than 0 or greater than 255.
- ValueError – if *socket_id* is less than 0 or greater than 255.

static create_packet(*raw*, *operating_mode*)

Override method.

Returns *SocketCreateResponsePacket*.

Raises

- InvalidPacketException – if the bytearray length is less than 8. (start delim. + length (2 bytes) + frame type + frame id + socket id + status + checksum = 8 bytes).
- InvalidPacketException – if the length field of 'raw' is different from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is not `ApiFrameType.SOCKET_CREATE_RESPONSE`.
- `InvalidOperatingModeException` – if *operating_mode* is not supported.

See also:

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

`needs_id``()`
Override method.

See also:

`XBeeAPIPacket.needs_id()`

`socket_id`
Returns the the socket ID.

Returns the socket ID.

Return type Integer

`status`
Returns the socket create status.

Returns the status.

Return type *`SocketStatus`*

See also:

`SocketStatus`

`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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info of the XBeePacket fields.

Return type Dictionary

static unescape_data(*data*)

Un-escapes the provided bytearray data.

Parameters **data** (*Bytearray*) – the bytearray to unescape.

Returns *data* unescaped.

Return type Bytearray

class digi.xbee.packets.socket.**SocketOptionRequestPacket**(*frame_id, socket_id, option, option_data=None, op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents a Socket Option Request packet. Packet is built using the parameters of the constructor.

Use this frame to modify the behavior of sockets to be different from the normal default behavior.

If the Option Data field is zero-length, the Socket Option Response Packet (0xC1) reports the current effective value.

See also:

SocketOptionResponsePacket
XBeeAPIPacket

Class constructor. Instantiates a new *SocketOptionRequestPacket* object with the provided parameters.

Parameters

- **frame_id** (*Integer*) – the frame ID of the packet.
- **socket_id** (*Integer*) – the socket ID to modify.
- **option** (*SocketOption*) – the socket option of the parameter to change.
- **option_data** (*Bytearray, optional*) – the option data.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

See also:

XBeeAPIPacket
SocketOption

Raises

- ValueError – if *frame_id* is less than 0 or greater than 255.
- ValueError – if *socket_id* is less than 0 or greater than 255.

static create_packet(*raw*, *operating_mode*)

Override method.

Returns *SocketOptionRequestPacket*.

Raises

- `InvalidPacketException` – if the bytearray length is less than 8. (start delim. + length (2 bytes) + frame type + frame id + socket id + option + checksum = 8 bytes).
- `InvalidPacketException` – if the length field of 'raw' is different from its real length. (length field: byte 2 and 3)
- `InvalidPacketException` – if the first byte of 'raw' is not the header byte. See *SpecialByte*.
- `InvalidPacketException` – if the calculated checksum is different from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is not `ApiFrameType.SOCKET_OPTION_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()

socket_id

Returns the the socket ID.

Returns the socket ID.

Return type Integer

option

Returns the socket option.

Returns the socket option.

Return type *SocketOption*

See also:

SocketOption

option_data

Returns the socket option data.

Returns the socket option data.

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info of the XBeePacket fields.

Return type Dictionary

static unescape_data(*data*)
Un-escapes the provided bytearray data.

Parameters **data** (*Bytearray*) – the bytearray to unescape.

Returns *data* unescaped.

Return type Bytearray

class digi.xbee.packets.socket.**SocketOptionResponsePacket**(*frame_id, socket_id, option, status, option_data=None, op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents a Socket Option Response packet. Packet is built using the parameters of the constructor.

Reports the status of requests made with the Socket Option Request (0x41) packet.

See also:

SocketOptionRequestPacket
XBeeAPIPacket

Class constructor. Instantiates a new *SocketOptionResponsePacket* object with the provided parameters.

Parameters

- **frame_id** (*Integer*) – the frame ID of the packet.
- **socket_id** (*Integer*) – the socket ID for which modification was requested.

- **option** (*SocketOption*) – the socket option of the parameter requested.
- **status** (*SocketStatus*) – the socket option status of the parameter requested.
- **option_data** (*Bytearray, optional*) – the option data.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

See also:

XBeeAPIPacket
SocketOption
SocketStatus

Raises

- ValueError – if *frame_id* is less than 0 or greater than 255.
- ValueError – if *socket_id* is less than 0 or greater than 255.

static create_packet(*raw*, *operating_mode*)
Override method.

Returns *SocketOptionResponsePacket*.

Raises

- InvalidPacketException – if the bytearray length is less than 9. (start delim. + length (2 bytes) + frame type + frame id + socket id + option + status + checksum = 9 bytes).
- InvalidPacketException – if the length field of 'raw' is different from 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 from the checksum field value (last byte).
- InvalidPacketException – if the frame type is not ApiFrameType.SOCKET_OPTION_RESPONSE.
- InvalidOperatingModeException – if *operating_mode* is not supported.

See also:

XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()

needs_id()
Override method.

See also:

XBeeAPIPacket.needs_id()

socket_id
Returns the the socket ID.

Returns the socket ID.

Return type Integer

option
Returns the socket option.

Returns the socket option.

Return type *SocketOption*

See also:

SocketOption

status
Returns the socket option status.

Returns the socket option status.

Return type *SocketStatus*

See also:

SocketStatus

option_data
Returns the socket option data.

Returns the socket option data.

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info of the XBeePacket fields.

Return type Dictionary

static unescape_data(*data*)
Un-escapes the provided bytearray data.

Parameters **data** (*Bytearray*) – the bytearray to unescape.

Returns *data* unescaped.

Return type Bytearray

class digi.xbee.packets.socket.**SocketConnectPacket**(*frame_id*, *socket_id*, *dest_port*, *dest_address_type*, *dest_address*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)
Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents a Socket Connect packet. Packet is built using the parameters of the constructor.

Use this frame to create a socket connect message that causes the device to connect a socket to the given address and port.

For a UDP socket, this filters out any received responses that are not from the specified remote address and port.

Two frames occur in response:

- Socket Connect Response frame (*SocketConnectResponsePacket*): Arrives immediately and confirms the request.
- Socket Status frame (*SocketStatePacket*): Indicates if the connection was successful.

See also:

SocketConnectResponsePacket
SocketStatePacket
XBeeAPIPacket

Class constructor. Instantiates a new *SocketConnectPacket* object with the provided parameters.

Parameters

- **frame_id** (*Integer*) – the frame ID of the packet.
- **socket_id** (*Integer*) – the ID of the socket to connect.
- **dest_port** (*Integer*) – the destination port number.
- **dest_address_type** (*Integer*) – the destination address type. One of *SocketConnectPacket.DEST_ADDRESS_BINARY* or *SocketConnectPacket.DEST_ADDRESS_STRING*.
- **dest_address** (*Bytearray or String*) – the destination address.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

See also:

SocketConnectPacket.DEST_ADDRESS_BINARY
SocketConnectPacket.DEST_ADDRESS_STRING

XBeeAPIPacket

Raises

- `ValueError` – if *frame_id* is less than 0 or greater than 255.
- `ValueError` – if *socket_id* is less than 0 or greater than 255.
- `ValueError` – if *dest_port* is less than 0 or greater than 65535.
- `ValueError` – if *dest_address_type* is different than *SocketConnectPacket.DEST_ADDRESS_BINARY* and *SocketConnectPacket.DEST_ADDRESS_STRING*.
- `ValueError` – if *dest_address* is *None* or does not follow the format specified in the configured type.

DEST_ADDRESS_BINARY = 0
Indicates the destination address field is a binary IPv4 address in network byte order.

DEST_ADDRESS_STRING = 1
Indicates the destination address field is a string containing either a dotted quad value or a domain name to be resolved.

static create_packet(*raw*, *operating_mode*)
Override method.

Returns *SocketConnectPacket*.

Raises

- `InvalidPacketException` – if the bytearray length is less than 11. (start delim. + length (2 bytes) + frame type + frame id + socket id + dest port (2 bytes) + dest address type + dest_address + checksum = 11 bytes).
- `InvalidPacketException` – if the length field of 'raw' is different from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is not `ApiFrameType.SOCKET_CONNECT`.
- `InvalidOperatingModeException` – if *operating_mode* is not supported.

See also:

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

needs_id()
Override method.

See also:

XBeeAPIPacket.needs_id()

socket_id
Returns the the socket ID.

Returns the socket ID.

Return type Integer

dest_port
Returns the destination port.

Returns the destination port.

Return type Integer

dest_address_type
Returns the destination address type.

Returns the destination address type.

Return type Integer

dest_address
Returns the destination address.

Returns the destination address.

Return type Bytearray or String

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

op_mode

Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info of the XBeePacket fields.

Return type Dictionary

static unescape_data(*data*)

Un-escapes the provided bytearray data.

Parameters **data** (*Bytearray*) – the bytearray to unescape.

Returns *data* unescaped.

Return type Bytearray

class digi.xbee.packets.socket.**SocketConnectResponsePacket**(*frame_id*, *socket_id*, *status*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents a Socket Connect Response packet. Packet is built using the parameters of the constructor.

The device sends this frame in response to a Socket Connect (0x42) frame. The frame contains a status regarding the initiation of the connect.

See also:

SocketConnectPacket
XBeeAPIPacket

Class constructor. Instantiates a new *SocketConnectPacket* object with the provided parameters.

Parameters

- **frame_id** (*Integer*) – the frame ID of the packet.
- **socket_id** (*Integer*) – the ID of the socket to connect.
- **status** (*SocketStatus*) – the socket connect status.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

See also:

XBeeAPIPacket
SocketStatus

Raises

- ValueError – if *frame_id* is less than 0 or greater than 255.
- ValueError – if *socket_id* is less than 0 or greater than 255.

static create_packet(*raw*, *operating_mode*)

Override method.

Returns *SocketConnectResponsePacket*.

Raises

- InvalidPacketException – if the bytearray length is less than 8. (start delim. + length (2 bytes) + frame type + frame id + socket id + status + checksum = 8 bytes).
- InvalidPacketException – if the length field of 'raw' is different from 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 from the checksum field value (last byte).

- `InvalidPacketException` – if the frame type is not `ApiFrameType.SOCKET_CONNECT_RESPONSE`.
- `InvalidOperatingModeException` – if *operating_mode* is not supported.

See also:

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

needs_id()
Override method.

See also:

XBeeAPIPacket.needs_id()

socket_id
Returns the the socket ID.

Returns the socket ID.

Return type Integer

status
Returns the socket connect status.

Returns the socket connect status.

Return type *SocketStatus*

See also:

SocketStatus

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info of the XBeePacket fields.

Return type Dictionary

static unescape_data(*data*)
Un-escapes the provided bytearray data.

Parameters **data** (*Bytearray*) – the bytearray to unescape.

Returns *data* unescaped.

Return type Bytearray

class digi.xbee.packets.socket.**SocketClosePacket**(*frame_id*, *socket_id*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents a Socket Close packet. Packet is built using the parameters of the constructor.

Use this frame to close a socket when given an identifier.

See also:

SocketCloseResponsePacket
XBeeAPIPacket

Class constructor. Instantiates a new *SocketClosePacket* object with the provided parameters.

Parameters

- **frame_id** (*Integer*) – the frame ID of the packet.
- **socket_id** (*Integer*) – the ID of the socket to close.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

See also:

XBeeAPIPacket

Raises

- ValueError – if *frame_id* is less than 0 or greater than 255.
- ValueError – if *socket_id* is less than 0 or greater than 255.

static create_packet(*raw*, *operating_mode*)
Override method.

Returns *SocketClosePacket*.

Raises

- InvalidPacketException – if the bytearray length is less than 7. (start delim. + length (2 bytes) + frame type + frame id + socket id + checksum = 7 bytes).
- InvalidPacketException – if the length field of 'raw' is different from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is not `ApiFrameType.SOCKET_CLOSE`.
- `InvalidOperatingModeException` – if *operating_mode* is not supported.

See also:

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

`needs_id``()`
Override method.

See also:

`XBeeAPIPacket.needs_id()`

`socket_id`
Returns the socket ID.

Returns the socket ID.

Return type Integer

`frame_id`
Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

`get_checksum``()`
Returns the checksum value of this XBeePacket. The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer

See also:

`factory`

`get_frame_spec_data``()`
Override method.

See also:

XBeePacket.get_frame_spec_data()

get_frame_type()
Returns the frame type of this packet.

Returns the frame type of this packet.

Return type *ApiFrameType*

See also:

ApiFrameType

get_frame_type_value()
Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer

See also:

ApiFrameType

is_broadcast()
Returns whether this packet is broadcast or not.

Returns *True* if this packet is broadcast, *False* otherwise.

Return type Boolean

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info of the XBeePacket fields.

Return type Dictionary

static unescape_data(*data*)
Un-escapes the provided bytearray data.

Parameters **data** (*Bytearray*) – the bytearray to unescape.

Returns *data* unescaped.

Return type Bytearray

class digi.xbee.packets.socket.**SocketCloseResponsePacket**(*frame_id*, *socket_id*, *status*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents a Socket Close Response packet. Packet is built using the parameters of the constructor.

The device sends this frame in response to a Socket Close (0x43) frame. Since a close will always succeed for a socket that exists, the status can be only one of two values:

- Success.
- Bad socket ID.

See also:

SocketClosePacket
XBeeAPIPacket

Class constructor. Instantiates a new *SocketCloseResponsePacket* object with the provided parameters.

Parameters

- **frame_id** (*Integer*) – the frame ID of the packet.
- **socket_id** (*Integer*) – the ID of the socket to close.
- **status** (*SocketStatus*) – the socket close status.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

See also:

XBeeAPIPacket
SocketStatus

Raises

- ValueError – if *frame_id* is less than 0 or greater than 255.
- ValueError – if *socket_id* is less than 0 or greater than 255.

static create_packet(*raw*, *operating_mode*)

Override method.

Returns *SocketCloseResponsePacket*.

Raises

- InvalidPacketException – if the bytearray length is less than 8. (start delim. + length (2 bytes) + frame type + frame id + socket id + status + checksum = 8 bytes).
- InvalidPacketException – if the length field of 'raw' is different from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is not `ApiFrameType.SOCKET_CLOSE_RESPONSE`.
- `InvalidOperatingModeException` – if *operating_mode* is not supported.

See also:

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

needs_id()
Override method.

See also:

XBeeAPIPacket.needs_id()

socket_id
Returns the the socket ID.

Returns the socket ID.

Return type Integer

status
Returns the socket close status.

Returns the socket close status.

Return type *SocketStatus*

See also:

SocketStatus

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info of the XBeePacket fields.

Return type Dictionary

static unescape_data(*data*)
Un-escapes the provided bytearray data.

Parameters **data** (*Bytearray*) – the bytearray to unescape.

Returns *data* unescaped.

Return type Bytearray

class digi.xbee.packets.socket.**SocketSendPacket**(*frame_id*, *socket_id*, *payload=None*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents a Socket Send packet. Packet is built using the parameters of the constructor.

A Socket Send message causes the device to transmit data using the current connection. For a nonzero frame ID, this will elicit a Transmit (TX) Status - 0x89 frame (*TransmitStatusPacket*).

This frame requires a successful Socket Connect - 0x42 frame first (*SocketConnectPacket*). For a socket that is not connected, the device responds with a Transmit (TX) Status - 0x89 frame with an error.

See also:

TransmitStatusPacket
XBeeAPIPacket

Class constructor. Instantiates a new *SocketSendPacket* object with the provided parameters.

Parameters

- **frame_id** (*Integer*) – the frame ID of the packet.
- **socket_id** (*Integer*) – the socket identifier.
- **payload** (*Bytearray, optional*) – data that is sent.
- **op_mode** (*OperatingMode*, optional, default=‘OperatingMode.API_MODE‘) – The mode in which the frame was captured.

Raises

- ValueError – if *frame_id* is less than 0 or greater than 255.
- ValueError – if *socket_id* is less than 0 or greater than 255.

See also:

XBeeAPIPacket

static create_packet(*raw*, *operating_mode*)

Override method.

Returns *SocketSendPacket*.

Raises

- `InvalidPacketException` – if the bytearray length is less than 7. (start delim. + length (2 bytes) + frame type + frame id + socket ID + checksum = 7 bytes).
- `InvalidPacketException` – if the length field of 'raw' is different from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is not `ApiFrameType.SOCKET_SEND`.
- `InvalidOperatingModeException` – if *operating_mode* is not supported.

See also:

XBeePacket.create_packet()

`XBeeAPIPacket._check_api_packet()`

needs_id()

Override method.

See also:

XBeeAPIPacket.needs_id()

socket_id

Returns the socket ID.

Returns the socket ID.

Return type Integer

payload

Returns the payload to send.

Returns the payload to send.

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info of the XBeePacket fields.

Return type Dictionary

static unescape_data (*data*)
Un-escapes the provided bytearray data.

Parameters **data** (*Bytearray*) – the bytearray to unescape.

Returns *data* unescaped.

Return type Bytearray

class digi.xbee.packets.socket.**SocketSendToPacket** (*frame_id*, *socket_id*, *dest_address*, *dest_port*, *payload=None*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents a Socket Send packet. Packet is built using the parameters of the constructor.

A Socket SendTo (Transmit Explicit Data) message causes the device to transmit data using an IPv4 address and port. For a non-zero frame ID, this will elicit a Transmit (TX) Status - 0x89 frame (*TransmitStatusPacket*).

If this frame is used with a TCP, SSL, or a connected UDP socket, the address and port fields are ignored.

See also:

TransmitStatusPacket
XBeeAPIPacket

Class constructor. Instantiates a new *SocketSendToPacket* object with the provided parameters.

Parameters

- **frame_id** (*Integer*) – the frame ID of the packet.
- **socket_id** (*Integer*) – the socket identifier.
- **dest_address** (IPv4Address) – IPv4 address of the destination device.
- **dest_port** (*Integer*) – destination port number.
- **payload** (*Bytearray, optional*) – data that is sent.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

Raises

- `ValueError` – if *frame_id* is less than 0 or greater than 255.
- `ValueError` – if *socket_id* is less than 0 or greater than 255.
- `ValueError` – if *dest_port* is less than 0 or greater than 65535.

See also:

XBeeAPIPacket

static create_packet(*raw*, *operating_mode*)

Override method.

Returns *SocketSendToPacket*.

Raises

- `InvalidPacketException` – if the bytearray length is less than 14. (start delim. + length (2 bytes) + frame type + frame id + socket ID + dest address (4 bytes) + dest port (2 bytes) + transmit options + checksum = 14 bytes).
- `InvalidPacketException` – if the length field of 'raw' is different from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is not `ApiFrameType.SOCKET_SENDTO`.
- `InvalidOperatingModeException` – if *operating_mode* is not supported.

See also:

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

needs_id()

Override method.

See also:

XBeeAPIPacket.needs_id()

socket_id

Returns the socket ID.

Returns the socket ID.

Return type Integer

dest_address

Returns the IPv4 address of the destination device.

Returns the IPv4 address of the destination device.

Return type `ipaddress.IPv4Address`

dest_port
Returns the destination port.

Returns the destination port.

Return type Integer

payload
Returns the payload to send.

Returns the payload to send.

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

op_mode

Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info of the XBeePacket fields.

Return type Dictionary

static unescape_data(*data*)

Un-escapes the provided bytearray data.

Parameters **data** (*Bytearray*) – the bytearray to unescape.

Returns *data* unescaped.

Return type Bytearray

class digi.xbee.packets.socket.**SocketBindListenPacket**(*frame_id*, *socket_id*, *src_port*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents a Socket Bind/Listen packet. Packet is built using the parameters of the constructor.

Opens a listener socket that listens for incoming connections.

When there is an incoming connection on the listener socket, a Socket New IPv4 Client - 0xCC frame (*SocketNewIPv4ClientPacket*) is sent, indicating the socket ID for the new connection along with the remote address information.

For a UDP socket, this frame binds the socket to a given port. A bound UDP socket can receive data with a Socket Receive From: IPv4 - 0xCE frame (`SocketReceiveFromIPv4Packet`).

See also:

SocketNewIPv4ClientPacket
SocketReceiveFromIPv4Packet
XBeeAPIPacket

Class constructor. Instantiates a new *SocketBindListenPacket* object with the provided parameters.

Parameters

- **frame_id** (*Integer*) – the frame ID of the packet.
- **socket_id** (*Integer*) – socket ID to listen on.
- **src_port** (*Integer*) – the port to listen on.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

Raises

- ValueError – if *frame_id* is less than 0 or greater than 255.
- ValueError – if *socket_id* is less than 0 or greater than 255.
- ValueError – if *source_port* is less than 0 or greater than 65535.

See also:

XBeeAPIPacket

static create_packet(*raw*, *operating_mode*)
Override method.

Returns *SocketBindListenPacket*.

Raises

- InvalidPacketException – if the bytearray length is less than 9. (start delim. + length (2 bytes) + frame type + frame id + socket ID + source port (2 bytes) + checksum = 9 bytes).
- InvalidPacketException – if the length field of 'raw' is different from 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 from the checksum field value (last byte).
- InvalidPacketException – if the frame type is not ApiFrameType.SOCKET_BIND.
- InvalidOperatingModeException – if *operating_mode* is not supported.

See also:

XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()

needs_id()

Override method.

See also:

XBeeAPIPacket.needs_id()

socket_id

Returns the socket ID.

Returns the socket ID.

Return type Integer

source_port

Returns the source port.

Returns the source port.

Return type Integer

frame_id

Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

get_checksum()

Returns the checksum value of this XBeePacket. The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer

See also:

factory

get_frame_spec_data()

Override method.

See also:

XBeePacket.get_frame_spec_data()

get_frame_type()

Returns the frame type of this packet.

Returns the frame type of this packet.

Return type *ApiFrameType*

See also:

ApiFrameType

get_frame_type_value()
Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer

See also:

ApiFrameType

is_broadcast()
Returns whether this packet is broadcast or not.

Returns *True* if this packet is broadcast, *False* otherwise.

Return type Boolean

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info of the XBeePacket fields.

Return type Dictionary

static unescape_data(*data*)
Un-escapes the provided bytearray data.

Parameters **data** (*Bytearray*) – the bytearray to unescape.

Returns *data* unescaped.

Return type Bytearray

class digi.xbee.packets.socket.**SocketListenResponsePacket**(*frame_id*, *socket_id*, *status*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents a Socket Listen Response packet. Packet is built using the parameters of the constructor.

The device sends this frame in response to a Socket Bind/Listen (0x46) frame (*SocketBindListenPacket*).

See also:

SocketBindListenPacket
XBeeAPIPacket

Class constructor. Instantiates a new *SocketListenResponsePacket* object with the provided parameters.

Parameters

- **frame_id** (*Integer*) – the frame ID of the packet.
- **socket_id** (*Integer*) – socket ID.
- **status** (*SocketStatus*) – socket listen status.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

Raises

- `ValueError` – if *frame_id* is less than 0 or greater than 255.
- `ValueError` – if *socket_id* is less than 0 or greater than 255.

See also:

XBeeAPIPacket
SocketStatus

static create_packet (*raw*, *operating_mode*)
Override method.

Returns *SocketListenResponsePacket*.

Raises

- `InvalidPacketException` – if the bytearray length is less than 8. (start delim. + length (2 bytes) + frame type + frame id + socket ID + status + checksum = 8 bytes).
- `InvalidPacketException` – if the length field of 'raw' is different from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is not `ApiFrameType.SOCKET_LISTEN_RESPONSE`.
- `InvalidOperatingModeException` – if *operating_mode* is not supported.

See also:

XBeePacket.create_packet()

XBeeAPIPacket._check_api_packet()

needs_id()

Override method.

See also:

XBeeAPIPacket.needs_id()

socket_id

Returns the socket ID.

Returns the socket ID.

Return type Integer

status

Returns the socket listen status.

Returns The socket listen status.

Return type *SocketStatus*

See also:

SocketStatus

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

op_mode

Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info of the XBeePacket fields.

Return type Dictionary

static unescape_data(*data*)

Un-escapes the provided bytearray data.

Parameters **data** (*Bytearray*) – the bytearray to unescape.

Returns *data* unescaped.

Return type Bytearray

class digi.xbee.packets.socket.**SocketNewIPv4ClientPacket**(*socket_id*, *client_socket_id*, *remote_address*, *remote_port*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents a Socket New IPv4 Client packet. Packet is built using the parameters of the constructor.

XBee Cellular modem uses this frame when an incoming connection is accepted on a listener socket.

This frame contains the original listener's socket ID and a new socket ID of the incoming connection, along with the connection's remote address information.

See also:

XBeeAPIPacket

Class constructor. Instantiates a new *SocketNewIPv4ClientPacket* object with the provided parameters.

Parameters

- **socket_id** (*Integer*) – the socket ID of the listener socket.
- **client_socket_id** (*Integer*) – the socket ID of the new connection.
- **remote_address** (IPv4Address) – the remote IPv4 address.
- **remote_port** (*Integer*) – the remote port number.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

Raises

- ValueError – if *socket_id* is less than 0 or greater than 255.
- ValueError – if *client_socket_id* is less than 0 or greater than 255.
- ValueError – if *remote_port* is less than 0 or greater than 65535.

See also:

XBeeAPIPacket

static create_packet(*raw*, *operating_mode*)

Override method.

Returns *SocketNewIPv4ClientPacket*.

Raises

- InvalidPacketException – if the bytearray length is less than 13. (start delim. + length (2 bytes) + frame type + socket ID + client socket ID + remote address (4 bytes) + remote port (2 bytes) + checksum = 13 bytes).
- InvalidPacketException – if the length field of 'raw' is different from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is not `ApiFrameType.SOCKET_NEW_IPV4_CLIENT`.
- `InvalidOperatingModeException` – if *operating_mode* is not supported.

See also:

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

needs_id()
Override method.

See also:

XBeeAPIPacket.needs_id()

socket_id
Returns the socket ID.

Returns the socket ID.

Return type Integer

client_socket_id
Returns the client socket ID.

Returns the client socket ID.

Return type Integer

remote_address
Returns the remote IPv4 address.

Returns the remote IPv4 address.

Return type `ipaddress.IPv4Address`

remote_port
Returns the remote port.

Returns the remote port.

Return type Integer

frame_id
Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

get_checksum()

Returns the checksum value of this XBeePacket. The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer

See also:

factory

get_frame_spec_data()

Override method.

See also:

XBeePacket.get_frame_spec_data()

get_frame_type()

Returns the frame type of this packet.

Returns the frame type of this packet.

Return type *ApiFrameType*

See also:

ApiFrameType

get_frame_type_value()

Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer

See also:

ApiFrameType

is_broadcast()

Returns whether this packet is broadcast or not.

Returns *True* if this packet is broadcast, *False* otherwise.

Return type Boolean

op_mode

Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info of the XBeePacket fields.

Return type Dictionary

static unescape_data (*data*)
Un-escapes the provided bytearray data.

Parameters **data** (*Bytearray*) – the bytearray to unescape.

Returns *data* unescaped.

Return type Bytearray

class digi.xbee.packets.socket.**SocketReceivePacket** (*frame_id*, *socket_id*, *payload=None*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents a Socket Receive packet. Packet is built using the parameters of the constructor.

XBee Cellular modem uses this frame when it receives RF data on the specified socket.

See also:

XBeeAPIPacket

Class constructor. Instantiates a new *SocketReceivePacket* object with the provided parameters.

Parameters

- **frame_id** (*Integer*) – the frame ID of the packet.
- **socket_id** (*Integer*) – the ID of the socket the data has been received on.
- **payload** (*Bytearray, optional*) – data that is received.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

Raises

- ValueError – if *frame_id* is less than 0 or greater than 255.
- ValueError – if *socket_id* is less than 0 or greater than 255.

See also:

XBeeAPIPacket

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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 *SocketReceivePacket*.

Raises

- `InvalidPacketException` – if the bytearray length is less than 7. (start delim. + length (2 bytes) + frame type + frame id + socket ID + checksum = 7 bytes).
- `InvalidPacketException` – if the length field of 'raw' is different from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is not `ApiFrameType.SOCKET_RECEIVE`.
- `InvalidOperatingModeException` – if *operating_mode* is not supported.

See also:

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

needs_id()

Override method.

See also:

XBeeAPIPacket.needs_id()

socket_id

Returns the socket ID.

Returns the socket ID.

Return type Integer

payload

Returns the payload that was received.

Returns the payload that was received.

Return type Bytearray

class digi.xbee.packets.socket.**SocketReceiveFromPacket**(*frame_id*, *socket_id*, *src_address*, *src_port*, *payload=None*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents a Socket Receive From packet. Packet is built using the parameters of the constructor.

XBee Cellular modem uses this frame when it receives RF data on the specified socket. The frame also contains addressing information about the source.

See also:

XBeeAPIPacket

Class constructor. Instantiates a new *SocketReceiveFromPacket* object with the provided parameters.

Parameters

- **frame_id** (*Integer*) – the frame ID of the packet.
- **socket_id** (*Integer*) – the ID of the socket the data has been received on.
- **src_address** (IPv4Address) – IPv4 address of the source device.
- **src_port** (*Integer*) – source port number.
- **payload** (*Bytearray, optional*) – data that is received.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

Raises

- ValueError – if *frame_id* is less than 0 or greater than 255.
- ValueError – if *socket_id* is less than 0 or greater than 255.
- ValueError – if *source_port* is less than 0 or greater than 65535.

See also:

XBeeAPIPacket

frame_id
Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

get_checksum()
Returns the checksum value of this XBeePacket. The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer

See also:

factory

get_frame_spec_data()
Override method.

See also:

XBeePacket.get_frame_spec_data()

get_frame_type()
Returns the frame type of this packet.

Returns the frame type of this packet.

Return type *ApiFrameType*

See also:

ApiFrameType

get_frame_type_value()
Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer

See also:

ApiFrameType

is_broadcast()

Returns whether this packet is broadcast or not.

Returns *True* if this packet is broadcast, *False* otherwise.

Return type Boolean

op_mode

Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info of the XBeePacket fields.

Return type Dictionary

static unescape_data(*data*)

Un-escapes the provided bytearray data.

Parameters **data** (*Bytearray*) – the bytearray to unescape.

Returns *data* unescaped.

Return type Bytearray

static create_packet(*raw*, *operating_mode*)

Override method.

Returns *SocketReceiveFromPacket*.

Raises

- `InvalidPacketException` – if the bytearray length is less than 13. (start delim. + length (2 bytes) + frame type + frame id + socket ID + source address (4 bytes) + source port (2 bytes) + status + Checksum = 14 bytes).
- `InvalidPacketException` – if the length field of 'raw' is different from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is not `ApiFrameType.SOCKET_RECEIVE_FROM`.
- `InvalidOperatingModeException` – if *operating_mode* is not supported.

See also:

XBeePacket.create_packet()
XBeeAPIPacket._check_api_packet()

needs_id()
Override method.

See also:

XBeeAPIPacket.needs_id()

socket_id
Returns the socket ID.

Returns the socket ID.

Return type Integer

source_address
Returns the IPv4 address of the source device.

Returns the IPv4 address of the source device.

Return type ipaddress.IPv4Address

source_port
Returns the source port.

Returns the source port.

Return type Integer

payload
Returns the payload to send.

Returns the payload that has been received.

Return type Bytearray

class digi.xbee.packets.socket.**SocketStatePacket**(*socket_id*, *state*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents a Socket State packet. Packet is built using the parameters of the constructor.

This frame is sent out the device's serial port to indicate the state related to the socket.

See also:

XBeeAPIPacket

Class constructor. Instantiates a new *SocketStatePacket* object with the provided parameters.

Parameters

- **socket_id** (*Integer*) – the socket identifier.
- **state** (*SocketState*) – socket status.

- **`op_mode`** (*`OperatingMode`*, optional, default=‘OperatingMode.API_MODE‘) – The mode in which the frame was captured.

Raises `ValueError` – if *socket_id* is less than 0 or greater than 255.

See also:

`SockeState`
`XBeeAPIPacket`

`frame_id`
Returns the frame ID of the packet.

Returns the frame ID of the packet.

Return type Integer

`get_checksum``()`
Returns the checksum value of this XBeePacket. The checksum is the last 8 bits of the sum of the bytes between the length field and the checksum field.

Returns checksum value of this XBeePacket.

Return type Integer

See also:

`factory`

`get_frame_spec_data``()`
Override method.

See also:

`XBeePacket.get_frame_spec_data()`

`get_frame_type``()`
Returns the frame type of this packet.

Returns the frame type of this packet.

Return type *`ApiFrameType`*

See also:

`ApiFrameType`

`get_frame_type_value``()`
Returns the frame type integer value of this packet.

Returns the frame type integer value of this packet.

Return type Integer

See also:

ApiFrameType

is_broadcast()
Returns whether this packet is broadcast or not.

Returns *True* if this packet is broadcast, *False* otherwise.

Return type Boolean

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info of the XBeePacket fields.

Return type Dictionary

static unescape_data(*data*)
Un-escapes the provided bytearray data.

Parameters **data** (*Bytearray*) – the bytearray to unescape.

Returns *data* unescaped.

Return type Bytearray

static create_packet(*raw*, *operating_mode*)
Override method.

Returns *SocketStatePacket*.

Raises

- `InvalidPacketException` – if the bytearray length is less than 7. (start delim. + length (2 bytes) + frame type + socket ID + state + checksum = 7 bytes).
- `InvalidPacketException` – if the length field of 'raw' is different from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is not `ApiFrameType.SOCKET_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()`

`socket_id`

Returns the socket ID.

Returns the socket ID.

Return type Integer

`state`

Returns the socket state.

Returns The socket state.

Return type *`SocketState`*

See also:

`SocketState`

digi.xbee.packets.wifi module

class `digi.xbee.packets.wifi.`**`IODataSampleRxIndicatorWifiPacket`**`(`*src_address*, *rssi*, *rx_options*, *rf_data=None*, *op_mode=<OperatingMode.API_MODE* *(1, 'API mode')>*`)`

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

- **src_address** (ipaddress.IPv4Address) – the 64-bit source address.
- **rssi** (*Integer*) – received signal strength indicator.
- **rx_options** (*Integer*) – bitfield indicating the receive options.
- **rf_data** (*Bytearray, optional*) – received RF data.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

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 from 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 from 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
Returns the IPv4 address of the source device.

Returns the IPv4 address of the source device.

Return type `ipaddress.IPv4Address`

See also:

`ipaddress.IPv4Address`

rssi
Returns the received Signal Strength Indicator (RSSI).

Returns the received Signal Strength Indicator (RSSI).

Return type Integer

receive_options
Returns the receive options bitfield.

Returns the receive options bitfield.

Return type Integer

See also:

ReceiveOptions

rf_data
Returns the received RF data.

Returns the received RF data.

Return type Bytearray

io_sample
Returns the IO sample corresponding to the data contained in the packet.

Returns

the IO sample of the packet, *None* if the packet has not any data or if the sample could not be generated correctly.

Return type *IOSample*

See also:

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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, tx_options, command, parameter=None, op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

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.
- **tx_options** (*Integer*) – bitfield of supported transmission options.
- **command** (*String*) – AT command to send.
- **parameter** (*Bytearray, optional*) – AT command parameter.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

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 from 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 from 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()
`XBeeAPIPacket._check_api_packet()`

needs_id()

Override method.

See also:

XBeeAPIPacket.needs_id()

dest_address
Returns the IPv4 address of the destination device.

Returns the IPv4 address of the destination device.

Return type `ipaddress.IPv4Address`

See also:

`ipaddress.IPv4Address`

transmit_options
Returns the transmit options bitfield.

Returns the transmit options bitfield.

Return type Integer

See also:

RemoteATCmdOptions

command
Returns the AT command.

Returns the AT command.

Return type String

parameter
Returns the AT command parameter.

Returns the AT command parameter.

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info of the XBeePacket fields.

Return type Dictionary

static unescape_data(*data*)
Un-escapes the provided bytearray data.

Parameters **data** (`Bytearray`) – the bytearray to unescape.

Returns *data* unescaped.

Return type Bytearray

class digi.xbee.packets.wifi.**RemoteATCommandResponseWifiPacket**(*frame_id*, *src_address*, *command*, *resp_status*, *comm_value=None*, *op_mode=<OperatingMode.API_MODE (1, 'API mode')>*)

Bases: `digi.xbee.packets.base.XBeeAPIPacket`

This class represents a remote AT command response (Wi-Fi) packet. Packet is built using the parameters of the constructor or providing a valid API payload.

If a module receives a remote command response RF data frame in response to a Remote AT Command Request, the module will send a Remote AT Command Response message out the UART. Some commands may send back multiple frames for example, Node Discover (*ND*) command.

This packet is received in response of a `RemoteATCommandPacket`.

Response also includes an `ATCommandStatus` object with the status of the AT command.

See also:

`RemoteATCommandWifiPacket`
`ATCommandStatus`
`XBeeAPIPacket`

Class constructor. Instantiates a new `RemoteATCommandResponseWifiPacket` object with the provided parameters.

Parameters

- **frame_id** (`Integer`) – the frame ID of the packet.
- **src_address** (`ipaddress.IPv4Address`) – the IPv4 address of the source device.
- **command** (`String`) – the AT command of the packet. Must be a string.
- **resp_status** (`ATCommandStatus`) – the status of the AT command.
- **comm_value** (`Bytearray, optional`) – the AT command response value.
- **op_mode** (`OperatingMode`, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

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

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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 *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 from 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 from 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()

source_address

Returns the IPv4 address of the source device.

Returns the IPv4 address of the source device.

Return type `ipaddress.IPv4Address`

See also:

`ipaddress.IPv4Address`

command

Returns the AT command of the packet.

Returns the AT command of the packet.

Return type String

status

Returns the AT command response status of the packet.

Returns the AT command response status of the packet.

Return type *ATCommandStatus*

See also:

ATCommandStatus

command_value

Returns the AT command response value.

Returns the AT command response value.

Return type Bytearray

digi.xbee.packets.zigbee module

class digi.xbee.packets.zigbee.**RegisterJoiningDevicePacket**(*frame_id*, *registrant_address*, *options*, *key*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents a Register Joining Device packet. Packet is built using the parameters of the constructor or providing a valid API payload.

Use this frame to securely register a joining device to a trust center. Registration is the process by which a node is authorized to join the network using a preconfigured link key or installation code that is conveyed to the trust center out-of-band (using a physical interface and not over-the-air).

If registering a device with a centralized trust center (EO = 2), then the key entry will only persist for KT seconds before expiring.

Registering devices in a distributed trust center (EO = 0) is persistent and the key entry will never expire unless explicitly removed.

To remove a key entry on a distributed trust center, this frame should be issued with a null (None) key. In a centralized trust center you cannot use this method to explicitly remove the key entries.

See also:

XBeeAPIPacket

Class constructor. Instantiates a new *RegisterJoiningDevicePacket* object with the provided parameters.

Parameters

- **frame_id** (*integer*) – the frame ID of the packet.
- **registrant_address** (*XBee64BitAddress*) – the 64-bit address of the destination device.
- **options** (*RegisterKeyOptions*) – the register options indicating the key source.
- **key** (*Bytearray*) – key of the device to register. Up to 16 bytes if entering a Link Key or up to 18 bytes (16-byte code + 2 byte CRC) if entering an Install Code.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

Raises ValueError – if *frame_id* is less than 0 or greater than 255.

See also:

XBee64BitAddress
XBeeAPIPacket
RegisterKeyOptions

static create_packet(*raw*, *operating_mode*)

Override method.

Returns `RegisterJoiningDevicePacket`.

Raises

- `InvalidPacketException` – if the bytearray length is less than 17. (start delim. + length (2 bytes) + frame type + frame id + 64-bit registrant addr. (8 bytes) + 16-bit registrant addr. (2 bytes) + options + checksum = 17 bytes).
- `InvalidPacketException` – if the length field of 'raw' is different from 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 from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is not `ApiFrameType.REGISTER_JOINING_DEVICE`.
- `InvalidOperatingModeException` – if *operating_mode* is not supported.

See also:

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

needs_id()

Override method.

See also:

`XBeeAPIPacket.needs_id()`

registrant_address

Returns the 64-bit registrant address.

Returns the 64-bit registrant address.

Return type `XBee64BitAddress`

See also:

`XBee64BitAddress`

options

Returns the register options value.

Returns the register options indicating the key source.

Return type `RegisterKeyOptions`

See also:

RegisterKeyOptions

key

Returns the register key.

Returns the register key.

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info of the XBeePacket fields.

Return type Dictionary

static unescape_data(*data*)
Un-escapes the provided bytearray data.

Parameters data (*Bytearray*) – the bytearray to unescape.

Returns *data* unescaped.

Return type Bytearray

class digi.xbee.packets.zigbee.**RegisterDeviceStatusPacket**(*frame_id*, *status*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents a Register Device Status packet. Packet is built using the parameters of the constructor or providing a valid API payload.

This frame is sent out of the UART of the trust center as a response to a 0x24 Register Device frame, indicating whether the registration was successful or not.

See also:

RegisterJoiningDevicePacket
XBeeAPIPacket

Class constructor. Instantiates a new *RegisterDeviceStatusPacket* object with the provided parameters.

Parameters

- **frame_id** (*integer*) – the frame ID of the packet.
- **status** (*ZigbeeRegisterStatus*) – status of the register device operation.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

Raises `ValueError` – if *frame_id* is less than 0 or greater than 255.

See also:

XBeeAPIPacket
ZigbeeRegisterStatus

static create_packet(*raw*, *operating_mode*)
Override method.

Returns *RegisterDeviceStatusPacket*.

Raises

- `InvalidPacketException` – if the bytearray length is less than 17. (start delim. + length (2 bytes) + frame type + frame id + status + checksum = 7 bytes).
- `InvalidPacketException` – if the length field of 'raw' is different from its real length. (length field: bytes 1 and 3)
- `InvalidPacketException` – if the first byte of 'raw' is not the header byte. See *SpecialByte*.
- `InvalidPacketException` – if the calculated checksum is different from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is not `ApiFrameType.REGISTER_JOINING_DEVICE_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
Returns the register device status.

Returns the register device status.

Return type *ZigbeeRegisterStatus*

See also:

ZigbeeRegisterStatus

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info of the XBeePacket fields.

Return type Dictionary

static unescape_data(*data*)
Un-escapes the provided bytearray data.

Parameters **data** (*Bytearray*) – the bytearray to unescape.

Returns *data* unescaped.

Return type Bytearray

class digi.xbee.packets.zigbee.**RouteRecordIndicatorPacket**(*x64bit_addr*, *x16bit_addr*, *rx_opts*, *hops=None*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents a Zigbee Route Record Indicator packet. Packet is built using the parameters of the constructor or providing a valid API payload.

The route record indicator is received whenever a device sends a Zigbee route record command. This is used with many-to-one routing to create source routes for devices in a network.

Among received data, some options can also be received indicating transmission parameters.

See also:

ReceiveOptions
XBeeAPIPacket

Class constructor. Instantiates a new *RouteRecordIndicatorPacket* object with the provided parameters.

Parameters

- **x64bit_addr** (*XBee64BitAddress*) – The 64-bit source address.
- **x16bit_addr** (*XBee16BitAddress*) – The 16-bit source address.
- **rx_opts** (*Integer*) – Bitfield indicating the receive options.
- **hops** (*List, optional, default=`None`*) – List of 16-bit address of intermediate hops in the source route (excluding source and destination).
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

See also:

ReceiveOptions
XBee16BitAddress
XBee64BitAddress
XBeeAPIPacket

static create_packet(*raw*, *operating_mode*)
Override method.

Returns *RouteRecordIndicatorPacket*.

Raises

- `InvalidPacketException` – If the bytearray length is less than 17. (start delim. + length (2 bytes) + frame type + 64bit addr. + 16bit addr. + Receive options + num of addrs + checksum = 17 bytes).
- `InvalidPacketException` – If the length field of *raw* is different from its real length. (length field: bytes 1 and 3)
- `InvalidPacketException` – If the first byte of 'raw' is not the header byte. See *SpecialByte*.
- `InvalidPacketException` – If the calculated checksum is different from the checksum field value (last byte).
- `InvalidPacketException` – If the frame type is not `ApiFrameType.ROUTE_RECORD_INDICATOR`.
- `InvalidPacketException` – If the number of hops does not match with the number of 16-bit addresses.
- `InvalidOperatingModeException` – If *operating_mode* is not supported.

See also:

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

needs_id()
Override method.

See also:

XBeeAPIPacket.needs_id()

is_broadcast()

Override method.

See also:

XBeeAPIPacket.is_broadcast()

x64bit_source_addr

Returns the 64-bit source address.

Returns The 64-bit source address.

Return type *XBee64BitAddress*

See also:

XBee64BitAddress

x16bit_source_addr

Returns the 16-bit source address.

Returns The 16-bit source address.

Return type *XBee16BitAddress*

See also:

XBee16BitAddress

receive_options

Returns the receive options bitfield.

Returns The receive options bitfield.

Return type Integer

See also:

ReceiveOptions

number_of_hops

Returns the number of intermediate hops in the source route (excluding source and destination).

Returns The number of addresses.

Return type Integer

hops

Returns the list of intermediate hops starting from the closest to destination hop and finishing with the closest to the source (excluding source and destination).

Returns The list of 16-bit addresses of intermediate hops.

Return type List

See also:

XBee16BitAddress

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info of the XBeePacket fields.

Return type Dictionary

static unescape_data (*data*)
Un-escapes the provided bytearray data.

Parameters **data** (*Bytearray*) – the bytearray to unescape.

Returns *data* unescaped.

Return type Bytearray

class digi.xbee.packets.zigbee.**CreateSourceRoutePacket** (*frame_id*, *x64bit_addr*, *x16bit_addr*, *route_options=0*, *hops=None*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents a Zigbee Create Source Route packet. This packet is built using the parameters of the constructor or providing a valid API payload.

This frame creates a source route in the node. A source route specifies the complete route a packet should travese to get from source to destination. Source routing should be used with many-to-one routing for best results.

Note: Both, 64-bit and 16-bit destination addresses are required when creating a source route. These are obtained when a Route Record Indicator (0xA1) frame is received.

See also:

RouteRecordIndicatorPacket
XBeeAPIPacket

Class constructor. Instantiates a new *CreateSourceRoutePacket* 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.
- **route_options** (*Integer*) – Route command options.
- **hops** (*List, optional, default=`None`*) – List of 16-bit addresses of intermediate hops in the source route (excluding source and destination).
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

See also:

XBee16BitAddress
XBee64BitAddress
XBeeAPIPacket

static create_packet(*raw*, *operating_mode*)

Override method.

Returns *CreateSourceRoutePacket*.

Raises

- `InvalidPacketException` – If the bytearray length is less than 18. (start delim. + length (2 bytes) + frame type + frame id + 64-bit addr. + 16-bit addr. + Route command options + num of addrs + hops 16-bit addrs + checksum = 18 bytes).
- `InvalidPacketException` – If the length field of *raw* is different from its real length. (length field: bytes 1 and 3)
- `InvalidPacketException` – If the first byte of 'raw' is not the header byte. See *SpecialByte*.
- `InvalidPacketException` – If the calculated checksum is different from the checksum field value (last byte).
- `InvalidPacketException` – If the frame type is not `ApiFrameType.CREATE_SOURCE_ROUTE`.
- `InvalidPacketException` – If the number of hops does not match with the number of 16-bit addresses.
- `InvalidOperatingModeException` – If *operating_mode* is not supported.

See also:

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

needs_id()

Override method.

See also:

XBeeAPIPacket.needs_id()

x64bit_dest_addr

Returns the 64-bit destination address.

Returns The 64-bit destination address.

Return type *XBee64BitAddress*

See also:

XBee64BitAddress

x16bit_dest_addr

Returns the 16-bit destination address.

Returns The 16-bit destination address.

Return type *XBee16BitAddress*

See also:

XBee16BitAddress

route_cmd_options

Returns the route command options bitfield.

Returns The route command options bitfield.

Return type Integer

number_of_hops

Returns the number of intermediate hops in the source route (excluding source and destination).

Returns The number of intermediate hops.

Return type Integer

hops

Returns the list of intermediate hops starting from the closest to destination hop and finishing with the closest to the source (excluding source and destination).

Returns The list of 16-bit addresses of intermediate hops.

Return type List

See also:

XBee16BitAddress

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info of the XBeePacket fields.

Return type Dictionary

static unescape_data(*data*)
Un-escapes the provided bytearray data.

Parameters **data** (*Bytearray*) – the bytearray to unescape.

Returns *data* unescaped.

Return type Bytearray

class digi.xbee.packets.zigbee.**OTAFirmwareUpdateStatusPacket**(*src_address_64*, *up-dater_address_16*, *rx_options*, *msg_type*, *block_number*, *target_address_64*, *op_mode=<OperatingMode.API_MODE: (1, 'API mode')>*)

Bases: *digi.xbee.packets.base.XBeeAPIPacket*

This class represents a an Over The Air Firmware Update Status packet. Packet is built using the parameters of the constructor or providing a valid API payload.

This frame provides a status indication of a firmware update transmission.

If a query request returns a 0x15 (NACK) status, the target is likely waiting for a firmware update image. If no messages are sent to it for about 75 seconds, the target will timeout and accept new query messages.

If a query status returns a 0x51 (QUERY) status, then the target's bootloader is not active and will not respond to query messages.

See also:

EmberBootloaderMessageType
XBeeAPIPacket

Class constructor. Instantiates a new *OTAFirmwareUpdateStatusPacket* object with the provided parameters.

Parameters

- **src_address_64** (*XBee64BitAddress*) – the 64-bit address of the device returning this answer.
- **updater_address_16** (*XBee16BitAddress*) – the 16-bit address of the updater device.
- **rx_options** (*Integer*) – bitfield indicating the receive options.
- **msg_type** (*EmberBootloaderMessageType*) – Ember bootloader message type
- **block_number** (*Integer*) – block number used in the update request.
- **target_address_64** (*XBee64BitAddress*) – the 64-bit address of the device that is being updated.
- **op_mode** (*OperatingMode*, optional, default='OperatingMode.API_MODE') – The mode in which the frame was captured.

See also:

XBeeAPIPacket
XBee16BitAddress
XBee64BitAddress
ReceiveOptions
EmberBootloaderMessageType

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

op_mode
Retrieves the operating mode in which this packet was read.

Returns The operating mode.

Return type *OperatingMode*

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 must be escaped.

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 info 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 *OTAFirmwareUpdateStatusPacket*.

Raises

- `InvalidPacketException` – if the bytearray length is less than 17. (start delim. + length (2 bytes) + frame type + source 64bit addr. (8 bytes) + updater 16bit addr. (2 bytes) + receive options + bootloader message type + block number + source 64bit addr. (8 bytes) + checksum = 27 bytes).
- `InvalidPacketException` – if the length field of 'raw' is different from its real length. (length field: bytes 1 and 3)
- `InvalidPacketException` – if the first byte of 'raw' is not the header byte. See *SpecialByte*.
- `InvalidPacketException` – if the calculated checksum is different from the checksum field value (last byte).
- `InvalidPacketException` – if the frame type is not `ApiFrameType.OTA_FIRMWARE_UPDATE_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()

x64bit_source_addr

Returns the 64-bit source address.

Returns the 64-bit source address.

Return type *XBee64BitAddress*

See also:

XBee64BitAddress

x16bit_updater_addr

Returns the 16-bit updater address.

Returns the 16-bit updater address.

Return type *XBee16BitAddress*

See also:

XBee16BitAddress

receive_options
Returns the receive options bitfield.

Returns the receive options bitfield.

Return type Integer

See also:

ReceiveOptions

bootloader_msg_type
Returns the bootloader message type.

Returns the bootloader message type.

Return type *EmberBootloaderMessageType*

See also:

EmberBootloaderMessageType

block_number
Returns the block number of the request.

Returns the block number of the request.

Return type Integer

x64bit_target_addr
Returns the 64-bit target address.

Returns the 64-bit target address.

Return type *XBee64BitAddress*

See also:

XBee64BitAddress

digi.xbee.packets.factory module

This module provides functionality to build XBee packets from bytearray returning the appropriate XBeePacket subclass.

All the API and API2 logic is already included so all packet reads are independent of the XBee operating mode.

Two API modes are supported and both can be enabled using the *AP* (API Enable) command:

API1 - API Without Escapes The data frame structure is defined as follows:

```
  Start Delimiter          Length                   Frame Data                   ␣
↪Checksum
      (Byte 1)            (Bytes 2-3)               (Bytes 4-n)                (Byte␣
↪n + 1)
+-----------------+  +-------------------+  +--------------------------- +  +----------
↪------+
|       0x7E      |  |   MSB   |   LSB   |  |   API-specific Structure   |  |     1␣
↪Byte     |
+-----------------+  +-------------------+  +----------------------------+  +----------
↪------+
               MSB = Most Significant Byte, LSB = Least Significant Byte
```

API2 - API With Escapes The data frame structure is defined as follows:

```
  Start Delimiter          Length                   Frame Data                   ␣
↪Checksum
      (Byte 1)            (Bytes 2-3)               (Bytes 4-n)                (Byte␣
↪n + 1)
+-----------------+  +-------------------+  +--------------------------- +  +----------
↪------+
|       0x7E      |  |   MSB   |   LSB   |  |   API-specific Structure   |  |     1␣
↪Byte     |
+-----------------+  +-------------------+  +----------------------------+  +----------
↪------+
                     \___________________________________  _____________________________
↪_____/
                                                         \/
                                             Characters Escaped If Needed

               MSB = Most Significant Byte, LSB = Least Significant Byte
```

When sending or receiving an API2 frame, specific data values must be escaped (flagged) so they do not interfere with the data frame sequencing. To escape an interfering data byte, the byte 0x7D is inserted before the byte to be escaped XOR'd with 0x20.

The data bytes that need to be escaped:

- *0x7E* - Frame Delimiter - `SpecialByte`.
- *0x7D* - Escape
- *0x11* - XON
- *0x13* - XOFF

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

The frame data forms an API-specific structure as follows:

```
  Start Delimiter          Length                   Frame Data                   ␣
↪Checksum
      (Byte 1)            (Bytes 2-3)               (Bytes 4-n)                (Byte␣
↪n + 1)
+-----------------+  +-------------------+  +--------------------------- +  +----------
↪------+
|       0x7E      |  |   MSB   |   LSB   |  |   API-specific Structure   |  |     1␣
↪Byte     |
```

(continues on next page)

(continued from previous page)

```
+-----------------+  +-------------------+  +---------------------------+  +----------
↪------+
                                          /                                          ␣
↪       \
                                         /  API Identifier        Identifier␣
↪specific data   \
                                         +------------------+  +---------------------
↪---------+
                                         |       cmdID      |  |           cmdData  ␣
↪         |
                                         +------------------+  +---------------------
↪---------+
```

The cmdID frame (API-identifier) indicates which API messages will be contained in the cmdData frame (Identifier-specific data).

To unit_test data integrity, a checksum is calculated and verified on non-escaped data.

See also:

XBeePacket
OperatingMode

`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.

Parameters

- **number** (*Integer*) – the number to check if a bit is enabled.
- **position** (*Integer*) – the position of the bit to check if is enabled in *number*.

Returns

***True* if the bit located at *position* within *number* is** enabled, *False* otherwise.

Return type Boolean

`digi.xbee.util.utils.`**`get_int_from_byte`**(*number*, *offset*, *length*)

Reads an integer value from the given byte using the provived bit offset and length.

Parameters

- **`number`** (*Integer*) – Byte to read the integer from.
- **`offset`** (*Integer*) – Bit offset inside the byte to start reading (LSB = 0, MSB = 7).
- **`length`** (*Integer*) – Number of bits to read.

Returns The integer value read.

Return type Integer

Raises `ValueError` – If *number is lower than 0 or higher than 255. If ‘offset* is lower than 0 or higher than 7. If *length* is lower than 0 or higher than 8. If *offset + length* is higher than 8.

`digi.xbee.util.utils.`**`hex_string_to_bytes`**(*hex_string*)

Converts a String (composed by hex. digits) into a bytearray with same digits.

Parameters **`hex_string`** (*String*) – String (made by hex. digits) with “0x” header or not.

Returns bytearray containing the numeric value of the hexadecimal digits.

Return type Bytearray

Raises `ValueError` – if invalid literal for int() with base 16 is provided.

Example

```
>>> a = "0xFFFE"
>>> for i in hex_string_to_bytes(a): print(i)
255
254
>>> print(type(hex_string_to_bytes(a)))
<type 'bytearray'>
```

```
>>> b = "FFFE"
>>> for i in hex_string_to_bytes(b): print(i)
255
254
>>> print(type(hex_string_to_bytes(b)))
<type 'bytearray'>
```

`digi.xbee.util.utils.`**`int_to_bytes`**(*number*, *num_bytes=None*)

Converts the provided integer into a bytearray.

If *number* has less bytes than *num_bytes*, the resultant bytearray is filled with zeros (0x00) starting at the beginning.

If *number* has more bytes than *num_bytes*, the resultant bytearray is returned without changes.

Parameters

- **`number`** (*Integer*) – the number to convert to a bytearray.
- **`num_bytes`** (*Integer*) – the number of bytes that the resultant bytearray will have.

Returns the bytearray corresponding to the provided number.

Return type Bytearray

Example

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

`digi.xbee.util.utils.`**`length_to_int`**`(`*byte_array*`)`

Calculates the length value for the given length field of a packet. Length field are bytes 1 and 2 of any packet.

Parameters **`byte_array`** (*`Bytearray`*) – length field of a packet.

Returns the length value.

Return type Integer

Raises `ValueError` – if *byte_array* is not a valid length field (it has length distinct than 0).

Example

```
>>> b = bytearray([13,14])
>>> c = length_to_int(b)
>>> print("0x%02X" % c)
0x1314
>>> print(c)
4884
```

`digi.xbee.util.utils.`**`bytes_to_int`**`(`*byte_array*`)`

Converts the provided bytearray in an Integer. This integer is result of concatenate all components of *byte_array* and convert that hex number to a decimal number.

Parameters **`byte_array`** (*`Bytearray`*) – bytearray to convert in integer.

Returns the integer corresponding to the provided bytearray.

Return type Integer

Example

```
>>> 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
```

`digi.xbee.util.utils.`**`ascii_to_int`**`(`*array*`)`

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.

Parameters **`array`** (*`Bytearray`*) – bytearray to convert in integer.

Example

```
>>> x = bytearray( [0x31,0x30,0x30] )  #0x31 => ASCII code for number 1.
                                       #0x31,0x30,0x30 <==> 1,0,0
>>> print(ascii_to_int(x))
100
```

`digi.xbee.util.utils.`**`int_to_ascii`**(*number*)

Converts an integer number to a bytearray. Each element of the bytearray is the ASCII code that corresponds to the digit of its position.

Parameters **`number`** (*`Integer`*) – the number to convert to an ASCII bytearray.

Returns the bytearray containing the ASCII value of each digit of the number.

Return type Bytearray

Example

```
>>> 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 The bytearray.

Return type Bytearray

Raises `ValueError` – if *number* is less than 0 or greater than 0xFFFF.

Example

```
>>> a = 0
>>> print(hex_to_string(int_to_length(a)))
00 00
```

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

```
>>> a = 200
>>> print(hex_to_string(int_to_length(a)))
00 C8
```

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

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

`digi.xbee.util.utils.`**`hex_to_string`**(*byte_array*, *pretty=True*)

Returns the provided bytearray in a pretty string format. All bytes are separated by blank spaces and printed in hex format.

Parameters

- **`byte_array`** (*`Bytearray`*) – the bytearray to print in pretty string.
- **`pretty`** (*`Boolean, optional`*) – *True* for pretty string format, *False* for plain string format. Default to *True*.

Returns the bytearray formatted in a string format.

Return type String

`digi.xbee.util.utils.`**`doc_enum`**(*enum_class*, *descriptions=None*)

Returns a string with the description of each value of an enumeration.

Parameters

- **`enum_class`** (*`Enumeration`*) – the Enumeration to get its values documentation.
- **`descriptions`** (*`dictionary`*) – each enumeration's item description. The key is the enumeration element name and the value is the description.

Returns the string listing all the enumeration values and their descriptions.

Return type String

`digi.xbee.util.utils.`**`enable_logger`**(*name*, *level=10*)

Enables a logger with the given name and level.

Parameters

- **`name`** (*`String`*) – name of the logger to enable.
- **`level`** (*`Integer`*) – logging level value.

Assigns a default formatter and a default handler (for console).

`digi.xbee.util.utils.`**`disable_logger`**(*name*)

Disables the logger with the give name.

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

`digi.xbee.util.utils.`**`deprecated`**(*version*, *details='None'*)

Decorates a method to mark as deprecated. This adds a deprecation note to the method docstring and also raises a `warning.DeprecationWarning`.

Parameters

- **`version`** (*`String`*) – Version that deprecates this feature.
- **`details`** (*`String, optional, default=`None``*) – Extra details to be added to the method docstring and warning.

digi.xbee.util.xmodem module

`exception` `digi.xbee.util.xmodem.`**`XModemException`**

Bases: `Exception`

This exception will be thrown when any problem related with the XModem/YModem transfer occurs.

All functionality of this class is the inherited from Exception.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception digi.xbee.util.xmodem.**XModemCancelException**
Bases: *digi.xbee.util.xmodem.XModemException*

This exception will be thrown when the XModem/YModem transfer is cancelled by the remote end.

All functionality of this class is the inherited from Exception.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

digi.xbee.util.xmodem.**send_file_xmodem**(*src_path*, *write_cb*, *read_cb*, *progress_cb=None*, *log=None*)
Sends a file using the XModem protocol to a remote end.

Parameters

- **src_path** (*String*) – absolute path of the file to transfer.
- **write_cb** (*Function*) – function to execute in order to write data to the remote end. Takes the following arguments:
  - The data to write as byte array.

  The function returns the following:

  Boolean: *True* if the write succeeded, *False* otherwise.
- **read_cb** (*Function*) – function to execute in order to read data from the remote end. Takes the following arguments:
  - The size of the data to read.
  - The timeout to wait for data. (seconds)

  The function returns the following:

  Bytearray: the read data, *None* if data could not be read
- **progress_cb** (*Function, optional*) – function to execute in order to receive progress information. Takes the following arguments:
  - The progress percentage as integer.
- **log** (Logger, optional) – logger used to log transfer debug messages

Raises

- ValueError – if any input value is not valid.
- *XModemCancelException* – if the transfer is cancelled by the remote end.
- *XModemException* – if there is any error during the file transfer.

digi.xbee.util.xmodem.**send_file_ymodem**(*src_path*, *write_cb*, *read_cb*, *progress_cb=None*, *log=None*)
Sends a file using the YModem protocol to a remote end.

Parameters

- **src_path** (*String*) – absolute path of the file to transfer.

- **write_cb** (*Function*) – function to execute in order to write data to the remote end. Takes the following arguments:
  - The data to write as byte array.

  The function returns the following:

  Boolean: *True* if the write succeeded, *False* otherwise
- **read_cb** (*Function*) – function to execute in order to read data from the remote end. Takes the following arguments:
  - The size of the data to read.
  - The timeout to wait for data. (seconds)

  The function returns the following:

  Bytearray: the read data, *None* if data could not be read
- **progress_cb** (*Function, optional*) – function to execute in order to receive progress information. Takes the following arguments:
  - The progress percentage as integer.
- **log** (Logger, optional) – logger used to log transfer debug messages

Raises

- ValueError – if any input value is not valid.
- *XModemCancelException* – if the transfer is cancelled by the remote end.
- *XModemException* – if there is any error during the file transfer.

digi.xbee.util.xmodem.**get_file_ymodem**(*dest_path*, *write_cb*, *read_cb*, *crc=True*, *progress_cb=None*, *log=None*)

Retrieves a file using the YModem protocol from a remote end.

Parameters

- **dest_path** (*String*) – absolute path to store downloaded file in.
- **write_cb** (*Function*) – function to execute in order to write data to the remote end. Takes the following arguments:
  - The data to write as byte array.

  The function returns the following:

  Boolean: *True* if the write succeeded, *False* otherwise
- **read_cb** (*Function*) – function to execute in order to read data from the remote end. Takes the following arguments:
  - The size of the data to read.
  - The timeout to wait for data. (seconds)

  The function returns the following:

  Bytearray: the read data, *None* if data could not be read
- **crc** (*Boolean, optional*) – *True* to use 16-bit CRC verification, *False* for standard 1 byte checksum. Defaults to *True*.
- **progress_cb** (*Function, optional*) – function to execute in order to receive progress information. Takes the following arguments:

  – The progress percentage as integer.
- **log** (Logger, optional) – logger used to log download debug messages

Raises

- ValueError – if any input value is not valid.
- *XModemCancelException* – if the file download is cancelled by the remote end.
- *XModemException* – if there is any error during the file download process.

Submodules

digi.xbee.comm_interface module

class digi.xbee.comm_interface.**XBeeCommunicationInterface**

Bases: object

This class represents the way the communication with the local XBee is established.

open()

Establishes the underlying hardware communication interface.

Subclasses may throw specific exceptions to signal implementation specific errors.

close()

Terminates the underlying hardware communication interface.

Subclasses may throw specific exceptions to signal implementation specific hardware errors.

is_interface_open

Returns whether the underlying hardware communication interface is active or not.

Returns *True* if the interface is active, *False* otherwise.

Return type Boolean

wait_for_frame(*operating_mode*)

Reads the next API frame packet.

This method blocks until:

- A complete frame is read, in which case returns it.
- The configured timeout goes by, in which case returns *None*.
- Another thread calls quit_reading, in which case returns *None*.

This method is not thread-safe, so no more than one thread should invoke it at the same time.

Subclasses may throw specific exceptions to signal implementation specific hardware errors.

Parameters **operating_mode** (*OperatingMode*) – The operating mode of the XBee connected to this hardware interface. Note: If this parameter does not match the connected XBee configuration, the behavior is undefined.

Returns

The read packet as bytearray if a packet is read, *None* otherwise.

Return type Bytearray

quit_reading()

Makes the thread (if any) blocking on wait_for_frame return.

If a thread was blocked on wait_for_frame, this method blocks (for a maximum of 'timeout' seconds) until the blocked thread is resumed.

write_frame(*frame*)

Writes an XBee frame to the underlying hardware interface.

Subclasses may throw specific exceptions to signal implementation specific hardware errors.

Parameters **frame** (*Bytearray*) – The XBee API frame packet to write. If the bytearray does not correctly represent an XBee frame, the behaviour is undefined.

get_network(*local_xbee*)

Returns the XBeeNetwork object associated to the XBeeDevice associated to this *XBeeCommunicationInterface*.

Some *XBeeCommunicationInterface implementations may need to handle the 'XBeeNetwork* associated to the *XBeeDevice* themselves. If that is the case, a implementation-specific XBeeNetwork object that complains to the generic *XBeeNetwork* class will be returned. Otherwise, this method returns *None* and the associated *XBeeNetwork* is handled as for a serial-connected *XBeeDevice*.

Parameters **local_xbee** (*XBeeDevice*) – The local XBee device.

Returns

class: .*XBeeNetwork*: *None* if the XBeeNetwork should handled as usual, otherwise a *XBeeNetwork* object.

get_local_xbee_info()

Returns a tuple with the local XBee information.

This is used when opening the local XBee. If this information is provided, it is used as internal XBee data, if not provided, the data is requested to the XBee.

Returns

Tuple with local XBee information: operation mode (int), hardware version (int), firmware version (int), 64-bit address (string), 16-bit address (string), node identifier (string), and role (int).

Return type Tuple

supports_update_firmware()

Returns if the interface supports the firmware update feature.

Returns *True* if it is supported, *False* otherwise.

Return type Boolean

update_firmware(*xbee*, *xml_fw_file*, *xbee_fw_file=None*, *bootloader_fw_file=None*, *timeout=None*, *progress_callback=None*)

Performs a firmware update operation of the provided XBee.

Parameters

- **xbee** (*AbstractXBeeDevice*) – Local or remote XBee node to be updated.
- **xml_fw_file** (*String*) – Path of the XML file that describes the firmware to upload.
- **xbee_fw_file** (*String, optional*) – Location of the XBee binary firmware file.
- **bootloader_fw_file** (*String, optional*) – Location of the bootloader binary firmware file.

- **timeout** (*Integer, optional*) – Maximum time to wait for target read operations during the update process.
- **progress_callback** (*Function, optional*) – Function to execute to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

Raises

- XBeeException – If the local XBee is not open.
- InvalidOperatingModeException – If the local XBee operating mode is invalid.
- OperationNotSupportedException – If the firmware update is not supported in the XBee.
- FirmwareUpdateException – If there is any error performing the firmware update.

supports_apply_profile()

Returns if the interface supports the apply profile feature.

Returns *True* if it is supported, *False* otherwise.

Return type Boolean

apply_profile(*xbee*, *profile_path*, *timeout=None*, *progress_callback=None*)

Applies the given XBee profile to the XBee device.

Parameters

- **xbee** (*AbstractXBeeDevice*) – Local or remote XBee node to be updated.
- **profile_path** (*String*) – Path of the XBee profile file to apply.
- **timeout** (*Integer, optional*) – Maximum time to wait for target read operations during the apply profile.
- **progress_callback** (*Function, optional*) – Function to execute to receive progress information. Receives two arguments:
  - The current apply profile task as a String
  - The current apply profile task percentage as an Integer

Raises

- XBeeException – If the local XBee is not open.
- InvalidOperatingModeException – If the local XBee operating mode is invalid.
- UpdateProfileException – If there is any error applying the XBee profile.
- OperationNotSupportedException – If XBee profiles are not supported in the XBee.

timeout

Returns the read timeout.

Returns Read timeout in seconds.

Return type Integer

digi.xbee.devices module

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

Bases: `object`

This class provides common functionality for all XBee devices.

Class constructor. Instantiates a new *AbstractXBeeDevice* object with the provided parameters.

Parameters

- **local_xbee_device** (*XBeeDevice*, optional, default=‘None‘) – Only necessary if XBee is remote. The local XBee to be the connection interface to communicate with the remote XBee one.
- **serial_port** (*XBeeSerialPort*, optional, default=‘None‘) – Only necessary if the XBee device is local. The serial port to communicate with this XBee.
- **(Integer, optional, default** (*sync_ops_timeout*) – `AbstractXBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS`): Timeout (in seconds) for all synchronous operations.
- **comm_iface** (*XBeeCommunicationInterface*, optional, default=‘None‘) – Only necessary if the XBee is local. The hardware interface to communicate with this XBee.

See also:

XBeeDevice

XBeeSerialPort

update_device_data_from(*device*)

Updates the current node information with provided data. This is only for internal use.

Parameters **device** (*AbstractXBeeDevice*) – XBee to get the data from.

Returns *True* if the node data has been updated, *False* otherwise.

Return type Boolean

get_parameter(*parameter*, *parameter_value=None*, *apply=None*)

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

Parameters

- **(String or** (*parameter*) – class: *.ATStringCommand*): Parameter to get.
- **parameter_value** (*Bytearray, optional, default=`None`*) – Value of the parameter to execute (if any).
- **apply** (*Boolean, optional, default=`None`*) – *True* to apply changes in XBee configuration, *False* not to apply them, *None* to use *is_apply_changes_enabled()* returned value.

Returns Parameter value.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`AbstractXBeeDevice.set_parameter()`
`AbstractXBeeDevice.execute_command()`
`AbstractXBeeDevice.apply_changes()`
`AbstractXBeeDevice.write_changes()`

`set_parameter`(*parameter*, *value*, *apply=None*)

Sets the value of a parameter via an AT Command.

Any parameter changes are applied automatically, if *apply* is *True* or if it is *None* and apply flag is enabled (*is_apply_changes_enabled()*)

You can set this flag via the method *`AbstractXBeeDevice.enable_apply_changes()`*.

This only applies modified values in the XBee configuration, to save changed parameters permanently (between resets), use *`AbstractXBeeDevice.write_changes()`*.

Parameters

- **`(String or`** (*`parameter`*) – class: *.ATStringCommand*): Parameter to set.
- **`value`** (*`Bytearray`*) – Value of the parameter.
- **`apply`** (*`Boolean, optional, default=`None``*) – *True* to apply changes, *False* otherwise, *None* to use *is_apply_changes_enabled()* returned value.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `ValueError` – If *parameter* is *None* or *value* is *None*.

See also:

`AbstractXBeeDevice.get_parameter()`
`AbstractXBeeDevice.execute_command()`
`AbstractXBeeDevice.apply_changes()`
`AbstractXBeeDevice.write_changes()`
`AbstractXBeeDevice.is_apply_changes_enabled()`
`AbstractXBeeDevice.enable_apply_changes()`

execute_command(*parameter*, *value=None*, *apply=None*)

Executes the provided command.

Parameters

- **(String or** (*parameter*) – class: *.ATStringCommand*): AT command to execute.
- **value** (*bytearray, optional, default=`None`*) – Command value (if any).
- **apply** (*Boolean, optional, default=`None`*) – *True* to apply changes in XBee configuration, *False* not to apply them, *None* to use *is_apply_changes_enabled()* returned value.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

AbstractXBeeDevice.get_parameter()
AbstractXBeeDevice.set_parameter()
AbstractXBeeDevice.apply_changes()
AbstractXBeeDevice.write_changes()
AbstractXBeeDevice.is_apply_changes_enabled()
AbstractXBeeDevice.enable_apply_changes()

apply_changes()

Applies changes via 'AC' command.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

write_changes()

Writes configurable parameter values to the non-volatile memory of the XBee so that parameter modifications persist through subsequent resets.

Parameters values remain in the device's memory until overwritten by subsequent use of this method.

If changes are made without writing them, the XBee 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_changes_enabled()* to get its status and *enable_apply_changes()* to enable/disable the option. Method *apply_changes()* can be used in order to manually apply the changes.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

reset()

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

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

read_device_info(*init=True*, *fire_event=True*)

Updates all instance parameters reading them from the XBee.

Parameters

- **init** (*Boolean, optional, default=`True`*) – If *False* only not initialized parameters are read, all if *True*.
- **fire_event** (*Boolean, optional, default=`True`*) – *True* to throw and update event if any parameter changed, *False* otherwise.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

AbstractXBeeDevice.is_device_info_complete()

determine_protocol(*hardware_version*, *firmware_version*)

Determines the XBee protocol based on the given hardware and firmware versions.

Parameters

- **hardware_version** (*Integer*) – Hardware version to get its protocol.
- **firmware_version** (*Bytearray*) – Firmware version to get its protocol.

Returns

XBee protocol corresponding to the given hardware and firmware versions.

Return type *XBeeProtocol*

is_device_info_complete()

Returns whether XBee node information is complete.

Returns *True* if node information is complete, *False* otherwise.

Return type Boolean

See also:

AbstractXBeeDevice.read_device_info()

get_node_id()

Returns the node identifier ('NI') value of the XBee.

Returns Node identifier ('NI') of the XBee.

Return type String

set_node_id(*node_id*)

Sets the node identifier ('NI') value of the XBee.

Parameters **node_id** (*String*) – New node identifier ('NI') of the XBee.

Raises

- `ValueError` – If *node_id* is *None* or its length is greater than 20.
- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

get_hardware_version()

Returns the hardware version of the XBee.

Returns Hardware version of the XBee.

Return type *HardwareVersion*

See also:

HardwareVersion

get_firmware_version()

Returns the firmware version of the XBee.

Returns Firmware version of the XBee.

Return type Bytearray

get_protocol()

Returns the current protocol of the XBee.

Returns Current protocol of the XBee.

Return type *XBeeProtocol*

See also:

XBeeProtocol

get_16bit_addr()

Returns the 16-bit address of the XBee.

Returns 16-bit address of the XBee.

Return type *XBee16BitAddress*

See also:

XBee16BitAddress

set_16bit_addr(*value*)

Sets the 16-bit address of the XBee.

Parameters **value** (*XBee16BitAddress*) – New 16-bit address of the XBee.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If the protocol is not 802.15.4.

get_64bit_addr()

Returns the 64-bit address of the XBee.

Returns 64-bit address of the XBee.

Return type *XBee64BitAddress*

See also:

XBee64BitAddress

get_role()

Gets the XBee role.

Returns the role of the XBee.

Return type *Role*

See also:

Role

get_current_frame_id()
Returns the last used frame ID.

Returns Last used frame ID.

Return type Integer

enable_apply_changes(*value*)
Sets apply changes flag.

Parameters **value** (*Boolean*) – *True* to enable apply changes flag, *False* to disable it.

is_apply_changes_enabled()
Returns whether apply changes flag is enabled.

Returns *True* if apply changes flag is enabled, *False* otherwise.

Return type Boolean

is_remote()
Determines whether XBee is remote.

Returns *True* if the XBee 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*) – Read timeout in seconds.

get_sync_ops_timeout()
Returns the serial port read timeout.

Returns Serial port read timeout in seconds.

Return type Integer

get_dest_address()
Returns the 64-bit address of the XBee that is data destination.

Returns 64-bit address of destination XBee.

Return type *XBee64BitAddress*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

XBee64BitAddress
set_dest_address()

set_dest_address(*addr*)
Sets the 64-bit address of the XBee that is data destination.

Parameters **addr** (*XBee64BitAddress* or *RemoteXBeeDevice*) – Address itself or remote XBee to be data destination.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `ValueError` – If *addr* is *None*.

See also:

XBee64BitAddress
get_dest_address()

get_pan_id()

Returns the operating PAN ID of the XBee.

Returns Operating PAN ID of the XBee.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

set_pan_id()

set_pan_id(*value*)

Sets the operating PAN ID of the XBee.

Parameters **value** (*Bytearray*) – New operating PAN ID of the XBee. Must have only 1 or 2 bytes.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

get_pan_id()

get_power_level()

Returns the power level of the XBee.

Returns Power level of the XBee.

Return type *PowerLevel*

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

See also:

PowerLevel
set_power_level()

set_power_level(*power_level*)

Sets the power level of the XBee.

Parameters **power_level** (*PowerLevel*) – New power level of the XBee.

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

See also:

PowerLevel
get_power_level()

set_io_configuration(*io_line*, *io_mode*)

Sets the configuration of the provided IO line.

Parameters

- **io_line** (*IOLine*) – IO line to configure.
- **io_mode** (*IOMode*) – IO mode to set to the IO line.

Raises

- TimeoutException – If response is not received before the read timeout expires.

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`IOLine`
`IOMode`
`get_io_configuration()`

`get_io_configuration`(*io_line*)
Returns the configuration of the provided IO line.

Parameters **`io_line`** (*`IOLine`*) – IO line to get its configuration.

Returns IO mode of the IO line provided.

Return type *`IOMode`*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`IOLine`
`IOMode`
`set_io_configuration()`

`get_io_sampling_rate`()
Returns the IO sampling rate of the XBee.

Returns IO sampling rate of XBee.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

set_io_sampling_rate()

set_io_sampling_rate(*rate*)

Sets the IO sampling rate of the XBee in seconds. A sample rate of 0 means the IO sampling feature is disabled.

Parameters **rate** (*Integer*) – New IO sampling rate of the XBee in seconds.

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

See also:

get_io_sampling_rate()

read_io_sample()

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

Returns IO sample read from the XBee.

Return type *IOSample*

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If 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*) – IO line to get its ADC value.

Returns Analog value corresponding to the provided IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If response does not contain the value for the given IO line.

See also:

IOLine

set_io_configuration()

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*) – IO Line to be assigned.
- **cycle** (*Integer*) – Duty cycle in % to be assigned. Must be between 0 and 100.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – IO line to get its PWM duty cycle.

Returns PWM duty cycle of the given IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.

- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `ValueError` – If *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 gets its digital value.

Returns current value of the provided IO line.

Return type *`IOValue`*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If response does not contain the value for the given IO line.

See also:

`IOLine`
`IOValue`
`set_io_configuration()`

`set_dio_value`(*io_line*, *io_value*)

Sets the digital value (high or low) to the provided IO line.

Parameters

- **`io_line`** (*`IOLine`*) – Digital IO line to sets its value.
- **`io_value`** (*`IOValue`*) – IO value to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

- `ATCommandException` – If 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 response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`IOLine`

`get_api_output_mode`()
Deprecated since version 1.3: Use *`get_api_output_mode_value()`*

Returns the API output mode of the XBee.

The API output mode determines the format of the data through the serial interface of the XBee.

Returns API output mode of the XBee.

Return type *`APIOutputMode`*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`APIOutputMode`

get_api_output_mode_value()

Returns the API output mode of the XBee.

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

Returns the parameter value.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

digi.xbee.models.mode.APIOutputModeBit

set_api_output_mode(*api_output_mode*)

Deprecated since version 1.3: Use *set_api_output_mode_value()*

Sets the API output mode of the XBee.

Parameters **api_output_mode** (*APIOutputMode*) – New API output mode.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

APIOutputMode

set_api_output_mode_value(*api_output_mode*)

Sets the API output mode of the XBee.

Parameters **api_output_mode** (*Integer*) – New API output mode options. Calculate this value using the method `APIOutputModeBit.calculate_api_output_mode_value()` with a set of *APIOutputModeBit*.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

APIOutputModeBit

enable_bluetooth()

Enables the Bluetooth interface of this XBee.

To work with this interface, you must also configure the Bluetooth password if not done previously. Use method *AbstractXBeeDevice.update_bluetooth_password()*.

Note that your XBee must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

disable_bluetooth()

Disables the Bluetooth interface of this XBee.

Note that your device must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

get_bluetooth_mac_addr()

Reads and returns the EUI-48 Bluetooth MAC address of this XBee following the format *00112233AABB*.

Note that your device must include Bluetooth Low Energy support.

Returns The Bluetooth MAC address.

Return type String

Raises

- `TimeoutException` – If response is not received before the read timeout expires.

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

`update_bluetooth_password`(*new_password*)

Changes the Bluetooth password of this XBee with the new one provided.

Note that your device must include Bluetooth Low Energy support.

Parameters **`new_password`** (*`String`*) – New Bluetooth password.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

`update_firmware`(*xml_firmware_file*, *xbee_firmware_file=None*, *bootloader_firmware_file=None*, *timeout=None*, *progress_callback=None*)

Performs a firmware update operation of the XBee.

Parameters

- **`xml_firmware_file`** (*`String`*) – Path of the XML file that describes the firmware to upload.
- **`xbee_firmware_file`** (*`String, optional, default=`None``*) – Location of the XBee binary firmware file.
- **`bootloader_firmware_file`** (*`String, optional, default=`None``*) – Location of the bootloader binary firmware file.
- **`timeout`** (*`Integer, optional, default=`None``*) – Maximum time to wait for target read operations during the update process (seconds).
- **`progress_callback`** (*`Function, optional, default=`None``*) – Function to to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

Raises

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `OperationNotSupportedException` – If XBee does not support firmware update.
- `FirmwareUpdateException` – If there is any error during the firmware update.

`apply_profile`(*profile_path*, *timeout=None*, *progress_callback=None*)

Applies the given XBee profile to the XBee.

Parameters

- **`profile_path`** (*`String`*) – Path of the XBee profile file to apply.

- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the apply profile (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to receive progress information. Receives two arguments:
  - The current apply profile task as a String
  - The current apply profile task percentage as an Integer

Raises

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `UpdateProfileException` – If there is any error applying the XBee profile.

get_file_manager()

Returns the file system manager for the XBee.

Returns The file system manager.

Return type *FileSystemManager*

Raises `FileSystemNotSupportedException` – If the XBee does not support filesystem.

reachable

Returns whether the XBee is reachable.

Returns *True* if the device is reachable, *False* otherwise.

Return type Boolean

scan_counter

Returns the scan counter for this node.

Returns The scan counter for this node.

Return type Integer

log

Returns the XBee logger.

Returns The XBee device logger.

Return type `Logger`

class digi.xbee.devices.**XBeeDevice**(*port=None, baud_rate=None, data_bits=<sphinx.ext.autodoc.importer._MockObject object>, stop_bits=<sphinx.ext.autodoc.importer._MockObject object>, parity=<sphinx.ext.autodoc.importer._MockObject object>, flow_control=<FlowControl.NONE: None>, _sync_ops_timeout=4, comm_iface=None*)

Bases: *digi.xbee.devices.AbstractXBeeDevice*

This class represents a non-remote generic XBee.

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** (*String*) – Serial port identifier. Depends on operating system. e.g. '/dev/ttyUSB0' on 'GNU/Linux' or 'COM3' on Windows.
- **baud_rate** (*Integer, optional, default=`None`*) – Serial port baud rate.
- **(Integer, default** (*_sync_ops_timeout*) – `serial.EIGHTBITS`): Port bit-size.
- **(Integer, default** – `serial.STOPBITS_ONE`): Port stop bits.
- **(Character, default** (*parity*) – `serial.PARITY_NONE`): Port parity.
- **(Integer, default** – `FlowControl.NONE`): Port flow control.
- **(Integer, default** – 4): Read timeout (in seconds).
- **comm_iface** (*XBeeCommunicationInterface*) – Communication interface.

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 XBee object created.

Return type *XBeeDevice*

Raises `SerialException` – If the port to open does not exist or is already opened.

See also:

XBeeDevice

open (*force_settings=False*)
Opens the communication with the XBee and loads information about it.

Parameters **force_settings** (*Boolean, optional, default=`False`*) – *True* to open the device ensuring/forcing that the specified serial settings are applied even if the current configuration is different, *False* to open the device with the current configuration.

Raises

- `TimeoutException` – If there is any problem with the communication.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the XBee is already opened.

close()

Closes the communication with the XBee.

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

serial_port

Returns the serial port associated to the XBee, if any.

Returns

Serial port of the XBee. ***None*** **if the** local XBee does not use serial communication.

Return type *XBeeSerialPort*

See also:

XBeeSerialPort

comm_iface

Returns the hardware interface associated to the XBee.

Returns Hardware interface of the XBee.

Return type *XBeeCommunicationInterface*

See also:

XBeeCommunicationInterface

operating_mode

Returns the operating mode of this XBee.

Returns *OperatingMode*. This XBee operating mode.

get_parameter(*parameter*, *parameter_value=None*, *apply=None*)

Override.

See also:

AbstractXBeeDevice.get_parameter()

set_parameter(*parameter*, *value*, *apply=None*)

Override.

See: *AbstractXBeeDevice.set_parameter()*

send_data(*remote_xbee*, *data*, *transmit_options=0*)

Blocking method. This method sends data to a remote XBee synchronously.

This method will wait for the packet response. The default timeout is `XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS`.

Parameters

- **remote_xbee** (*RemoteXBeeDevice*) – Remote XBee to send data to.
- **data** (*String or Bytearray*) – Raw data to send.
- **transmit_options** (*Integer, optional*) – Transmit options, bitfield of *TransmitOptions*. Default to *TransmitOptions.NONE.value*.

Returns The response.

Return type *XBeePacket*

Raises

- `ValueError` – If *remote_xbee* is *None*.
- `TimeoutException` – If response is not received before the read timeout expires.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TransmitException` – If the status of the response received is not OK.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

RemoteXBeeDevice
XBeePacket

send_data_async(*remote_xbee*, *data*, *transmit_options=0*)

Non-blocking method. This method sends data to a remote XBee.

This method does not wait for a response.

Parameters

- **remote_xbee** (*RemoteXBeeDevice*) – the remote XBee to send data to.
- **data** (*String or Bytearray*) – Raw data to send.
- **transmit_options** (*Integer, optional*) – Transmit options, bitfield of *TransmitOptions*. Default to *TransmitOptions.NONE.value*.

Raises

- `ValueError` – If *remote_xbee* is *None*.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the XBee's communication interface 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 until a success or error transmit status arrives or the configured receive timeout expires.

The received timeout is configured using method `AbstractXBeeDevice.set_sync_ops_timeout()` 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 response is not received before the read timeout expires.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TransmitException` – If the status of the response received is not OK.
- `XBeeException` – If the XBee's communication interface is closed.

`send_user_data_relay`(*local_interface*, *data*)

Sends the given data to the given XBee local interface.

Parameters

- **`local_interface`** (*`XBeeLocalInterface`*) – Destination XBee local interface.
- **`data`** (*`Bytearray`*) – Data to send.

Raises

- `InvalidOperatingModeException` – If the XBee'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'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'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.

If *timeout* is specified, this method blocks until new data is received or the timeout expires, throwing a *TimeoutException* in this case.

Parameters **timeout** (*Integer, optional*) – Read timeout in seconds. If *None*, this method is non-blocking and returns *None* if no data is available.

Returns

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's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- XBeeException – If the XBee's communication interface is closed.

See also:

XBeeMessage

read_data_from(*remote_xbee*, *timeout=None*)

Reads new data received from the given remote XBee.

If *timeout* is specified, this method blocks until new data is received or the timeout expires, throwing a `TimeoutException` in this case.

Parameters

- **remote_xbee** (`RemoteXBeeDevice`) – Remote XBee that sent the data.
- **timeout** (`Integer, optional`) – Read timeout in seconds. If *None*, this method is non-blocking and returns *None* if no data is available.

Returns

Read message sent by *remote_xbee* 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's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

`XBeeMessage`
`RemoteXBeeDevice`

has_packets()

Returns if there are pending packets to read. This does not include explicit packets.

Returns *True* if there are pending packets, *False* otherwise.

Return type Boolean

See also:

`XBeeDevice.has_explicit_packets()`

has_explicit_packets()

Returns if there are pending explicit packets to read. This does not include non-explicit packets.

Returns *True* if there are pending packets, *False* otherwise.

Return type Boolean

See also:

`XBeeDevice.has_packets()`

flush_queues()
Flushes the packets queue.

reset()
Override method.

See also:

AbstractXBeeDevice.reset()

add_packet_received_callback(*callback*)
Adds a callback for the event *PacketReceived*.

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

- The received packet as a *XBeeAPIPacket*.

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*.

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

- The modem status as a *ModemStatus*.

add_io_sample_received_callback(*callback*)
Adds a callback for the event *IOSampleReceived*.

Parameters **callback** (*Function*) – The callback. Receives three arguments.

- The received IO sample as an *IOSample*.
- The remote XBee which sent the packet as a *RemoteXBeeDevice*.
- The time in which the packet was received as an Integer.

add_expl_data_received_callback(*callback*)
Adds a callback for the event *ExplicitDataReceived*.

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

- The explicit data received as a *ExplicitXBeeMessage*.

add_user_data_relay_received_callback(*callback*)
Adds a callback for the event *RelayDataReceived*.

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

- The relay data as a *UserDataRelayMessage*.

add_bluetooth_data_received_callback(*callback*)
Adds a callback for the event *BluetoothDataReceived*.

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

- The Bluetooth data as a Bytearray.

add_micropython_data_received_callback(*callback*)
Adds a callback for the event *MicroPythonDataReceived*.

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

- The MicroPython data as a Bytearray.

add_socket_state_received_callback(*callback*)
Adds a callback for the event *SocketStateReceived*.

Parameters **callback** (*Function*) – The callback. Receives two arguments.

- The socket ID as an Integer.
- The state received as a *SocketState*.

add_socket_data_received_callback(*callback*)
Adds a callback for the event *SocketDataReceived*.

Parameters **callback** (*Function*) – The callback. Receives two arguments.

- The socket ID as an Integer.
- The data received as Bytearray.

add_socket_data_received_from_callback(*callback*)
Adds a callback for the event *SocketDataReceivedFrom*.

Parameters **callback** (*Function*) – The callback. Receives three arguments.

- The socket ID as an Integer.
- **Source address pair (host, port) where host is a string** representing an IPv4 address like '100.50.200.5', and port is an integer.
- The data received as Bytearray.

add_fs_frame_received_callback(*callback*)
Adds a callback for the event *FileSystemFrameReceived*.

Parameters **callback** (*Function*) – The callback. Receives four arguments.

- Source (*AbstractXBeeDevice*): The node that sent the file system frame.
- Frame id (Integer): The received frame id.
- Command (*FSCmd*): The file system command.
- Receive options (Integer): Bitfield indicating receive options.

See also:

AbstractXBeeDevice
FSCmd
ReceiveOptions

del_packet_received_callback(*callback*)
Deletes a callback for the callback list of *PacketReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_data_received_callback(*callback*)
Deletes a callback for the callback list of *DataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_modem_status_received_callback(*callback*)
Deletes a callback for the callback list of *ModemStatusReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_io_sample_received_callback(*callback*)
Deletes a callback for the callback list of *IOSampleReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_expl_data_received_callback(*callback*)
Deletes a callback for the callback list of *ExplicitDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_user_data_relay_received_callback(*callback*)
Deletes a callback for the callback list of *RelayDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_bluetooth_data_received_callback(*callback*)
Deletes a callback for the callback list of *BluetoothDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_micropython_data_received_callback(*callback*)
Deletes a callback for the callback list of *MicroPythonDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_socket_state_received_callback(*callback*)
Deletes a callback for the callback list of *SocketStateReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_socket_data_received_callback(*callback*)
Deletes a callback for the callback list of *SocketDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_socket_data_received_from_callback(*callback*)
Deletes a callback for the callback list of *SocketDataReceivedFrom* event.

Parameters **callback** (*Function*) – The callback to delete.

del_fs_frame_received_callback(*callback*)
Deletes a callback for the callback list of *FileSystemFrameReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

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 are executed before user callbacks.

Returns *PacketReceived*

is_open()
Returns whether this XBee is open.

Returns Boolean. *True* if this XBee is open, *False* otherwise.

is_remote()

Override method.

See also:

AbstractXBeeDevice.is_remote()

get_network()

Returns the network of this XBee.

Returns The XBee network.

Return type *XBeeNetwork*

send_packet_sync_and_get_response(*packet_to_send*, *timeout=None*)

Sends the packet and waits for its corresponding response.

Parameters

- **packet_to_send** (*XBeePacket*) – The packet to transmit.
- **timeout** (*Integer, optional, default=`None`*) – Number of seconds to wait. -1 to wait indefinitely.

Returns Received response packet.

Return type *XBeePacket*

Raises

- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TimeoutException` – If response is not received in the configured timeout.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

XBeePacket

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

Sends the packet and waits for the response. The packet to send is escaped depending on the current operating mode.

This method can be synchronous or asynchronous.

If synchronous, this method discards all response packets until it finds the one that has the appropriate frame ID, that is, the sent packet's frame ID.

If asynchronous, this method does not wait for any response and returns *None*.

Parameters

- **packet** (*XBeePacket*) – The packet to send.
- **sync** (*Boolean*) – *True* to wait for the response of the sent packet and return it, *False* otherwise.

Returns

Response packet if *sync* is *True*, *None* otherwise.

Return type *XBeePacket*

Raises

- `TimeoutException` – If *sync* is *True* and the response packet for the sent one cannot be read.
- `InvalidOperatingModeException` – If the XBee operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the packet listener is not running or the XBee's communication interface is closed.

See also:

XBeePacket

get_next_frame_id()

Returns the next frame ID of the XBee.

Returns The next frame ID of the XBee.

Return type Integer

add_route_received_callback(*callback*)

Adds a callback for the event *RouteReceived*. This works for Zigbee and Digimesh devices.

Parameters **callback** (*Function*) – The callback. Receives three arguments.

- source (*XBeeDevice*): The source node.
- destination (*RemoteXBeeDevice*): The destination node.
- **hops (List): List of intermediate hops from closest to source** to closest to destination (*RemoteXBeeDevice*).

See also:

XBeeDevice.del_route_received_callback()

del_route_received_callback(*callback*)

Deletes a callback for the callback list of *RouteReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

See also:

XBeeDevice.add_route_received_callback()

get_route_to_node(*remote*, *timeout=10*, *force=True*)

Gets the route from this XBee to the given remote node.

For Zigbee:

- 'AR' parameter of the local node must be configured with a value different from 'FF'.

- Set *force* to *True* to force the Zigbee remote node to return its route independently of the local node configuration as high or low RAM concentrator ('DO' of the local value)

Parameters

- **remote** (*RemoteXBeeDevice*) – The remote node.
- **timeout** (*Float, optional, default=10*) – Maximum number of seconds to wait for the route.
- **force** (*Boolean*) – *True* to force asking for the route, *False* otherwise. Only for Zigbee.

Returns

Tuple containing route data:

- status (*TransmitStatus*): The transmit status.
- Tuple with route data (*None* if the route was not read in the provided timeout):
  - source (*RemoteXBeeDevice*): The source node of the route.
  - destination (*RemoteXBeeDevice*): The destination node of the route.
  - hops (List): List of intermediate nodes (*RemoteXBeeDevice*) ordered from closest to source to closest to destination node (source and destination not included).

Return type Tuple

apply_changes()

Applies changes via 'AC' command.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

apply_profile(*profile_path*, *timeout=None*, *progress_callback=None*)

Applies the given XBee profile to the XBee.

Parameters

- **profile_path** (*String*) – Path of the XBee profile file to apply.
- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the apply profile (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to receive progress information. Receives two arguments:
  - The current apply profile task as a String
  - The current apply profile task percentage as an Integer

Raises

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

- `UpdateProfileException` – If there is any error applying the XBee profile.

`determine_protocol`(*hardware_version*, *firmware_version*)

Determines the XBee protocol based on the given hardware and firmware versions.

Parameters

- **`hardware_version`** (*`Integer`*) – Hardware version to get its protocol.
- **`firmware_version`** (*`Bytearray`*) – Firmware version to get its protocol.

Returns

XBee protocol corresponding to the given hardware and firmware versions.

Return type *`XBeeProtocol`*

`disable_bluetooth`()

Disables the Bluetooth interface of this XBee.

Note that your device must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

`enable_apply_changes`(*value*)

Sets apply changes flag.

Parameters **`value`** (*`Boolean`*) – *True* to enable apply changes flag, *False* to disable it.

`enable_bluetooth`()

Enables the Bluetooth interface of this XBee.

To work with this interface, you must also configure the Bluetooth password if not done previously. Use method *`AbstractXBeeDevice.update_bluetooth_password()`*.

Note that your XBee must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

`execute_command`(*parameter*, *value=None*, *apply=None*)

Executes the provided command.

Parameters

- **`(String or`** (*`parameter`*) – class: *.ATStringCommand*): AT command to execute.
- **`value`** (*`bytearray, optional, default=`None``*) – Command value (if any).

- **apply** (*Boolean, optional, default=`None`*) – *True* to apply changes in XBee configuration, *False* not to apply them, *None* to use *is_apply_changes_enabled()* returned value.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

AbstractXBeeDevice.get_parameter()
AbstractXBeeDevice.set_parameter()
AbstractXBeeDevice.apply_changes()
AbstractXBeeDevice.write_changes()
AbstractXBeeDevice.is_apply_changes_enabled()
AbstractXBeeDevice.enable_apply_changes()

get_16bit_addr()
Returns the 16-bit address of the XBee.

Returns 16-bit address of the XBee.

Return type *XBee16BitAddress*

See also:

XBee16BitAddress

get_64bit_addr()
Returns the 64-bit address of the XBee.

Returns 64-bit address of the XBee.

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*) – IO line to get its ADC value.

Returns Analog value corresponding to the provided IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If response does not contain the value for the given IO line.

See also:

`IOLine`
`set_io_configuration()`

`get_api_output_mode``()`

Deprecated since version 1.3: Use *`get_api_output_mode_value()`*

Returns the API output mode of the XBee.

The API output mode determines the format of the data through the serial interface of the XBee.

Returns API output mode of the XBee.

Return type *`APIOutputMode`*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`APIOutputMode`

`get_api_output_mode_value``()`

Returns the API output mode of the XBee.

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

Returns the parameter value.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

digi.xbee.models.mode.APIOutputModeBit

get_bluetooth_mac_addr()

Reads and returns the EUI-48 Bluetooth MAC address of this XBee following the format *00112233AABB*.

Note that your device must include Bluetooth Low Energy support.

Returns The Bluetooth MAC address.

Return type String

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

get_current_frame_id()

Returns the last used frame ID.

Returns Last used frame ID.

Return type Integer

get_dest_address()

Returns the 64-bit address of the XBee that is data destination.

Returns 64-bit address of destination XBee.

Return type *XBee64BitAddress*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

XBee64BitAddress
set_dest_address()

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 response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.
- OperationNotSupportedException – If response does not contain the value for the given IO line.

See also:

IOLine
IOValue
set_io_configuration()

get_file_manager()
Returns the file system manager for the XBee.

Returns The file system manager.

Return type *FileSystemManager*

Raises FileSystemNotSupportedException – If the XBee does not support filesystem.

get_firmware_version()
Returns the firmware version of the XBee.

Returns Firmware version of the XBee.

Return type Bytearray

get_hardware_version()
Returns the hardware version of the XBee.

Returns Hardware version of the XBee.

Return type *HardwareVersion*

See also:

HardwareVersion

get_io_configuration(*io_line*)
Returns the configuration of the provided IO line.

Parameters **io_line** (*IOLine*) – IO line to get its configuration.

Returns IO mode of the IO line provided.

Return type *IOMode*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOLine
IOMode
set_io_configuration()

get_io_sampling_rate()
Returns the IO sampling rate of the XBee.

Returns IO sampling rate of XBee.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

set_io_sampling_rate()

get_node_id()
Returns the node identifier ('NI') value of the XBee.

Returns Node identifier ('NI') of the XBee.

Return type String

get_pan_id()
Returns the operating PAN ID of the XBee.

Returns Operating PAN ID of the XBee.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`set_pan_id()`

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

Returns Power level of the XBee.

Return type *`PowerLevel`*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`PowerLevel`
`set_power_level()`

`get_protocol`()
Returns the current protocol of the XBee.

Returns Current protocol of the XBee.

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`*) – IO line to get its PWM duty cycle.

Returns PWM duty cycle of the given IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `ValueError` – If *io_line* has no PWM capability.

See also:

IOLine

get_role()

Gets the XBee role.

Returns the role of the XBee.

Return type *Role*

See also:

Role

get_sync_ops_timeout()

Returns the serial port read timeout.

Returns Serial port read timeout in seconds.

Return type Integer

is_apply_changes_enabled()

Returns whether apply changes flag is enabled.

Returns *True* if apply changes flag is enabled, *False* otherwise.

Return type Boolean

is_device_info_complete()

Returns whether XBee node information is complete.

Returns *True* if node information is complete, *False* otherwise.

Return type Boolean

See also:

AbstractXBeeDevice.read_device_info()

log

Returns the XBee logger.

Returns The XBee device logger.

Return type `Logger`

`reachable`
Returns whether the XBee is reachable.

Returns *True* if the device is reachable, *False* otherwise.

Return type Boolean

`read_device_info`(*init=True*, *fire_event=True*)
Updates all instance parameters reading them from the XBee.

Parameters

- **`init`** (*`Boolean, optional, default=`True``*) – If *False* only not initialized parameters are read, all if *True*.
- **`fire_event`** (*`Boolean, optional, default=`True``*) – *True* to throw and update event if any parameter changed, *False* otherwise.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`AbstractXBeeDevice.is_device_info_complete()`

`read_io_sample`()
Returns an IO sample from the XBee containing the value of all enabled digital IO and analog input channels.

Returns IO sample read from the XBee.

Return type *`IOSample`*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`IOSample`

`scan_counter`
Returns the scan counter for this node.

Returns The scan counter for this node.

Return type Integer

set_16bit_addr(*value*)
Sets the 16-bit address of the XBee.

Parameters **value** (*XBee16BitAddress*) – New 16-bit address of the XBee.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If the protocol is not 802.15.4.

set_api_output_mode(*api_output_mode*)
Deprecated since version 1.3: Use *set_api_output_mode_value()*

Sets the API output mode of the XBee.

Parameters **api_output_mode** (*APIOutputMode*) – New API output mode.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

APIOutputMode

set_api_output_mode_value(*api_output_mode*)
Sets the API output mode of the XBee.

Parameters **api_output_mode** (*Integer*) – New API output mode options. Calculate this value using the method `APIOutputModeBit.calculate_api_output_mode_value()` with a set of *APIOutputModeBit*.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

APIOutputModeBit

set_dest_address (*addr*)

Sets the 64-bit address of the XBee that is data destination.

Parameters addr (*XBee64BitAddress* or *RemoteXBeeDevice*) – Address itself or remote XBee to be data destination.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `ValueError` – If *addr* is *None*.

See also:

XBee64BitAddress
get_dest_address()

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 response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – Digital IO line to sets its value.
- **io_value** (*IOValue*) – IO value to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – IO line to configure.
- **io_mode** (*IOMode*) – IO mode to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOLine
IOMode
get_io_configuration()

set_io_sampling_rate(*rate*)
Sets the IO sampling rate of the XBee in seconds. A sample rate of 0 means the IO sampling feature is disabled.

Parameters **rate** (*Integer*) – New IO sampling rate of the XBee in seconds.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

- ATCommandException – If response is not as expected.

See also:

get_io_sampling_rate()

set_node_id(*node_id*)

Sets the node identifier ('NI') value of the XBee.

Parameters node_id (*String*) – New node identifier ('NI') of the XBee.

Raises

- ValueError – If *node_id* is *None* or its length is greater than 20.
- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

set_pan_id(*value*)

Sets the operating PAN ID of the XBee.

Parameters value (*Bytearray*) – New operating PAN ID of the XBee. Must have only 1 or 2 bytes.

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

See also:

get_pan_id()

set_power_level(*power_level*)

Sets the power level of the XBee.

Parameters power_level (*PowerLevel*) – New power level of the XBee.

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

See also:

PowerLevel

get_power_level()

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*) – IO Line to be assigned.
- **cycle** (*Integer*) – Duty cycle in % to be assigned. Must be between 0 and 100.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – Read timeout in seconds.

update_bluetooth_password(*new_password*)
Changes the Bluetooth password of this XBee with the new one provided.

Note that your device must include Bluetooth Low Energy support.

Parameters **new_password** (*String*) – New Bluetooth password.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

update_device_data_from(*device*)
Updates the current node information with provided data. This is only for internal use.

Parameters **device** (*AbstractXBeeDevice*) – XBee to get the data from.

Returns *True* if the node data has been updated, *False* otherwise.

Return type Boolean

update_firmware(*xml_firmware_file*, *xbee_firmware_file=None*, *bootloader_firmware_file=None*, *timeout=None*, *progress_callback=None*)

Performs a firmware update operation of the XBee.

Parameters

- **xml_firmware_file** (*String*) – Path of the XML file that describes the firmware to upload.
- **xbee_firmware_file** (*String, optional, default=`None`*) – Location of the XBee binary firmware file.
- **bootloader_firmware_file** (*String, optional, default=`None`*) – Location of the bootloader binary firmware file.
- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the update process (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

Raises

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `OperationNotSupportedException` – If XBee does not support firmware update.
- `FirmwareUpdateException` – If there is any error during the firmware update.

write_changes()

Writes configurable parameter values to the non-volatile memory of the XBee so that parameter modifications persist through subsequent resets.

Parameters values remain in the device's memory until overwritten by subsequent use of this method.

If changes are made without writing them, the XBee 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_changes_enabled()` to get its status and `enable_apply_changes()` to enable/disable the option. Method `apply_changes()` can be used in order to manually apply the changes.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

class digi.xbee.devices.**Raw802Device**(*port=None*, *baud_rate=None*, *data_bits=<sphinx.ext.autodoc.importer._MockObject object>*, *stop_bits=<sphinx.ext.autodoc.importer._MockObject object>*, *parity=<sphinx.ext.autodoc.importer._MockObject object>*, *flow_control=<FlowControl.NONE: None>*, *_sync_ops_timeout=4*, *comm_iface=None*)

Bases: *digi.xbee.devices.XBeeDevice*

This class represents a local 802.15.4 XBee.

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

Parameters

- **port** (*String*) – Serial port identifier. Depends on operating system. e.g. '/dev/ttyUSB0' on 'GNU/Linux' or 'COM3' on Windows.
- **baud_rate** (*Integer*) – Serial port baud rate.
- **(Integer, default** (*flow_control*) – serial.EIGHTBITS): Port bitsize.
- **(Integer, default** – serial.STOPBITS_ONE): Port stop bits.
- **(Character, default** (*parity*) – serial.PARITY_NONE): Port parity.
- **(Integer, default** – FlowControl.NONE): Port flow control.

_sync_ops_timeout (Integer, default: 3): Read timeout (in seconds). comm_iface (*XBeeCommunicationInterface*): Communication interface.

Raises All exceptions raised by XBeeDevice.__init__() constructor.

See also:

XBeeDevice
XBeeDevice.__init__()

open(*force_settings=False*)
Override.

See also:

XBeeDevice.open()

get_protocol()
Override.

See also:

XBeeDevice.get_protocol()

get_ai_status()
Returns the current association status of this XBee. It indicates occurrences of errors during the modem initialization and connection.

Returns

The XBee association indication status.

Return type *AssociationIndicationStatus*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

send_data_64 (*x64addr*, *data*, *transmit_options=0*)

Blocking method. This method sends data to a remote XBee with the given 64-bit address.

This method waits for the packet response. The default timeout is `XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS`.

Parameters

- **x64addr** (*XBee64BitAddress*) – 64-bit address of the destination XBee.
- **data** (*String or Bytearray*) – Raw data to send.
- **transmit_options** (*Integer, optional*) – Transmit options, bitfield of *TransmitOptions*. Default to *TransmitOptions.NONE.value*.

Returns The response.

Return type *XBeePacket*

Raises

- `ValueError` – If *x64addr* or *data* is *None*.
- `TimeoutException` – If response is not received before the read timeout expires.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TransmitException` – If the status of the response received is not OK.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

XBee64BitAddress
XBeePacket

send_data_async_64 (*x64addr*, *data*, *transmit_options=0*)

Non-blocking method. This method sends data to a remote XBee with the given 64-bit address.

This method does not wait for a response.

Parameters

- **x64addr** (*XBee64BitAddress*) – 64-bit address of the destination XBee.
- **data** (*String or Bytearray*) – Raw data to send.

- **transmit_options** (*Integer, optional*) – Transmit options, bitfield of *TransmitOptions*. Default to *TransmitOptions.NONE.value*.

Raises

- ValueError – If *x64addr* or *data* is *None*.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- XBeeException – If the XBee's communication interface is closed.

See also:

XBee64BitAddress
XBeePacket

send_data_16 (*x16addr*, *data*, *transmit_options=0*)
Blocking method. This method sends data to a remote XBee with the given 16-bit address.

This method will wait for the packet response. The default timeout is XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS.

Parameters

- **x16addr** (*XBee16BitAddress*) – 16-bit address of the destination XBee.
- **data** (*String or Bytearray*) – Raw data to send.
- **transmit_options** (*Integer, optional*) – Transmit options, bitfield of *TransmitOptions*. Default to *TransmitOptions.NONE.value*.

Returns The response.

Return type *XBeePacket*

Raises

- ValueError – If *x16addr* or *data* is *None*.
- TimeoutException – If response is not received before the read timeout expires.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- TransmitException – If the status of the response received is not OK.
- XBeeException – If the XBee's communication interface is closed.

See also:

XBee16BitAddress
XBeePacket

send_data_async_16 (*x16addr*, *data*, *transmit_options=0*)
Non-blocking method. This method sends data to a remote XBee with the given 16-bit address.

This method does not wait for a response.

Parameters

- **x16addr** (*XBee16BitAddress*) – 16-bit address of the destination XBee.
- **data** (*String or Bytearray*) – Raw data to send.
- **transmit_options** (*Integer, optional*) – Transmit options, bitfield of *TransmitOptions*. Default to *TransmitOptions.NONE.value*.

Raises

- `ValueError` – If *x16addr* or *data* is *None*.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

XBee16BitAddress
XBeePacket

add_bluetooth_data_received_callback (*callback*)
Adds a callback for the event *BluetoothDataReceived*.

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

- The Bluetooth data as a Bytearray.

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

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

- The data received as an *XBeeMessage*.

add_expl_data_received_callback (*callback*)
Adds a callback for the event *ExplicitDataReceived*.

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

- The explicit data received as a *ExplicitXBeeMessage*.

add_fs_frame_received_callback (*callback*)
Adds a callback for the event *FileSystemFrameReceived*.

Parameters callback (*Function*) – The callback. Receives four arguments.

- Source (*AbstractXBeeDevice*): The node that sent the file system frame.
- Frame id (Integer): The received frame id.
- Command (*FSCmd*): The file system command.
- Receive options (Integer): Bitfield indicating receive options.

See also:

AbstractXBeeDevice
FSCmd
ReceiveOptions

add_io_sample_received_callback(*callback*)
Adds a callback for the event *IOSampleReceived*.

Parameters **callback** (*Function*) – The callback. Receives three arguments.

- The received IO sample as an *IOSample*.
- The remote XBee which sent the packet as a *RemoteXBeeDevice*.
- The time in which the packet was received as an Integer.

add_micropython_data_received_callback(*callback*)
Adds a callback for the event *MicroPythonDataReceived*.

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

- The MicroPython data as a Bytearray.

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

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

- The modem status as a *ModemStatus*.

add_packet_received_callback(*callback*)
Adds a callback for the event *PacketReceived*.

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

- The received packet as a *XBeeAPIPacket*.

add_route_received_callback(*callback*)
Adds a callback for the event *RouteReceived*. This works for Zigbee and Digimesh devices.

Parameters **callback** (*Function*) – The callback. Receives three arguments.

- source (*XBeeDevice*): The source node.
- destination (*RemoteXBeeDevice*): The destination node.
- **hops (List): List of intermediate hops from closest to source** to closest to destination (*RemoteXBeeDevice*).

See also:

XBeeDevice.del_route_received_callback()

add_socket_data_received_callback(*callback*)
Adds a callback for the event *SocketDataReceived*.

Parameters **callback** (*Function*) – The callback. Receives two arguments.

- The socket ID as an Integer.
- The data received as Bytearray.

add_socket_data_received_from_callback(*callback*)
Adds a callback for the event *SocketDataReceivedFrom*.

Parameters **callback** (*Function*) – The callback. Receives three arguments.

- The socket ID as an Integer.

- **Source address pair (host, port) where host is a string** representing an IPv4 address like '100.50.200.5', and port is an integer.
- The data received as Bytearray.

add_socket_state_received_callback(*callback*)
Adds a callback for the event *SocketStateReceived*.

Parameters callback (*Function*) – The callback. Receives two arguments.

- The socket ID as an Integer.
- The state received as a *SocketState*.

add_user_data_relay_received_callback(*callback*)
Adds a callback for the event *RelayDataReceived*.

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

- The relay data as a *UserDataRelayMessage*.

apply_changes()
Applies changes via 'AC' command.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

apply_profile(*profile_path*, *timeout=None*, *progress_callback=None*)
Applies the given XBee profile to the XBee.

Parameters

- **profile_path** (*String*) – Path of the XBee profile file to apply.
- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the apply profile (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to receive progress information. Receives two arguments:
  - The current apply profile task as a String
  - The current apply profile task percentage as an Integer

Raises

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `UpdateProfileException` – If there is any error applying the XBee profile.

close()
Closes the communication with the XBee.

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

comm_iface
Returns the hardware interface associated to the XBee.

Returns Hardware interface of the XBee.

Return type *XBeeCommunicationInterface*

See also:

XBeeCommunicationInterface

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 XBee object created.

Return type *XBeeDevice*

Raises `SerialException` – If the port to open does not exist or is already opened.

See also:

XBeeDevice

del_bluetooth_data_received_callback(*callback*)
Deletes a callback for the callback list of *BluetoothDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_data_received_callback(*callback*)
Deletes a callback for the callback list of *DataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_expl_data_received_callback(*callback*)
Deletes a callback for the callback list of *ExplicitDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_fs_frame_received_callback(*callback*)
Deletes a callback for the callback list of *FileSystemFrameReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_io_sample_received_callback(*callback*)
Deletes a callback for the callback list of *IOSampleReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_micropython_data_received_callback(*callback*)
Deletes a callback for the callback list of *MicroPythonDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_modem_status_received_callback(*callback*)
Deletes a callback for the callback list of *ModemStatusReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_packet_received_callback(*callback*)
Deletes a callback for the callback list of *PacketReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_route_received_callback(*callback*)
Deletes a callback for the callback list of *RouteReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

See also:

XBeeDevice.add_route_received_callback()

del_socket_data_received_callback(*callback*)
Deletes a callback for the callback list of *SocketDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_socket_data_received_from_callback(*callback*)
Deletes a callback for the callback list of *SocketDataReceivedFrom* event.

Parameters **callback** (*Function*) – The callback to delete.

del_socket_state_received_callback(*callback*)
Deletes a callback for the callback list of *SocketStateReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_user_data_relay_received_callback(*callback*)
Deletes a callback for the callback list of *RelayDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

determine_protocol(*hardware_version*, *firmware_version*)
Determines the XBee protocol based on the given hardware and firmware versions.

Parameters

- **hardware_version** (*Integer*) – Hardware version to get its protocol.
- **firmware_version** (*Bytearray*) – Firmware version to get its protocol.

Returns

XBee protocol corresponding to the given hardware and firmware versions.

Return type *XBeeProtocol*

disable_bluetooth()
Disables the Bluetooth interface of this XBee.

Note that your device must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

`enable_apply_changes`(*value*)

Sets apply changes flag.

Parameters **`value`** (*`Boolean`*) – *True* to enable apply changes flag, *False* to disable it.

`enable_bluetooth`()

Enables the Bluetooth interface of this XBee.

To work with this interface, you must also configure the Bluetooth password if not done previously. Use method *`AbstractXBeeDevice.update_bluetooth_password()`*.

Note that your XBee must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

`execute_command`(*parameter*, *value=None*, *apply=None*)

Executes the provided command.

Parameters

- **`(String or`** (*`parameter`*) – class: *.ATStringCommand*): AT command to execute.
- **`value`** (*`bytearray, optional, default=`None``*) – Command value (if any).
- **`apply`** (*`Boolean, optional, default=`None``*) – *True* to apply changes in XBee configuration, *False* not to apply them, *None* to use *is_apply_changes_enabled()* returned value.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`AbstractXBeeDevice.get_parameter()`

AbstractXBeeDevice.set_parameter()
AbstractXBeeDevice.apply_changes()
AbstractXBeeDevice.write_changes()
AbstractXBeeDevice.is_apply_changes_enabled()
AbstractXBeeDevice.enable_apply_changes()

flush_queues()
Flushes the packets queue.

get_16bit_addr()
Returns the 16-bit address of the XBee.

Returns 16-bit address of the XBee.

Return type *XBee16BitAddress*

See also:

XBee16BitAddress

get_64bit_addr()
Returns the 64-bit address of the XBee.

Returns 64-bit address of the XBee.

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*) – IO line to get its ADC value.

Returns Analog value corresponding to the provided IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If response does not contain the value for the given IO line.

See also:

IOLine
set_io_configuration()

get_api_output_mode()

Deprecated since version 1.3: Use *get_api_output_mode_value()*

Returns the API output mode of the XBee.

The API output mode determines the format of the data through the serial interface of the XBee.

Returns API output mode of the XBee.

Return type *APIOutputMode*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

APIOutputMode

get_api_output_mode_value()

Returns the API output mode of the XBee.

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

Returns the parameter value.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

digi.xbee.models.mode.APIOutputModeBit

get_bluetooth_mac_addr()

Reads and returns the EUI-48 Bluetooth MAC address of this XBee following the format *00112233AABB*.

Note that your device must include Bluetooth Low Energy support.

Returns The Bluetooth MAC address.

Return type String

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

get_current_frame_id()

Returns the last used frame ID.

Returns Last used frame ID.

Return type Integer

get_dest_address()

Returns the 64-bit address of the XBee that is data destination.

Returns 64-bit address of destination XBee.

Return type *XBee64BitAddress*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

XBee64BitAddress
set_dest_address()

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 response is not received before the read timeout expires.

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If response does not contain the value for the given IO line.

See also:

`IOLine`
`IOValue`
`set_io_configuration()`

`get_file_manager`()
Returns the file system manager for the XBee.

Returns The file system manager.

Return type *`FileSystemManager`*

Raises `FileSystemNotSupportedException` – If the XBee does not support filesystem.

`get_firmware_version`()
Returns the firmware version of the XBee.

Returns Firmware version of the XBee.

Return type Bytearray

`get_hardware_version`()
Returns the hardware version of the XBee.

Returns Hardware version of the XBee.

Return type *`HardwareVersion`*

See also:

`HardwareVersion`

`get_io_configuration`(*io_line*)
Returns the configuration of the provided IO line.

Parameters **`io_line`** (*`IOLine`*) – IO line to get its configuration.

Returns IO mode of the IO line provided.

Return type *`IOMode`*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.

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

See also:

IOLine
IOMode
set_io_configuration()

get_io_sampling_rate()
Returns the IO sampling rate of the XBee.

Returns IO sampling rate of XBee.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

set_io_sampling_rate()

get_network()
Returns the network of this XBee.

Returns The XBee network.

Return type *XBeeNetwork*

get_next_frame_id()
Returns the next frame ID of the XBee.

Returns The next frame ID of the XBee.

Return type Integer

get_node_id()
Returns the node identifier ('NI') value of the XBee.

Returns Node identifier ('NI') of the XBee.

Return type String

get_pan_id()
Returns the operating PAN ID of the XBee.

Returns Operating PAN ID of the XBee.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

set_pan_id()

get_parameter(*parameter*, *parameter_value=None*, *apply=None*)
Override.

See also:

AbstractXBeeDevice.get_parameter()

get_power_level()
Returns the power level of the XBee.

Returns Power level of the XBee.

Return type *PowerLevel*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

PowerLevel
set_power_level()

get_pwm_duty_cycle(*io_line*)
Returns the PWM duty cycle in % corresponding to the provided IO line.

Parameters **io_line** (*IOLine*) – IO line to get its PWM duty cycle.

Returns PWM duty cycle of the given IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `ValueError` – If *io_line* has no PWM capability.

See also:

`IOLine`

`get_role`()

Gets the XBee role.

Returns the role of the XBee.

Return type *`Role`*

See also:

`Role`

`get_route_to_node`(*remote*, *timeout=10*, *force=True*)

Gets the route from this XBee to the given remote node.

For Zigbee:

- 'AR' parameter of the local node must be configured with a value different from 'FF'.
- Set *force* to *True* to force the Zigbee remote node to return its route independently of the local node configuration as high or low RAM concentrator ('DO' of the local value)

Parameters

- **`remote`** (*`RemoteXBeeDevice`*) – The remote node.
- **`timeout`** (*`Float, optional, default=10`*) – Maximum number of seconds to wait for the route.
- **`force`** (*`Boolean`*) – *True* to force asking for the route, *False* otherwise. Only for Zigbee.

Returns

Tuple containing route data:

- status (*`TransmitStatus`*): The transmit status.
- Tuple with route data (*None* if the route was not read in the provided timeout):
  - source (*`RemoteXBeeDevice`*): The source node of the route.
  - destination (*`RemoteXBeeDevice`*): The destination node of the route.
  - hops (List): List of intermediate nodes (*`RemoteXBeeDevice`*) ordered from closest to source to closest to destination node (source and destination not included).

Return type Tuple

get_sync_ops_timeout()
Returns the serial port read timeout.

Returns 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 are executed before user callbacks.

Returns *PacketReceived*

has_explicit_packets()
Returns if there are pending explicit packets to read. This does not include non-explicit packets.

Returns *True* if there are pending packets, *False* otherwise.

Return type Boolean

See also:

XBeeDevice.has_packets()

has_packets()
Returns if there are pending packets to read. This does not include explicit packets.

Returns *True* if there are pending packets, *False* otherwise.

Return type Boolean

See also:

XBeeDevice.has_explicit_packets()

is_apply_changes_enabled()
Returns whether apply changes flag is enabled.

Returns *True* if apply changes flag is enabled, *False* otherwise.

Return type Boolean

is_device_info_complete()
Returns whether XBee node information is complete.

Returns *True* if node information is complete, *False* otherwise.

Return type Boolean

See also:

AbstractXBeeDevice.read_device_info()

is_open()

Returns whether this XBee is open.

Returns Boolean. *True* if this XBee is open, *False* otherwise.

is_remote()

Override method.

See also:

AbstractXBeeDevice.is_remote()

log

Returns the XBee logger.

Returns The XBee device logger.

Return type Logger

operating_mode

Returns the operating mode of this XBee.

Returns *OperatingMode*. This XBee operating mode.

reachable

Returns whether the XBee is reachable.

Returns *True* if the device is reachable, *False* otherwise.

Return type Boolean

read_data(*timeout=None*)

Reads new data received by this XBee.

If *timeout* is specified, this method blocks until new data is received or the timeout expires, throwing a *TimeoutException* in this case.

Parameters timeout (*Integer, optional*) – Read timeout in seconds. If *None*, this method is non-blocking and returns *None* if no data is available.

Returns

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's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- XBeeException – If the XBee's communication interface is closed.

See also:

XBeeMessage

read_data_from(*remote_xbee*, *timeout=None*)

Reads new data received from the given remote XBee.

If *timeout* is specified, this method blocks until new data is received or the timeout expires, throwing a *TimeoutException* in this case.

Parameters

- **remote_xbee** (*RemoteXBeeDevice*) – Remote XBee that sent the data.
- **timeout** (*Integer, optional*) – Read timeout in seconds. If *None*, this method is non-blocking and returns *None* if no data is available.

Returns

Read message sent by *remote_xbee* 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's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- XBeeException – If the XBee's communication interface is closed.

See also:

XBeeMessage
RemoteXBeeDevice

read_device_info(*init=True*, *fire_event=True*)

Updates all instance parameters reading them from the XBee.

Parameters

- **init** (*Boolean, optional, default=`True`*) – If *False* only not initialized parameters are read, all if *True*.
- **fire_event** (*Boolean, optional, default=`True`*) – *True* to throw and update event if any parameter changed, *False* otherwise.

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

See also:

AbstractXBeeDevice.is_device_info_complete()

read_io_sample()

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

Returns IO sample read from the XBee.

Return type *IOSample*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOSample

reset()

Override method.

See also:

AbstractXBeeDevice.reset()

scan_counter

Returns the scan counter for this node.

Returns The scan counter for this node.

Return type Integer

send_bluetooth_data(*data*)

Sends the given data to the Bluetooth interface using a User Data Relay frame.

Parameters **data** (*Bytearray*) – Data to send.

Raises

- `InvalidOperatingModeException` – If the XBee'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*, *data*, *transmit_options=0*)

Blocking method. This method sends data to a remote XBee synchronously.

This method will wait for the packet response. The default timeout is `XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS`.

Parameters

- **remote_xbee** (*RemoteXBeeDevice*) – Remote XBee to send data to.
- **data** (*String or Bytearray*) – Raw data to send.
- **transmit_options** (*Integer, optional*) – Transmit options, bitfield of *TransmitOptions*. Default to *TransmitOptions.NONE.value*.

Returns The response.

Return type *XBeePacket*

Raises

- `ValueError` – If *remote_xbee* is *None*.
- `TimeoutException` – If response is not received before the read timeout expires.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TransmitException` – If the status of the response received is not OK.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

RemoteXBeeDevice
XBeePacket

send_data_async (*remote_xbee*, *data*, *transmit_options=0*)

Non-blocking method. This method sends data to a remote XBee.

This method does not wait for a response.

Parameters

- **remote_xbee** (*RemoteXBeeDevice*) – the remote XBee to send data to.
- **data** (*String or Bytearray*) – Raw data to send.
- **transmit_options** (*Integer, optional*) – Transmit options, bitfield of *TransmitOptions*. Default to *TransmitOptions.NONE.value*.

Raises

- `ValueError` – If *remote_xbee* is *None*.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the XBee's communication interface 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 until a success or error transmit status arrives or the configured receive timeout expires.

The received timeout is configured using method *AbstractXBeeDevice.set_sync_ops_timeout()* 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 response is not received before the read timeout expires.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TransmitException` – If the status of the response received is not OK.
- `XBeeException` – If the XBee's communication interface is closed.

send_micropython_data (*data*)

Sends the given data to the MicroPython interface using a User Data Relay frame.

Parameters **data** (*Bytearray*) – Data to send.

Raises

- `InvalidOperatingModeException` – If the XBee'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*)

Sends the packet and waits for the response. The packet to send is escaped depending on the current operating mode.

This method can be synchronous or asynchronous.

If synchronous, this method discards all response packets until it finds the one that has the appropriate frame ID, that is, the sent packet's frame ID.

If asynchronous, this method does not wait for any response and returns *None*.

Parameters

- **packet** (*XBeePacket*) – The packet to send.
- **sync** (*Boolean*) – *True* to wait for the response of the sent packet and return it, *False* otherwise.

Returns

Response packet if *sync* is *True*, *None* otherwise.

Return type *XBeePacket*

Raises

- TimeoutException – If *sync* is *True* and the response packet for the sent one cannot be read.
- InvalidOperatingModeException – If the XBee operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- XBeeException – If the packet listener is not running or the XBee's communication interface is closed.

See also:

XBeePacket

send_packet_sync_and_get_response(*packet_to_send*, *timeout=None*)
Sends the packet and waits for its corresponding response.

Parameters

- **packet_to_send** (*XBeePacket*) – The packet to transmit.
- **timeout** (*Integer, optional, default=`None`*) – Number of seconds to wait. -1 to wait indefinitely.

Returns Received response packet.

Return type *XBeePacket*

Raises

- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- TimeoutException – If response is not received in the configured timeout.
- XBeeException – If the XBee's communication interface is closed.

See also:

XBeePacket

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'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, if any.

Returns

Serial port of the XBee. *None* if the local XBee does not use serial communication.

Return type *XBeeSerialPort*

See also:

XBeeSerialPort

set_16bit_addr(*value*)

Sets the 16-bit address of the XBee.

Parameters **value** (*XBee16BitAddress*) – New 16-bit address of the XBee.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If the protocol is not 802.15.4.

set_api_output_mode(*api_output_mode*)

Deprecated since version 1.3: Use *set_api_output_mode_value()*

Sets the API output mode of the XBee.

Parameters **api_output_mode** (*APIOutputMode*) – New API output mode.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

APIOutputMode

`set_api_output_mode_value`(*api_output_mode*)

Sets the API output mode of the XBee.

Parameters **`api_output_mode`** (*Integer*) – New API output mode options. Calculate this value using the method `APIOutputModeBit.calculate_api_output_mode_value()` with a set of *APIOutputModeBit*.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

APIOutputModeBit

`set_dest_address`(*addr*)

Sets the 64-bit address of the XBee that is data destination.

Parameters **`addr`** (*XBee64BitAddress* or *RemoteXBeeDevice*) – Address itself or remote XBee to be data destination.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `ValueError` – If *addr* is *None*.

See also:

XBee64BitAddress
get_dest_address()

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 response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – Digital IO line to sets its value.
- **io_value** (*IOValue*) – IO value to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – IO line to configure.
- **io_mode** (*IOMode*) – IO mode to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.

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

See also:

`IOLine`
`IOMode`
`get_io_configuration()`

`set_io_sampling_rate`(*rate*)
Sets the IO sampling rate of the XBee in seconds. A sample rate of 0 means the IO sampling feature is disabled.

Parameters **`rate`** (*`Integer`*) – New IO sampling rate of the XBee in seconds.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`get_io_sampling_rate()`

`set_node_id`(*node_id*)
Sets the node identifier ('NI') value of the XBee.

Parameters **`node_id`** (*`String`*) – New node identifier ('NI') of the XBee.

Raises

- `ValueError` – If *node_id* is *None* or its length is greater than 20.
- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

`set_pan_id`(*value*)
Sets the operating PAN ID of the XBee.

Parameters **`value`** (*`Bytearray`*) – New operating PAN ID of the XBee. Must have only 1 or 2 bytes.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`get_pan_id()`

`set_parameter`(*parameter*, *value*, *apply=None*)
Override.

See: *`AbstractXBeeDevice.set_parameter()`*

`set_power_level`(*power_level*)
Sets the power level of the XBee.

Parameters **`power_level`** (*`PowerLevel`*) – New power level of the XBee.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`PowerLevel`
`get_power_level()`

`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`*) – IO Line to be assigned.
- **`cycle`** (*`Integer`*) – Duty cycle in % to be assigned. Must be between 0 and 100.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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`) – Read timeout in seconds.

`update_bluetooth_password`(*new_password*)

Changes the Bluetooth password of this XBee with the new one provided.

Note that your device must include Bluetooth Low Energy support.

Parameters **`new_password`** (`String`) – New Bluetooth password.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

`update_device_data_from`(*device*)

Updates the current node information with provided data. This is only for internal use.

Parameters **`device`** (`AbstractXBeeDevice`) – XBee to get the data from.

Returns *True* if the node data has been updated, *False* otherwise.

Return type Boolean

`update_firmware`(*xml_firmware_file*, *xbee_firmware_file=None*, *bootloader_firmware_file=None*, *timeout=None*, *progress_callback=None*)

Performs a firmware update operation of the XBee.

Parameters

- **`xml_firmware_file`** (`String`) – Path of the XML file that describes the firmware to upload.
- **`xbee_firmware_file`** (`String, optional, default=`None``) – Location of the XBee binary firmware file.
- **`bootloader_firmware_file`** (`String, optional, default=`None``) – Location of the bootloader binary firmware file.
- **`timeout`** (`Integer, optional, default=`None``) – Maximum time to wait for target read operations during the update process (seconds).
- **`progress_callback`** (`Function, optional, default=`None``) – Function to to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

Raises

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `OperationNotSupportedException` – If XBee does not support firmware update.
- `FirmwareUpdateException` – If there is any error during the firmware update.

write_changes()

Writes configurable parameter values to the non-volatile memory of the XBee so that parameter modifications persist through subsequent resets.

Parameters values remain in the device's memory until overwritten by subsequent use of this method.

If changes are made without writing them, the XBee 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_changes_enabled()* to get its status and *enable_apply_changes()* to enable/disable the option. Method *apply_changes()* can be used in order to manually apply the changes.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

class digi.xbee.devices.**DigiMeshDevice**(*port=None*, *baud_rate=None*, *data_bits=<sphinx.ext.autodoc.importer._MockObject object>*, *stop_bits=<sphinx.ext.autodoc.importer._MockObject object>*, *parity=<sphinx.ext.autodoc.importer._MockObject object>*, *flow_control=<FlowControl.NONE: None>*, *_sync_ops_timeout=4*, *comm_iface=None*)

Bases: *digi.xbee.devices.XBeeDevice*

This class represents a local DigiMesh XBee.

Class constructor. Instantiates a new *DigiMeshDevice* with the provided parameters.

Parameters

- **port** (*String*) – serial port identifier. Depends on operating system. e.g. '/dev/ttyUSB0' on 'GNU/Linux' or 'COM3' on Windows.
- **baud_rate** (*Integer*) – Serial port baud rate.
- **(Integer, default** (*flow_control*) – `serial.EIGHTBITS`): Port bitsize.
- **(Integer, default** – `serial.STOPBITS_ONE`): Port stop bits.
- **(Character, default** (*parity*) – `serial.PARITY_NONE`): Port parity.
- **(Integer, default** – `FlowControl.NONE`): port flow control.

  _sync_ops_timeout (Integer, default: 3): Read timeout (in seconds). comm_iface (*XBeeCommunicationInterface*): Communication interface.

Raises All exceptions raised by `XBeeDevice.__init__()` constructor.

See also:

XBeeDevice

XBeeDevice.__init__()

open (*force_settings=False*)

Override.

See also:

XBeeDevice.open()

get_protocol ()

Override.

See also:

XBeeDevice.get_protocol()

build_aggregate_routes ()

Forces all nodes in the network to automatically build routes to this node. The receiving node establishes a route back to this node.

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

send_data_64 (*x64addr*, *data*, *transmit_options=0*)

Blocking method. This method sends data to a remote XBee with the given 64-bit address.

This method waits for the packet response. The default timeout is XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS.

Parameters

- **x64addr** (*XBee64BitAddress*) – 64-bit address of the destination XBee.
- **data** (*String or Bytearray*) – Raw data to send.
- **transmit_options** (*Integer, optional*) – Transmit options, bitfield of *TransmitOptions*. Default to *TransmitOptions.NONE.value*.

Returns The response.

Return type *XBeePacket*

Raises

- ValueError – If *x64addr* or *data* is *None*.

- `TimeoutException` – If response is not received before the read timeout expires.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TransmitException` – If the status of the response received is not OK.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

`XBee64BitAddress`
`XBeePacket`

`send_data_async_64`(*x64addr*, *data*, *transmit_options=0*)
Non-blocking method. This method sends data to a remote XBee with the given 64-bit address.

This method does not wait for a response.

Parameters

- **`x64addr`** (*`XBee64BitAddress`*) – 64-bit address of the destination XBee.
- **`data`** (*`String or Bytearray`*) – Raw data to send.
- **`transmit_options`** (*`Integer, optional`*) – Transmit options, bitfield of *`TransmitOptions`*. Default to *TransmitOptions.NONE.value*.

Raises

- `ValueError` – If *x64addr* or *data* is *None*.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

`XBee64BitAddress`
`XBeePacket`

`read_expl_data`(*timeout=None*)
Reads new explicit data received by this XBee.

If *timeout* is specified, this method blocks until new data is received or the timeout expires, throwing a *`TimeoutException`* in this case.

Parameters **`timeout`** (*`Integer, optional`*) – Read timeout in seconds. If *None*, this method is non-blocking and returns *None* if there is no explicit data available.

Returns

Read message or *None* if this XBee did not receive new explicit data.

Return type *`ExplicitXBeeMessage`*

Raises

- `ValueError` – If a timeout is specified and is less than 0.

- `TimeoutException` – If a timeout is specified and no explicit data was received during that time.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

`ExplicitXBeeMessage`

`read_expl_data_from`(*remote_xbee*, *timeout=None*)

Reads new explicit data received from the given remote XBee.

If *timeout* is specified, this method blocks until new data is received or the timeout expires, throwing a *`TimeoutException`* in this case.

Parameters

- **`remote_xbee`** (*`RemoteXBeeDevice`*) – Remote XBee that sent the explicit data.
- **`timeout`** (*`Integer, optional`*) – Read timeout in seconds. If *None*, this method is non-blocking and returns *None* if there is no data available.

Returns

Read message sent by *remote_xbee* or *None* if this XBee did not receive new data from that node.

Return type *`ExplicitXBeeMessage`*

Raises

- `ValueError` – If a timeout is specified and is less than 0.
- `TimeoutException` – If a timeout is specified and no explicit data was received during that time.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

`ExplicitXBeeMessage`
`RemoteXBeeDevice`

`send_expl_data`(*remote_xbee*, *data*, *src_endpoint*, *dest_endpoint*, *cluster_id*, *profile_id*, *transmit_options=0*)

Blocking method. Sends the provided explicit data to the given XBee, source and destination end points, cluster and profile ids.

This method blocks until a success or error response arrives or the configured receive timeout expires. The default timeout is `XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS`.

Parameters

- **remote_xbee** (*RemoteXBeeDevice*) – Remote XBee to send data to.
- **data** (*String or Bytearray*) – 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 (between 0x0 and 0xFFFF)
- **profile_id** (*Integer*) – Profile ID of the transmission (between 0x0 and 0xFFFF)
- **transmit_options** (*Integer, optional*) – Transmit options, bitfield of *TransmitOptions*. Default to *TransmitOptions.NONE.value*.

Returns Response packet obtained after sending data.

Return type *XBeePacket*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TransmitException` – If the status of the response received is not OK.
- `XBeeException` – If the XBee's communication interface is closed.
- `ValueError` – if *cluster_id* or *profile_id* is less than 0x0 or greater than 0xFFFF.

See also:

RemoteXBeeDevice
XBeePacket

send_expl_data_broadcast (*data*, *src_endpoint*, *dest_endpoint*, *cluster_id*, *profile_id*, *transmit_options=0*)

Sends the provided explicit data to all the XBee nodes of the network (broadcast) using provided source and destination end points, cluster and profile ids.

This method blocks until 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 method *AbstractXBeeDevice.get_sync_ops_timeout()*.

Parameters

- **data** (*String or Bytearray*) – 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 (between 0x0 and 0xFFFF)
- **profile_id** (*Integer*) – Profile ID of the transmission (between 0x0 and 0xFFFF)

- **transmit_options** (*Integer, optional*) – Transmit options, bitfield of *TransmitOptions*. Default to *TransmitOptions.NONE.value*.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TransmitException` – If the status of the response received is not OK.
- `XBeeException` – If the XBee's communication interface is closed.
- `ValueError` – if *cluster_id* or *profile_id* is less than 0x0 or greater than 0xFFFF.

See also:

`XBeeDevice._send_expl_data()`

send_expl_data_async(*remote_xbee*, *data*, *src_endpoint*, *dest_endpoint*, *cluster_id*, *profile_id*, *transmit_options=0*)

Non-blocking method. Sends the provided explicit data to the given XBee, source and destination end points, cluster and profile ids.

Parameters

- **remote_xbee** (*RemoteXBeeDevice*) – Remote XBee to send data to.
- **data** (*String or Bytearray*) – 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 (between 0x0 and 0xFFFF)
- **profile_id** (*Integer*) – Profile ID of the transmission (between 0x0 and 0xFFFF)
- **transmit_options** (*Integer, optional*) – Transmit options, bitfield of *TransmitOptions*. Default to *TransmitOptions.NONE.value*.

Raises

- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the XBee's communication interface is closed.
- `ValueError` – if *cluster_id* or *profile_id* is less than 0x0 or greater than 0xFFFF.

See also:

RemoteXBeeDevice

get_neighbors(*neighbor_cb=None*, *finished_cb=None*, *timeout=None*)

Returns the neighbors of this XBee. If *neighbor_cb* is not defined, the process blocks during the specified timeout.

Parameters

- **neighbor_cb** (*Function, optional, default=`None`*) – Method called when a new neighbor is received. Receives two arguments:
  - The XBee that owns this new neighbor.
  - The new neighbor.
- **finished_cb** (*Function, optional, default=`None`*) – Method to execute when the process finishes. Receives two arguments:
  - The XBee that is searching for its neighbors.
  - A list with the discovered neighbors.
  - An error message if something went wrong.
- **timeout** (*Float, optional, default=`NeighborFinder.DEFAULT_TIMEOUT`*) – The timeout in seconds.

Returns

List of `Neighbor` when *neighbor_cb* is not defined, *None* otherwise (in this case neighbors are received in the callback).

Return type List

Raises `OperationNotSupportedException` – If XBee protocol is not DigiMesh.

See also:

`com.digi.models.zdo.Neighbor`

add_bluetooth_data_received_callback(*callback*)
Adds a callback for the event *BluetoothDataReceived*.

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

- The Bluetooth data as a Bytearray.

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

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

- The data received as an *XBeeMessage*.

add_expl_data_received_callback(*callback*)
Adds a callback for the event *ExplicitDataReceived*.

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

- The explicit data received as a *ExplicitXBeeMessage*.

add_fs_frame_received_callback(*callback*)
Adds a callback for the event *FileSystemFrameReceived*.

Parameters callback (*Function*) – The callback. Receives four arguments.

- Source (*AbstractXBeeDevice*): The node that sent the file system frame.
- Frame id (Integer): The received frame id.
- Command (*FSCmd*): The file system command.

- Receive options (Integer): Bitfield indicating receive options.

See also:

AbstractXBeeDevice
FSCmd
ReceiveOptions

add_io_sample_received_callback(*callback*)
Adds a callback for the event *IOSampleReceived*.

Parameters **callback** (*Function*) – The callback. Receives three arguments.

- The received IO sample as an *IOSample*.
- The remote XBee which sent the packet as a *RemoteXBeeDevice*.
- The time in which the packet was received as an Integer.

add_micropython_data_received_callback(*callback*)
Adds a callback for the event *MicroPythonDataReceived*.

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

- The MicroPython data as a Bytearray.

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

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

- The modem status as a *ModemStatus*.

add_packet_received_callback(*callback*)
Adds a callback for the event *PacketReceived*.

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

- The received packet as a *XBeeAPIPacket*.

add_route_received_callback(*callback*)
Adds a callback for the event *RouteReceived*. This works for Zigbee and Digimesh devices.

Parameters **callback** (*Function*) – The callback. Receives three arguments.

- source (*XBeeDevice*): The source node.
- destination (*RemoteXBeeDevice*): The destination node.
- **hops (List): List of intermediate hops from closest to source** to closest to destination (*RemoteXBeeDevice*).

See also:

XBeeDevice.del_route_received_callback()

add_socket_data_received_callback(*callback*)
Adds a callback for the event *SocketDataReceived*.

Parameters **callback** (*Function*) – The callback. Receives two arguments.

- The socket ID as an Integer.
- The data received as Bytearray.

add_socket_data_received_from_callback(*callback*)
Adds a callback for the event *SocketDataReceivedFrom*.

Parameters **callback** (*Function*) – The callback. Receives three arguments.

- The socket ID as an Integer.
- **Source address pair (host, port) where host is a string** representing an IPv4 address like '100.50.200.5', and port is an integer.
- The data received as Bytearray.

add_socket_state_received_callback(*callback*)
Adds a callback for the event *SocketStateReceived*.

Parameters **callback** (*Function*) – The callback. Receives two arguments.

- The socket ID as an Integer.
- The state received as a *SocketState*.

add_user_data_relay_received_callback(*callback*)
Adds a callback for the event *RelayDataReceived*.

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

- The relay data as a *UserDataRelayMessage*.

apply_changes()
Applies changes via 'AC' command.

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

apply_profile(*profile_path*, *timeout=None*, *progress_callback=None*)
Applies the given XBee profile to the XBee.

Parameters

- **profile_path** (*String*) – Path of the XBee profile file to apply.
- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the apply profile (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to receive progress information. Receives two arguments:
  - The current apply profile task as a String
  - The current apply profile task percentage as an Integer

Raises

- XBeeException – If the XBee's communication interface is closed.

- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `UpdateProfileException` – If there is any error applying the XBee profile.

close()

Closes the communication with the XBee.

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

comm_iface

Returns the hardware interface associated to the XBee.

Returns Hardware interface of the XBee.

Return type *XBeeCommunicationInterface*

See also:

XBeeCommunicationInterface

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 XBee object created.

Return type *XBeeDevice*

Raises `SerialException` – If the port to open does not exist or is already opened.

See also:

XBeeDevice

del_bluetooth_data_received_callback(*callback*)

Deletes a callback for the callback list of *BluetoothDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_data_received_callback(*callback*)

Deletes a callback for the callback list of *DataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_expl_data_received_callback(*callback*)
Deletes a callback for the callback list of *ExplicitDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_fs_frame_received_callback(*callback*)
Deletes a callback for the callback list of *FileSystemFrameReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_io_sample_received_callback(*callback*)
Deletes a callback for the callback list of *IOSampleReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_micropython_data_received_callback(*callback*)
Deletes a callback for the callback list of *MicroPythonDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_modem_status_received_callback(*callback*)
Deletes a callback for the callback list of *ModemStatusReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_packet_received_callback(*callback*)
Deletes a callback for the callback list of *PacketReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_route_received_callback(*callback*)
Deletes a callback for the callback list of *RouteReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

See also:

XBeeDevice.add_route_received_callback()

del_socket_data_received_callback(*callback*)
Deletes a callback for the callback list of *SocketDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_socket_data_received_from_callback(*callback*)
Deletes a callback for the callback list of *SocketDataReceivedFrom* event.

Parameters **callback** (*Function*) – The callback to delete.

del_socket_state_received_callback(*callback*)
Deletes a callback for the callback list of *SocketStateReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_user_data_relay_received_callback(*callback*)
Deletes a callback for the callback list of *RelayDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

determine_protocol(*hardware_version*, *firmware_version*)
Determines the XBee protocol based on the given hardware and firmware versions.

Parameters

- **hardware_version** (*Integer*) – Hardware version to get its protocol.
- **firmware_version** (*Bytearray*) – Firmware version to get its protocol.

Returns

XBee protocol corresponding to the given hardware and firmware versions.

Return type *XBeeProtocol*

disable_bluetooth()

Disables the Bluetooth interface of this XBee.

Note that your device must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

enable_apply_changes(*value*)

Sets apply changes flag.

Parameters **value** (*Boolean*) – *True* to enable apply changes flag, *False* to disable it.

enable_bluetooth()

Enables the Bluetooth interface of this XBee.

To work with this interface, you must also configure the Bluetooth password if not done previously. Use method *AbstractXBeeDevice.update_bluetooth_password()*.

Note that your XBee must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

execute_command(*parameter*, *value=None*, *apply=None*)

Executes the provided command.

Parameters

- **(String or** (*parameter*) – class: *.ATStringCommand*): AT command to execute.
- **value** (*bytearray, optional, default=`None`*) – Command value (if any).
- **apply** (*Boolean, optional, default=`None`*) – *True* to apply changes in XBee configuration, *False* not to apply them, *None* to use *is_apply_changes_enabled()* returned value.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.

- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

See also:

AbstractXBeeDevice.get_parameter()
AbstractXBeeDevice.set_parameter()
AbstractXBeeDevice.apply_changes()
AbstractXBeeDevice.write_changes()
AbstractXBeeDevice.is_apply_changes_enabled()
AbstractXBeeDevice.enable_apply_changes()

flush_queues()
Flushes the packets queue.

get_16bit_addr()
Returns the 16-bit address of the XBee.

Returns 16-bit address of the XBee.

Return type *XBee16BitAddress*

See also:

XBee16BitAddress

get_64bit_addr()
Returns the 64-bit address of the XBee.

Returns 64-bit address of the XBee.

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*) – IO line to get its ADC value.

Returns Analog value corresponding to the provided IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If response does not contain the value for the given IO line.

See also:

`IOLine`
`set_io_configuration()`

`get_api_output_mode``()`
Deprecated since version 1.3: Use *`get_api_output_mode_value()`*

Returns the API output mode of the XBee.

The API output mode determines the format of the data through the serial interface of the XBee.

Returns API output mode of the XBee.

Return type *`APIOutputMode`*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`APIOutputMode`

`get_api_output_mode_value``()`
Returns the API output mode of the XBee.

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

Returns the parameter value.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

`digi.xbee.models.mode.APIOutputModeBit`

`get_bluetooth_mac_addr``()`

Reads and returns the EUI-48 Bluetooth MAC address of this XBee following the format *00112233AABB*.

Note that your device must include Bluetooth Low Energy support.

Returns The Bluetooth MAC address.

Return type String

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

`get_current_frame_id``()`

Returns the last used frame ID.

Returns Last used frame ID.

Return type Integer

`get_dest_address``()`

Returns the 64-bit address of the XBee that is data destination.

Returns 64-bit address of destination XBee.

Return type *`XBee64BitAddress`*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`XBee64BitAddress`
`set_dest_address()`

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 response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.
- OperationNotSupportedException – If response does not contain the value for the given IO line.

See also:

IOLine
IOValue
set_io_configuration()

get_file_manager()

Returns the file system manager for the XBee.

Returns The file system manager.

Return type *FileSystemManager*

Raises FileSystemNotSupportedException – If the XBee does not support filesystem.

get_firmware_version()

Returns the firmware version of the XBee.

Returns Firmware version of the XBee.

Return type Bytearray

get_hardware_version()

Returns the hardware version of the XBee.

Returns Hardware version of the XBee.

Return type *HardwareVersion*

See also:

HardwareVersion

get_io_configuration(*io_line*)
Returns the configuration of the provided IO line.

Parameters **io_line** (*IOLine*) – IO line to get its configuration.

Returns IO mode of the IO line provided.

Return type *IOMode*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOLine
IOMode
set_io_configuration()

get_io_sampling_rate()
Returns the IO sampling rate of the XBee.

Returns IO sampling rate of XBee.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

set_io_sampling_rate()

get_network()
Returns the network of this XBee.

Returns The XBee network.

Return type *XBeeNetwork*

get_next_frame_id()
Returns the next frame ID of the XBee.

Returns The next frame ID of the XBee.

Return type Integer

get_node_id()

Returns the node identifier ('NI') value of the XBee.

Returns Node identifier ('NI') of the XBee.

Return type String

get_pan_id()

Returns the operating PAN ID of the XBee.

Returns Operating PAN ID of the XBee.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

set_pan_id()

get_parameter(*parameter*, *parameter_value=None*, *apply=None*)

Override.

See also:

AbstractXBeeDevice.get_parameter()

get_power_level()

Returns the power level of the XBee.

Returns Power level of the XBee.

Return type *PowerLevel*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

PowerLevel
set_power_level()

get_pwm_duty_cycle(*io_line*)

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

Parameters **io_line** (*IOLine*) – IO line to get its PWM duty cycle.

Returns PWM duty cycle of the given IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `ValueError` – If *io_line* has no PWM capability.

See also:

IOLine

get_role()

Gets the XBee role.

Returns the role of the XBee.

Return type *Role*

See also:

Role

get_route_to_node(*remote*, *timeout=10*, *force=True*)

Gets the route from this XBee to the given remote node.

For Zigbee:

- 'AR' parameter of the local node must be configured with a value different from 'FF'.
- Set *force* to *True* to force the Zigbee remote node to return its route independently of the local node configuration as high or low RAM concentrator ('DO' of the local value)

Parameters

- **remote** (*RemoteXBeeDevice*) – The remote node.
- **timeout** (*Float, optional, default=10*) – Maximum number of seconds to wait for the route.
- **force** (*Boolean*) – *True* to force asking for the route, *False* otherwise. Only for Zigbee.

Returns

Tuple containing route data:

- status (`TransmitStatus`): The transmit status.
- Tuple with route data (*None* if the route was not read in the provided timeout):
  - source (`RemoteXBeeDevice`): The source node of the route.
  - destination (`RemoteXBeeDevice`): The destination node of the route.
  - hops (List): List of intermediate nodes (`RemoteXBeeDevice`) ordered from closest to source to closest to destination node (source and destination not included).

Return type Tuple

get_sync_ops_timeout()

Returns the serial port read timeout.

Returns 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 are executed before user callbacks.

Returns `PacketReceived`

has_explicit_packets()

Returns if there are pending explicit packets to read. This does not include non-explicit packets.

Returns *True* if there are pending packets, *False* otherwise.

Return type Boolean

See also:

`XBeeDevice.has_packets()`

has_packets()

Returns if there are pending packets to read. This does not include explicit packets.

Returns *True* if there are pending packets, *False* otherwise.

Return type Boolean

See also:

`XBeeDevice.has_explicit_packets()`

is_apply_changes_enabled()

Returns whether apply changes flag is enabled.

Returns *True* if apply changes flag is enabled, *False* otherwise.

Return type Boolean

is_device_info_complete()

Returns whether XBee node information is complete.

Returns *True* if node information is complete, *False* otherwise.

Return type Boolean

See also:

AbstractXBeeDevice.read_device_info()

is_open()

Returns whether this XBee is open.

Returns Boolean. *True* if this XBee is open, *False* otherwise.

is_remote()

Override method.

See also:

AbstractXBeeDevice.is_remote()

log

Returns the XBee logger.

Returns The XBee device logger.

Return type `Logger`

operating_mode

Returns the operating mode of this XBee.

Returns *OperatingMode*. This XBee operating mode.

reachable

Returns whether the XBee is reachable.

Returns *True* if the device is reachable, *False* otherwise.

Return type Boolean

read_data(*timeout=None*)

Reads new data received by this XBee.

If *timeout* is specified, this method blocks until new data is received or the timeout expires, throwing a *TimeoutException* in this case.

Parameters **timeout** (*Integer, optional*) – Read timeout in seconds. If *None*, this method is non-blocking and returns *None* if no data is available.

Returns

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's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

XBeeMessage

read_data_from(*remote_xbee*, *timeout=None*)

Reads new data received from the given remote XBee.

If *timeout* is specified, this method blocks until new data is received or the timeout expires, throwing a *TimeoutException* in this case.

Parameters

- **remote_xbee** (*RemoteXBeeDevice*) – Remote XBee that sent the data.
- **timeout** (*Integer, optional*) – Read timeout in seconds. If *None*, this method is non-blocking and returns *None* if no data is available.

Returns

Read message sent by *remote_xbee* 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's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

XBeeMessage
RemoteXBeeDevice

read_device_info(*init=True*, *fire_event=True*)

Updates all instance parameters reading them from the XBee.

Parameters

- **init** (*Boolean, optional, default=`True`*) – If *False* only not initialized parameters are read, all if *True*.
- **fire_event** (*Boolean, optional, default=`True`*) – *True* to throw and update event if any parameter changed, *False* otherwise.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`AbstractXBeeDevice.is_device_info_complete()`

`read_io_sample``()`

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

Returns IO sample read from the XBee.

Return type *`IOSample`*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`IOSample`

`reset``()`

Override method.

See also:

`AbstractXBeeDevice.reset()`

`scan_counter`

Returns the scan counter for this node.

Returns The scan counter for this node.

Return type Integer

`send_bluetooth_data``(`*data*`)`

Sends the given data to the Bluetooth interface using a User Data Relay frame.

Parameters **`data`** (*`Bytearray`*) – Data to send.

Raises

- `InvalidOperatingModeException` – If the XBee'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*, *data*, *transmit_options=0*)
Blocking method. This method sends data to a remote XBee synchronously.

This method will wait for the packet response. The default timeout is `XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS`.

Parameters

- **remote_xbee** (*RemoteXBeeDevice*) – Remote XBee to send data to.
- **data** (*String or Bytearray*) – Raw data to send.
- **transmit_options** (*Integer, optional*) – Transmit options, bitfield of *TransmitOptions*. Default to *TransmitOptions.NONE.value*.

Returns The response.

Return type *XBeePacket*

Raises

- `ValueError` – If *remote_xbee* is *None*.
- `TimeoutException` – If response is not received before the read timeout expires.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TransmitException` – If the status of the response received is not OK.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

RemoteXBeeDevice
XBeePacket

send_data_async(*remote_xbee*, *data*, *transmit_options=0*)
Non-blocking method. This method sends data to a remote XBee.

This method does not wait for a response.

Parameters

- **remote_xbee** (*RemoteXBeeDevice*) – the remote XBee to send data to.
- **data** (*String or Bytearray*) – Raw data to send.
- **transmit_options** (*Integer, optional*) – Transmit options, bitfield of *TransmitOptions*. Default to *TransmitOptions.NONE.value*.

Raises

- `ValueError` – If *remote_xbee* is *None*.

- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the XBee's communication interface 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 until a success or error transmit status arrives or the configured receive timeout expires.

The received timeout is configured using method *`AbstractXBeeDevice.set_sync_ops_timeout()`* 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 response is not received before the read timeout expires.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TransmitException` – If the status of the response received is not OK.
- `XBeeException` – If the XBee's communication interface is closed.

`send_micropython_data`(*data*)

Sends the given data to the MicroPython interface using a User Data Relay frame.

Parameters **`data`** (*`Bytearray`*) – Data to send.

Raises

- `InvalidOperatingModeException` – If the XBee'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*)

Sends the packet and waits for the response. The packet to send is escaped depending on the current operating mode.

This method can be synchronous or asynchronous.

If synchronous, this method discards all response packets until it finds the one that has the appropriate frame ID, that is, the sent packet's frame ID.

If asynchronous, this method does not wait for any response and returns *None*.

Parameters

- **`packet`** (*`XBeePacket`*) – The packet to send.
- **`sync`** (*`Boolean`*) – *True* to wait for the response of the sent packet and return it, *False* otherwise.

Returns

Response packet if *sync* is *True*, *None* otherwise.

Return type *`XBeePacket`*

Raises

- `TimeoutException` – If *sync* is *True* and the response packet for the sent one cannot be read.
- `InvalidOperatingModeException` – If the XBee operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the packet listener is not running or the XBee's communication interface is closed.

See also:

`XBeePacket`

`send_packet_sync_and_get_response`(*packet_to_send*, *timeout=None*)

Sends the packet and waits for its corresponding response.

Parameters

- **`packet_to_send`** (*`XBeePacket`*) – The packet to transmit.
- **`timeout`** (*`Integer, optional, default=`None``*) – Number of seconds to wait. -1 to wait indefinitely.

Returns Received response packet.

Return type *`XBeePacket`*

Raises

- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TimeoutException` – If response is not received in the configured timeout.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

`XBeePacket`

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'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, if any.

Returns

Serial port of the XBee. *None* if the local XBee does not use serial communication.

Return type *XBeeSerialPort*

See also:

XBeeSerialPort

set_16bit_addr(*value*)
Sets the 16-bit address of the XBee.

Parameters **value** (*XBee16BitAddress*) – New 16-bit address of the XBee.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If the protocol is not 802.15.4.

set_api_output_mode(*api_output_mode*)
Deprecated since version 1.3: Use *set_api_output_mode_value()*

Sets the API output mode of the XBee.

Parameters **api_output_mode** (*APIOutputMode*) – New API output mode.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

`APIOutputMode`

`set_api_output_mode_value`(*api_output_mode*)

Sets the API output mode of the XBee.

Parameters **`api_output_mode`** (*`Integer`*) – New API output mode options. Calculate this value using the method `APIOutputModeBit.calculate_api_output_mode_value()` with a set of *`APIOutputModeBit`*.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

`APIOutputModeBit`

`set_dest_address`(*addr*)

Sets the 64-bit address of the XBee that is data destination.

Parameters **`addr`** (*`XBee64BitAddress`* or *`RemoteXBeeDevice`*) – Address itself or remote XBee to be data destination.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

- `ValueError` – If *addr* is *None*.

See also:

XBee64BitAddress
get_dest_address()

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 response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – Digital IO line to sets its value.
- **io_value** (*IOValue*) – IO value to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – IO line to configure.
- **io_mode** (*IOMode*) – IO mode to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOLine
IOMode
get_io_configuration()

set_io_sampling_rate(*rate*)
Sets the IO sampling rate of the XBee in seconds. A sample rate of 0 means the IO sampling feature is disabled.

Parameters **rate** (*Integer*) – New IO sampling rate of the XBee in seconds.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

get_io_sampling_rate()

set_node_id(*node_id*)
Sets the node identifier ('NI') value of the XBee.

Parameters **node_id** (*String*) – New node identifier ('NI') of the XBee.

Raises

- `ValueError` – If *node_id* is *None* or its length is greater than 20.
- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

set_pan_id(*value*)
Sets the operating PAN ID of the XBee.

Parameters **value** (*Bytearray*) – New operating PAN ID of the XBee. Must have only 1 or 2 bytes.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

get_pan_id()

set_parameter(*parameter*, *value*, *apply=None*)
Override.

See: *AbstractXBeeDevice.set_parameter()*

set_power_level(*power_level*)
Sets the power level of the XBee.

Parameters **power_level** (*PowerLevel*) – New power level of the XBee.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

PowerLevel
get_power_level()

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*) – IO Line to be assigned.
- **cycle** (*Integer*) – Duty cycle in % to be assigned. Must be between 0 and 100.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – Read timeout in seconds.

update_bluetooth_password(*new_password*)
Changes the Bluetooth password of this XBee with the new one provided.

Note that your device must include Bluetooth Low Energy support.

Parameters **new_password** (*String*) – New Bluetooth password.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

update_device_data_from(*device*)
Updates the current node information with provided data. This is only for internal use.

Parameters **device** (*AbstractXBeeDevice*) – XBee to get the data from.

Returns *True* if the node data has been updated, *False* otherwise.

Return type Boolean

update_firmware(*xml_firmware_file*, *xbee_firmware_file=None*, *bootloader_firmware_file=None*, *timeout=None*, *progress_callback=None*)
Performs a firmware update operation of the XBee.

Parameters

- **xml_firmware_file** (*String*) – Path of the XML file that describes the firmware to upload.
- **xbee_firmware_file** (*String, optional, default=`None`*) – Location of the XBee binary firmware file.
- **bootloader_firmware_file** (*String, optional, default=`None`*) – Location of the bootloader binary firmware file.

- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the update process (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

Raises

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `OperationNotSupportedException` – If XBee does not support firmware update.
- `FirmwareUpdateException` – If there is any error during the firmware update.

write_changes()

Writes configurable parameter values to the non-volatile memory of the XBee so that parameter modifications persist through subsequent resets.

Parameters values remain in the device's memory until overwritten by subsequent use of this method.

If changes are made without writing them, the XBee 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_changes_enabled()`* to get its status and *`enable_apply_changes()`* to enable/disable the option. Method *`apply_changes()`* can be used in order to manually apply the changes.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

class digi.xbee.devices.**DigiPointDevice**(*port=None, baud_rate=None, data_bits=<sphinx.ext.autodoc.importer._MockObject object>, stop_bits=<sphinx.ext.autodoc.importer._MockObject object>, parity=<sphinx.ext.autodoc.importer._MockObject object>, flow_control=<FlowControl.NONE: None>, _sync_ops_timeout=4, comm_iface=None*)

Bases: *`digi.xbee.devices.XBeeDevice`*

This class represents a local DigiPoint XBee.

Class constructor. Instantiates a new *`DigiPointDevice`* with the provided parameters.

Parameters

- **port** (*String*) – Serial port identifier. Depends on operating system. e.g. '/dev/ttyUSB0' on 'GNU/Linux' or 'COM3' on Windows.
- **baud_rate** (*Integer*) – Serial port baud rate.

- **(Integer, default** (*_sync_ops_timeout*) – serial.EIGHTBITS): Port bitsize.
- **(Integer, default** – serial.STOPBITS_ONE): Port stop bits.
- **(Character, default** (*parity*) – serial.PARITY_NONE): Port parity.
- **(Integer, default** – FlowControl.NONE): Port flow control.
- **(Integer, default** – 3): Read timeout (in seconds).
- **comm_iface** (*XBeeCommunicationInterface*) – Communication interface.

Raises All exceptions raised by `XBeeDevice.__init__()` constructor.

See also:

XBeeDevice
`XBeeDevice.__init__()`

open (*force_settings=False*)
Override.

See also:

XBeeDevice.open()

get_protocol ()
Override.

See also:

XBeeDevice.get_protocol()

send_data_64_16 (*x64addr*, *x16addr*, *data*, *transmit_options=0*)
Blocking method. This method sends data to the remote XBee with the given 64-bit/16-bit address.

This method waits for the packet response. The default timeout is `XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS`.

Parameters

- **x64addr** (*XBee64BitAddress*) – 64-bit address of the destination XBee.
- **x16addr** (*XBee16BitAddress*) – 16-bit address of the destination XBee, *XBee16BitAddress.UNKNOWN_ADDRESS* if unknown.
- **data** (*String or Bytearray*) – Raw data to send.
- **transmit_options** (*Integer, optional*) – Transmit options, bitfield of *TransmitOptions*. Default to *TransmitOptions.NONE.value*.

Returns The response.

Return type *XBeePacket*

Raises

- `ValueError` – If *x64addr*, *x16addr* or *data* is *None*.
- `TimeoutException` – If response is not received before the read timeout expires.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TransmitException` – If the status of the response received is not OK.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

`XBee64BitAddress`
`XBee16BitAddress`
`XBeePacket`

`send_data_async_64_16`(*x64addr*, *x16addr*, *data*, *transmit_options=0*)

Non-blocking method. This method sends data to a remote XBee with the given 64-bit/16-bit address.

This method does not wait for a response.

Parameters

- **`x64addr`** (*`XBee64BitAddress`*) – 64-bit address of the destination XBee.
- **`x16addr`** (*`XBee16BitAddress`*) – 16-bit address of the destination XBee, *`XBee16BitAddress.UNKNOWN_ADDRESS`* if unknown.
- **`data`** (*`String or Bytearray`*) – Raw data to send.
- **`transmit_options`** (*`Integer, optional`*) – Transmit options, bitfield of *`TransmitOptions`*. Default to *TransmitOptions.NONE.value*.

Raises

- `ValueError` – If *x64addr*, *x16addr* or *data* is *None*.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

`XBee64BitAddress`
`XBee16BitAddress`
`XBeePacket`

`read_expl_data`(*timeout=None*)

Reads new explicit data received by this XBee.

If *timeout* is specified, this method blocks until new data is received or the timeout expires, throwing a *`TimeoutException`* in this case.

Parameters **`timeout`** (*`Integer, optional`*) – Read timeout in seconds. If *None*, this method is non-blocking and returns *None* if there is no explicit data available.

Returns

Read message or *None* if this XBee did not receive new explicit data.

Return type *ExplicitXBeeMessage*

Raises

- `ValueError` – If a timeout is specified and is less than 0.
- `TimeoutException` – If a timeout is specified and no explicit data was received during that time.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

ExplicitXBeeMessage

read_expl_data_from(*remote_xbee*, *timeout=None*)

Reads new explicit data received from the given remote XBee.

If *timeout* is specified, this method blocks until new data is received or the timeout expires, throwing a *TimeoutException* in this case.

Parameters

- **remote_xbee** (*RemoteXBeeDevice*) – Remote XBee that sent the explicit data.
- **timeout** (*Integer, optional*) – Read timeout in seconds. If *None*, this method is non-blocking and returns *None* if there is no data available.

Returns

Read message sent by *remote_xbee* or *None* if this XBee did not receive new data from that node.

Return type *ExplicitXBeeMessage*

Raises

- `ValueError` – If a timeout is specified and is less than 0.
- `TimeoutException` – If a timeout is specified and no explicit data was received during that time.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

ExplicitXBeeMessage
RemoteXBeeDevice

send_expl_data(*remote_xbee*, *data*, *src_endpoint*, *dest_endpoint*, *cluster_id*, *profile_id*, *transmit_options=0*)

Blocking method. Sends the provided explicit data to the given XBee, source and destination end points, cluster and profile ids.

This method blocks until a success or error response arrives or the configured receive timeout expires. The default timeout is `XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS`.

Parameters

- **remote_xbee** (*RemoteXBeeDevice*) – Remote XBee to send data to.
- **data** (*String or Bytearray*) – 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 (between 0x0 and 0xFFFF)
- **profile_id** (*Integer*) – Profile ID of the transmission (between 0x0 and 0xFFFF)
- **transmit_options** (*Integer, optional*) – Transmit options, bitfield of *TransmitOptions*. Default to *TransmitOptions.NONE.value*.

Returns Response packet obtained after sending data.

Return type *XBeePacket*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TransmitException` – If the status of the response received is not OK.
- `XBeeException` – If the XBee's communication interface is closed.
- `ValueError` – if *cluster_id* or *profile_id* is less than 0x0 or greater than 0xFFFF.

See also:

RemoteXBeeDevice
XBeePacket

send_expl_data_broadcast(*data*, *src_endpoint*, *dest_endpoint*, *cluster_id*, *profile_id*, *transmit_options=0*)

Sends the provided explicit data to all the XBee nodes of the network (broadcast) using provided source and destination end points, cluster and profile ids.

This method blocks until 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 method *AbstractXBeeDevice.get_sync_ops_timeout()*.

Parameters

- **data** (*String or Bytearray*) – 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 (between 0x0 and 0xFFFF)
- **profile_id** (*Integer*) – Profile ID of the transmission (between 0x0 and 0xFFFF)
- **transmit_options** (*Integer, optional*) – Transmit options, bitfield of *TransmitOptions*. Default to *TransmitOptions.NONE.value*.

Raises

- TimeoutException – If response is not received before the read timeout expires.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- TransmitException – If the status of the response received is not OK.
- XBeeException – If the XBee's communication interface is closed.
- ValueError – if *cluster_id* or *profile_id* is less than 0x0 or greater than 0xFFFF.

See also:

XBeeDevice._send_expl_data()

send_expl_data_async(*remote_xbee*, *data*, *src_endpoint*, *dest_endpoint*, *cluster_id*, *profile_id*, *transmit_options=0*)

Non-blocking method. Sends the provided explicit data to the given XBee, source and destination end points, cluster and profile ids.

Parameters

- **remote_xbee** (*RemoteXBeeDevice*) – Remote XBee to send data to.
- **data** (*String or Bytearray*) – 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 (between 0x0 and 0xFFFF)
- **profile_id** (*Integer*) – Profile ID of the transmission (between 0x0 and 0xFFFF)
- **transmit_options** (*Integer, optional*) – Transmit options, bitfield of *TransmitOptions*. Default to *TransmitOptions.NONE.value*.

Raises

- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- XBeeException – If the XBee's communication interface is closed.
- ValueError – if *cluster_id* or *profile_id* is less than 0x0 or greater than 0xFFFF.

See also:

RemoteXBeeDevice

add_bluetooth_data_received_callback(*callback*)
Adds a callback for the event *BluetoothDataReceived*.

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

- The Bluetooth data as a Bytearray.

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

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

- The data received as an *XBeeMessage*.

add_expl_data_received_callback(*callback*)
Adds a callback for the event *ExplicitDataReceived*.

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

- The explicit data received as a *ExplicitXBeeMessage*.

add_fs_frame_received_callback(*callback*)
Adds a callback for the event *FileSystemFrameReceived*.

Parameters **callback** (*Function*) – The callback. Receives four arguments.

- Source (*AbstractXBeeDevice*): The node that sent the file system frame.
- Frame id (Integer): The received frame id.
- Command (*FSCmd*): The file system command.
- Receive options (Integer): Bitfield indicating receive options.

See also:

AbstractXBeeDevice
FSCmd
ReceiveOptions

add_io_sample_received_callback(*callback*)
Adds a callback for the event *IOSampleReceived*.

Parameters **callback** (*Function*) – The callback. Receives three arguments.

- The received IO sample as an *IOSample*.
- The remote XBee which sent the packet as a *RemoteXBeeDevice*.
- The time in which the packet was received as an Integer.

add_micropython_data_received_callback(*callback*)
Adds a callback for the event *MicroPythonDataReceived*.

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

- The MicroPython data as a Bytearray.

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

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

- The modem status as a *ModemStatus*.

add_packet_received_callback(*callback*)
Adds a callback for the event *PacketReceived*.

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

- The received packet as a *XBeeAPIPacket*.

add_route_received_callback(*callback*)
Adds a callback for the event *RouteReceived*. This works for Zigbee and Digimesh devices.

Parameters **callback** (*Function*) – The callback. Receives three arguments.

- source (*XBeeDevice*): The source node.
- destination (*RemoteXBeeDevice*): The destination node.
- **hops (List): List of intermediate hops from closest to source** to closest to destination (*RemoteXBeeDevice*).

See also:

XBeeDevice.del_route_received_callback()

add_socket_data_received_callback(*callback*)
Adds a callback for the event *SocketDataReceived*.

Parameters **callback** (*Function*) – The callback. Receives two arguments.

- The socket ID as an Integer.
- The data received as Bytearray.

add_socket_data_received_from_callback(*callback*)
Adds a callback for the event *SocketDataReceivedFrom*.

Parameters **callback** (*Function*) – The callback. Receives three arguments.

- The socket ID as an Integer.
- **Source address pair (host, port) where host is a string** representing an IPv4 address like '100.50.200.5', and port is an integer.
- The data received as Bytearray.

add_socket_state_received_callback(*callback*)
Adds a callback for the event *SocketStateReceived*.

Parameters **callback** (*Function*) – The callback. Receives two arguments.

- The socket ID as an Integer.
- The state received as a *SocketState*.

add_user_data_relay_received_callback(*callback*)
Adds a callback for the event *RelayDataReceived*.

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

- The relay data as a *UserDataRelayMessage*.

apply_changes()

Applies changes via 'AC' command.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

apply_profile(*profile_path*, *timeout=None*, *progress_callback=None*)

Applies the given XBee profile to the XBee.

Parameters

- **profile_path** (*String*) – Path of the XBee profile file to apply.
- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the apply profile (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to receive progress information. Receives two arguments:
  - The current apply profile task as a String
  - The current apply profile task percentage as an Integer

Raises

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `UpdateProfileException` – If there is any error applying the XBee profile.

close()

Closes the communication with the XBee.

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

comm_iface

Returns the hardware interface associated to the XBee.

Returns Hardware interface of the XBee.

Return type *XBeeCommunicationInterface*

See also:

XBeeCommunicationInterface

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 XBee object created.

Return type *XBeeDevice*

Raises `SerialException` – If the port to open does not exist or is already opened.

See also:

XBeeDevice

`del_bluetooth_data_received_callback`(*callback*)
Deletes a callback for the callback list of *BluetoothDataReceived* event.

Parameters **`callback`** (*Function*) – The callback to delete.

`del_data_received_callback`(*callback*)
Deletes a callback for the callback list of *DataReceived* event.

Parameters **`callback`** (*Function*) – The callback to delete.

`del_expl_data_received_callback`(*callback*)
Deletes a callback for the callback list of *ExplicitDataReceived* event.

Parameters **`callback`** (*Function*) – The callback to delete.

`del_fs_frame_received_callback`(*callback*)
Deletes a callback for the callback list of *FileSystemFrameReceived* event.

Parameters **`callback`** (*Function*) – The callback to delete.

`del_io_sample_received_callback`(*callback*)
Deletes a callback for the callback list of *IOSampleReceived* event.

Parameters **`callback`** (*Function*) – The callback to delete.

`del_micropython_data_received_callback`(*callback*)
Deletes a callback for the callback list of *MicroPythonDataReceived* event.

Parameters **`callback`** (*Function*) – The callback to delete.

`del_modem_status_received_callback`(*callback*)
Deletes a callback for the callback list of *ModemStatusReceived* event.

Parameters **`callback`** (*Function*) – The callback to delete.

`del_packet_received_callback`(*callback*)
Deletes a callback for the callback list of *PacketReceived* event.

Parameters **`callback`** (*Function*) – The callback to delete.

`del_route_received_callback`(*callback*)
Deletes a callback for the callback list of *RouteReceived* event.

Parameters **`callback`** (*`Function`*) – The callback to delete.

See also:

`XBeeDevice.add_route_received_callback()`

`del_socket_data_received_callback`(*callback*)
Deletes a callback for the callback list of *`SocketDataReceived`* event.

Parameters **`callback`** (*`Function`*) – The callback to delete.

`del_socket_data_received_from_callback`(*callback*)
Deletes a callback for the callback list of *`SocketDataReceivedFrom`* event.

Parameters **`callback`** (*`Function`*) – The callback to delete.

`del_socket_state_received_callback`(*callback*)
Deletes a callback for the callback list of *`SocketStateReceived`* event.

Parameters **`callback`** (*`Function`*) – The callback to delete.

`del_user_data_relay_received_callback`(*callback*)
Deletes a callback for the callback list of *`RelayDataReceived`* event.

Parameters **`callback`** (*`Function`*) – The callback to delete.

`determine_protocol`(*hardware_version*, *firmware_version*)
Determines the XBee protocol based on the given hardware and firmware versions.

Parameters

- **`hardware_version`** (*`Integer`*) – Hardware version to get its protocol.
- **`firmware_version`** (*`Bytearray`*) – Firmware version to get its protocol.

Returns

XBee protocol corresponding to the given hardware and firmware versions.

Return type *`XBeeProtocol`*

`disable_bluetooth`()
Disables the Bluetooth interface of this XBee.

Note that your device must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

`enable_apply_changes`(*value*)
Sets apply changes flag.

Parameters **`value`** (*`Boolean`*) – *True* to enable apply changes flag, *False* to disable it.

enable_bluetooth()

Enables the Bluetooth interface of this XBee.

To work with this interface, you must also configure the Bluetooth password if not done previously. Use method *AbstractXBeeDevice.update_bluetooth_password()*.

Note that your XBee must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

execute_command(*parameter*, *value=None*, *apply=None*)

Executes the provided command.

Parameters

- **(String or** (*parameter*) – class: *.ATStringCommand*): AT command to execute.
- **value** (*bytearray, optional, default=`None`*) – Command value (if any).
- **apply** (*Boolean, optional, default=`None`*) – *True* to apply changes in XBee configuration, *False* not to apply them, *None* to use *is_apply_changes_enabled()* returned value.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

AbstractXBeeDevice.get_parameter()
AbstractXBeeDevice.set_parameter()
AbstractXBeeDevice.apply_changes()
AbstractXBeeDevice.write_changes()
AbstractXBeeDevice.is_apply_changes_enabled()
AbstractXBeeDevice.enable_apply_changes()

flush_queues()

Flushes the packets queue.

get_16bit_addr()

Returns the 16-bit address of the XBee.

Returns 16-bit address of the XBee.

Return type *XBee16BitAddress*

See also:

XBee16BitAddress

get_64bit_addr()
Returns the 64-bit address of the XBee.

Returns 64-bit address of the XBee.

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*) – IO line to get its ADC value.

Returns Analog value corresponding to the provided IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If response does not contain the value for the given IO line.

See also:

IOLine
set_io_configuration()

get_api_output_mode()
Deprecated since version 1.3: Use *get_api_output_mode_value()*

Returns the API output mode of the XBee.

The API output mode determines the format of the data through the serial interface of the XBee.

Returns API output mode of the XBee.

Return type *APIOutputMode*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

APIOutputMode

get_api_output_mode_value()

Returns the API output mode of the XBee.

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

Returns the parameter value.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

digi.xbee.models.mode.APIOutputModeBit

get_bluetooth_mac_addr()

Reads and returns the EUI-48 Bluetooth MAC address of this XBee following the format *00112233AABB*.

Note that your device must include Bluetooth Low Energy support.

Returns The Bluetooth MAC address.

Return type String

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.

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

`get_current_frame_id`()

Returns the last used frame ID.

Returns Last used frame ID.

Return type Integer

`get_dest_address`()

Returns the 64-bit address of the XBee that is data destination.

Returns 64-bit address of destination XBee.

Return type *`XBee64BitAddress`*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`XBee64BitAddress`
`set_dest_address()`

`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 response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If response does not contain the value for the given IO line.

See also:

IOLine
IOValue
set_io_configuration()

get_file_manager()
Returns the file system manager for the XBee.

Returns The file system manager.

Return type *FileSystemManager*

Raises `FileSystemNotSupportedException` – If the XBee does not support filesystem.

get_firmware_version()
Returns the firmware version of the XBee.

Returns Firmware version of the XBee.

Return type Bytearray

get_hardware_version()
Returns the hardware version of the XBee.

Returns Hardware version of the XBee.

Return type *HardwareVersion*

See also:

HardwareVersion

get_io_configuration(*io_line*)
Returns the configuration of the provided IO line.

Parameters **io_line** (*IOLine*) – IO line to get its configuration.

Returns IO mode of the IO line provided.

Return type *IOMode*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOLine
IOMode
set_io_configuration()

get_io_sampling_rate()
Returns the IO sampling rate of the XBee.

Returns IO sampling rate of XBee.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

set_io_sampling_rate()

get_network()
Returns the network of this XBee.

Returns The XBee network.

Return type *XBeeNetwork*

get_next_frame_id()
Returns the next frame ID of the XBee.

Returns The next frame ID of the XBee.

Return type Integer

get_node_id()
Returns the node identifier ('NI') value of the XBee.

Returns Node identifier ('NI') of the XBee.

Return type String

get_pan_id()
Returns the operating PAN ID of the XBee.

Returns Operating PAN ID of the XBee.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

set_pan_id()

get_parameter(*parameter*, *parameter_value=None*, *apply=None*)

Override.

See also:

AbstractXBeeDevice.get_parameter()

get_power_level()

Returns the power level of the XBee.

Returns Power level of the XBee.

Return type *PowerLevel*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

PowerLevel
set_power_level()

get_pwm_duty_cycle(*io_line*)

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

Parameters **io_line** (*IOLine*) – IO line to get its PWM duty cycle.

Returns PWM duty cycle of the given IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `ValueError` – If *io_line* has no PWM capability.

See also:

IOLine

get_role()

Gets the XBee role.

Returns the role of the XBee.

Return type *Role*

See also:

Role

get_route_to_node(*remote*, *timeout=10*, *force=True*)

Gets the route from this XBee to the given remote node.

For Zigbee:

- 'AR' parameter of the local node must be configured with a value different from 'FF'.
- Set *force* to *True* to force the Zigbee remote node to return its route independently of the local node configuration as high or low RAM concentrator ('DO' of the local value)

Parameters

- **remote** (*RemoteXBeeDevice*) – The remote node.
- **timeout** (*Float, optional, default=10*) – Maximum number of seconds to wait for the route.
- **force** (*Boolean*) – *True* to force asking for the route, *False* otherwise. Only for Zigbee.

Returns

Tuple containing route data:

- status (*TransmitStatus*): The transmit status.
- Tuple with route data (*None* if the route was not read in the provided timeout):
  - source (*RemoteXBeeDevice*): The source node of the route.
  - destination (*RemoteXBeeDevice*): The destination node of the route.
  - hops (List): List of intermediate nodes (*RemoteXBeeDevice*) ordered from closest to source to closest to destination node (source and destination not included).

Return type Tuple

get_sync_ops_timeout()

Returns the serial port read timeout.

Returns 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 are executed before user callbacks.

Returns *PacketReceived*

has_explicit_packets()

Returns if there are pending explicit packets to read. This does not include non-explicit packets.

Returns *True* if there are pending packets, *False* otherwise.

Return type Boolean

See also:

XBeeDevice.has_packets()

has_packets()

Returns if there are pending packets to read. This does not include explicit packets.

Returns *True* if there are pending packets, *False* otherwise.

Return type Boolean

See also:

XBeeDevice.has_explicit_packets()

is_apply_changes_enabled()

Returns whether apply changes flag is enabled.

Returns *True* if apply changes flag is enabled, *False* otherwise.

Return type Boolean

is_device_info_complete()

Returns whether XBee node information is complete.

Returns *True* if node information is complete, *False* otherwise.

Return type Boolean

See also:

AbstractXBeeDevice.read_device_info()

is_open()

Returns whether this XBee is open.

Returns Boolean. *True* if this XBee is open, *False* otherwise.

is_remote()

Override method.

See also:

AbstractXBeeDevice.is_remote()

log

Returns the XBee logger.

Returns The XBee device logger.

Return type `Logger`

operating_mode

Returns the operating mode of this XBee.

Returns *OperatingMode*. This XBee operating mode.

reachable

Returns whether the XBee is reachable.

Returns *True* if the device is reachable, *False* otherwise.

Return type Boolean

read_data (*timeout=None*)

Reads new data received by this XBee.

If *timeout* is specified, this method blocks until new data is received or the timeout expires, throwing a *TimeoutException* in this case.

Parameters timeout (*Integer, optional*) – Read timeout in seconds. If *None*, this method is non-blocking and returns *None* if no data is available.

Returns

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's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

XBeeMessage

read_data_from (*remote_xbee, timeout=None*)

Reads new data received from the given remote XBee.

If *timeout* is specified, this method blocks until new data is received or the timeout expires, throwing a *TimeoutException* in this case.

Parameters

- **remote_xbee** (*RemoteXBeeDevice*) – Remote XBee that sent the data.
- **timeout** (*Integer, optional*) – Read timeout in seconds. If *None*, this method is non-blocking and returns *None* if no data is available.

Returns

Read message sent by *remote_xbee* 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's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

XBeeMessage

RemoteXBeeDevice

read_device_info(*init=True*, *fire_event=True*)

Updates all instance parameters reading them from the XBee.

Parameters

- **init** (*Boolean, optional, default=`True`*) – If *False* only not initialized parameters are read, all if *True*.
- **fire_event** (*Boolean, optional, default=`True`*) – *True* to throw and update event if any parameter changed, *False* otherwise.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

AbstractXBeeDevice.is_device_info_complete()

read_io_sample()

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

Returns IO sample read from the XBee.

Return type *IOSample*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.

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

See also:

`IOSample`

`reset`()

Override method.

See also:

`AbstractXBeeDevice.reset()`

`scan_counter`

Returns the scan counter for this node.

Returns The scan counter for this node.

Return type Integer

`send_bluetooth_data`(*data*)

Sends the given data to the Bluetooth interface using a User Data Relay frame.

Parameters **`data`** (*`Bytearray`*) – Data to send.

Raises

- `InvalidOperatingModeException` – If the XBee'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*, *data*, *transmit_options=0*)

Blocking method. This method sends data to a remote XBee synchronously.

This method will wait for the packet response. The default timeout is `XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS`.

Parameters

- **`remote_xbee`** (*`RemoteXBeeDevice`*) – Remote XBee to send data to.
- **`data`** (*`String or Bytearray`*) – Raw data to send.
- **`transmit_options`** (*`Integer, optional`*) – Transmit options, bitfield of *`TransmitOptions`*. Default to *TransmitOptions.NONE.value*.

Returns The response.

Return type *XBeePacket*

Raises

- `ValueError` – If *remote_xbee* is *None*.
- `TimeoutException` – If response is not received before the read timeout expires.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TransmitException` – If the status of the response received is not OK.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

RemoteXBeeDevice
XBeePacket

send_data_async (*remote_xbee*, *data*, *transmit_options=0*)
Non-blocking method. This method sends data to a remote XBee.

This method does not wait for a response.

Parameters

- **remote_xbee** (*RemoteXBeeDevice*) – the remote XBee to send data to.
- **data** (*String or Bytearray*) – Raw data to send.
- **transmit_options** (*Integer, optional*) – Transmit options, bitfield of *TransmitOptions*. Default to *TransmitOptions.NONE.value*.

Raises

- `ValueError` – If *remote_xbee* is *None*.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the XBee's communication interface 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 until a success or error transmit status arrives or the configured receive timeout expires.

The received timeout is configured using method *AbstractXBeeDevice.set_sync_ops_timeout()* 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 response is not received before the read timeout expires.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- TransmitException – If the status of the response received is not OK.
- XBeeException – If the XBee's communication interface is closed.

send_micropython_data(*data*)

Sends the given data to the MicroPython interface using a User Data Relay frame.

Parameters **data** (*Bytearray*) – Data to send.

Raises

- InvalidOperatingModeException – If the XBee'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*)

Sends the packet and waits for the response. The packet to send is escaped depending on the current operating mode.

This method can be synchronous or asynchronous.

If synchronous, this method discards all response packets until it finds the one that has the appropriate frame ID, that is, the sent packet's frame ID.

If asynchronous, this method does not wait for any response and returns *None*.

Parameters

- **packet** (*XBeePacket*) – The packet to send.
- **sync** (*Boolean*) – *True* to wait for the response of the sent packet and return it, *False* otherwise.

Returns

Response packet if *sync* is *True*, *None* otherwise.

Return type *XBeePacket*

Raises

- TimeoutException – If *sync* is *True* and the response packet for the sent one cannot be read.
- InvalidOperatingModeException – If the XBee operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

- `XBeeException` – If the packet listener is not running or the XBee's communication interface is closed.

See also:

XBeePacket

send_packet_sync_and_get_response (*packet_to_send*, *timeout=None*)
Sends the packet and waits for its corresponding response.

Parameters

- **packet_to_send** (*XBeePacket*) – The packet to transmit.
- **timeout** (*Integer, optional, default=`None`*) – Number of seconds to wait. -1 to wait indefinitely.

Returns Received response packet.

Return type *XBeePacket*

Raises

- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TimeoutException` – If response is not received in the configured timeout.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

XBeePacket

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'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, if any.

Returns

Serial port of the XBee. ***None*** **if the** local XBee does not use serial communication.

Return type *XBeeSerialPort*

See also:

XBeeSerialPort

set_16bit_addr(*value*)
Sets the 16-bit address of the XBee.

Parameters **value** (*XBee16BitAddress*) – New 16-bit address of the XBee.

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.
- OperationNotSupportedException – If the protocol is not 802.15.4.

set_api_output_mode(*api_output_mode*)
Deprecated since version 1.3: Use *set_api_output_mode_value()*

Sets the API output mode of the XBee.

Parameters **api_output_mode** (*APIOutputMode*) – New API output mode.

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.
- OperationNotSupportedException – If it is not supported by the current protocol.

See also:

APIOutputMode

set_api_output_mode_value(*api_output_mode*)
Sets the API output mode of the XBee.

Parameters **api_output_mode** (*Integer*) – New API output mode options. Calculate this value using the method `APIOutputModeBit.calculate_api_output_mode_value()` with a set of *APIOutputModeBit*.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

APIOutputModeBit

set_dest_address(*addr*)

Sets the 64-bit address of the XBee that is data destination.

Parameters **addr** (*XBee64BitAddress* or *RemoteXBeeDevice*) – Address itself or remote XBee to be data destination.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `ValueError` – If *addr* is *None*.

See also:

XBee64BitAddress

get_dest_address()

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 response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

- `ATCommandException` – If 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`*) – Digital IO line to sets its value.
- **`io_value`** (*`IOValue`*) – IO value to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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`*) – IO line to configure.
- **`io_mode`** (*`IOMode`*) – IO mode to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`IOLine`
`IOMode`
`get_io_configuration()`

set_io_sampling_rate(*rate*)

Sets the IO sampling rate of the XBee in seconds. A sample rate of 0 means the IO sampling feature is disabled.

Parameters **rate** (*Integer*) – New IO sampling rate of the XBee in seconds.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

get_io_sampling_rate()

set_node_id(*node_id*)

Sets the node identifier ('NI') value of the XBee.

Parameters **node_id** (*String*) – New node identifier ('NI') of the XBee.

Raises

- `ValueError` – If *node_id* is *None* or its length is greater than 20.
- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

set_pan_id(*value*)

Sets the operating PAN ID of the XBee.

Parameters **value** (*Bytearray*) – New operating PAN ID of the XBee. Must have only 1 or 2 bytes.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

get_pan_id()

set_parameter(*parameter*, *value*, *apply=None*)
Override.

See: *AbstractXBeeDevice.set_parameter()*

set_power_level(*power_level*)
Sets the power level of the XBee.

Parameters **power_level** (*PowerLevel*) – New power level of the XBee.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

PowerLevel
get_power_level()

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*) – IO Line to be assigned.
- **cycle** (*Integer*) – Duty cycle in % to be assigned. Must be between 0 and 100.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – Read timeout in seconds.

update_bluetooth_password(*new_password*)

Changes the Bluetooth password of this XBee with the new one provided.

Note that your device must include Bluetooth Low Energy support.

Parameters **new_password** (*String*) – New Bluetooth password.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

update_device_data_from(*device*)

Updates the current node information with provided data. This is only for internal use.

Parameters **device** (*AbstractXBeeDevice*) – XBee to get the data from.

Returns *True* if the node data has been updated, *False* otherwise.

Return type Boolean

update_firmware(*xml_firmware_file*, *xbee_firmware_file=None*, *bootloader_firmware_file=None*, *timeout=None*, *progress_callback=None*)

Performs a firmware update operation of the XBee.

Parameters

- **xml_firmware_file** (*String*) – Path of the XML file that describes the firmware to upload.
- **xbee_firmware_file** (*String, optional, default=`None`*) – Location of the XBee binary firmware file.
- **bootloader_firmware_file** (*String, optional, default=`None`*) – Location of the bootloader binary firmware file.
- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the update process (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

Raises

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `OperationNotSupportedException` – If XBee does not support firmware update.
- `FirmwareUpdateException` – If there is any error during the firmware update.

write_changes()

Writes configurable parameter values to the non-volatile memory of the XBee so that parameter modifications persist through subsequent resets.

Parameters values remain in the device's memory until overwritten by subsequent use of this method.

If changes are made without writing them, the XBee 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_changes_enabled()` to get its status and `enable_apply_changes()` to enable/disable the option. Method `apply_changes()` can be used in order to manually apply the changes.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

class digi.xbee.devices.**ZigBeeDevice**(*port=None*, *baud_rate=None*, *data_bits=<sphinx.ext.autodoc.importer._MockObject object>*, *stop_bits=<sphinx.ext.autodoc.importer._MockObject object>*, *parity=<sphinx.ext.autodoc.importer._MockObject object>*, *flow_control=<FlowControl.NONE: None>*, *_sync_ops_timeout=4*, *comm_iface=None*)

Bases: `digi.xbee.devices.XBeeDevice`

This class represents a local Zigbee XBee.

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

Parameters

- **port** (*String*) – Serial port identifier. Depends on operating system. e.g. '/dev/ttyUSB0' on 'GNU/Linux' or 'COM3' on Windows.
- **baud_rate** (*Integer*) – Serial port baud rate.
- **(Integer, default** (*flow_control*) – `serial.EIGHTBITS`): Port bitsize.
- **(Integer, default** – `serial.STOPBITS_ONE`): Port stop bits.
- **(Character, default** (*parity*) – `serial.PARITY_NONE`): Port parity.
- **(Integer, default** – `FlowControl.NONE`): Port flow control.

_sync_ops_timeout (Integer, default: 3): Read timeout (in seconds). comm_iface (`XBeeCommunicationInterface`): Communication interface.

Raises All exceptions raised by `XBeeDevice.__init__()` constructor.

See also:

`XBeeDevice`

`XBeeDevice.__init__()`

open(*force_settings=False*)

Override.

See also:

XBeeDevice.open()

get_protocol()
Override.

See also:

XBeeDevice.get_protocol()

get_ai_status()
Returns the current association status of this XBee. It indicates occurrences of errors during the modem initialization and connection.

Returns

The XBee association indication status.

Return type *AssociationIndicationStatus*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

force_disassociate()
Forces this XBee to immediately disassociate from the network and re-attempt to associate.

Only valid for End Devices.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

get_many_to_one_broadcasting_time()
Returns the time between aggregation route broadcast in tenths of a second.

Returns

The number of tenths of a second between aggregation route broadcasts. -1 if it is disabled.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.

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

set_many_to_one_broadcasting_time(*tenths_second*)
Configures the time between aggregation route broadcast in tenths of a second.

Parameters **tenths_second** (*Integer*) – The number of tenths of a second between aggregation route broadcasts. -1 to disable. 0 to only send one broadcast.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `ValueError` – If *tenths_second* is *None* or is lower than -1, or bigger than 254.

send_data_64_16(*x64addr*, *x16addr*, *data*, *transmit_options=0*)
Blocking method. This method sends data to the remote XBee with the given 64-bit/16-bit address.

This method waits for the packet response. The default timeout is `XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS`.

Parameters

- **x64addr** (*XBee64BitAddress*) – 64-bit address of the destination XBee.
- **x16addr** (*XBee16BitAddress*) – 16-bit address of the destination XBee, *XBee16BitAddress.UNKNOWN_ADDRESS* if unknown.
- **data** (*String or Bytearray*) – Raw data to send.
- **transmit_options** (*Integer, optional*) – Transmit options, bitfield of *TransmitOptions*. Default to *TransmitOptions.NONE.value*.

Returns The response.

Return type *XBeePacket*

Raises

- `ValueError` – If *x64addr*, *x16addr* or *data* is *None*.
- `TimeoutException` – If response is not received before the read timeout expires.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TransmitException` – If the status of the response received is not OK.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

XBee64BitAddress
XBee16BitAddress
XBeePacket

send_data_async_64_16(*x64addr*, *x16addr*, *data*, *transmit_options=0*)

Non-blocking method. This method sends data to a remote XBee with the given 64-bit/16-bit address.

This method does not wait for a response.

Parameters

- **x64addr** (*XBee64BitAddress*) – 64-bit address of the destination XBee.
- **x16addr** (*XBee16BitAddress*) – 16-bit address of the destination XBee, *XBee16BitAddress.UNKNOWN_ADDRESS* if unknown.
- **data** (*String or Bytearray*) – Raw data to send.
- **transmit_options** (*Integer, optional*) – Transmit options, bitfield of *TransmitOptions*. Default to *TransmitOptions.NONE.value*.

Raises

- ValueError – If *x64addr*, *x16addr* or *data* is *None*.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- XBeeException – If the XBee's communication interface is closed.

See also:

XBee64BitAddress
XBee16BitAddress
XBeePacket

read_expl_data(*timeout=None*)

Reads new explicit data received by this XBee.

If *timeout* is specified, this method blocks until new data is received or the timeout expires, throwing a *TimeoutException* in this case.

Parameters timeout (*Integer, optional*) – Read timeout in seconds. If *None*, this method is non-blocking and returns *None* if there is no explicit data available.

Returns

Read message or *None* if this XBee did not receive new explicit data.

Return type *ExplicitXBeeMessage*

Raises

- ValueError – If a timeout is specified and is less than 0.
- TimeoutException – If a timeout is specified and no explicit data was received during that time.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- XBeeException – If the XBee's communication interface is closed.

See also:

ExplicitXBeeMessage

read_expl_data_from(*remote_xbee*, *timeout=None*)

Reads new explicit data received from the given remote XBee.

If *timeout* is specified, this method blocks until new data is received or the timeout expires, throwing a *TimeoutException* in this case.

Parameters

- **remote_xbee** (*RemoteXBeeDevice*) – Remote XBee that sent the explicit data.
- **timeout** (*Integer, optional*) – Read timeout in seconds. If *None*, this method is non-blocking and returns *None* if there is no data available.

Returns

Read message sent by *remote_xbee* or *None* if this XBee did not receive new data from that node.

Return type *ExplicitXBeeMessage*

Raises

- ValueError – If a timeout is specified and is less than 0.
- TimeoutException – If a timeout is specified and no explicit data was received during that time.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- XBeeException – If the XBee's communication interface is closed.

See also:

ExplicitXBeeMessage
RemoteXBeeDevice

send_expl_data(*remote_xbee*, *data*, *src_endpoint*, *dest_endpoint*, *cluster_id*, *profile_id*, *transmit_options=0*)

Blocking method. Sends the provided explicit data to the given XBee, source and destination end points, cluster and profile ids.

This method blocks until a success or error response arrives or the configured receive timeout expires. The default timeout is `XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS`.

Parameters

- **remote_xbee** (*RemoteXBeeDevice*) – Remote XBee to send data to.
- **data** (*String or Bytearray*) – 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 (between 0x0 and 0xFFFF)
- **profile_id** (*Integer*) – Profile ID of the transmission (between 0x0 and 0xFFFF)
- **transmit_options** (*Integer, optional*) – Transmit options, bitfield of *TransmitOptions*. Default to *TransmitOptions.NONE.value*.

Returns Response packet obtained after sending data.

Return type *XBeePacket*

Raises

- TimeoutException – If response is not received before the read timeout expires.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- TransmitException – If the status of the response received is not OK.
- XBeeException – If the XBee's communication interface is closed.
- ValueError – if *cluster_id* or *profile_id* is less than 0x0 or greater than 0xFFFF.

See also:

RemoteXBeeDevice
XBeePacket

send_expl_data_broadcast (*data*, *src_endpoint*, *dest_endpoint*, *cluster_id*, *profile_id*, *transmit_options=0*)

Sends the provided explicit data to all the XBee nodes of the network (broadcast) using provided source and destination end points, cluster and profile ids.

This method blocks until 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 method *AbstractXBeeDevice.get_sync_ops_timeout()*.

Parameters

- **data** (*String or Bytearray*) – 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 (between 0x0 and 0xFFFF)
- **profile_id** (*Integer*) – Profile ID of the transmission (between 0x0 and 0xFFFF)
- **transmit_options** (*Integer, optional*) – Transmit options, bitfield of *TransmitOptions*. Default to *TransmitOptions.NONE.value*.

Raises

- TimeoutException – If response is not received before the read timeout expires.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- TransmitException – If the status of the response received is not OK.
- XBeeException – If the XBee's communication interface is closed.
- ValueError – if *cluster_id* or *profile_id* is less than 0x0 or greater than 0xFFFF.

See also:

XBeeDevice._send_expl_data()

send_expl_data_async(*remote_xbee*, *data*, *src_endpoint*, *dest_endpoint*, *cluster_id*, *profile_id*, *transmit_options=0*)

Non-blocking method. Sends the provided explicit data to the given XBee, source and destination end points, cluster and profile ids.

Parameters

- **remote_xbee** (*RemoteXBeeDevice*) – Remote XBee to send data to.
- **data** (*String or Bytearray*) – 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 (between 0x0 and 0xFFFF)
- **profile_id** (*Integer*) – Profile ID of the transmission (between 0x0 and 0xFFFF)
- **transmit_options** (*Integer, optional*) – Transmit options, bitfield of *TransmitOptions*. Default to *TransmitOptions.NONE.value*.

Raises

- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- XBeeException – If the XBee's communication interface is closed.
- ValueError – if *cluster_id* or *profile_id* is less than 0x0 or greater than 0xFFFF.

See also:

RemoteXBeeDevice

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*) – 16-bit address of the multicast group.
- **data** (*Bytearray*) – 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 (between 0x0 and 0xFFFF)
- **profile_id** (*Integer*) – Profile ID of the transmission (between 0x0 and 0xFFFF)

Returns the response packet.

Return type *XBeePacket*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

`XBee16BitAddress`
`XBeePacket`

send_multicast_data_async(*group_id*, *data*, *src_endpoint*, *dest_endpoint*, *cluster_id*, *profile_id*)

Non-blocking method. This method sends multicast data to the provided group ID.

This method does not wait for a response.

Parameters

- **group_id** (*XBee16BitAddress*) – 16-bit address of the multicast group.
- **data** (*Bytearray*) – 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 (between 0x0 and 0xFFFF)
- **profile_id** (*Integer*) – Profile ID of the transmission (between 0x0 and 0xFFFF)

Raises

- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

`XBee16BitAddress`

register_joining_device(*registrant_address*, *options*, *key*)

Securely registers a joining device to a trust center. Registration is the process by which a node is authorized to join the network using a preconfigured link key or installation code that is conveyed to the trust center out-of-band (using a physical interface and not over-the-air).

This method is synchronous, it sends the register joining device request and waits for the answer of the operation. Then, returns the corresponding status.

Parameters

- **registrant_address** (XBee64BitAddress) – 64-bit address of the device to register.
- **options** (RegisterKeyOptions) – Register options indicating the key source.
- **key** (*Bytearray*) – Key of the device to register.

Returns

Register device operation status or *None* if the answer is not a *RegisterDeviceStatusPacket*.

Return type *ZigbeeRegisterStatus*

Raises

- TimeoutException – If the answer is not received in the configured timeout.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- XBeeException – If the XBee's communication interface is closed.
- ValueError – If *registrant_address* or *options* is *None*.

See also:

```
RegisterKeyOptions
XBee64BitAddress
ZigbeeRegisterStatus
```

register_joining_device_async (*registrant_address*, *options*, *key*)

Securely registers a joining device to a trust center. Registration is the process by which a node is authorized to join the network using a preconfigured link key or installation code that is conveyed to the trust center out-of-band (using a physical interface and not over-the-air).

This method is asynchronous, which means that it does not wait for an answer after sending the request.

Parameters

- **registrant_address** (XBee64BitAddress) – 64-bit address of the device to register.
- **options** (RegisterKeyOptions) – Register options indicating the key source.
- **key** (*Bytearray*) – Key of the device to register.

Raises

- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- XBeeException – If the XBee's communication interface is closed.
- ValueError – if *registrant_address* or *options* is *None*.

See also:

```
RegisterKeyOptions
XBee64BitAddress
```

unregister_joining_device(*unregistrant_address*)
Unregisters a joining device from a trust center.

This method is synchronous, it sends the unregister joining device request and waits for the answer of the operation. Then, returns the corresponding status.

Parameters **unregistrant_address** (XBee64BitAddress) – 64-bit address of the device to unregister.

Returns

Unregister device operation status or *None* if the answer is not a *RegisterDeviceStatusPacket*.

Return type *ZigbeeRegisterStatus*

Raises

- TimeoutException – If the answer is not received in the configured timeout.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- XBeeException – If the XBee's communication interface is closed.
- ValueError – If *registrant_address* is *None*.

See also:

XBee64BitAddress
ZigbeeRegisterStatus

unregister_joining_device_async(*unregistrant_address*)
Unregisters a joining device from a trust center.

This method is asynchronous, which means that it will not wait for an answer after sending the unregister request.

Parameters **unregistrant_address** (XBee64BitAddress) – 64-bit address of the device to unregister.

Raises

- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- XBeeException – If the XBee's communication interface is closed.
- ValueError – If *registrant_address* is *None*.

See also:

XBee64BitAddress

get_routes(*route_cb=None*, *finished_cb=None*, *timeout=None*)
Returns the routes of this XBee. If *route_cb* is not defined, the process blocks until the complete routing table is read.

Parameters

- **route_cb** (*Function, optional, default=`None`*) – Method called when a new route is received. Receives two arguments:
  - The XBee that owns this new route.
  - The new route.
- **finished_cb** (*Function, optional, default=`None`*) – Method to execute when the process finishes. Receives three arguments:
  - The XBee that executed the ZDO command.
  - A list with the discovered routes.
  - An error message if something went wrong.
- **timeout** (*Float, optional, default=`RouteTableReader.DEFAULT_TIMEOUT`*) – The ZDO command timeout in seconds.

Returns

List of `Route` when *route_cb* is not defined, *None* otherwise (in this case routes are received in the callback).

Return type List

Raises

- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `OperationNotSupportedException` – If XBee is not Zigbee or Smart Energy.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

`com.digi.models.zdo.Route`

get_neighbors(*neighbor_cb=None*, *finished_cb=None*, *timeout=None*)

Returns the neighbors of this XBee. If *neighbor_cb* is not defined, the process blocks until the complete neighbor table is read.

Parameters

- **neighbor_cb** (*Function, optional, default=`None`*) – Method called when a new neighbor is received. Receives two arguments:
  - The XBee that owns this new neighbor.
  - The new neighbor.
- **finished_cb** (*Function, optional, default=`None`*) – Method to execute when the process finishes. Receives three arguments:
  - The XBee that executed the ZDO command.
  - A list with the discovered neighbors.
  - An error message if something went wrong.
- **timeout** (*Float, optional, default=`NeighborTableReader.DEFAULT_TIMEOUT`*) – The ZDO command timeout in seconds.

Returns

List of `Neighbor` when *neighbor_cb* is not defined, *None* otherwise (in this case neighbors are received in the callback).

Return type List

Raises `OperationNotSupportedException` – If XBee is not Zigbee or Smart Energy.

See also:

`com.digi.models.zdo.Neighbor`

`create_source_route`(*dest_node*, *hops*)

Creates a source route for the provided destination node. A source route specifies the complete route a packet traverses to get from source to destination.

For best results, use source routing with many-to-one routing.

Parameters

- **`dest_node`** (*`RemoteXBeeDevice`*) – The destination node.
- **`hops`** (*`List`*) – List of intermediate nodes (*`RemoteXBeeDevice`*) ordered from closest to source to closest to destination node (source and destination excluded).

Raises

- `ValueError` – If *dest_node* is *None*, or if it is a local node, or if its protocol is not Zigbee based, or if its 64-bit address or 16-bit address is *None*, unknown, or invalid.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the packet listener is not running or the XBee's communication interface is closed.

`add_bluetooth_data_received_callback`(*callback*)

Adds a callback for the event *`BluetoothDataReceived`*.

Parameters **`callback`** (*`Function`*) – The callback. Receives one argument.

- The Bluetooth data as a Bytearray.

`add_data_received_callback`(*callback*)

Adds a callback for the event *`DataReceived`*.

Parameters **`callback`** (*`Function`*) – The callback. Receives one argument.

- The data received as an *`XBeeMessage`*.

`add_expl_data_received_callback`(*callback*)

Adds a callback for the event *`ExplicitDataReceived`*.

Parameters **`callback`** (*`Function`*) – The callback. Receives one argument.

- The explicit data received as a *`ExplicitXBeeMessage`*.

`add_fs_frame_received_callback`(*callback*)

Adds a callback for the event *`FileSystemFrameReceived`*.

Parameters **`callback`** (*`Function`*) – The callback. Receives four arguments.

- Source (*`AbstractXBeeDevice`*): The node that sent the file system frame.

- Frame id (Integer): The received frame id.
- Command (*FSCmd*): The file system command.
- Receive options (Integer): Bitfield indicating receive options.

See also:

AbstractXBeeDevice
FSCmd
ReceiveOptions

add_io_sample_received_callback(*callback*)
Adds a callback for the event *IOSampleReceived*.

Parameters **callback** (*Function*) – The callback. Receives three arguments.

- The received IO sample as an *IOSample*.
- The remote XBee which sent the packet as a *RemoteXBeeDevice*.
- The time in which the packet was received as an Integer.

add_micropython_data_received_callback(*callback*)
Adds a callback for the event *MicroPythonDataReceived*.

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

- The MicroPython data as a Bytearray.

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

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

- The modem status as a *ModemStatus*.

add_packet_received_callback(*callback*)
Adds a callback for the event *PacketReceived*.

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

- The received packet as a *XBeeAPIPacket*.

add_route_received_callback(*callback*)
Adds a callback for the event *RouteReceived*. This works for Zigbee and Digimesh devices.

Parameters **callback** (*Function*) – The callback. Receives three arguments.

- source (*XBeeDevice*): The source node.
- destination (*RemoteXBeeDevice*): The destination node.
- **hops (List): List of intermediate hops from closest to source** to closest to destination (*RemoteXBeeDevice*).

See also:

XBeeDevice.del_route_received_callback()

add_socket_data_received_callback(*callback*)

Adds a callback for the event *SocketDataReceived*.

Parameters **callback** (*Function*) – The callback. Receives two arguments.

- The socket ID as an Integer.
- The data received as Bytearray.

add_socket_data_received_from_callback(*callback*)

Adds a callback for the event *SocketDataReceivedFrom*.

Parameters **callback** (*Function*) – The callback. Receives three arguments.

- The socket ID as an Integer.
- **Source address pair (host, port) where host is a string** representing an IPv4 address like '100.50.200.5', and port is an integer.
- The data received as Bytearray.

add_socket_state_received_callback(*callback*)

Adds a callback for the event *SocketStateReceived*.

Parameters **callback** (*Function*) – The callback. Receives two arguments.

- The socket ID as an Integer.
- The state received as a *SocketState*.

add_user_data_relay_received_callback(*callback*)

Adds a callback for the event *RelayDataReceived*.

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

- The relay data as a *UserDataRelayMessage*.

apply_changes()

Applies changes via 'AC' command.

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

apply_profile(*profile_path*, *timeout=None*, *progress_callback=None*)

Applies the given XBee profile to the XBee.

Parameters

- **profile_path** (*String*) – Path of the XBee profile file to apply.
- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the apply profile (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to receive progress information. Receives two arguments:
  - The current apply profile task as a String
  - The current apply profile task percentage as an Integer

Raises

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `UpdateProfileException` – If there is any error applying the XBee profile.

close()

Closes the communication with the XBee.

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

comm_iface

Returns the hardware interface associated to the XBee.

Returns Hardware interface of the XBee.

Return type *XBeeCommunicationInterface*

See also:

XBeeCommunicationInterface

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 XBee object created.

Return type *XBeeDevice*

Raises `SerialException` – If the port to open does not exist or is already opened.

See also:

XBeeDevice

del_bluetooth_data_received_callback(*callback*)

Deletes a callback for the callback list of *BluetoothDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_data_received_callback(*callback*)
Deletes a callback for the callback list of *DataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_expl_data_received_callback(*callback*)
Deletes a callback for the callback list of *ExplicitDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_fs_frame_received_callback(*callback*)
Deletes a callback for the callback list of *FileSystemFrameReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_io_sample_received_callback(*callback*)
Deletes a callback for the callback list of *IOSampleReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_micropython_data_received_callback(*callback*)
Deletes a callback for the callback list of *MicroPythonDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_modem_status_received_callback(*callback*)
Deletes a callback for the callback list of *ModemStatusReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_packet_received_callback(*callback*)
Deletes a callback for the callback list of *PacketReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_route_received_callback(*callback*)
Deletes a callback for the callback list of *RouteReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

See also:

XBeeDevice.add_route_received_callback()

del_socket_data_received_callback(*callback*)
Deletes a callback for the callback list of *SocketDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_socket_data_received_from_callback(*callback*)
Deletes a callback for the callback list of *SocketDataReceivedFrom* event.

Parameters **callback** (*Function*) – The callback to delete.

del_socket_state_received_callback(*callback*)
Deletes a callback for the callback list of *SocketStateReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_user_data_relay_received_callback(*callback*)
Deletes a callback for the callback list of *RelayDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

determine_protocol(*hardware_version*, *firmware_version*)

Determines the XBee protocol based on the given hardware and firmware versions.

Parameters

- **hardware_version** (*Integer*) – Hardware version to get its protocol.
- **firmware_version** (*Bytearray*) – Firmware version to get its protocol.

Returns

XBee protocol corresponding to the given hardware and firmware versions.

Return type *XBeeProtocol*

disable_bluetooth()

Disables the Bluetooth interface of this XBee.

Note that your device must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

enable_apply_changes(*value*)

Sets apply changes flag.

Parameters **value** (*Boolean*) – *True* to enable apply changes flag, *False* to disable it.

enable_bluetooth()

Enables the Bluetooth interface of this XBee.

To work with this interface, you must also configure the Bluetooth password if not done previously. Use method *AbstractXBeeDevice.update_bluetooth_password()*.

Note that your XBee must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

execute_command(*parameter*, *value=None*, *apply=None*)

Executes the provided command.

Parameters

- **(String or** (*parameter*) – class: *.ATStringCommand*): AT command to execute.
- **value** (*bytearray, optional, default=`None`*) – Command value (if any).

- **apply** (*Boolean, optional, default=`None`*) – *True* to apply changes in XBee configuration, *False* not to apply them, *None* to use *is_apply_changes_enabled()* returned value.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

AbstractXBeeDevice.get_parameter()
AbstractXBeeDevice.set_parameter()
AbstractXBeeDevice.apply_changes()
AbstractXBeeDevice.write_changes()
AbstractXBeeDevice.is_apply_changes_enabled()
AbstractXBeeDevice.enable_apply_changes()

flush_queues()
Flushes the packets queue.

get_16bit_addr()
Returns the 16-bit address of the XBee.

Returns 16-bit address of the XBee.

Return type *XBee16BitAddress*

See also:

XBee16BitAddress

get_64bit_addr()
Returns the 64-bit address of the XBee.

Returns 64-bit address of the XBee.

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*) – IO line to get its ADC value.

Returns Analog value corresponding to the provided IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If response does not contain the value for the given IO line.

See also:

IOLine
set_io_configuration()

get_api_output_mode()

Deprecated since version 1.3: Use *get_api_output_mode_value()*

Returns the API output mode of the XBee.

The API output mode determines the format of the data through the serial interface of the XBee.

Returns API output mode of the XBee.

Return type *APIOutputMode*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

APIOutputMode

get_api_output_mode_value()

Returns the API output mode of the XBee.

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

Returns the parameter value.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

`digi.xbee.models.mode.APIOutputModeBit`

`get_bluetooth_mac_addr()`

Reads and returns the EUI-48 Bluetooth MAC address of this XBee following the format *00112233AABB*.

Note that your device must include Bluetooth Low Energy support.

Returns The Bluetooth MAC address.

Return type String

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

`get_current_frame_id()`

Returns the last used frame ID.

Returns Last used frame ID.

Return type Integer

`get_dest_address()`

Returns the 64-bit address of the XBee that is data destination.

Returns 64-bit address of destination XBee.

Return type *`XBee64BitAddress`*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

XBee64BitAddress
set_dest_address()

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 response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.
- OperationNotSupportedException – If response does not contain the value for the given IO line.

See also:

IOLine
IOValue
set_io_configuration()

get_file_manager()
Returns the file system manager for the XBee.

Returns The file system manager.

Return type *FileSystemManager*

Raises FileSystemNotSupportedException – If the XBee does not support filesystem.

get_firmware_version()
Returns the firmware version of the XBee.

Returns Firmware version of the XBee.

Return type Bytearray

get_hardware_version()
Returns the hardware version of the XBee.

Returns Hardware version of the XBee.

Return type *HardwareVersion*

See also:

HardwareVersion

get_io_configuration(*io_line*)
Returns the configuration of the provided IO line.

Parameters **io_line** (*IOLine*) – IO line to get its configuration.

Returns IO mode of the IO line provided.

Return type *IOMode*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOLine
IOMode
set_io_configuration()

get_io_sampling_rate()
Returns the IO sampling rate of the XBee.

Returns IO sampling rate of XBee.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

set_io_sampling_rate()

get_network()
Returns the network of this XBee.

Returns The XBee network.

Return type *XBeeNetwork*

get_next_frame_id()
Returns the next frame ID of the XBee.

Returns The next frame ID of the XBee.

Return type Integer

get_node_id()
Returns the node identifier ('NI') value of the XBee.

Returns Node identifier ('NI') of the XBee.

Return type String

get_pan_id()
Returns the operating PAN ID of the XBee.

Returns Operating PAN ID of the XBee.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

set_pan_id()

get_parameter(*parameter*, *parameter_value=None*, *apply=None*)
Override.

See also:

AbstractXBeeDevice.get_parameter()

get_power_level()
Returns the power level of the XBee.

Returns Power level of the XBee.

Return type *PowerLevel*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

PowerLevel

set_power_level()

get_pwm_duty_cycle(*io_line*)

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

Parameters **io_line** (*IOLine*) – IO line to get its PWM duty cycle.

Returns PWM duty cycle of the given IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `ValueError` – If *io_line* has no PWM capability.

See also:

IOLine

get_role()

Gets the XBee role.

Returns the role of the XBee.

Return type *Role*

See also:

Role

get_route_to_node(*remote*, *timeout=10*, *force=True*)

Gets the route from this XBee to the given remote node.

For Zigbee:

- 'AR' parameter of the local node must be configured with a value different from 'FF'.
- Set *force* to *True* to force the Zigbee remote node to return its route independently of the local node configuration as high or low RAM concentrator ('DO' of the local value)

Parameters

- **remote** (*RemoteXBeeDevice*) – The remote node.
- **timeout** (*Float, optional, default=10*) – Maximum number of seconds to wait for the route.

- **force** (*Boolean*) – *True* to force asking for the route, *False* otherwise. Only for Zigbee.

Returns

Tuple containing route data:

- status (*TransmitStatus*): The transmit status.
- Tuple with route data (*None* if the route was not read in the provided timeout):
  - source (*RemoteXBeeDevice*): The source node of the route.
  - destination (*RemoteXBeeDevice*): The destination node of the route.
  - hops (List): List of intermediate nodes (*RemoteXBeeDevice*) ordered from closest to source to closest to destination node (source and destination not included).

Return type Tuple

get_sync_ops_timeout()
Returns the serial port read timeout.

Returns 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 are executed before user callbacks.

Returns *PacketReceived*

has_explicit_packets()
Returns if there are pending explicit packets to read. This does not include non-explicit packets.

Returns *True* if there are pending packets, *False* otherwise.

Return type Boolean

See also:

XBeeDevice.has_packets()

has_packets()
Returns if there are pending packets to read. This does not include explicit packets.

Returns *True* if there are pending packets, *False* otherwise.

Return type Boolean

See also:

XBeeDevice.has_explicit_packets()

is_apply_changes_enabled()
Returns whether apply changes flag is enabled.

Returns *True* if apply changes flag is enabled, *False* otherwise.

Return type Boolean

`is_device_info_complete``()`

Returns whether XBee node information is complete.

Returns *True* if node information is complete, *False* otherwise.

Return type Boolean

See also:

`AbstractXBeeDevice.read_device_info()`

`is_open``()`

Returns whether this XBee is open.

Returns Boolean. *True* if this XBee is open, *False* otherwise.

`is_remote``()`

Override method.

See also:

`AbstractXBeeDevice.is_remote()`

`log`

Returns the XBee logger.

Returns The XBee device logger.

Return type `Logger`

`operating_mode`

Returns the operating mode of this XBee.

Returns *`OperatingMode`*. This XBee operating mode.

`reachable`

Returns whether the XBee is reachable.

Returns *True* if the device is reachable, *False* otherwise.

Return type Boolean

`read_data``(`*timeout=None*`)`

Reads new data received by this XBee.

If *timeout* is specified, this method blocks until new data is received or the timeout expires, throwing a *`TimeoutException`* in this case.

Parameters **`timeout`** (*`Integer, optional`*) – Read timeout in seconds. If *None*, this method is non-blocking and returns *None* if no data is available.

Returns

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's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

`XBeeMessage`

`read_data_from`(*remote_xbee*, *timeout=None*)

Reads new data received from the given remote XBee.

If *timeout* is specified, this method blocks until new data is received or the timeout expires, throwing a *`TimeoutException`* in this case.

Parameters

- **`remote_xbee`** (*`RemoteXBeeDevice`*) – Remote XBee that sent the data.
- **`timeout`** (*`Integer, optional`*) – Read timeout in seconds. If *None*, this method is non-blocking and returns *None* if no data is available.

Returns

Read message sent by *remote_xbee* 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's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

`XBeeMessage`
`RemoteXBeeDevice`

`read_device_info`(*init=True*, *fire_event=True*)

Updates all instance parameters reading them from the XBee.

Parameters

- **`init`** (*`Boolean, optional, default=`True``*) – If *False* only not initialized parameters are read, all if *True*.

- **fire_event** (*Boolean, optional, default=`True`*) – *True* to throw and update event if any parameter changed, *False* otherwise.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

AbstractXBeeDevice.is_device_info_complete()

read_io_sample()

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

Returns IO sample read from the XBee.

Return type *IOSample*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOSample

reset()

Override method.

See also:

AbstractXBeeDevice.reset()

scan_counter

Returns the scan counter for this node.

Returns The scan counter for this node.

Return type Integer

send_bluetooth_data(*data*)

Sends the given data to the Bluetooth interface using a User Data Relay frame.

Parameters **data** (*Bytearray*) – Data to send.

Raises

- `InvalidOperatingModeException` – If the XBee'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*, *data*, *transmit_options=0*)

Blocking method. This method sends data to a remote XBee synchronously.

This method will wait for the packet response. The default timeout is `XBeeDevice._DEFAULT_TIMEOUT_SYNC_OPERATIONS`.

Parameters

- **remote_xbee** (*RemoteXBeeDevice*) – Remote XBee to send data to.
- **data** (*String or Bytearray*) – Raw data to send.
- **transmit_options** (*Integer, optional*) – Transmit options, bitfield of *TransmitOptions*. Default to *TransmitOptions.NONE.value*.

Returns The response.

Return type *XBeePacket*

Raises

- `ValueError` – If *remote_xbee* is *None*.
- `TimeoutException` – If response is not received before the read timeout expires.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TransmitException` – If the status of the response received is not OK.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

RemoteXBeeDevice

XBeePacket

send_data_async(*remote_xbee*, *data*, *transmit_options=0*)

Non-blocking method. This method sends data to a remote XBee.

This method does not wait for a response.

Parameters

- **remote_xbee** (*RemoteXBeeDevice*) – the remote XBee to send data to.
- **data** (*String or Bytearray*) – Raw data to send.

- **transmit_options** (*Integer, optional*) – Transmit options, bitfield of *TransmitOptions*. Default to *TransmitOptions.NONE.value*.

Raises

- ValueError – If *remote_xbee* is *None*.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- XBeeException – If the XBee's communication interface 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 until a success or error transmit status arrives or the configured receive timeout expires.

The received timeout is configured using method *AbstractXBeeDevice.set_sync_ops_timeout()* 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 response is not received before the read timeout expires.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- TransmitException – If the status of the response received is not OK.
- XBeeException – If the XBee's communication interface is closed.

send_micropython_data(*data*)

Sends the given data to the MicroPython interface using a User Data Relay frame.

Parameters data (*Bytearray*) – Data to send.

Raises

- InvalidOperatingModeException – If the XBee'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*)

Sends the packet and waits for the response. The packet to send is escaped depending on the current operating mode.

This method can be synchronous or asynchronous.

If synchronous, this method discards all response packets until it finds the one that has the appropriate frame ID, that is, the sent packet's frame ID.

If asynchronous, this method does not wait for any response and returns *None*.

Parameters

- **packet** (*XBeePacket*) – The packet to send.
- **sync** (*Boolean*) – *True* to wait for the response of the sent packet and return it, *False* otherwise.

Returns

Response packet if *sync* is *True*, *None* otherwise.

Return type *XBeePacket*

Raises

- `TimeoutException` – If *sync* is *True* and the response packet for the sent one cannot be read.
- `InvalidOperatingModeException` – If the XBee operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the packet listener is not running or the XBee's communication interface is closed.

See also:

XBeePacket

send_packet_sync_and_get_response (*packet_to_send*, *timeout=None*)

Sends the packet and waits for its corresponding response.

Parameters

- **packet_to_send** (*XBeePacket*) – The packet to transmit.
- **timeout** (*Integer, optional, default=`None`*) – Number of seconds to wait. -1 to wait indefinitely.

Returns Received response packet.

Return type *XBeePacket*

Raises

- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TimeoutException` – If response is not received in the configured timeout.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

XBeePacket

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'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, if any.

Returns

Serial port of the XBee. *None* if the local XBee does not use serial communication.

Return type *XBeeSerialPort*

See also:

XBeeSerialPort

set_16bit_addr(*value*)

Sets the 16-bit address of the XBee.

Parameters **value** (*XBee16BitAddress*) – New 16-bit address of the XBee.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If the protocol is not 802.15.4.

set_api_output_mode(*api_output_mode*)
Deprecated since version 1.3: Use `set_api_output_mode_value()`

Sets the API output mode of the XBee.

Parameters **api_output_mode** (`APIOutputMode`) – New API output mode.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

`APIOutputMode`

set_api_output_mode_value(*api_output_mode*)
Sets the API output mode of the XBee.

Parameters **api_output_mode** (`Integer`) – New API output mode options. Calculate this value using the method `APIOutputModeBit.calculate_api_output_mode_value()` with a set of `APIOutputModeBit`.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

`APIOutputModeBit`

set_dest_address(*addr*)
Sets the 64-bit address of the XBee that is data destination.

Parameters **addr** (`XBee64BitAddress` or `RemoteXBeeDevice`) – Address itself or remote XBee to be data destination.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `ValueError` – If *addr* is *None*.

See also:

`XBee64BitAddress`
`get_dest_address()`

`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 response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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`*) – Digital IO line to sets its value.
- **`io_value`** (*`IOValue`*) – IO value to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – IO line to configure.
- **io_mode** (*IOMode*) – IO mode to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOLine
IOMode
get_io_configuration()

set_io_sampling_rate(*rate*)
Sets the IO sampling rate of the XBee in seconds. A sample rate of 0 means the IO sampling feature is disabled.

Parameters rate (*Integer*) – New IO sampling rate of the XBee in seconds.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

get_io_sampling_rate()

set_node_id(*node_id*)
Sets the node identifier ('NI') value of the XBee.

Parameters node_id (*String*) – New node identifier ('NI') of the XBee.

Raises

- `ValueError` – If *node_id* is *None* or its length is greater than 20.
- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.

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

set_pan_id(*value*)

Sets the operating PAN ID of the XBee.

Parameters **value** (*Bytearray*) – New operating PAN ID of the XBee. Must have only 1 or 2 bytes.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

get_pan_id()

set_parameter(*parameter*, *value*, *apply=None*)

Override.

See: *AbstractXBeeDevice.set_parameter()*

set_power_level(*power_level*)

Sets the power level of the XBee.

Parameters **power_level** (*PowerLevel*) – New power level of the XBee.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

PowerLevel
get_power_level()

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*) – IO Line to be assigned.

- **cycle** (*Integer*) – Duty cycle in % to be assigned. Must be between 0 and 100.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – Read timeout in seconds.

update_bluetooth_password(*new_password*)
Changes the Bluetooth password of this XBee with the new one provided.

Note that your device must include Bluetooth Low Energy support.

Parameters **new_password** (*String*) – New Bluetooth password.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

update_device_data_from(*device*)
Updates the current node information with provided data. This is only for internal use.

Parameters **device** (*AbstractXBeeDevice*) – XBee to get the data from.

Returns *True* if the node data has been updated, *False* otherwise.

Return type Boolean

update_firmware(*xml_firmware_file*, *xbee_firmware_file=None*, *bootloader_firmware_file=None*, *timeout=None*, *progress_callback=None*)
Performs a firmware update operation of the XBee.

Parameters

- **xml_firmware_file** (*String*) – Path of the XML file that describes the firmware to upload.
- **xbee_firmware_file** (*String, optional, default=`None`*) – Location of the XBee binary firmware file.

- **bootloader_firmware_file** (*String, optional, default=`None`*) – Location of the bootloader binary firmware file.
- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the update process (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

Raises

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `OperationNotSupportedException` – If XBee does not support firmware update.
- `FirmwareUpdateException` – If there is any error during the firmware update.

write_changes()

Writes configurable parameter values to the non-volatile memory of the XBee so that parameter modifications persist through subsequent resets.

Parameters values remain in the device's memory until overwritten by subsequent use of this method.

If changes are made without writing them, the XBee 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_changes_enabled()* to get its status and *enable_apply_changes()* to enable/disable the option. Method *apply_changes()* can be used in order to manually apply the changes.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

class digi.xbee.devices.**IPDevice**(*port=None, baud_rate=None, data_bits=<sphinx.ext.autodoc.importer._MockObject object>, stop_bits=<sphinx.ext.autodoc.importer._MockObject object>, parity=<sphinx.ext.autodoc.importer._MockObject object>, flow_control=<FlowControl.NONE: None>, _sync_ops_timeout=4, comm_iface=None*)

Bases: *digi.xbee.devices.XBeeDevice*

This class provides common functionality for XBee IP devices.

Class constructor. Instantiates a new *IPDevice* with the provided parameters.

Parameters

- **port** (*String*) – Serial port identifier. Depends on operating system. e.g. '/dev/ttyUSB0' on 'GNU/Linux' or 'COM3' on Windows.

- **baud_rate** (*Integer*) – Serial port baud rate.
- **(Integer, default** (*_sync_ops_timeout*) – `serial.EIGHTBITS`): Port bitsize.
- **(Integer, default** – `serial.STOPBITS_ONE`): Port stop bits.
- **(Character, default** (*parity*) – `serial.PARITY_NONE`): Port parity.
- **(Integer, default** – `FlowControl.NONE`): Port flow control.
- **(Integer, default** – 3): Read timeout (in seconds).
- **comm_iface** (*XBeeCommunicationInterface*) – Communication interface.

Raises All exceptions raised by `XBeeDevice.__init__()` constructor.

See also:

XBeeDevice
`XBeeDevice.__init__()`

is_device_info_complete()

Override.

See also:

AbstractXBeeDevice.is_device_info_complete()

get_ip_addr()

Returns the IP address of this IP XBee.

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 Configured destination IP address.

Return type `ipaddress.IPv4Address`

Raises

- `TimeoutException` – If there is a timeout getting the destination IP address.
- `XBeeException` – If there is any other XBee related exception.

See also:

`ipaddress.IPv4Address`

add_ip_data_received_callback(*callback*)
Adds a callback for the event *IPDataReceived*.

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

- The data received as an *IPMessage*

del_ip_data_received_callback(*callback*)
Deletes a callback for the callback list of *IPDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

start_listening(*src_port*)
Starts listening for incoming IP transmissions in the provided port.

Parameters **src_port** (*Integer*) – Port to listen for incoming transmissions.

Raises

- `ValueError` – If *source_port* is less than 0 or greater than 65535.
- `TimeoutException` – If there is a timeout setting the source port.
- `XBeeException` – If there is any other XBee related exception.

stop_listening()
Stops listening for incoming IP transmissions.

Raises

- `TimeoutException` – If there is a timeout processing the operation.
- `XBeeException` – If there is any other XBee related exception.

send_ip_data(*ip_addr*, *dest_port*, *protocol*, *data*, *close_socket=False*)
Sends the provided IP data to the given IP address and port using the specified IP protocol. For TCP and TCP SSL protocols, you can also indicate if the socket should be closed when data is sent.

This method blocks until 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, default=`False`*) – *True* to close the socket just after the transmission. *False* to keep it open.

Raises

- `ValueError` – If *ip_addr* or *protocol* or *data* is *None* or *dest_port* is less than 0 or greater than 65535.
- `OperationNotSupportedException` – If the XBee 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, default=`False`*) – *True* to close the socket just after the transmission. *False* to keep it open.

Raises

- `ValueError` – If *ip_addr* or *protocol* or *data* is *None* or *dest_port* is less than 0 or greater than 65535.
- `OperationNotSupportedException` – If the XBee 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 until 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* or *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 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 raises an AttributeError.

set_dest_address(*addr*)
Deprecated.

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

get_pan_id()
Deprecated.

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

set_pan_id(*value*)
Deprecated.

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

add_data_received_callback(*callback*)
Deprecated.

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

del_data_received_callback(*callback*)
Deprecated.

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

add_expl_data_received_callback(*callback*)
Deprecated.

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

del_expl_data_received_callback(*callback*)
Deprecated.

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

read_data(*timeout=None*)
Deprecated.

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

read_data_from(*remote_xbee*, *timeout=None*)
Deprecated.

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

send_data_broadcast(*data*, *transmit_options=0*)
Deprecated.

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

send_data(*remote_xbee*, *data*, *transmit_options=0*)
Deprecated.

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

send_data_async(*remote_xbee*, *data*, *transmit_options=0*)
Deprecated.

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

add_bluetooth_data_received_callback(*callback*)
Adds a callback for the event *BluetoothDataReceived*.

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

- The Bluetooth data as a Bytearray.

add_fs_frame_received_callback(*callback*)
Adds a callback for the event *FileSystemFrameReceived*.

Parameters **`callback`** (*`Function`*) – The callback. Receives four arguments.

- Source (*`AbstractXBeeDevice`*): The node that sent the file system frame.
- Frame id (Integer): The received frame id.
- Command (*`FSCmd`*): The file system command.
- Receive options (Integer): Bitfield indicating receive options.

See also:

`AbstractXBeeDevice`
`FSCmd`
`ReceiveOptions`

`add_io_sample_received_callback`(*callback*)
Adds a callback for the event *`IOSampleReceived`*.

Parameters **`callback`** (*`Function`*) – The callback. Receives three arguments.

- The received IO sample as an *`IOSample`*.
- The remote XBee which sent the packet as a *`RemoteXBeeDevice`*.
- The time in which the packet was received as an Integer.

`add_micropython_data_received_callback`(*callback*)
Adds a callback for the event *`MicroPythonDataReceived`*.

Parameters **`callback`** (*`Function`*) – The callback. Receives one argument.

- The MicroPython data as a Bytearray.

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

Parameters **`callback`** (*`Function`*) – The callback. Receives one argument.

- The modem status as a *`ModemStatus`*.

`add_packet_received_callback`(*callback*)
Adds a callback for the event *`PacketReceived`*.

Parameters **`callback`** (*`Function`*) – The callback. Receives one argument.

- The received packet as a *`XBeeAPIPacket`*.

`add_route_received_callback`(*callback*)
Adds a callback for the event *`RouteReceived`*. This works for Zigbee and Digimesh devices.

Parameters **`callback`** (*`Function`*) – The callback. Receives three arguments.

- source (*`XBeeDevice`*): The source node.
- destination (*`RemoteXBeeDevice`*): The destination node.
- **hops (List): List of intermediate hops from closest to source** to closest to destination (*`RemoteXBeeDevice`*).

See also:

`XBeeDevice.del_route_received_callback()`

add_socket_data_received_callback(*callback*)
Adds a callback for the event *SocketDataReceived*.

Parameters **callback** (*Function*) – The callback. Receives two arguments.

- The socket ID as an Integer.
- The data received as Bytearray.

add_socket_data_received_from_callback(*callback*)
Adds a callback for the event *SocketDataReceivedFrom*.

Parameters **callback** (*Function*) – The callback. Receives three arguments.

- The socket ID as an Integer.
- **Source address pair (host, port) where host is a string** representing an IPv4 address like '100.50.200.5', and port is an integer.
- The data received as Bytearray.

add_socket_state_received_callback(*callback*)
Adds a callback for the event *SocketStateReceived*.

Parameters **callback** (*Function*) – The callback. Receives two arguments.

- The socket ID as an Integer.
- The state received as a *SocketState*.

add_user_data_relay_received_callback(*callback*)
Adds a callback for the event *RelayDataReceived*.

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

- The relay data as a *UserDataRelayMessage*.

apply_changes()
Applies changes via 'AC' command.

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

apply_profile(*profile_path*, *timeout=None*, *progress_callback=None*)
Applies the given XBee profile to the XBee.

Parameters

- **profile_path** (*String*) – Path of the XBee profile file to apply.
- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the apply profile (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to receive progress information. Receives two arguments:
  - The current apply profile task as a String
  - The current apply profile task percentage as an Integer

Raises

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `UpdateProfileException` – If there is any error applying the XBee profile.

close()

Closes the communication with the XBee.

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

comm_iface

Returns the hardware interface associated to the XBee.

Returns Hardware interface of the XBee.

Return type *XBeeCommunicationInterface*

See also:

XBeeCommunicationInterface

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 XBee object created.

Return type *XBeeDevice*

Raises `SerialException` – If the port to open does not exist or is already opened.

See also:

XBeeDevice

del_bluetooth_data_received_callback(*callback*)

Deletes a callback for the callback list of *BluetoothDataReceived* event.

Parameters callback (*Function*) – The callback to delete.

del_fs_frame_received_callback(*callback*)
Deletes a callback for the callback list of *FileSystemFrameReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_io_sample_received_callback(*callback*)
Deletes a callback for the callback list of *IOSampleReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_micropython_data_received_callback(*callback*)
Deletes a callback for the callback list of *MicroPythonDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_modem_status_received_callback(*callback*)
Deletes a callback for the callback list of *ModemStatusReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_packet_received_callback(*callback*)
Deletes a callback for the callback list of *PacketReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_route_received_callback(*callback*)
Deletes a callback for the callback list of *RouteReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

See also:

XBeeDevice.add_route_received_callback()

del_socket_data_received_callback(*callback*)
Deletes a callback for the callback list of *SocketDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_socket_data_received_from_callback(*callback*)
Deletes a callback for the callback list of *SocketDataReceivedFrom* event.

Parameters **callback** (*Function*) – The callback to delete.

del_socket_state_received_callback(*callback*)
Deletes a callback for the callback list of *SocketStateReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_user_data_relay_received_callback(*callback*)
Deletes a callback for the callback list of *RelayDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

determine_protocol(*hardware_version*, *firmware_version*)
Determines the XBee protocol based on the given hardware and firmware versions.

Parameters

- **hardware_version** (*Integer*) – Hardware version to get its protocol.
- **firmware_version** (*Bytearray*) – Firmware version to get its protocol.

Returns

XBee protocol corresponding to the given hardware and firmware versions.

Return type *XBeeProtocol*

disable_bluetooth()

Disables the Bluetooth interface of this XBee.

Note that your device must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

enable_apply_changes(*value*)

Sets apply changes flag.

Parameters **value** (*Boolean*) – *True* to enable apply changes flag, *False* to disable it.

enable_bluetooth()

Enables the Bluetooth interface of this XBee.

To work with this interface, you must also configure the Bluetooth password if not done previously. Use method *AbstractXBeeDevice.update_bluetooth_password()*.

Note that your XBee must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

execute_command(*parameter*, *value=None*, *apply=None*)

Executes the provided command.

Parameters

- **(String or** (*parameter*) – class: *.ATStringCommand*): AT command to execute.
- **value** (*bytearray, optional, default=`None`*) – Command value (if any).
- **apply** (*Boolean, optional, default=`None`*) – *True* to apply changes in XBee configuration, *False* not to apply them, *None* to use *is_apply_changes_enabled()* returned value.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

- `ATCommandException` – If response is not as expected.

See also:

`AbstractXBeeDevice.get_parameter()`
`AbstractXBeeDevice.set_parameter()`
`AbstractXBeeDevice.apply_changes()`
`AbstractXBeeDevice.write_changes()`
`AbstractXBeeDevice.is_apply_changes_enabled()`
`AbstractXBeeDevice.enable_apply_changes()`

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

`get_64bit_addr``()`
Returns the 64-bit address of the XBee.

Returns 64-bit address of the XBee.

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`*) – IO line to get its ADC value.

Returns Analog value corresponding to the provided IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If response does not contain the value for the given IO line.

See also:

`IOLine`
`set_io_configuration()`

get_api_output_mode()

Deprecated since version 1.3: Use *get_api_output_mode_value()*

Returns the API output mode of the XBee.

The API output mode determines the format of the data through the serial interface of the XBee.

Returns API output mode of the XBee.

Return type *APIOutputMode*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

APIOutputMode

get_api_output_mode_value()

Returns the API output mode of the XBee.

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

Returns the parameter value.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

digi.xbee.models.mode.APIOutputModeBit

get_bluetooth_mac_addr()

Reads and returns the EUI-48 Bluetooth MAC address of this XBee following the format *00112233AABB*.

Note that your device must include Bluetooth Low Energy support.

Returns The Bluetooth MAC address.

Return type String

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

`get_current_frame_id`()

Returns the last used frame ID.

Returns 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 response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If response does not contain the value for the given IO line.

See also:

`IOLine`
`IOValue`
`set_io_configuration()`

`get_file_manager`()

Returns the file system manager for the XBee.

Returns The file system manager.

Return type *`FileSystemManager`*

Raises `FileSystemNotSupportedException` – If the XBee does not support filesystem.

`get_firmware_version`()

Returns the firmware version of the XBee.

Returns Firmware version of the XBee.

Return type Bytearray

get_hardware_version()

Returns the hardware version of the XBee.

Returns Hardware version of the XBee.

Return type *HardwareVersion*

See also:

HardwareVersion

get_io_configuration(*io_line*)

Returns the configuration of the provided IO line.

Parameters **io_line** (*IOLine*) – IO line to get its configuration.

Returns IO mode of the IO line provided.

Return type *IOMode*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOLine
IOMode
set_io_configuration()

get_io_sampling_rate()

Returns the IO sampling rate of the XBee.

Returns IO sampling rate of XBee.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

set_io_sampling_rate()

get_next_frame_id()
Returns the next frame ID of the XBee.

Returns The next frame ID of the XBee.

Return type Integer

get_node_id()
Returns the node identifier ('NI') value of the XBee.

Returns Node identifier ('NI') of the XBee.

Return type String

get_parameter(*parameter*, *parameter_value=None*, *apply=None*)
Override.

See also:

AbstractXBeeDevice.get_parameter()

get_power_level()
Returns the power level of the XBee.

Returns Power level of the XBee.

Return type *PowerLevel*

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

See also:

PowerLevel
set_power_level()

get_protocol()
Returns the current protocol of the XBee.

Returns Current protocol of the XBee.

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*) – IO line to get its PWM duty cycle.

Returns PWM duty cycle of the given IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `ValueError` – If *io_line* has no PWM capability.

See also:

IOLine

get_role()
Gets the XBee role.

Returns the role of the XBee.

Return type *Role*

See also:

Role

get_route_to_node(*remote*, *timeout=10*, *force=True*)
Gets the route from this XBee to the given remote node.

For Zigbee:

- 'AR' parameter of the local node must be configured with a value different from 'FF'.
- Set *force* to *True* to force the Zigbee remote node to return its route independently of the local node configuration as high or low RAM concentrator ('DO' of the local value)

Parameters

- **remote** (*RemoteXBeeDevice*) – The remote node.
- **timeout** (*Float, optional, default=10*) – Maximum number of seconds to wait for the route.
- **force** (*Boolean*) – *True* to force asking for the route, *False* otherwise. Only for Zigbee.

Returns

Tuple containing route data:

- status (*TransmitStatus*): The transmit status.
- Tuple with route data (*None* if the route was not read in the provided timeout):
  - source (*RemoteXBeeDevice*): The source node of the route.
  - destination (*RemoteXBeeDevice*): The destination node of the route.
  - hops (List): List of intermediate nodes (*RemoteXBeeDevice*) ordered from closest to source to closest to destination node (source and destination not included).

Return type Tuple

get_sync_ops_timeout()
Returns the serial port read timeout.

Returns 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 are executed before user callbacks.

Returns *PacketReceived*

has_explicit_packets()
Returns if there are pending explicit packets to read. This does not include non-explicit packets.

Returns *True* if there are pending packets, *False* otherwise.

Return type Boolean

See also:

XBeeDevice.has_packets()

has_packets()
Returns if there are pending packets to read. This does not include explicit packets.

Returns *True* if there are pending packets, *False* otherwise.

Return type Boolean

See also:

XBeeDevice.has_explicit_packets()

is_apply_changes_enabled()
Returns whether apply changes flag is enabled.

Returns *True* if apply changes flag is enabled, *False* otherwise.

Return type Boolean

is_open()
Returns whether this XBee is open.

Returns Boolean. *True* if this XBee is open, *False* otherwise.

is_remote()

Override method.

See also:

AbstractXBeeDevice.is_remote()

log

Returns the XBee logger.

Returns The XBee device logger.

Return type `Logger`

open(*force_settings=False*)

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

Parameters **force_settings** (*Boolean, optional, default=`False`*) – *True* to open the device ensuring/forcing that the specified serial settings are applied even if the current configuration is different, *False* to open the device with the current configuration.

Raises

- `TimeoutException` – If there is any problem with the communication.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the XBee is already opened.

operating_mode

Returns the operating mode of this XBee.

Returns *OperatingMode*. This XBee operating mode.

reachable

Returns whether the XBee is reachable.

Returns *True* if the device is reachable, *False* otherwise.

Return type Boolean

read_device_info(*init=True*, *fire_event=True*)

Updates all instance parameters reading them from the XBee.

Parameters

- **init** (*Boolean, optional, default=`True`*) – If *False* only not initialized parameters are read, all if *True*.
- **fire_event** (*Boolean, optional, default=`True`*) – *True* to throw and update event if any parameter changed, *False* otherwise.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.

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

See also:

AbstractXBeeDevice.is_device_info_complete()

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

Returns IO sample read from the XBee.

Return type *IOSample*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOSample

reset()
Override method.

See also:

AbstractXBeeDevice.reset()

scan_counter
Returns the scan counter for this node.

Returns The scan counter for this node.

Return type Integer

send_bluetooth_data(*data*)
Sends the given data to the Bluetooth interface using a User Data Relay frame.

Parameters **data** (*Bytearray*) – Data to send.

Raises

- `InvalidOperatingModeException` – If the XBee'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'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*)
Sends the packet and waits for the response. The packet to send is escaped depending on the current operating mode.

This method can be synchronous or asynchronous.

If synchronous, this method discards all response packets until it finds the one that has the appropriate frame ID, that is, the sent packet's frame ID.

If asynchronous, this method does not wait for any response and returns *None*.

Parameters

- **packet** (*XBeePacket*) – The packet to send.
- **sync** (`Boolean`) – *True* to wait for the response of the sent packet and return it, *False* otherwise.

Returns

Response packet if *sync* is *True*, *None* otherwise.

Return type *XBeePacket*

Raises

- `TimeoutException` – If *sync* is *True* and the response packet for the sent one cannot be read.
- `InvalidOperatingModeException` – If the XBee operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the packet listener is not running or the XBee's communication interface is closed.

See also:

XBeePacket

send_packet_sync_and_get_response(*packet_to_send*, *timeout=None*)
Sends the packet and waits for its corresponding response.

Parameters

- **packet_to_send** (*XBeePacket*) – The packet to transmit.
- **timeout** (*Integer, optional, default=`None`*) – Number of seconds to wait. -1 to wait indefinitely.

Returns Received response packet.

Return type *XBeePacket*

Raises

- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TimeoutException` – If response is not received in the configured timeout.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

XBeePacket

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'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, if any.

Returns

Serial port of the XBee. *None* if the local XBee does not use serial communication.

Return type *XBeeSerialPort*

See also:

XBeeSerialPort

set_16bit_addr(*value*)

Sets the 16-bit address of the XBee.

Parameters **value** (*XBee16BitAddress*) – New 16-bit address of the XBee.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If the protocol is not 802.15.4.

set_api_output_mode(*api_output_mode*)

Deprecated since version 1.3: Use *set_api_output_mode_value()*

Sets the API output mode of the XBee.

Parameters **api_output_mode** (*APIOutputMode*) – New API output mode.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

APIOutputMode

set_api_output_mode_value(*api_output_mode*)

Sets the API output mode of the XBee.

Parameters **api_output_mode** (*Integer*) – New API output mode options. Calculate this value using the method `APIOutputModeBit.calculate_api_output_mode_value()` with a set of *APIOutputModeBit*.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.

- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

`APIOutputModeBit`

`set_dio_change_detection`(*io_lines_set*)
Sets the digital IO lines to be monitored and sampled whenever their status changes. A *None* set of lines disables this feature.

Parameters **`io_lines_set`** – Set of *`IOLine`*.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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`*) – Digital IO line to sets its value.
- **`io_value`** (*`IOValue`*) – IO value to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – IO line to configure.
- **io_mode** (*IOMode*) – IO mode to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOLine

IOMode

get_io_configuration()

set_io_sampling_rate(*rate*)

Sets the IO sampling rate of the XBee in seconds. A sample rate of 0 means the IO sampling feature is disabled.

Parameters **rate** (*Integer*) – New IO sampling rate of the XBee in seconds.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

get_io_sampling_rate()

set_node_id(*node_id*)

Sets the node identifier ('NI') value of the XBee.

Parameters **node_id** (*String*) – New node identifier ('NI') of the XBee.

Raises

- `ValueError` – If *node_id* is *None* or its length is greater than 20.
- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.

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

`set_parameter`(*parameter*, *value*, *apply=None*)
Override.

See: *`AbstractXBeeDevice.set_parameter()`*

`set_power_level`(*power_level*)
Sets the power level of the XBee.

Parameters **`power_level`** (*`PowerLevel`*) – New power level of the XBee.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`PowerLevel`
`get_power_level()`

`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`*) – IO Line to be assigned.
- **`cycle`** (*`Integer`*) – Duty cycle in % to be assigned. Must be between 0 and 100.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – Read timeout in seconds.

update_bluetooth_password(*new_password*)
Changes the Bluetooth password of this XBee with the new one provided.

Note that your device must include Bluetooth Low Energy support.

Parameters **new_password** (*String*) – New Bluetooth password.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

update_device_data_from(*device*)
Updates the current node information with provided data. This is only for internal use.

Parameters **device** (*AbstractXBeeDevice*) – XBee to get the data from.

Returns *True* if the node data has been updated, *False* otherwise.

Return type Boolean

update_firmware(*xml_firmware_file*, *xbee_firmware_file=None*, *bootloader_firmware_file=None*, *timeout=None*, *progress_callback=None*)
Performs a firmware update operation of the XBee.

Parameters

- **xml_firmware_file** (*String*) – Path of the XML file that describes the firmware to upload.
- **xbee_firmware_file** (*String, optional, default=`None`*) – Location of the XBee binary firmware file.
- **bootloader_firmware_file** (*String, optional, default=`None`*) – Location of the bootloader binary firmware file.
- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the update process (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

Raises

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `OperationNotSupportedException` – If XBee does not support firmware update.
- `FirmwareUpdateException` – If there is any error during the firmware update.

write_changes()

Writes configurable parameter values to the non-volatile memory of the XBee so that parameter modifications persist through subsequent resets.

Parameters values remain in the device's memory until overwritten by subsequent use of this method.

If changes are made without writing them, the XBee 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_changes_enabled()* to get its status and *enable_apply_changes()* to enable/disable the option. Method *apply_changes()* can be used in order to manually apply the changes.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

class digi.xbee.devices.**CellularDevice**(*port=None*, *baud_rate=None*, *data_bits=<sphinx.ext.autodoc.importer._MockObject object>*, *stop_bits=<sphinx.ext.autodoc.importer._MockObject object>*, *parity=<sphinx.ext.autodoc.importer._MockObject object>*, *flow_control=<FlowControl.NONE: None>*, *_sync_ops_timeout=4*, *comm_iface=None*)

Bases: *digi.xbee.devices.IPDevice*

This class represents a local Cellular device.

Class constructor. Instantiates a new *CellularDevice* with the provided parameters.

Parameters

- **port** (*String*) – Serial port identifier. Depends on operating system. e.g. '/dev/ttyUSB0' on 'GNU/Linux' or 'COM3' on Windows.
- **baud_rate** (*Integer*) – Serial port baud rate.
- **(Integer, default** (*_sync_ops_timeout*) – `serial.EIGHTBITS`): Port bitsize.
- **(Integer, default** – `serial.STOPBITS_ONE`): Port stop bits.
- **(Character, default** (*parity*) – `serial.PARITY_NONE`): Port parity.
- **(Integer, default** – `FlowControl.NONE`): Port flow control.
- **(Integer, default** – 3): Read timeout (in seconds).
- **comm_iface** (*XBeeCommunicationInterface*) – Communication interface.

Raises All exceptions raised by `XBeeDevice.__init__()` constructor.

See also:

XBeeDevice

`XBeeDevice.__init__()`

open(*force_settings=False*)
Override.

See also:

XBeeDevice.open()

get_protocol()
Override.

See also:

XBeeDevice.get_protocol()

is_device_info_complete()
Override.

See also:

AbstractXBeeDevice.is_device_info_complete()

is_connected()
Returns whether the device is connected to the Internet.

Returns *True* if 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.

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 until 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* or *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* or *data* is *None*.
- `OperationNotSupportedException` – If the device is remote.
- `XBeeException` – If there is any other XBee related exception.

get_sockets_list ()

Returns a list with the IDs of all active (open) sockets.

Returns

list with the IDs of all active (open) sockets, or empty list if there is not any active socket.

Return type List

Raises

- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TimeoutException` – If the response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.

get_socket_info(*socket_id*)
Returns the information of the socket with the given socket ID.

Parameters **socket_id** (*Integer*) – ID of the socket.

Returns

The socket information, or *None* if the socket with that ID does not exist.

Return type *SocketInfo*

Raises

- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TimeoutException` – If the response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

SocketInfo

get_64bit_addr()
Deprecated.

Cellular protocol does not have an associated 64-bit address.

add_io_sample_received_callback(*callback*)
Deprecated.

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

del_io_sample_received_callback(*callback*)
Deprecated.

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

set_dio_change_detection(*io_lines_set*)
Deprecated.

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

get_io_sampling_rate()
Deprecated.

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

set_io_sampling_rate(*rate*)
Deprecated.

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

get_node_id()
Deprecated.

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

set_node_id(*node_id*)
Deprecated.

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

get_power_level()
Deprecated.

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

set_power_level(*power_level*)
Deprecated.

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

add_bluetooth_data_received_callback(*callback*)
Adds a callback for the event *BluetoothDataReceived*.

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

- The Bluetooth data as a Bytearray.

add_data_received_callback(*callback*)
Deprecated.

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

add_expl_data_received_callback(*callback*)
Deprecated.

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

add_fs_frame_received_callback(*callback*)
Adds a callback for the event *FileSystemFrameReceived*.

Parameters **callback** (*Function*) – The callback. Receives four arguments.

- Source (*AbstractXBeeDevice*): The node that sent the file system frame.
- Frame id (Integer): The received frame id.
- Command (*FSCmd*): The file system command.
- Receive options (Integer): Bitfield indicating receive options.

See also:

AbstractXBeeDevice
FSCmd
ReceiveOptions

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*)
Adds a callback for the event *MicroPythonDataReceived*.

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

- The MicroPython data as a Bytearray.

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

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

- The modem status as a *ModemStatus*.

add_packet_received_callback(*callback*)
Adds a callback for the event *PacketReceived*.

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

- The received packet as a *XBeeAPIPacket*.

add_route_received_callback(*callback*)
Adds a callback for the event *RouteReceived*. This works for Zigbee and Digimesh devices.

Parameters **callback** (*Function*) – The callback. Receives three arguments.

- source (*XBeeDevice*): The source node.
- destination (*RemoteXBeeDevice*): The destination node.
- **hops (List): List of intermediate hops from closest to source** to closest to destination (*RemoteXBeeDevice*).

See also:

XBeeDevice.del_route_received_callback()

add_socket_data_received_callback(*callback*)
Adds a callback for the event *SocketDataReceived*.

Parameters **callback** (*Function*) – The callback. Receives two arguments.

- The socket ID as an Integer.
- The data received as Bytearray.

add_socket_data_received_from_callback(*callback*)
Adds a callback for the event *SocketDataReceivedFrom*.

Parameters **callback** (*Function*) – The callback. Receives three arguments.

- The socket ID as an Integer.
- **Source address pair (host, port) where host is a string** representing an IPv4 address like '100.50.200.5', and port is an integer.
- The data received as Bytearray.

add_socket_state_received_callback(*callback*)
Adds a callback for the event *SocketStateReceived*.

Parameters **callback** (*Function*) – The callback. Receives two arguments.

- The socket ID as an Integer.

- The state received as a *SocketState*.

add_user_data_relay_received_callback(*callback*)
Adds a callback for the event *RelayDataReceived*.

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

- The relay data as a *UserDataRelayMessage*.

apply_changes()
Applies changes via 'AC' command.

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

apply_profile(*profile_path*, *timeout=None*, *progress_callback=None*)
Applies the given XBee profile to the XBee.

Parameters

- **profile_path** (*String*) – Path of the XBee profile file to apply.
- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the apply profile (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to receive progress information. Receives two arguments:
  - The current apply profile task as a String
  - The current apply profile task percentage as an Integer

Raises

- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- UpdateProfileException – If there is any error applying the XBee profile.

close()
Closes the communication with the XBee.

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

comm_iface
Returns the hardware interface associated to the XBee.

Returns Hardware interface of the XBee.

Return type *XBeeCommunicationInterface*

See also:

XBeeCommunicationInterface

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 XBee object created.

Return type *XBeeDevice*

Raises `SerialException` – If the port to open does not exist or is already opened.

See also:

XBeeDevice

del_bluetooth_data_received_callback(*callback*)

Deletes a callback for the callback list of *BluetoothDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_data_received_callback(*callback*)

Deprecated.

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

del_expl_data_received_callback(*callback*)

Deprecated.

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

del_fs_frame_received_callback(*callback*)

Deletes a callback for the callback list of *FileSystemFrameReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_ip_data_received_callback(*callback*)

Deletes a callback for the callback list of *IPDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_micropython_data_received_callback(*callback*)

Deletes a callback for the callback list of *MicroPythonDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_modem_status_received_callback(*callback*)

Deletes a callback for the callback list of *ModemStatusReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_packet_received_callback(*callback*)
Deletes a callback for the callback list of *PacketReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_route_received_callback(*callback*)
Deletes a callback for the callback list of *RouteReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

See also:

XBeeDevice.add_route_received_callback()

del_socket_data_received_callback(*callback*)
Deletes a callback for the callback list of *SocketDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_socket_data_received_from_callback(*callback*)
Deletes a callback for the callback list of *SocketDataReceivedFrom* event.

Parameters **callback** (*Function*) – The callback to delete.

del_socket_state_received_callback(*callback*)
Deletes a callback for the callback list of *SocketStateReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_user_data_relay_received_callback(*callback*)
Deletes a callback for the callback list of *RelayDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

determine_protocol(*hardware_version*, *firmware_version*)
Determines the XBee protocol based on the given hardware and firmware versions.

Parameters

- **hardware_version** (*Integer*) – Hardware version to get its protocol.
- **firmware_version** (*Bytearray*) – Firmware version to get its protocol.

Returns

XBee protocol corresponding to the given hardware and firmware versions.

Return type *XBeeProtocol*

disable_bluetooth()
Disables the Bluetooth interface of this XBee.

Note that your device must include Bluetooth Low Energy support.

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

enable_apply_changes(*value*)
Sets apply changes flag.

Parameters **value** (*Boolean*) – *True* to enable apply changes flag, *False* to disable it.

enable_bluetooth()
Enables the Bluetooth interface of this XBee.

To work with this interface, you must also configure the Bluetooth password if not done previously. Use method *AbstractXBeeDevice.update_bluetooth_password()*.

Note that your XBee must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

execute_command(*parameter*, *value=None*, *apply=None*)
Executes the provided command.

Parameters

- **(String or** (*parameter*) – class: *.ATStringCommand*): AT command to execute.
- **value** (*bytearray, optional, default=`None`*) – Command value (if any).
- **apply** (*Boolean, optional, default=`None`*) – *True* to apply changes in XBee configuration, *False* not to apply them, *None* to use *is_apply_changes_enabled()* returned value.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

AbstractXBeeDevice.get_parameter()
AbstractXBeeDevice.set_parameter()
AbstractXBeeDevice.apply_changes()
AbstractXBeeDevice.write_changes()
AbstractXBeeDevice.is_apply_changes_enabled()
AbstractXBeeDevice.enable_apply_changes()

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*) – IO line to get its ADC value.

Returns Analog value corresponding to the provided IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If response does not contain the value for the given IO line.

See also:

IOLine
set_io_configuration()

get_api_output_mode()

Deprecated since version 1.3: Use *get_api_output_mode_value()*

Returns the API output mode of the XBee.

The API output mode determines the format of the data through the serial interface of the XBee.

Returns API output mode of the XBee.

Return type *APIOutputMode*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

APIOutputMode

get_api_output_mode_value()
Returns the API output mode of the XBee.

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

Returns the parameter value.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

digi.xbee.models.mode.APIOutputModeBit

get_bluetooth_mac_addr()
Reads and returns the EUI-48 Bluetooth MAC address of this XBee following the format *00112233AABB*.

Note that your device must include Bluetooth Low Energy support.

Returns The Bluetooth MAC address.

Return type String

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

get_current_frame_id()
Returns the last used frame ID.

Returns 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 raises an `AttributeError`.

get_dest_ip_addr()
Returns the destination IP address.

Returns Configured destination IP address.

Return type `ipaddress.IPv4Address`

Raises

- `TimeoutException` – If there is a timeout getting the destination IP address.
- `XBeeException` – If there is any other XBee related exception.

See also:

`ipaddress.IPv4Address`

`get_dio_value`(*io_line*)

Returns the digital value of the provided IO line.

The provided IO line must be previously configured as digital I/O. To do so, use *`AbstractXBeeDevice.set_io_configuration()`*.

Parameters **`io_line`** (*`IOLine`*) – the DIO line to gets its digital value.

Returns current value of the provided IO line.

Return type *`IOValue`*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If response does not contain the value for the given IO line.

See also:

`IOLine`
`IOValue`
`set_io_configuration()`

`get_file_manager`()

Returns the file system manager for the XBee.

Returns The file system manager.

Return type *`FileSystemManager`*

Raises `FileSystemNotSupportedException` – If the XBee does not support filesystem.

`get_firmware_version`()

Returns the firmware version of the XBee.

Returns Firmware version of the XBee.

Return type Bytearray

get_hardware_version()
Returns the hardware version of the XBee.

Returns Hardware version of the XBee.

Return type *HardwareVersion*

See also:

HardwareVersion

get_io_configuration(*io_line*)
Returns the configuration of the provided IO line.

Parameters **io_line** (*IOLine*) – IO line to get its configuration.

Returns IO mode of the IO line provided.

Return type *IOMode*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOLine
IOMode
set_io_configuration()

get_ip_addr()
Returns the IP address of this IP XBee.

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.

Returns The next frame ID of the XBee.

Return type Integer

get_pan_id()

Deprecated.

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

get_parameter(*parameter*, *parameter_value=None*, *apply=None*)

Override.

See also:

AbstractXBeeDevice.get_parameter()

get_pwm_duty_cycle(*io_line*)

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

Parameters **io_line** (*IOLine*) – IO line to get its PWM duty cycle.

Returns PWM duty cycle of the given IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `ValueError` – If *io_line* has no PWM capability.

See also:

IOLine

get_role()

Gets the XBee role.

Returns the role of the XBee.

Return type *Role*

See also:

Role

get_route_to_node (*remote*, *timeout=10*, *force=True*)
Gets the route from this XBee to the given remote node.

For Zigbee:

- 'AR' parameter of the local node must be configured with a value different from 'FF'.
- Set *force* to *True* to force the Zigbee remote node to return its route independently of the local node configuration as high or low RAM concentrator ('DO' of the local value)

Parameters

- **remote** (*RemoteXBeeDevice*) – The remote node.
- **timeout** (*Float, optional, default=10*) – Maximum number of seconds to wait for the route.
- **force** (*Boolean*) – *True* to force asking for the route, *False* otherwise. Only for Zigbee.

Returns

Tuple containing route data:

- status (*TransmitStatus*): The transmit status.
- Tuple with route data (*None* if the route was not read in the provided timeout):
  - source (*RemoteXBeeDevice*): The source node of the route.
  - destination (*RemoteXBeeDevice*): The destination node of the route.
  - hops (List): List of intermediate nodes (*RemoteXBeeDevice*) ordered from closest to source to closest to destination node (source and destination not included).

Return type Tuple

get_sync_ops_timeout ()
Returns the serial port read timeout.

Returns 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 are executed before user callbacks.

Returns *PacketReceived*

has_explicit_packets ()
Returns if there are pending explicit packets to read. This does not include non-explicit packets.

Returns *True* if there are pending packets, *False* otherwise.

Return type Boolean

See also:

XBeeDevice.has_packets()

has_packets()
Returns if there are pending packets to read. This does not include explicit packets.

Returns *True* if there are pending packets, *False* otherwise.

Return type Boolean

See also:

XBeeDevice.has_explicit_packets()

is_apply_changes_enabled()
Returns whether apply changes flag is enabled.

Returns *True* if apply changes flag is enabled, *False* otherwise.

Return type Boolean

is_open()
Returns whether this XBee is open.

Returns Boolean. *True* if this XBee is open, *False* otherwise.

is_remote()
Override method.

See also:

AbstractXBeeDevice.is_remote()

log
Returns the XBee logger.

Returns The XBee device logger.

Return type `Logger`

operating_mode
Returns the operating mode of this XBee.

Returns *OperatingMode*. This XBee operating mode.

reachable
Returns whether the XBee is reachable.

Returns *True* if the device is reachable, *False* otherwise.

Return type Boolean

read_data(*timeout=None*)
Deprecated.

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

read_data_from(*remote_xbee*, *timeout=None*)
Deprecated.

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

read_device_info(*init=True*, *fire_event=True*)

Updates all instance parameters reading them from the XBee.

Parameters

- **init** (*Boolean, optional, default=`True`*) – If *False* only not initialized parameters are read, all if *True*.
- **fire_event** (*Boolean, optional, default=`True`*) – *True* to throw and update event if any parameter changed, *False* otherwise.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

AbstractXBeeDevice.is_device_info_complete()

read_io_sample()

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

Returns IO sample read from the XBee.

Return type *IOSample*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOSample

read_ip_data(*timeout=3*)

Reads new IP data received by this XBee during the provided timeout.

This method blocks until new IP data is received or the provided timeout expires.

For non-blocking operations, register a callback and use the method *IPDevice.add_ip_data_received_callback()*.

Before reading IP data you need to start listening for incoming IP data at a specific port. Use the method *IPDevice.start_listening()* for that purpose. When finished, you can use the method *IPDevice.stop_listening()* to stop listening for incoming IP data.

Parameters **timeout** (*Integer, optional*) – The time to wait for new IP data in seconds.

Returns IP message, *None* if this device did not receive new data.

Return type *IPMessage*

Raises ValueError – If *timeout* is less than 0.

read_ip_data_from(*ip_addr*, *timeout=3*)

Reads new IP data received from the given IP address during the provided timeout.

This method blocks until new IP data from the provided IP address is received or the given timeout expires.

For non-blocking operations, register a callback and use the method *IPDevice.add_ip_data_received_callback()*.

Before reading IP data you need to start listening for incoming IP data at a specific port. Use the method *IPDevice.start_listening()* for that purpose. When finished, you can use the method *IPDevice.stop_listening()* to stop listening for incoming IP data.

Parameters

- **ip_addr** (ipaddress.IPv4Address) – The IP address to read data from.
- **timeout** (*Integer, optional*) – The time to wait for new IP data in seconds.

Returns

IP message, *None* if this device did not receive new data from the provided IP address.

Return type *IPMessage*

Raises ValueError – If *timeout* is less than 0.

reset()

Override method.

See also:

AbstractXBeeDevice.reset()

scan_counter

Returns the scan counter for this node.

Returns The scan counter for this node.

Return type Integer

send_bluetooth_data(*data*)

Sends the given data to the Bluetooth interface using a User Data Relay frame.

Parameters **data** (*Bytearray*) – Data to send.

Raises

- InvalidOperatingModeException – If the XBee'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*, *data*, *transmit_options=0*)
Deprecated.

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

send_data_async(*remote_xbee*, *data*, *transmit_options=0*)
Deprecated.

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

send_data_broadcast(*data*, *transmit_options=0*)
Deprecated.

Operation not supported in this protocol. This method raises 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 until 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, default=`False`*) – *True* to close the socket just after the transmission. *False* to keep it open.

Raises

- `ValueError` – If *ip_addr* or *protocol* or *data* is *None* or *dest_port* is less than 0 or greater than 65535.
- `OperationNotSupportedException` – If the XBee 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, default=`False`*) – *True* to close the socket just after the transmission. *False* to keep it open.

Raises

- `ValueError` – If *ip_addr* or *protocol* or *data* is *None* or *dest_port* is less than 0 or greater than 65535.
- `OperationNotSupportedException` – If the XBee 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 until 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* or *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'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*)

Sends the packet and waits for the response. The packet to send is escaped depending on the current operating mode.

This method can be synchronous or asynchronous.

If synchronous, this method discards all response packets until it finds the one that has the appropriate frame ID, that is, the sent packet's frame ID.

If asynchronous, this method does not wait for any response and returns *None*.

Parameters

- **packet** (*XBeePacket*) – The packet to send.
- **sync** (*Boolean*) – *True* to wait for the response of the sent packet and return it, *False* otherwise.

Returns

Response packet if *sync* is *True*, *None* otherwise.

Return type *XBeePacket*

Raises

- `TimeoutException` – If *sync* is *True* and the response packet for the sent one cannot be read.
- `InvalidOperatingModeException` – If the XBee operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the packet listener is not running or the XBee's communication interface is closed.

See also:

XBeePacket

send_packet_sync_and_get_response(*packet_to_send*, *timeout=None*)
Sends the packet and waits for its corresponding response.

Parameters

- **packet_to_send** (*XBeePacket*) – The packet to transmit.
- **timeout** (*Integer, optional, default=`None`*) – Number of seconds to wait. -1 to wait indefinitely.

Returns Received response packet.

Return type *XBeePacket*

Raises

- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TimeoutException` – If response is not received in the configured timeout.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

XBeePacket

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'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, if any.

Returns

Serial port of the XBee. *None* if the local XBee does not use serial communication.

Return type *XBeeSerialPort*

See also:

XBeeSerialPort

set_16bit_addr(*value*)

Sets the 16-bit address of the XBee.

Parameters value (*XBee16BitAddress*) – New 16-bit address of the XBee.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If the protocol is not 802.15.4.

set_api_output_mode(*api_output_mode*)

Deprecated since version 1.3: Use *set_api_output_mode_value()*

Sets the API output mode of the XBee.

Parameters api_output_mode (*APIOutputMode*) – New API output mode.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

APIOutputMode

set_api_output_mode_value(*api_output_mode*)

Sets the API output mode of the XBee.

Parameters **api_output_mode** (*Integer*) – New API output mode options. Calculate this value using the method `APIOutputModeBit.calculate_api_output_mode_value()` with a set of *APIOutputModeBit*.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

APIOutputModeBit

set_dest_address(*addr*)

Deprecated.

Operation not supported in this protocol. Use *IPDevice.set_dest_ip_addr()* instead. This method raises 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*) – Digital IO line to sets its value.
- **io_value** (*IOValue*) – IO value to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – IO line to configure.
- **io_mode** (*IOMode*) – IO mode to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOLine
IOMode
get_io_configuration()

set_pan_id(*value*)

Deprecated.

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

set_parameter(*parameter*, *value*, *apply=None*)

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*) – IO Line to be assigned.
- **cycle** (*Integer*) – Duty cycle in % to be assigned. Must be between 0 and 100.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – Read timeout in seconds.

start_listening(*src_port*)
Starts listening for incoming IP transmissions in the provided port.

Parameters **src_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 Bluetooth password of this XBee with the new one provided.

Note that your device must include Bluetooth Low Energy support.

Parameters **new_password** (*String*) – New Bluetooth password.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

`update_device_data_from`(*device*)

Updates the current node information with provided data. This is only for internal use.

Parameters **`device`** (*`AbstractXBeeDevice`*) – XBee to get the data from.

Returns *True* if the node data has been updated, *False* otherwise.

Return type Boolean

`update_firmware`(*xml_firmware_file*, *xbee_firmware_file=None*, *bootloader_firmware_file=None*, *timeout=None*, *progress_callback=None*)

Performs a firmware update operation of the XBee.

Parameters

- **`xml_firmware_file`** (*`String`*) – Path of the XML file that describes the firmware to upload.
- **`xbee_firmware_file`** (*`String, optional, default=`None``*) – Location of the XBee binary firmware file.
- **`bootloader_firmware_file`** (*`String, optional, default=`None``*) – Location of the bootloader binary firmware file.
- **`timeout`** (*`Integer, optional, default=`None``*) – Maximum time to wait for target read operations during the update process (seconds).
- **`progress_callback`** (*`Function, optional, default=`None``*) – Function to to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

Raises

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `OperationNotSupportedException` – If XBee does not support firmware update.
- `FirmwareUpdateException` – If there is any error during the firmware update.

`write_changes`()

Writes configurable parameter values to the non-volatile memory of the XBee so that parameter modifications persist through subsequent resets.

Parameters values remain in the device's memory until overwritten by subsequent use of this method.

If changes are made without writing them, the XBee 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_changes_enabled()`*

to get its status and *enable_apply_changes()* to enable/disable the option. Method *apply_changes()* can be used in order to manually apply the changes.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

class digi.xbee.devices.**LPWANDevice**(*port=None*, *baud_rate=None*, *data_bits=<sphinx.ext.autodoc.importer._MockObject object>*, *stop_bits=<sphinx.ext.autodoc.importer._MockObject object>*, *parity=<sphinx.ext.autodoc.importer._MockObject object>*, *flow_control=<FlowControl.NONE: None>*, *_sync_ops_timeout=4*, *comm_iface=None*)

Bases: *digi.xbee.devices.CellularDevice*

This class provides common functionality for XBee Low-Power Wide-Area Network devices.

Class constructor. Instantiates a new *LPWANDevice* with the provided parameters.

Parameters

- **port** (*String*) – Serial port identifier. Depends on operating system. e.g. '/dev/ttyUSB0' on 'GNU/Linux' or 'COM3' on Windows.
- **baud_rate** (*Integer*) – Serial port baud rate.
- **(Integer, default** (*_sync_ops_timeout*) – `serial.EIGHTBITS`): Port bitsize.
- **(Integer, default** – `serial.STOPBITS_ONE`): Port stop bits.
- **(Character, default** (*parity*) – `serial.PARITY_NONE`): Port parity.
- **(Integer, default** – `FlowControl.NONE`): Port flow control.
- **(Integer, default** – 3): Read timeout (in seconds).
- **comm_iface** (*XBeeCommunicationInterface*) – Communication interface.

Raises All exceptions raised by `XBeeDevice.__init__()` constructor.

See also:

CellularDevice

`CellularDevice.__init__()`

send_ip_data(*ip_addr*, *dest_port*, *protocol*, *data*, *close_socket=False*)

Sends the provided IP data to the given IP address and port using the specified IP protocol.

This method blocks until a success or error response arrives or the configured receive timeout expires.

Parameters

- **ip_addr** (`ipaddress.IPv4Address`) – The IP address to send IP data to.
- **dest_port** (*Integer*) – The destination port of the transmission.

- **protocol** (*IPProtocol*) – The IP protocol used for the transmission.
- **data** (*String or Bytearray*) – The IP data to be sent.
- **close_socket** (*Boolean, optional*) – Must be *False*.

Raises `ValueError` – If *protocol* is not UDP.

send_ip_data_async (*ip_addr*, *dest_port*, *protocol*, *data*, *close_socket=False*)
Sends the provided IP data to the given IP address and port asynchronously using the specified IP protocol.

Asynchronous transmissions do not wait for answer from the remote device or for transmit status packet.

Parameters

- **ip_addr** (`ipaddress.IPv4Address`) – The IP address to send IP data to.
- **dest_port** (*Integer*) – The destination port of the transmission.
- **protocol** (*IPProtocol*) – The IP protocol used for the transmission.
- **data** (*String or Bytearray*) – The IP data to be sent.
- **close_socket** (*Boolean, optional*) – Must be *False*.

Raises `ValueError` – If *protocol* is not UDP.

add_sms_callback (*callback*)
Deprecated.

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

del_sms_callback (*callback*)
Deprecated.

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

send_sms (*phone_number*, *data*)
Deprecated.

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

send_sms_async (*phone_number*, *data*)
Deprecated.

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

add_bluetooth_data_received_callback (*callback*)
Adds a callback for the event *BluetoothDataReceived*.

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

- The Bluetooth data as a Bytearray.

add_data_received_callback (*callback*)
Deprecated.

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

add_expl_data_received_callback (*callback*)
Deprecated.

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

add_fs_frame_received_callback (*callback*)
Adds a callback for the event *FileSystemFrameReceived*.

Parameters **callback** (*Function*) – The callback. Receives four arguments.

- Source (*AbstractXBeeDevice*): The node that sent the file system frame.
- Frame id (Integer): The received frame id.
- Command (*FSCmd*): The file system command.
- Receive options (Integer): Bitfield indicating receive options.

See also:

AbstractXBeeDevice
FSCmd
ReceiveOptions

add_io_sample_received_callback(*callback*)
Deprecated.

Operation not supported in this protocol. This method raises 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*)
Adds a callback for the event *MicroPythonDataReceived*.

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

- The MicroPython data as a Bytearray.

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

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

- The modem status as a *ModemStatus*.

add_packet_received_callback(*callback*)
Adds a callback for the event *PacketReceived*.

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

- The received packet as a *XBeeAPIPacket*.

add_route_received_callback(*callback*)
Adds a callback for the event *RouteReceived*. This works for Zigbee and Digimesh devices.

Parameters **callback** (*Function*) – The callback. Receives three arguments.

- source (*XBeeDevice*): The source node.
- destination (*RemoteXBeeDevice*): The destination node.
- **hops (List): List of intermediate hops from closest to source** to closest to destination (*RemoteXBeeDevice*).

See also:

XBeeDevice.del_route_received_callback()

add_socket_data_received_callback (*callback*)
Adds a callback for the event *SocketDataReceived*.

Parameters **callback** (*Function*) – The callback. Receives two arguments.

- The socket ID as an Integer.
- The data received as Bytearray.

add_socket_data_received_from_callback (*callback*)
Adds a callback for the event *SocketDataReceivedFrom*.

Parameters **callback** (*Function*) – The callback. Receives three arguments.

- The socket ID as an Integer.
- **Source address pair (host, port) where host is a string** representing an IPv4 address like ‘100.50.200.5’, and port is an integer.
- The data received as Bytearray.

add_socket_state_received_callback (*callback*)
Adds a callback for the event *SocketStateReceived*.

Parameters **callback** (*Function*) – The callback. Receives two arguments.

- The socket ID as an Integer.
- The state received as a *SocketState*.

add_user_data_relay_received_callback (*callback*)
Adds a callback for the event *RelayDataReceived*.

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

- The relay data as a *UserDataRelayMessage*.

apply_changes ()
Applies changes via ‘AC’ command.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee’s communication interface is closed.
- `InvalidOperatingModeException` – If the XBee’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

apply_profile (*profile_path*, *timeout=None*, *progress_callback=None*)
Applies the given XBee profile to the XBee.

Parameters

- **profile_path** (*String*) – Path of the XBee profile file to apply.
- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the apply profile (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to receive progress information. Receives two arguments:
  - The current apply profile task as a String
  - The current apply profile task percentage as an Integer

Raises

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `UpdateProfileException` – If there is any error applying the XBee profile.

close()

Closes the communication with the XBee.

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

comm_iface

Returns the hardware interface associated to the XBee.

Returns Hardware interface of the XBee.

Return type *XBeeCommunicationInterface*

See also:

XBeeCommunicationInterface

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 XBee object created.

Return type *XBeeDevice*

Raises `SerialException` – If the port to open does not exist or is already opened.

See also:

XBeeDevice

del_bluetooth_data_received_callback(*callback*)

Deletes a callback for the callback list of *BluetoothDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_data_received_callback(*callback*)
Deprecated.

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

del_expl_data_received_callback(*callback*)
Deprecated.

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

del_fs_frame_received_callback(*callback*)
Deletes a callback for the callback list of *FileSystemFrameReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_io_sample_received_callback(*callback*)
Deprecated.

Operation not supported in this protocol. This method raises 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.

del_micropython_data_received_callback(*callback*)
Deletes a callback for the callback list of *MicroPythonDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_modem_status_received_callback(*callback*)
Deletes a callback for the callback list of *ModemStatusReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_packet_received_callback(*callback*)
Deletes a callback for the callback list of *PacketReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_route_received_callback(*callback*)
Deletes a callback for the callback list of *RouteReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

See also:

XBeeDevice.add_route_received_callback()

del_socket_data_received_callback(*callback*)
Deletes a callback for the callback list of *SocketDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_socket_data_received_from_callback(*callback*)
Deletes a callback for the callback list of *SocketDataReceivedFrom* event.

Parameters **callback** (*Function*) – The callback to delete.

del_socket_state_received_callback(*callback*)
Deletes a callback for the callback list of *SocketStateReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_user_data_relay_received_callback(*callback*)

Deletes a callback for the callback list of *RelayDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

determine_protocol(*hardware_version*, *firmware_version*)

Determines the XBee protocol based on the given hardware and firmware versions.

Parameters

- **hardware_version** (*Integer*) – Hardware version to get its protocol.
- **firmware_version** (*Bytearray*) – Firmware version to get its protocol.

Returns

XBee protocol corresponding to the given hardware and firmware versions.

Return type *XBeeProtocol*

disable_bluetooth()

Disables the Bluetooth interface of this XBee.

Note that your device must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

enable_apply_changes(*value*)

Sets apply changes flag.

Parameters **value** (*Boolean*) – *True* to enable apply changes flag, *False* to disable it.

enable_bluetooth()

Enables the Bluetooth interface of this XBee.

To work with this interface, you must also configure the Bluetooth password if not done previously. Use method *AbstractXBeeDevice.update_bluetooth_password()*.

Note that your XBee must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

execute_command(*parameter*, *value=None*, *apply=None*)

Executes the provided command.

Parameters

- **(String or** (*parameter*) – class: *.ATStringCommand*): AT command to execute.

- **value** (*bytearray, optional, default=`None`*) – Command value (if any).
- **apply** (*Boolean, optional, default=`None`*) – *True* to apply changes in XBee configuration, *False* not to apply them, *None* to use *is_apply_changes_enabled()* returned value.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

AbstractXBeeDevice.get_parameter()
AbstractXBeeDevice.set_parameter()
AbstractXBeeDevice.apply_changes()
AbstractXBeeDevice.write_changes()
AbstractXBeeDevice.is_apply_changes_enabled()
AbstractXBeeDevice.enable_apply_changes()

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*) – IO line to get its ADC value.

Returns Analog value corresponding to the provided IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

- `OperationNotSupportedException` – If response does not contain the value for the given IO line.

See also:

IOLine

set_io_configuration()

get_api_output_mode()

Deprecated since version 1.3: Use *get_api_output_mode_value()*

Returns the API output mode of the XBee.

The API output mode determines the format of the data through the serial interface of the XBee.

Returns API output mode of the XBee.

Return type *APIOutputMode*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

APIOutputMode

get_api_output_mode_value()

Returns the API output mode of the XBee.

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

Returns the parameter value.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

digi.xbee.models.mode.APIOutputModeBit

get_bluetooth_mac_addr()
Reads and returns the EUI-48 Bluetooth MAC address of this XBee following the format *00112233AABB*.

Note that your device must include Bluetooth Low Energy support.

Returns The Bluetooth MAC address.

Return type String

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

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 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 raises an `AttributeError`.

get_dest_ip_addr()
Returns the destination IP address.

Returns Configured destination IP address.

Return type `ipaddress.IPv4Address`

Raises

- `TimeoutException` – If there is a timeout getting the destination IP address.
- `XBeeException` – If there is any other XBee related exception.

See also:

ipaddress.IPv4Address

get_dio_value(*io_line*)

Returns the digital value of the provided IO line.

The provided IO line must be previously configured as digital I/O. To do so, use *AbstractXBeeDevice.set_io_configuration()*.

Parameters **io_line** (*IOLine*) – the DIO line to gets its digital value.

Returns current value of the provided IO line.

Return type *IOValue*

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.
- OperationNotSupportedException – If response does not contain the value for the given IO line.

See also:

IOLine
IOValue
set_io_configuration()

get_file_manager()

Returns the file system manager for the XBee.

Returns The file system manager.

Return type *FileSystemManager*

Raises FileSystemNotSupportedException – If the XBee does not support filesystem.

get_firmware_version()

Returns the firmware version of the XBee.

Returns Firmware version of the XBee.

Return type Bytearray

get_hardware_version()

Returns the hardware version of the XBee.

Returns Hardware version of the XBee.

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*) – IO line to get its configuration.

Returns IO mode of the IO line provided.

Return type *IOMode*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOLine
IOMode
set_io_configuration()

get_io_sampling_rate()

Deprecated.

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

get_ip_addr()

Returns the IP address of this IP XBee.

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.

Returns The next frame ID of the XBee.

Return type Integer

get_node_id()
Deprecated.

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

get_pan_id()
Deprecated.

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

get_parameter(*parameter*, *parameter_value=None*, *apply=None*)
Override.

See also:

AbstractXBeeDevice.get_parameter()

get_power_level()
Deprecated.

Operation not supported in this protocol. This method raises 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*) – IO line to get its PWM duty cycle.

Returns PWM duty cycle of the given IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `ValueError` – If *io_line* has no PWM capability.

See also:

IOLine

get_role()

Gets the XBee role.

Returns the role of the XBee.

Return type *Role*

See also:

Role

get_route_to_node(*remote*, *timeout=10*, *force=True*)

Gets the route from this XBee to the given remote node.

For Zigbee:

- 'AR' parameter of the local node must be configured with a value different from 'FF'.
- Set *force* to *True* to force the Zigbee remote node to return its route independently of the local node configuration as high or low RAM concentrator ('DO' of the local value)

Parameters

- **remote** (*RemoteXBeeDevice*) – The remote node.
- **timeout** (*Float, optional, default=10*) – Maximum number of seconds to wait for the route.
- **force** (*Boolean*) – *True* to force asking for the route, *False* otherwise. Only for Zigbee.

Returns

Tuple containing route data:

- status (*TransmitStatus*): The transmit status.
- Tuple with route data (*None* if the route was not read in the provided timeout):
  - source (*RemoteXBeeDevice*): The source node of the route.
  - destination (*RemoteXBeeDevice*): The destination node of the route.
  - hops (List): List of intermediate nodes (*RemoteXBeeDevice*) ordered from closest to source to closest to destination node (source and destination not included).

Return type Tuple

get_socket_info(*socket_id*)

Returns the information of the socket with the given socket ID.

Parameters **socket_id** (*Integer*) – ID of the socket.

Returns

The socket information, or *None* if the socket with that ID does not exist.

Return type *SocketInfo*

Raises

- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TimeoutException` – If the response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

`SocketInfo`

`get_sockets_list``()`

Returns a list with the IDs of all active (open) sockets.

Returns

list with the IDs of all active (open) sockets, or empty list if there is not any active socket.

Return type List

Raises

- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TimeoutException` – If the response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.

`get_sync_ops_timeout``()`

Returns the serial port read timeout.

Returns 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 are executed before user callbacks.

Returns *`PacketReceived`*

`has_explicit_packets``()`

Returns if there are pending explicit packets to read. This does not include non-explicit packets.

Returns *True* if there are pending packets, *False* otherwise.

Return type Boolean

See also:

`XBeeDevice.has_packets()`

has_packets()

Returns if there are pending packets to read. This does not include explicit packets.

Returns *True* if there are pending packets, *False* otherwise.

Return type Boolean

See also:

XBeeDevice.has_explicit_packets()

is_apply_changes_enabled()

Returns whether apply changes flag is enabled.

Returns *True* if apply changes flag is enabled, *False* otherwise.

Return type Boolean

is_connected()

Returns whether the device is connected to the Internet.

Returns *True* if 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_device_info_complete()

Override.

See also:

AbstractXBeeDevice.is_device_info_complete()

is_open()

Returns whether this XBee is open.

Returns Boolean. *True* if this XBee is open, *False* otherwise.

is_remote()

Override method.

See also:

AbstractXBeeDevice.is_remote()

log

Returns the XBee logger.

Returns The XBee device logger.

Return type `Logger`

open (*force_settings=False*)
Override.

See also:

XBeeDevice.open()

operating_mode
Returns the operating mode of this XBee.

Returns *OperatingMode*. This XBee operating mode.

reachable
Returns whether the XBee is reachable.

Returns *True* if the device is reachable, *False* otherwise.

Return type Boolean

read_data (*timeout=None*)
Deprecated.

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

read_data_from (*remote_xbee*, *timeout=None*)
Deprecated.

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

read_device_info (*init=True*, *fire_event=True*)
Updates all instance parameters reading them from the XBee.

Parameters

- **init** (*Boolean, optional, default=`True`*) – If *False* only not initialized parameters are read, all if *True*.
- **fire_event** (*Boolean, optional, default=`True`*) – *True* to throw and update event if any parameter changed, *False* otherwise.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

AbstractXBeeDevice.is_device_info_complete()

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

Returns IO sample read from the XBee.

Return type *IOSample*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOSample

read_ip_data(*timeout=3*)

Reads new IP data received by this XBee during the provided timeout.

This method blocks until new IP data is received or the provided timeout expires.

For non-blocking operations, register a callback and use the method *IPDevice.add_ip_data_received_callback()*.

Before reading IP data you need to start listening for incoming IP data at a specific port. Use the method *IPDevice.start_listening()* for that purpose. When finished, you can use the method *IPDevice.stop_listening()* to stop listening for incoming IP data.

Parameters **timeout** (*Integer, optional*) – The time to wait for new IP data in seconds.

Returns IP message, *None* if this device did not receive new data.

Return type *IPMessage*

Raises `ValueError` – If *timeout* is less than 0.

read_ip_data_from(*ip_addr*, *timeout=3*)

Reads new IP data received from the given IP address during the provided timeout.

This method blocks until new IP data from the provided IP address is received or the given timeout expires.

For non-blocking operations, register a callback and use the method *IPDevice.add_ip_data_received_callback()*.

Before reading IP data you need to start listening for incoming IP data at a specific port. Use the method *IPDevice.start_listening()* for that purpose. When finished, you can use the method *IPDevice.stop_listening()* to stop listening for incoming IP data.

Parameters

- **ip_addr** (`ipaddress.IPv4Address`) – The IP address to read data from.
- **timeout** (*Integer, optional*) – The time to wait for new IP data in seconds.

Returns

IP message, *None* if this device did not receive new data from the provided IP address.

Return type *IPMessage*

Raises `ValueError` – If *timeout* is less than 0.

reset()
Override method.

See also:

AbstractXBeeDevice.reset()

scan_counter
Returns the scan counter for this node.

Returns The scan counter for this node.

Return type Integer

send_bluetooth_data(*data*)
Sends the given data to the Bluetooth interface using a User Data Relay frame.

Parameters **data** (*Bytearray*) – Data to send.

Raises

- `InvalidOperatingModeException` – If the XBee'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, data, transmit_options=0*)
Deprecated.

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

send_data_async(*remote_xbee, data, transmit_options=0*)
Deprecated.

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

send_data_broadcast(*data, transmit_options=0*)
Deprecated.

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

send_ip_data_broadcast(*dest_port, data*)
Sends the provided IP data to all clients.

This method blocks until 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* or *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'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*)

Sends the packet and waits for the response. The packet to send is escaped depending on the current operating mode.

This method can be synchronous or asynchronous.

If synchronous, this method discards all response packets until it finds the one that has the appropriate frame ID, that is, the sent packet's frame ID.

If asynchronous, this method does not wait for any response and returns *None*.

Parameters

- **`packet`** (*`XBeePacket`*) – The packet to send.
- **`sync`** (*`Boolean`*) – *True* to wait for the response of the sent packet and return it, *False* otherwise.

Returns

Response packet if *sync* is *True*, *None* otherwise.

Return type *`XBeePacket`*

Raises

- `TimeoutException` – If *sync* is *True* and the response packet for the sent one cannot be read.
- `InvalidOperatingModeException` – If the XBee operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the packet listener is not running or the XBee's communication interface is closed.

See also:

`XBeePacket`

send_packet_sync_and_get_response(*packet_to_send*, *timeout=None*)
Sends the packet and waits for its corresponding response.

Parameters

- **packet_to_send** (*XBeePacket*) – The packet to transmit.
- **timeout** (*Integer, optional, default=`None`*) – Number of seconds to wait. -1 to wait indefinitely.

Returns Received response packet.

Return type *XBeePacket*

Raises

- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TimeoutException` – If response is not received in the configured timeout.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

XBeePacket

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'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, if any.

Returns

Serial port of the XBee. *None* if the local XBee does not use serial communication.

Return type *XBeeSerialPort*

See also:

XBeeSerialPort

set_16bit_addr(*value*)

Sets the 16-bit address of the XBee.

Parameters **value** (*XBee16BitAddress*) – New 16-bit address of the XBee.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If the protocol is not 802.15.4.

set_api_output_mode(*api_output_mode*)

Deprecated since version 1.3: Use *set_api_output_mode_value()*

Sets the API output mode of the XBee.

Parameters **api_output_mode** (*APIOutputMode*) – New API output mode.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

APIOutputMode

set_api_output_mode_value(*api_output_mode*)

Sets the API output mode of the XBee.

Parameters **api_output_mode** (*Integer*) – New API output mode options. Calculate this value using the method `APIOutputModeBit.calculate_api_output_mode_value()` with a set of *APIOutputModeBit*.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

`APIOutputModeBit`

`set_dest_address`(*addr*)
Deprecated.

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

`set_dest_ip_addr`(*address*)
Sets the destination IP address.

Parameters **`address`** (`ipaddress.IPv4Address`) – Destination IP address.

Raises

- `ValueError` – If *address* is *None*.
- `TimeoutException` – If there is a timeout setting the destination IP address.
- `XBeeException` – If there is any other XBee related exception.

See also:

`ipaddress.IPv4Address`

`set_dio_change_detection`(*io_lines_set*)
Deprecated.

Operation not supported in this protocol. This method raises 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`*) – Digital IO line to sets its value.
- **`io_value`** (*`IOValue`*) – IO value to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – IO line to configure.
- **io_mode** (*IOMode*) – IO mode to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOLine
IOMode
get_io_configuration()

set_io_sampling_rate(*rate*)
Deprecated.

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

set_node_id(*node_id*)
Deprecated.

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

set_pan_id(*value*)
Deprecated.

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

set_parameter(*parameter*, *value*, *apply=None*)
Override.

See: *AbstractXBeeDevice.set_parameter()*

set_power_level(*power_level*)
Deprecated.

Operation not supported in this protocol. This method raises 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*) – IO Line to be assigned.
- **cycle** (*Integer*) – Duty cycle in % to be assigned. Must be between 0 and 100.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – Read timeout in seconds.

start_listening(*src_port*)
Starts listening for incoming IP transmissions in the provided port.

Parameters **src_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 Bluetooth password of this XBee with the new one provided.

Note that your device must include Bluetooth Low Energy support.

Parameters **new_password** (*String*) – New Bluetooth password.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

update_device_data_from(*device*)
Updates the current node information with provided data. This is only for internal use.

Parameters **device** (*AbstractXBeeDevice*) – XBee to get the data from.

Returns *True* if the node data has been updated, *False* otherwise.

Return type Boolean

update_firmware(*xml_firmware_file*, *xbee_firmware_file=None*, *bootloader_firmware_file=None*, *timeout=None*, *progress_callback=None*)

Performs a firmware update operation of the XBee.

Parameters

- **xml_firmware_file** (*String*) – Path of the XML file that describes the firmware to upload.
- **xbee_firmware_file** (*String, optional, default=`None`*) – Location of the XBee binary firmware file.
- **bootloader_firmware_file** (*String, optional, default=`None`*) – Location of the bootloader binary firmware file.
- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the update process (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

Raises

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `OperationNotSupportedException` – If XBee does not support firmware update.
- `FirmwareUpdateException` – If there is any error during the firmware update.

write_changes()

Writes configurable parameter values to the non-volatile memory of the XBee so that parameter modifications persist through subsequent resets.

Parameters values remain in the device's memory until overwritten by subsequent use of this method.

If changes are made without writing them, the XBee 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_changes_enabled()* to get its status and *enable_apply_changes()* to enable/disable the option. Method *apply_changes()* can be used in order to manually apply the changes.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

class digi.xbee.devices.**NBIoTDevice**(*port=None*, *baud_rate=None*, *data_bits=<sphinx.ext.autodoc.importer._MockObject object>*, *stop_bits=<sphinx.ext.autodoc.importer._MockObject object>*, *parity=<sphinx.ext.autodoc.importer._MockObject object>*, *flow_control=<FlowControl.NONE: None>*, *_sync_ops_timeout=4*, *comm_iface=None*)

Bases: *digi.xbee.devices.LPWANDevice*

This class represents a local NB-IoT device.

Class constructor. Instantiates a new *NBIoTDevice* with the provided parameters.

Parameters

- **port** (*String*) – Serial port identifier. Depends on operating system. e.g. '/dev/ttyUSB0' on 'GNU/Linux' or 'COM3' on Windows.
- **baud_rate** (*Integer*) – Serial port baud rate.
- **(Integer, default** (*_sync_ops_timeout*) – serial.EIGHTBITS): Port bitsize.
- **(Integer, default** – serial.STOPBITS_ONE): Port stop bits.
- **(Character, default** (*parity*) – serial.PARITY_NONE): Port parity.
- **(Integer, default** – FlowControl.NONE): Port flow control.
- **(Integer, default** – 3): Read timeout (in seconds).
- **comm_iface** (*XBeeCommunicationInterface*) – Communication interface.

Raises All exceptions raised by XBeeDevice.__init__() constructor.

See also:

LPWANDevice

LPWANDevice.__init__()

open(*force_settings=False*)

Override.

See also:

XBeeDevice.open()

get_protocol()

Override.

See also:

XBeeDevice.get_protocol()

add_bluetooth_data_received_callback(*callback*)

Adds a callback for the event *BluetoothDataReceived*.

Parameters **`callback`** (*`Function`*) – The callback. Receives one argument.

- The Bluetooth data as a Bytearray.

`add_data_received_callback`(*callback*)

Deprecated.

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

`add_expl_data_received_callback`(*callback*)

Deprecated.

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

`add_fs_frame_received_callback`(*callback*)

Adds a callback for the event *`FileSystemFrameReceived`*.

Parameters **`callback`** (*`Function`*) – The callback. Receives four arguments.

- Source (*`AbstractXBeeDevice`*): The node that sent the file system frame.
- Frame id (Integer): The received frame id.
- Command (*`FSCmd`*): The file system command.
- Receive options (Integer): Bitfield indicating receive options.

See also:

`AbstractXBeeDevice`
`FSCmd`
`ReceiveOptions`

`add_io_sample_received_callback`(*callback*)

Deprecated.

Operation not supported in this protocol. This method raises 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*)

Adds a callback for the event *`MicroPythonDataReceived`*.

Parameters **`callback`** (*`Function`*) – The callback. Receives one argument.

- The MicroPython data as a Bytearray.

`add_modem_status_received_callback`(*callback*)

Adds a callback for the event *`ModemStatusReceived`*.

Parameters **`callback`** (*`Function`*) – The callback. Receives one argument.

- The modem status as a *`ModemStatus`*.

`add_packet_received_callback`(*callback*)

Adds a callback for the event *`PacketReceived`*.

Parameters **`callback`** (*`Function`*) – The callback. Receives one argument.

- The received packet as a `XBeeAPIPacket`.

add_route_received_callback(*callback*)
Adds a callback for the event `RouteReceived`. This works for Zigbee and Digimesh devices.

Parameters **callback** (`Function`) – The callback. Receives three arguments.

- source (`XBeeDevice`): The source node.
- destination (`RemoteXBeeDevice`): The destination node.
- **hops (List): List of intermediate hops from closest to source** to closest to destination (`RemoteXBeeDevice`).

See also:

`XBeeDevice.del_route_received_callback()`

add_sms_callback(*callback*)
Deprecated.

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

add_socket_data_received_callback(*callback*)
Adds a callback for the event `SocketDataReceived`.

Parameters **callback** (`Function`) – The callback. Receives two arguments.

- The socket ID as an Integer.
- The data received as Bytearray.

add_socket_data_received_from_callback(*callback*)
Adds a callback for the event `SocketDataReceivedFrom`.

Parameters **callback** (`Function`) – The callback. Receives three arguments.

- The socket ID as an Integer.
- **Source address pair (host, port) where host is a string** representing an IPv4 address like '100.50.200.5', and port is an integer.
- The data received as Bytearray.

add_socket_state_received_callback(*callback*)
Adds a callback for the event `SocketStateReceived`.

Parameters **callback** (`Function`) – The callback. Receives two arguments.

- The socket ID as an Integer.
- The state received as a `SocketState`.

add_user_data_relay_received_callback(*callback*)
Adds a callback for the event `RelayDataReceived`.

Parameters **callback** (`Function`) – The callback. Receives one argument.

- The relay data as a `UserDataRelayMessage`.

apply_changes()
Applies changes via 'AC' command.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

apply_profile (*profile_path*, *timeout=None*, *progress_callback=None*)
Applies the given XBee profile to the XBee.

Parameters

- **profile_path** (*String*) – Path of the XBee profile file to apply.
- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the apply profile (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to receive progress information. Receives two arguments:
  - The current apply profile task as a String
  - The current apply profile task percentage as an Integer

Raises

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `UpdateProfileException` – If there is any error applying the XBee profile.

close ()
Closes the communication with the XBee.

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

comm_iface
Returns the hardware interface associated to the XBee.

Returns Hardware interface of the XBee.

Return type *XBeeCommunicationInterface*

See also:

XBeeCommunicationInterface

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 XBee object created.

Return type *XBeeDevice*

Raises `SerialException` – If the port to open does not exist or is already opened.

See also:

XBeeDevice

del_bluetooth_data_received_callback(*callback*)
Deletes a callback for the callback list of *BluetoothDataReceived* event.

Parameters callback (*Function*) – The callback to delete.

del_data_received_callback(*callback*)
Deprecated.

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

del_expl_data_received_callback(*callback*)
Deprecated.

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

del_fs_frame_received_callback(*callback*)
Deletes a callback for the callback list of *FileSystemFrameReceived* event.

Parameters callback (*Function*) – The callback to delete.

del_io_sample_received_callback(*callback*)
Deprecated.

Operation not supported in this protocol. This method raises 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.

del_micropython_data_received_callback(*callback*)
Deletes a callback for the callback list of *MicroPythonDataReceived* event.

Parameters callback (*Function*) – The callback to delete.

del_modem_status_received_callback(*callback*)
Deletes a callback for the callback list of *ModemStatusReceived* event.

Parameters callback (*Function*) – The callback to delete.

del_packet_received_callback(*callback*)
Deletes a callback for the callback list of *PacketReceived* event.

Parameters callback (*Function*) – The callback to delete.

del_route_received_callback(*callback*)
Deletes a callback for the callback list of *RouteReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

See also:

XBeeDevice.add_route_received_callback()

del_sms_callback(*callback*)
Deprecated.

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

del_socket_data_received_callback(*callback*)
Deletes a callback for the callback list of *SocketDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_socket_data_received_from_callback(*callback*)
Deletes a callback for the callback list of *SocketDataReceivedFrom* event.

Parameters **callback** (*Function*) – The callback to delete.

del_socket_state_received_callback(*callback*)
Deletes a callback for the callback list of *SocketStateReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_user_data_relay_received_callback(*callback*)
Deletes a callback for the callback list of *RelayDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

determine_protocol(*hardware_version*, *firmware_version*)
Determines the XBee protocol based on the given hardware and firmware versions.

Parameters

- **hardware_version** (*Integer*) – Hardware version to get its protocol.
- **firmware_version** (*Bytearray*) – Firmware version to get its protocol.

Returns

XBee protocol corresponding to the given hardware and firmware versions.

Return type *XBeeProtocol*

disable_bluetooth()
Disables the Bluetooth interface of this XBee.

Note that your device must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

enable_apply_changes(*value*)
Sets apply changes flag.

Parameters **value** (*Boolean*) – *True* to enable apply changes flag, *False* to disable it.

enable_bluetooth()

Enables the Bluetooth interface of this XBee.

To work with this interface, you must also configure the Bluetooth password if not done previously. Use method *AbstractXBeeDevice.update_bluetooth_password()*.

Note that your XBee must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

execute_command(*parameter*, *value=None*, *apply=None*)

Executes the provided command.

Parameters

- **(String or** (*parameter*) – class: *.ATStringCommand*): AT command to execute.
- **value** (*bytearray, optional, default=`None`*) – Command value (if any).
- **apply** (*Boolean, optional, default=`None`*) – *True* to apply changes in XBee configuration, *False* not to apply them, *None* to use *is_apply_changes_enabled()* returned value.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

AbstractXBeeDevice.get_parameter()
AbstractXBeeDevice.set_parameter()
AbstractXBeeDevice.apply_changes()
AbstractXBeeDevice.write_changes()
AbstractXBeeDevice.is_apply_changes_enabled()
AbstractXBeeDevice.enable_apply_changes()

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`) – IO line to get its ADC value.

Returns Analog value corresponding to the provided IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If response does not contain the value for the given IO line.

See also:

`IOLine`
`set_io_configuration()`

get_api_output_mode()

Deprecated since version 1.3: Use `get_api_output_mode_value()`

Returns the API output mode of the XBee.

The API output mode determines the format of the data through the serial interface of the XBee.

Returns API output mode of the XBee.

Return type `APIOutputMode`

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

APIOutputMode

get_api_output_mode_value()

Returns the API output mode of the XBee.

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

Returns the parameter value.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

digi.xbee.models.mode.APIOutputModeBit

get_bluetooth_mac_addr()

Reads and returns the EUI-48 Bluetooth MAC address of this XBee following the format *00112233AABB*.

Note that your device must include Bluetooth Low Energy support.

Returns The Bluetooth MAC address.

Return type String

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

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 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 raises an `AttributeError`.

get_dest_ip_addr()

Returns the destination IP address.

Returns Configured destination IP address.

Return type `ipaddress.IPv4Address`

Raises

- `TimeoutException` – If there is a timeout getting the destination IP address.
- `XBeeException` – If there is any other XBee related exception.

See also:

`ipaddress.IPv4Address`

get_dio_value(*io_line*)

Returns the digital value of the provided IO line.

The provided IO line must be previously configured as digital I/O. To do so, use *`AbstractXBeeDevice.set_io_configuration()`*.

Parameters **io_line** (*`IOLine`*) – the DIO line to gets its digital value.

Returns current value of the provided IO line.

Return type *`IOValue`*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If response does not contain the value for the given IO line.

See also:

`IOLine`

IOValue
set_io_configuration()

get_file_manager()
Returns the file system manager for the XBee.

Returns The file system manager.

Return type *FileSystemManager*

Raises FileSystemNotSupportedException – If the XBee does not support filesystem.

get_firmware_version()
Returns the firmware version of the XBee.

Returns Firmware version of the XBee.

Return type Bytearray

get_hardware_version()
Returns the hardware version of the XBee.

Returns Hardware version of the XBee.

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*) – IO line to get its configuration.

Returns IO mode of the IO line provided.

Return type *IOMode*

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

See also:

IOLine
IOMode
set_io_configuration()

get_io_sampling_rate()
Deprecated.

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

get_ip_addr()
Returns the IP address of this IP XBee.

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.

Returns The next frame ID of the XBee.

Return type Integer

get_node_id()
Deprecated.

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

get_pan_id()
Deprecated.

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

get_parameter(*parameter*, *parameter_value=None*, *apply=None*)
Override.

See also:

AbstractXBeeDevice.get_parameter()

get_power_level()
Deprecated.

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

get_pwm_duty_cycle(*io_line*)
Returns the PWM duty cycle in % corresponding to the provided IO line.

Parameters **io_line** (*IOLine*) – IO line to get its PWM duty cycle.

Returns PWM duty cycle of the given IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `ValueError` – If *io_line* has no PWM capability.

See also:

IOLine

get_role()

Gets the XBee role.

Returns the role of the XBee.

Return type *Role*

See also:

Role

get_route_to_node(*remote*, *timeout=10*, *force=True*)

Gets the route from this XBee to the given remote node.

For Zigbee:

- 'AR' parameter of the local node must be configured with a value different from 'FF'.
- Set *force* to *True* to force the Zigbee remote node to return its route independently of the local node configuration as high or low RAM concentrator ('DO' of the local value)

Parameters

- **remote** (*RemoteXBeeDevice*) – The remote node.
- **timeout** (*Float, optional, default=10*) – Maximum number of seconds to wait for the route.
- **force** (*Boolean*) – *True* to force asking for the route, *False* otherwise. Only for Zigbee.

Returns

Tuple containing route data:

- status (*TransmitStatus*): The transmit status.
- Tuple with route data (*None* if the route was not read in the provided timeout):

- source (*RemoteXBeeDevice*): The source node of the route.
- destination (*RemoteXBeeDevice*): The destination node of the route.
- hops (List): List of intermediate nodes (*RemoteXBeeDevice*) ordered from closest to source to closest to destination node (source and destination not included).

Return type Tuple

get_socket_info(*socket_id*)
Returns the information of the socket with the given socket ID.

Parameters **socket_id** (*Integer*) – ID of the socket.

Returns

The socket information, or *None* if the socket with that ID does not exist.

Return type *SocketInfo*

Raises

- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TimeoutException` – If the response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

SocketInfo

get_sockets_list()
Returns a list with the IDs of all active (open) sockets.

Returns

list with the IDs of all active (open) sockets, or empty list if there is not any active socket.

Return type List

Raises

- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TimeoutException` – If the response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.

get_sync_ops_timeout()
Returns the serial port read timeout.

Returns 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 are executed before user callbacks.

Returns *PacketReceived*

has_explicit_packets()

Returns if there are pending explicit packets to read. This does not include non-explicit packets.

Returns *True* if there are pending packets, *False* otherwise.

Return type Boolean

See also:

XBeeDevice.has_packets()

has_packets()

Returns if there are pending packets to read. This does not include explicit packets.

Returns *True* if there are pending packets, *False* otherwise.

Return type Boolean

See also:

XBeeDevice.has_explicit_packets()

is_apply_changes_enabled()

Returns whether apply changes flag is enabled.

Returns *True* if apply changes flag is enabled, *False* otherwise.

Return type Boolean

is_connected()

Returns whether the device is connected to the Internet.

Returns *True* if 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_device_info_complete()

Override.

See also:

AbstractXBeeDevice.is_device_info_complete()

is_open()
Returns whether this XBee is open.

Returns Boolean. *True* if this XBee is open, *False* otherwise.

is_remote()
Override method.

See also:

AbstractXBeeDevice.is_remote()

log
Returns the XBee logger.

Returns The XBee device logger.

Return type `Logger`

operating_mode
Returns the operating mode of this XBee.

Returns *OperatingMode*. This XBee operating mode.

reachable
Returns whether the XBee is reachable.

Returns *True* if the device is reachable, *False* otherwise.

Return type Boolean

read_data(*timeout=None*)
Deprecated.

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

read_data_from(*remote_xbee*, *timeout=None*)
Deprecated.

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

read_device_info(*init=True*, *fire_event=True*)
Updates all instance parameters reading them from the XBee.

Parameters

- **init** (*Boolean, optional, default=`True`*) – If *False* only not initialized parameters are read, all if *True*.
- **fire_event** (*Boolean, optional, default=`True`*) – *True* to throw and update event if any parameter changed, *False* otherwise.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

AbstractXBeeDevice.is_device_info_complete()

read_io_sample()

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

Returns IO sample read from the XBee.

Return type *IOSample*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOSample

read_ip_data(*timeout=3*)

Reads new IP data received by this XBee during the provided timeout.

This method blocks until new IP data is received or the provided timeout expires.

For non-blocking operations, register a callback and use the method *IPDevice.add_ip_data_received_callback()*.

Before reading IP data you need to start listening for incoming IP data at a specific port. Use the method *IPDevice.start_listening()* for that purpose. When finished, you can use the method *IPDevice.stop_listening()* to stop listening for incoming IP data.

Parameters **timeout** (*Integer, optional*) – The time to wait for new IP data in seconds.

Returns IP message, *None* if this device did not receive new data.

Return type *IPMessage*

Raises `ValueError` – If *timeout* is less than 0.

read_ip_data_from(*ip_addr*, *timeout=3*)

Reads new IP data received from the given IP address during the provided timeout.

This method blocks until new IP data from the provided IP address is received or the given timeout expires.

For non-blocking operations, register a callback and use the method *IPDevice.add_ip_data_received_callback()*.

Before reading IP data you need to start listening for incoming IP data at a specific port. Use the method *IPDevice.start_listening()* for that purpose. When finished, you can use the method *IPDevice.stop_listening()* to stop listening for incoming IP data.

Parameters

- **ip_addr** (`ipaddress.IPv4Address`) – The IP address to read data from.

- **timeout** (*Integer, optional*) – The time to wait for new IP data in seconds.

Returns

IP message, *None* if this device did not receive new data from the provided IP address.

Return type *IPMessage*

Raises `ValueError` – If *timeout* is less than 0.

reset()

Override method.

See also:

AbstractXBeeDevice.reset()

scan_counter

Returns the scan counter for this node.

Returns The scan counter for this node.

Return type Integer

send_bluetooth_data(*data*)

Sends the given data to the Bluetooth interface using a User Data Relay frame.

Parameters **data** (*Bytearray*) – Data to send.

Raises

- `InvalidOperatingModeException` – If the XBee'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, data, transmit_options=0*)

Deprecated.

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

send_data_async(*remote_xbee, data, transmit_options=0*)

Deprecated.

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

send_data_broadcast(*data, transmit_options=0*)

Deprecated.

Operation not supported in this protocol. This method raises 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.

This method blocks until 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.

send_ip_data_broadcast(*dest_port*, *data*)

Sends the provided IP data to all clients.

This method blocks until 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* or *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'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*)

Sends the packet and waits for the response. The packet to send is escaped depending on the current operating mode.

This method can be synchronous or asynchronous.

If synchronous, this method discards all response packets until it finds the one that has the appropriate frame ID, that is, the sent packet's frame ID.

If asynchronous, this method does not wait for any response and returns *None*.

Parameters

- **packet** (*XBeePacket*) – The packet to send.
- **sync** (*Boolean*) – *True* to wait for the response of the sent packet and return it, *False* otherwise.

Returns

Response packet if *sync* is *True*, *None* otherwise.

Return type *XBeePacket*

Raises

- `TimeoutException` – If *sync* is *True* and the response packet for the sent one cannot be read.
- `InvalidOperatingModeException` – If the XBee operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the packet listener is not running or the XBee's communication interface is closed.

See also:

XBeePacket

send_packet_sync_and_get_response (*packet_to_send*, *timeout=None*)

Sends the packet and waits for its corresponding response.

Parameters

- **packet_to_send** (*XBeePacket*) – The packet to transmit.
- **timeout** (*Integer, optional, default=`None`*) – Number of seconds to wait. -1 to wait indefinitely.

Returns Received response packet.

Return type *XBeePacket*

Raises

- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TimeoutException` – If response is not received in the configured timeout.

- `XBeeException` – If the XBee's communication interface is closed.

See also:

`XBeePacket`

`send_sms`(*phone_number*, *data*)
Deprecated.

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

`send_sms_async`(*phone_number*, *data*)
Deprecated.

Operation not supported in this protocol. This method raises 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'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, if any.

Returns

Serial port of the XBee. *None* if the local XBee does not use serial communication.

Return type *`XBeeSerialPort`*

See also:

`XBeeSerialPort`

`set_16bit_addr`(*value*)
Sets the 16-bit address of the XBee.

Parameters **`value`** (*`XBee16BitAddress`*) – New 16-bit address of the XBee.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If the protocol is not 802.15.4.

`set_api_output_mode`(*api_output_mode*)

Deprecated since version 1.3: Use *`set_api_output_mode_value()`*

Sets the API output mode of the XBee.

Parameters **`api_output_mode`** (*`APIOutputMode`*) – New API output mode.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

`APIOutputMode`

`set_api_output_mode_value`(*api_output_mode*)

Sets the API output mode of the XBee.

Parameters **`api_output_mode`** (*`Integer`*) – New API output mode options. Calculate this value using the method `APIOutputModeBit.calculate_api_output_mode_value()` with a set of *`APIOutputModeBit`*.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

`APIOutputModeBit`

set_dest_address(*addr*)
Deprecated.

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

set_dest_ip_addr(*address*)
Sets the destination IP address.

Parameters **address** (`ipaddress.IPv4Address`) – Destination IP address.

Raises

- `ValueError` – If *address* is *None*.
- `TimeoutException` – If there is a timeout setting the destination IP address.
- `XBeeException` – If there is any other XBee related exception.

See also:

`ipaddress.IPv4Address`

set_dio_change_detection(*io_lines_set*)
Deprecated.

Operation not supported in this protocol. This method raises 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*) – Digital IO line to sets its value.
- **io_value** (*IOValue*) – IO value to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – IO line to configure.
- **io_mode** (*IOMode*) – IO mode to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`IOLine`
`IOMode`
`get_io_configuration()`

`set_io_sampling_rate`(*rate*)
Deprecated.

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

`set_node_id`(*node_id*)
Deprecated.

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

`set_pan_id`(*value*)
Deprecated.

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

`set_parameter`(*parameter*, *value*, *apply=None*)
Override.

See: *`AbstractXBeeDevice.set_parameter()`*

`set_power_level`(*power_level*)
Deprecated.

Operation not supported in this protocol. This method raises 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`*) – IO Line to be assigned.
- **`cycle`** (*`Integer`*) – Duty cycle in % to be assigned. Must be between 0 and 100.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – Read timeout in seconds.

start_listening(*src_port*)
Starts listening for incoming IP transmissions in the provided port.

Parameters **src_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 Bluetooth password of this XBee with the new one provided.

Note that your device must include Bluetooth Low Energy support.

Parameters **new_password** (*String*) – New Bluetooth password.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

update_device_data_from(*device*)
Updates the current node information with provided data. This is only for internal use.

Parameters **device** (*AbstractXBeeDevice*) – XBee to get the data from.

Returns *True* if the node data has been updated, *False* otherwise.

Return type Boolean

update_firmware(*xml_firmware_file*, *xbee_firmware_file=None*, *bootloader_firmware_file=None*, *timeout=None*, *progress_callback=None*)

Performs a firmware update operation of the XBee.

Parameters

- **xml_firmware_file** (*String*) – Path of the XML file that describes the firmware to upload.
- **xbee_firmware_file** (*String, optional, default=`None`*) – Location of the XBee binary firmware file.
- **bootloader_firmware_file** (*String, optional, default=`None`*) – Location of the bootloader binary firmware file.
- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the update process (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

Raises

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `OperationNotSupportedException` – If XBee does not support firmware update.
- `FirmwareUpdateException` – If there is any error during the firmware update.

write_changes()

Writes configurable parameter values to the non-volatile memory of the XBee so that parameter modifications persist through subsequent resets.

Parameters values remain in the device's memory until overwritten by subsequent use of this method.

If changes are made without writing them, the XBee 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_changes_enabled()* to get its status and *enable_apply_changes()* to enable/disable the option. Method *apply_changes()* can be used in order to manually apply the changes.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

class digi.xbee.devices.**WiFiDevice**(*port=None*, *baud_rate=None*, *data_bits=<sphinx.ext.autodoc.importer._MockObject object>*, *stop_bits=<sphinx.ext.autodoc.importer._MockObject object>*, *parity=<sphinx.ext.autodoc.importer._MockObject object>*, *flow_control=<FlowControl.NONE: None>*, *_sync_ops_timeout=4*, *comm_iface=None*)

Bases: `digi.xbee.devices.IPDevice`

This class represents a local Wi-Fi XBee.

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

Parameters

- **port** (`String`) – Serial port identifier. Depends on operating system. e.g. '/dev/ttyUSB0' on 'GNU/Linux' or 'COM3' on Windows.
- **baud_rate** (`Integer`) – Serial port baud rate.
- **(Integer, default** (`_sync_ops_timeout`) – `serial.EIGHTBITS`): Port bitsize.
- **(Integer, default** – `serial.STOPBITS_ONE`): Port stop bits.
- **(Character, default** (`parity`) – `serial.PARITY_NONE`): Port parity.
- **(Integer, default** – `FlowControl.NONE`): Port flow control.
- **(Integer, default** – 3): Read timeout (in seconds).
- **comm_iface** (`XBeeCommunicationInterface`) – Communication interface.

Raises All exceptions raised by `XBeeDevice.__init__()` constructor.

See also:

`IPDevice`
`v.__init__()`

open(*force_settings=False*)
Override.

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

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*) – SSID of the access point to get.

Returns

Discovered access point with the provided SID, 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 List of *AccessPoint* objects discovered.

Return type List

Raises

- `TimeoutException` – If there is a timeout scanning the access points.
- `XBeeException` – If there is any other XBee related exception.

See also:

AccessPoint

connect_by_ap (*access_point*, *password=None*)

Connects to the provided access point.

This method blocks until the connection with the access point is established or the configured access point timeout expires.

The access point timeout is configured using the *WiFiDevice.set_access_point_timeout()* method and can be consulted with *WiFiDevice.get_access_point_timeout()* method.

Once the module is connected to the access point, you can issue the *WiFiDevice.write_changes()* method to save the connection settings. This way the module will try to connect to the access point every time it is powered on.

Parameters

- **access_point** (*AccessPoint*) – The access point to connect to.
- **password** (*String, optional*) – The password for the access point, *None* if it does not have any encryption enabled.

Returns

***True* if the module connected to the access point** successfully, *False* otherwise.

Return type Boolean

Raises

- `ValueError` – If *access_point* is *None*.
- `TimeoutException` – If there is a timeout sending the connect commands.
- `XBeeException` – If there is any other XBee related exception.

See also:

WiFiDevice.connect_by_ssid()
WiFiDevice.disconnect()
WiFiDevice.get_access_point()
WiFiDevice.get_access_point_timeout()
WiFiDevice.scan_access_points()
WiFiDevice.set_access_point_timeout()

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*) – 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.

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()

disconnect()

Disconnects from the access point that the device is connected to.

This method blocks until the device disconnects totally from the access point or the configured access point timeout expires.

The access point timeout is configured using the *WiFiDevice.set_access_point_timeout()* method and can be consulted with *WiFiDevice.get_access_point_timeout()* method.

Returns

***True* if the module disconnected from the access point** successfully, *False* otherwise.

Return type Boolean

Raises

- `TimeoutException` – If there is a timeout sending the disconnect command.
- `XBeeException` – If there is any other XBee related exception.

See also:

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:

`WiFiDevice.get_wifi_ai_status()`
`WiFiAssociationIndicationStatus`

`get_access_point_timeout`()
Returns the configured access point timeout for connecting, disconnecting and scanning access points.

Returns The current access point timeout in milliseconds.

Return type Integer

See also:

`WiFiDevice.set_access_point_timeout()`

`set_access_point_timeout`(*ap_timeout*)
Configures the access point timeout in milliseconds for connecting, disconnecting and scanning access points.

Parameters **`ap_timeout`** (*`Integer`*) – The new access point timeout in milliseconds.

Raises `ValueError` – If *ap_timeout* is less than 0.

See also:

`WiFiDevice.get_access_point_timeout()`

`get_ip_addressing_mode`()
Returns the IP addressing mode of the device.

Returns The IP addressing mode.

Return type *`IPAddressingMode`*

Raises `TimeoutException` – If there is a timeout reading the IP addressing mode.

See also:

`WiFiDevice.set_ip_addressing_mode()`
`IPAddressingMode`

`set_ip_addressing_mode`(*mode*)
Sets the IP addressing mode of the device.

Parameters **`mode`** (*`IPAddressingMode`*) – The new IP addressing mode to set.

Raises `TimeoutException` – If there is a timeout setting the IP addressing mode.

See also:

`WiFiDevice.get_ip_addressing_mode()`
`IPAddressingMode`

`set_ip_address`(*ip_address*)

Sets the IP address of the module.

This method can only be called if the module is configured in `IPAddressingMode.STATIC` mode. Otherwise an *XBeeException* will be thrown.

Parameters **`ip_address`** (`ipaddress.IPv4Address`) – 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`) – 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*)
Adds a callback for the event *BluetoothDataReceived*.

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

- The Bluetooth data as a Bytearray.

add_data_received_callback(*callback*)
Deprecated.

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

add_expl_data_received_callback(*callback*)
Deprecated.

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

add_fs_frame_received_callback(*callback*)
Adds a callback for the event *FileSystemFrameReceived*.

Parameters **callback** (*Function*) – The callback. Receives four arguments.

- Source (*AbstractXBeeDevice*): The node that sent the file system frame.
- Frame id (Integer): The received frame id.
- Command (*FSCmd*): The file system command.
- Receive options (Integer): Bitfield indicating receive options.

See also:

AbstractXBeeDevice
FSCmd
ReceiveOptions

add_io_sample_received_callback(*callback*)
Adds a callback for the event *IOSampleReceived*.

Parameters **callback** (*Function*) – The callback. Receives three arguments.

- The received IO sample as an *IOSample*.
- The remote XBee which sent the packet as a *RemoteXBeeDevice*.
- The time in which the packet was received as an Integer.

add_ip_data_received_callback(*callback*)
Adds a callback for the event *IPDataReceived*.

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

- The data received as an *IPMessage*

add_micropython_data_received_callback(*callback*)
Adds a callback for the event *MicroPythonDataReceived*.

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

- The MicroPython data as a Bytearray.

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

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

- The modem status as a *ModemStatus*.

add_packet_received_callback(*callback*)
Adds a callback for the event *PacketReceived*.

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

- The received packet as a *XBeeAPIPacket*.

add_route_received_callback(*callback*)
Adds a callback for the event *RouteReceived*. This works for Zigbee and Digimesh devices.

Parameters **callback** (*Function*) – The callback. Receives three arguments.

- source (*XBeeDevice*): The source node.
- destination (*RemoteXBeeDevice*): The destination node.
- **hops (List): List of intermediate hops from closest to source** to closest to destination (*RemoteXBeeDevice*).

See also:

XBeeDevice.del_route_received_callback()

add_socket_data_received_callback(*callback*)
Adds a callback for the event *SocketDataReceived*.

Parameters **callback** (*Function*) – The callback. Receives two arguments.

- The socket ID as an Integer.
- The data received as Bytearray.

add_socket_data_received_from_callback(*callback*)
Adds a callback for the event *SocketDataReceivedFrom*.

Parameters **callback** (*Function*) – The callback. Receives three arguments.

- The socket ID as an Integer.
- **Source address pair (host, port) where host is a string** representing an IPv4 address like ‘100.50.200.5’, and port is an integer.
- The data received as Bytearray.

add_socket_state_received_callback(*callback*)
Adds a callback for the event *SocketStateReceived*.

Parameters **callback** (*Function*) – The callback. Receives two arguments.

- The socket ID as an Integer.
- The state received as a *SocketState*.

add_user_data_relay_received_callback(*callback*)
Adds a callback for the event *RelayDataReceived*.

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

- The relay data as a *UserDataRelayMessage*.

apply_changes()
Applies changes via 'AC' command.

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

apply_profile(*profile_path*, *timeout=None*, *progress_callback=None*)
Applies the given XBee profile to the XBee.

Parameters

- **profile_path** (*String*) – Path of the XBee profile file to apply.
- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the apply profile (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to receive progress information. Receives two arguments:
  - The current apply profile task as a String
  - The current apply profile task percentage as an Integer

Raises

- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- UpdateProfileException – If there is any error applying the XBee profile.

close()
Closes the communication with the XBee.

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

comm_iface
Returns the hardware interface associated to the XBee.

Returns Hardware interface of the XBee.

Return type *XBeeCommunicationInterface*

See also:

XBeeCommunicationInterface

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 XBee object created.

Return type *XBeeDevice*

Raises `SerialException` – If the port to open does not exist or is already opened.

See also:

XBeeDevice

del_bluetooth_data_received_callback (*callback*)
Deletes a callback for the callback list of *BluetoothDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_data_received_callback (*callback*)
Deprecated.

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

del_expl_data_received_callback (*callback*)
Deprecated.

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

del_fs_frame_received_callback (*callback*)
Deletes a callback for the callback list of *FileSystemFrameReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_io_sample_received_callback (*callback*)
Deletes a callback for the callback list of *IOSampleReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_ip_data_received_callback (*callback*)
Deletes a callback for the callback list of *IPDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_micropython_data_received_callback (*callback*)
Deletes a callback for the callback list of *MicroPythonDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_modem_status_received_callback (*callback*)
Deletes a callback for the callback list of *ModemStatusReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_packet_received_callback(*callback*)
Deletes a callback for the callback list of *PacketReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_route_received_callback(*callback*)
Deletes a callback for the callback list of *RouteReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

See also:

XBeeDevice.add_route_received_callback()

del_socket_data_received_callback(*callback*)
Deletes a callback for the callback list of *SocketDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_socket_data_received_from_callback(*callback*)
Deletes a callback for the callback list of *SocketDataReceivedFrom* event.

Parameters **callback** (*Function*) – The callback to delete.

del_socket_state_received_callback(*callback*)
Deletes a callback for the callback list of *SocketStateReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

del_user_data_relay_received_callback(*callback*)
Deletes a callback for the callback list of *RelayDataReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

determine_protocol(*hardware_version*, *firmware_version*)
Determines the XBee protocol based on the given hardware and firmware versions.

Parameters

- **hardware_version** (*Integer*) – Hardware version to get its protocol.
- **firmware_version** (*Bytearray*) – Firmware version to get its protocol.

Returns

XBee protocol corresponding to the given hardware and firmware versions.

Return type *XBeeProtocol*

disable_bluetooth()
Disables the Bluetooth interface of this XBee.

Note that your device must include Bluetooth Low Energy support.

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

enable_apply_changes(*value*)

Sets apply changes flag.

Parameters **value** (*Boolean*) – *True* to enable apply changes flag, *False* to disable it.

enable_bluetooth()

Enables the Bluetooth interface of this XBee.

To work with this interface, you must also configure the Bluetooth password if not done previously. Use method *AbstractXBeeDevice.update_bluetooth_password()*.

Note that your XBee must include Bluetooth Low Energy support.

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

execute_command(*parameter*, *value=None*, *apply=None*)

Executes the provided command.

Parameters

- **(String or** (*parameter*) – class: *.ATStringCommand*): AT command to execute.
- **value** (*bytearray, optional, default=`None`*) – Command value (if any).
- **apply** (*Boolean, optional, default=`None`*) – *True* to apply changes in XBee configuration, *False* not to apply them, *None* to use *is_apply_changes_enabled()* returned value.

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

See also:

AbstractXBeeDevice.get_parameter()
AbstractXBeeDevice.set_parameter()
AbstractXBeeDevice.apply_changes()
AbstractXBeeDevice.write_changes()
AbstractXBeeDevice.is_apply_changes_enabled()
AbstractXBeeDevice.enable_apply_changes()

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.

Returns 64-bit address of the XBee.

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`) – IO line to get its ADC value.

Returns Analog value corresponding to the provided IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If response does not contain the value for the given IO line.

See also:

`IOLine`
`set_io_configuration()`

get_api_output_mode()
Deprecated since version 1.3: Use `get_api_output_mode_value()`

Returns the API output mode of the XBee.

The API output mode determines the format of the data through the serial interface of the XBee.

Returns API output mode of the XBee.

Return type `APIOutputMode`

Raises

- `TimeoutException` – If response is not received before the read timeout expires.

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

APIOutputMode

get_api_output_mode_value()

Returns the API output mode of the XBee.

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

Returns the parameter value.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

digi.xbee.models.mode.APIOutputModeBit

get_bluetooth_mac_addr()

Reads and returns the EUI-48 Bluetooth MAC address of this XBee following the format *00112233AABB*.

Note that your device must include Bluetooth Low Energy support.

Returns The Bluetooth MAC address.

Return type String

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

get_current_frame_id()

Returns the last used frame ID.

Returns 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 raises an `AttributeError`.

get_dest_ip_addr()
Returns the destination IP address.

Returns Configured destination IP address.

Return type `ipaddress.IPv4Address`

Raises

- `TimeoutException` – If there is a timeout getting the destination IP address.
- `XBeeException` – If there is any other XBee related exception.

See also:

`ipaddress.IPv4Address`

get_dio_value(*io_line*)
Returns the digital value of the provided IO line.

The provided IO line must be previously configured as digital I/O. To do so, use *AbstractXBeeDevice.set_io_configuration()*.

Parameters **io_line** (*IOLine*) – the DIO line to gets its digital value.

Returns current value of the provided IO line.

Return type *IOValue*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If response does not contain the value for the given IO line.

See also:

IOLine
IOValue
set_io_configuration()

get_file_manager()

Returns the file system manager for the XBee.

Returns The file system manager.

Return type *FileSystemManager*

Raises `FileSystemNotSupportedException` – If the XBee does not support filesystem.

get_firmware_version()

Returns the firmware version of the XBee.

Returns Firmware version of the XBee.

Return type Bytearray

get_hardware_version()

Returns the hardware version of the XBee.

Returns Hardware version of the XBee.

Return type *HardwareVersion*

See also:

HardwareVersion

get_io_configuration(*io_line*)

Returns the configuration of the provided IO line.

Parameters **io_line** (*IOLine*) – IO line to get its configuration.

Returns IO mode of the IO line provided.

Return type *IOMode*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOLine
IOMode
set_io_configuration()

get_io_sampling_rate()

Returns the IO sampling rate of the XBee.

Returns IO sampling rate of XBee.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`set_io_sampling_rate()`

`get_ip_addr`()

Returns the IP address of this IP XBee.

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.

Returns The next frame ID of the XBee.

Return type Integer

`get_node_id`()

Returns the node identifier ('NI') value of the XBee.

Returns Node identifier ('NI') of the XBee.

Return type String

`get_pan_id`()

Deprecated.

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

`get_parameter`(*parameter*, *parameter_value=None*, *apply=None*)

Override.

See also:

`AbstractXBeeDevice.get_parameter()`

get_power_level()
Returns the power level of the XBee.

Returns Power level of the XBee.

Return type *PowerLevel*

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

See also:

PowerLevel
set_power_level()

get_pwm_duty_cycle(*io_line*)
Returns the PWM duty cycle in % corresponding to the provided IO line.

Parameters **io_line** (*IOLine*) – IO line to get its PWM duty cycle.

Returns PWM duty cycle of the given IO line.

Return type Integer

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.
- ValueError – If *io_line* has no PWM capability.

See also:

IOLine

get_role()
Gets the XBee role.

Returns the role of the XBee.

Return type *Role*

See also:

Role

get_route_to_node(*remote*, *timeout=10*, *force=True*)

Gets the route from this XBee to the given remote node.

For Zigbee:

- 'AR' parameter of the local node must be configured with a value different from 'FF'.
- Set *force* to *True* to force the Zigbee remote node to return its route independently of the local node configuration as high or low RAM concentrator ('DO' of the local value)

Parameters

- **remote** (*RemoteXBeeDevice*) – The remote node.
- **timeout** (*Float, optional, default=10*) – Maximum number of seconds to wait for the route.
- **force** (*Boolean*) – *True* to force asking for the route, *False* otherwise. Only for Zigbee.

Returns

Tuple containing route data:

- status (*TransmitStatus*): The transmit status.
- Tuple with route data (*None* if the route was not read in the provided timeout):
  - source (*RemoteXBeeDevice*): The source node of the route.
  - destination (*RemoteXBeeDevice*): The destination node of the route.
  - hops (List): List of intermediate nodes (*RemoteXBeeDevice*) ordered from closest to source to closest to destination node (source and destination not included).

Return type Tuple

get_sync_ops_timeout()

Returns the serial port read timeout.

Returns 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 are executed before user callbacks.

Returns *PacketReceived*

has_explicit_packets()

Returns if there are pending explicit packets to read. This does not include non-explicit packets.

Returns *True* if there are pending packets, *False* otherwise.

Return type Boolean

See also:

XBeeDevice.has_packets()

has_packets()

Returns if there are pending packets to read. This does not include explicit packets.

Returns *True* if there are pending packets, *False* otherwise.

Return type Boolean

See also:

XBeeDevice.has_explicit_packets()

is_apply_changes_enabled()

Returns whether apply changes flag is enabled.

Returns *True* if apply changes flag is enabled, *False* otherwise.

Return type Boolean

is_device_info_complete()

Override.

See also:

AbstractXBeeDevice.is_device_info_complete()

is_open()

Returns whether this XBee is open.

Returns Boolean. *True* if this XBee is open, *False* otherwise.

is_remote()

Override method.

See also:

AbstractXBeeDevice.is_remote()

log

Returns the XBee logger.

Returns The XBee device logger.

Return type `Logger`

operating_mode

Returns the operating mode of this XBee.

Returns *OperatingMode*. This XBee operating mode.

reachable

Returns whether the XBee is reachable.

Returns *True* if the device is reachable, *False* otherwise.

Return type Boolean

read_data(*timeout=None*)
Deprecated.

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

read_data_from(*remote_xbee*, *timeout=None*)
Deprecated.

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

read_device_info(*init=True*, *fire_event=True*)
Updates all instance parameters reading them from the XBee.

Parameters

- **init** (*Boolean, optional, default=`True`*) – If *False* only not initialized parameters are read, all if *True*.
- **fire_event** (*Boolean, optional, default=`True`*) – *True* to throw and update event if any parameter changed, *False* otherwise.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

AbstractXBeeDevice.is_device_info_complete()

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

Returns IO sample read from the XBee.

Return type *IOSample*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOSample

read_ip_data(*timeout=3*)
Reads new IP data received by this XBee during the provided timeout.

This method blocks until new IP data is received or the provided timeout expires.

For non-blocking operations, register a callback and use the method *IPDevice.add_ip_data_received_callback()*.

Before reading IP data you need to start listening for incoming IP data at a specific port. Use the method *IPDevice.start_listening()* for that purpose. When finished, you can use the method *IPDevice.stop_listening()* to stop listening for incoming IP data.

Parameters **timeout** (*Integer, optional*) – The time to wait for new IP data in seconds.

Returns IP message, *None* if this device did not receive new data.

Return type *IPMessage*

Raises ValueError – If *timeout* is less than 0.

read_ip_data_from(*ip_addr*, *timeout=3*)
Reads new IP data received from the given IP address during the provided timeout.

This method blocks until new IP data from the provided IP address is received or the given timeout expires.

For non-blocking operations, register a callback and use the method *IPDevice.add_ip_data_received_callback()*.

Before reading IP data you need to start listening for incoming IP data at a specific port. Use the method *IPDevice.start_listening()* for that purpose. When finished, you can use the method *IPDevice.stop_listening()* to stop listening for incoming IP data.

Parameters

- **ip_addr** (ipaddress.IPv4Address) – The IP address to read data from.
- **timeout** (*Integer, optional*) – The time to wait for new IP data in seconds.

Returns

IP message, *None* if this device did not receive new data from the provided IP address.

Return type *IPMessage*

Raises ValueError – If *timeout* is less than 0.

reset()
Override method.

See also:

AbstractXBeeDevice.reset()

scan_counter
Returns the scan counter for this node.

Returns The scan counter for this node.

Return type Integer

send_bluetooth_data(*data*)
Sends the given data to the Bluetooth interface using a User Data Relay frame.

Parameters **`data`** (*`Bytearray`*) – Data to send.

Raises

- `InvalidOperatingModeException` – If the XBee'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*, *data*, *transmit_options=0*)
Deprecated.

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

`send_data_async`(*remote_xbee*, *data*, *transmit_options=0*)
Deprecated.

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

`send_data_broadcast`(*data*, *transmit_options=0*)
Deprecated.

Operation not supported in this protocol. This method raises 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 until 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, default=`False``*) – *True* to close the socket just after the transmission. *False* to keep it open.

Raises

- `ValueError` – If *ip_addr* or *protocol* or *data* is *None* or *dest_port* is less than 0 or greater than 65535.
- `OperationNotSupportedException` – If the XBee 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, default=`False``*) – *True* to close the socket just after the transmission. *False* to keep it open.

Raises

- `ValueError` – If *ip_addr* or *protocol* or *data* is *None* or *dest_port* is less than 0 or greater than 65535.
- `OperationNotSupportedException` – If the XBee 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 until 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* or *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'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*)

Sends the packet and waits for the response. The packet to send is escaped depending on the current operating mode.

This method can be synchronous or asynchronous.

If synchronous, this method discards all response packets until it finds the one that has the appropriate frame ID, that is, the sent packet's frame ID.

If asynchronous, this method does not wait for any response and returns *None*.

Parameters

- **`packet`** (*`XBeePacket`*) – The packet to send.
- **`sync`** (*`Boolean`*) – *True* to wait for the response of the sent packet and return it, *False* otherwise.

Returns

Response packet if *sync* is *True*, *None* otherwise.

Return type *`XBeePacket`*

Raises

- `TimeoutException` – If *sync* is *True* and the response packet for the sent one cannot be read.
- `InvalidOperatingModeException` – If the XBee operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `XBeeException` – If the packet listener is not running or the XBee's communication interface is closed.

See also:

`XBeePacket`

`send_packet_sync_and_get_response`(*packet_to_send*, *timeout=None*)

Sends the packet and waits for its corresponding response.

Parameters

- **`packet_to_send`** (*`XBeePacket`*) – The packet to transmit.
- **`timeout`** (*`Integer, optional, default=`None``*) – Number of seconds to wait. -1 to wait indefinitely.

Returns Received response packet.

Return type *`XBeePacket`*

Raises

- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TimeoutException` – If response is not received in the configured timeout.
- `XBeeException` – If the XBee's communication interface is closed.

See also:

`XBeePacket`

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'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, if any.

Returns

Serial port of the XBee. *None* if the local XBee does not use serial communication.

Return type *XBeeSerialPort*

See also:

XBeeSerialPort

set_16bit_addr(*value*)

Sets the 16-bit address of the XBee.

Parameters value (*XBee16BitAddress*) – New 16-bit address of the XBee.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If the protocol is not 802.15.4.

set_api_output_mode(*api_output_mode*)

Deprecated since version 1.3: Use *set_api_output_mode_value()*

Sets the API output mode of the XBee.

Parameters api_output_mode (*APIOutputMode*) – New API output mode.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

`APIOutputMode`

`set_api_output_mode_value`(*api_output_mode*)
Sets the API output mode of the XBee.

Parameters **`api_output_mode`** (*`Integer`*) – New API output mode options. Calculate this value using the method `APIOutputModeBit.calculate_api_output_mode_value()` with a set of *`APIOutputModeBit`*.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

`APIOutputModeBit`

`set_dest_address`(*addr*)
Deprecated.

Operation not supported in this protocol. Use *`IPDevice.set_dest_ip_addr()`* instead. This method raises 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 response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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`*) – Digital IO line to sets its value.
- **`io_value`** (*`IOValue`*) – IO value to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – IO line to configure.
- **io_mode** (*IOMode*) – IO mode to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOLine
IOMode
get_io_configuration()

set_io_sampling_rate(*rate*)
Sets the IO sampling rate of the XBee in seconds. A sample rate of 0 means the IO sampling feature is disabled.

Parameters **rate** (*Integer*) – New IO sampling rate of the XBee in seconds.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

get_io_sampling_rate()

set_node_id(*node_id*)
Sets the node identifier ('NI') value of the XBee.

Parameters **node_id** (*String*) – New node identifier ('NI') of the XBee.

Raises

- `ValueError` – If *node_id* is *None* or its length is greater than 20.
- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

set_pan_id(*value*)
Deprecated.

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

set_parameter(*parameter*, *value*, *apply=None*)
Override.

See: *AbstractXBeeDevice.set_parameter()*

set_power_level(*power_level*)
Sets the power level of the XBee.

Parameters **power_level** (*PowerLevel*) – New power level of the XBee.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

PowerLevel
get_power_level()

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*) – IO Line to be assigned.
- **cycle** (*Integer*) – Duty cycle in % to be assigned. Must be between 0 and 100.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – Read timeout in seconds.

start_listening(*src_port*)
Starts listening for incoming IP transmissions in the provided port.

Parameters **src_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 Bluetooth password of this XBee with the new one provided.

Note that your device must include Bluetooth Low Energy support.

Parameters **new_password** (*String*) – New Bluetooth password.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

update_device_data_from(*device*)
Updates the current node information with provided data. This is only for internal use.

Parameters **device** (*AbstractXBeeDevice*) – XBee to get the data from.

Returns *True* if the node data has been updated, *False* otherwise.

Return type Boolean

update_firmware(*xml_firmware_file*, *xbee_firmware_file=None*, *bootloader_firmware_file=None*, *timeout=None*, *progress_callback=None*)
Performs a firmware update operation of the XBee.

Parameters

- **xml_firmware_file** (*String*) – Path of the XML file that describes the firmware to upload.
- **xbee_firmware_file** (*String, optional, default=`None`*) – Location of the XBee binary firmware file.
- **bootloader_firmware_file** (*String, optional, default=`None`*) – Location of the bootloader binary firmware file.

- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the update process (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

Raises

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `OperationNotSupportedException` – If XBee does not support firmware update.
- `FirmwareUpdateException` – If there is any error during the firmware update.

write_changes()

Writes configurable parameter values to the non-volatile memory of the XBee so that parameter modifications persist through subsequent resets.

Parameters values remain in the device's memory until overwritten by subsequent use of this method.

If changes are made without writing them, the XBee 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_changes_enabled()`* to get its status and *`enable_apply_changes()`* to enable/disable the option. Method *`apply_changes()`* can be used in order to manually apply the changes.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

class digi.xbee.devices.**RemoteXBeeDevice**(*local_xbee, x64bit_addr=<digi.xbee.models.address.XBee64BitAddress object>, x16bit_addr=<digi.xbee.models.address.XBee16BitAddress object>, node_id=None*)

Bases: *`digi.xbee.devices.AbstractXBeeDevice`*

This class represents a remote XBee.

Class constructor. Instantiates a new *`RemoteXBeeDevice`* with the provided parameters.

Parameters

- **local_xbee** (*`XBeeDevice`*) – Local XBee associated with the remote one.
- **x64bit_addr** (*`XBee64BitAddress`*) – 64-bit address of the remote XBee.
- **x16bit_addr** (*`XBee16BitAddress`*) – 16-bit address of the remote XBee.
- **node_id** (*String, optional*) – Node identifier of the remote XBee.

See also:

`XBee16BitAddress`
`XBee64BitAddress`
`XBeeDevice`

`get_parameter`(*parameter*, *parameter_value=None*, *apply=None*)
Override.

See also:

`AbstractXBeeDevice.get_parameter()`

`set_parameter`(*parameter*, *value*, *apply=None*)
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 associated to the remote one.

Returns Local XBee.

Return type `XBeeDevice`

`set_local_xbee_device`(*local_xbee_device*)
This methods associates a `XBeeDevice` to the remote XBee.

Parameters **`local_xbee_device`** (`XBeeDevice`) – New local XBee associated to the remote one.

See also:

`XBeeDevice`

get_serial_port()

Returns the serial port of the local XBee associated to the remote one.

Returns

Serial port of the local XBee associated to the remote one.

Return type `XBeeSerialPort`

See also:

`XBeeSerialPort`

get_comm_iface()

Returns the communication interface of the local XBee associated to the remote one.

Returns

Communication interface of the local XBee associated to the remote one.

Return type `XBeeCommunicationInterface`

See also:

`XBeeCommunicationInterface`

get_ota_max_block_size()

Returns the maximum number of bytes to send for ota updates.

Returns Maximum ota block size to send.

Return type Integer

set_ota_max_block_size(*size*)

Sets the maximum number of bytes to send for ota updates.

Parameters **size** (*Integer*) – Maximum ota block size to send.

Raises `ValueError` – If size is not between 0 and 255.

update_filesystem_image(*ota_filesystem_file*, *timeout=None*, *progress_callback=None*)

Performs a filesystem image update operation of the device.

Parameters

- **ota_filesystem_file** (*String*) – Location of the OTA filesystem image file.
- **timeout** (*Integer, optional*) – Maximum time to wait for target read operations during the update process.
- **progress_callback** (*Function, optional*) – Function to receive progress information. Receives two arguments:
  - The current update task as a String.
  - The current update task percentage as an Integer.

Raises

- `XBeeException` – If the device is not open.

- `InvalidOperatingModeException` – If the device operating mode is invalid.
- `FileSystemNotSupportedException` – If the filesystem update is not supported in the XBee.
- `FileSystemException` – If there is any error performing the filesystem update.

`apply_changes`()

Applies changes via 'AC' command.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

`apply_profile`(*profile_path*, *timeout=None*, *progress_callback=None*)

Applies the given XBee profile to the XBee.

Parameters

- **profile_path** (*String*) – Path of the XBee profile file to apply.
- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the apply profile (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to receive progress information. Receives two arguments:
  - The current apply profile task as a String
  - The current apply profile task percentage as an Integer

Raises

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `UpdateProfileException` – If there is any error applying the XBee profile.

`determine_protocol`(*hardware_version*, *firmware_version*)

Determines the XBee protocol based on the given hardware and firmware versions.

Parameters

- **hardware_version** (*Integer*) – Hardware version to get its protocol.
- **firmware_version** (*Bytearray*) – Firmware version to get its protocol.

Returns

XBee protocol corresponding to the given hardware and firmware versions.

Return type *XBeeProtocol*

`disable_bluetooth`()

Disables the Bluetooth interface of this XBee.

Note that your device must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

enable_apply_changes(*value*)
Sets apply changes flag.

Parameters **value** (*Boolean*) – *True* to enable apply changes flag, *False* to disable it.

enable_bluetooth()
Enables the Bluetooth interface of this XBee.

To work with this interface, you must also configure the Bluetooth password if not done previously. Use method *AbstractXBeeDevice.update_bluetooth_password()*.

Note that your XBee must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

execute_command(*parameter*, *value=None*, *apply=None*)
Executes the provided command.

Parameters

- **(String or** (*parameter*) – class: *.ATStringCommand*): AT command to execute.
- **value** (*bytearray, optional, default=`None`*) – Command value (if any).
- **apply** (*Boolean, optional, default=`None`*) – *True* to apply changes in XBee configuration, *False* not to apply them, *None* to use *is_apply_changes_enabled()* returned value.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

AbstractXBeeDevice.get_parameter()
AbstractXBeeDevice.set_parameter()
AbstractXBeeDevice.apply_changes()

AbstractXBeeDevice.write_changes()
AbstractXBeeDevice.is_apply_changes_enabled()
AbstractXBeeDevice.enable_apply_changes()

get_16bit_addr()

Returns the 16-bit address of the XBee.

Returns 16-bit address of the XBee.

Return type *XBee16BitAddress*

See also:

XBee16BitAddress

get_64bit_addr()

Returns the 64-bit address of the XBee.

Returns 64-bit address of the XBee.

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*) – IO line to get its ADC value.

Returns Analog value corresponding to the provided IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If response does not contain the value for the given IO line.

See also:

IOLine
set_io_configuration()

get_api_output_mode()

Deprecated since version 1.3: Use *get_api_output_mode_value()*

Returns the API output mode of the XBee.

The API output mode determines the format of the data through the serial interface of the XBee.

Returns API output mode of the XBee.

Return type *APIOutputMode*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

APIOutputMode

get_api_output_mode_value()

Returns the API output mode of the XBee.

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

Returns the parameter value.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

digi.xbee.models.mode.APIOutputModeBit

get_bluetooth_mac_addr()

Reads and returns the EUI-48 Bluetooth MAC address of this XBee following the format *00112233AABB*.

Note that your device must include Bluetooth Low Energy support.

Returns The Bluetooth MAC address.

Return type String

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

get_current_frame_id()

Returns the last used frame ID.

Returns Last used frame ID.

Return type Integer

get_dest_address()

Returns the 64-bit address of the XBee that is data destination.

Returns 64-bit address of destination XBee.

Return type *XBee64BitAddress*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

XBee64BitAddress
set_dest_address()

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 response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

- `OperationNotSupportedException` – If response does not contain the value for the given IO line.

See also:

IOLine
IOValue
set_io_configuration()

get_file_manager()
Returns the file system manager for the XBee.

Returns The file system manager.

Return type *FileSystemManager*

Raises `FileSystemNotSupportedException` – If the XBee does not support filesystem.

get_firmware_version()
Returns the firmware version of the XBee.

Returns Firmware version of the XBee.

Return type Bytearray

get_hardware_version()
Returns the hardware version of the XBee.

Returns Hardware version of the XBee.

Return type *HardwareVersion*

See also:

HardwareVersion

get_io_configuration(*io_line*)
Returns the configuration of the provided IO line.

Parameters **io_line** (*IOLine*) – IO line to get its configuration.

Returns IO mode of the IO line provided.

Return type *IOMode*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOLine
IOMode
set_io_configuration()

get_io_sampling_rate()
Returns the IO sampling rate of the XBee.

Returns IO sampling rate of XBee.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

set_io_sampling_rate()

get_node_id()
Returns the node identifier ('NI') value of the XBee.

Returns Node identifier ('NI') of the XBee.

Return type String

get_pan_id()
Returns the operating PAN ID of the XBee.

Returns Operating PAN ID of the XBee.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

set_pan_id()

get_power_level()
Returns the power level of the XBee.

Returns Power level of the XBee.

Return type *PowerLevel*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

PowerLevel
set_power_level()

get_protocol()

Returns the current protocol of the XBee.

Returns Current protocol of the XBee.

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*) – IO line to get its PWM duty cycle.

Returns PWM duty cycle of the given IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `ValueError` – If *io_line* has no PWM capability.

See also:

IOLine

get_role()

Gets the XBee role.

Returns the role of the XBee.

Return type `Role`

See also:

`Role`

`get_sync_ops_timeout()`
Returns the serial port read timeout.

Returns Serial port read timeout in seconds.

Return type Integer

`is_apply_changes_enabled()`
Returns whether apply changes flag is enabled.

Returns *True* if apply changes flag is enabled, *False* otherwise.

Return type Boolean

`is_device_info_complete()`
Returns whether XBee node information is complete.

Returns *True* if node information is complete, *False* otherwise.

Return type Boolean

See also:

`AbstractXBeeDevice.read_device_info()`

`log`
Returns the XBee logger.

Returns The XBee device logger.

Return type `Logger`

`reachable`
Returns whether the XBee is reachable.

Returns *True* if the device is reachable, *False* otherwise.

Return type Boolean

`read_device_info`(*init=True*, *fire_event=True*)
Updates all instance parameters reading them from the XBee.

Parameters

- **init** (*Boolean, optional, default=`True`*) – If *False* only not initialized parameters are read, all if *True*.
- **fire_event** (*Boolean, optional, default=`True`*) – *True* to throw and update event if any parameter changed, *False* otherwise.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`AbstractXBeeDevice.is_device_info_complete()`

`read_io_sample`()

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

Returns IO sample read from the XBee.

Return type *`IOSample`*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`IOSample`

`scan_counter`

Returns the scan counter for this node.

Returns The scan counter for this node.

Return type Integer

`set_16bit_addr`(*value*)

Sets the 16-bit address of the XBee.

Parameters **`value`** (*`XBee16BitAddress`*) – New 16-bit address of the XBee.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If the protocol is not 802.15.4.

set_api_output_mode(*api_output_mode*)

Deprecated since version 1.3: Use `set_api_output_mode_value()`

Sets the API output mode of the XBee.

Parameters **api_output_mode** (`APIOutputMode`) – New API output mode.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

`APIOutputMode`

set_api_output_mode_value(*api_output_mode*)

Sets the API output mode of the XBee.

Parameters **api_output_mode** (`Integer`) – New API output mode options. Calculate this value using the method `APIOutputModeBit.calculate_api_output_mode_value()` with a set of `APIOutputModeBit`.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

`APIOutputModeBit`

set_dest_address(*addr*)

Sets the 64-bit address of the XBee that is data destination.

Parameters **addr** (`XBee64BitAddress` or `RemoteXBeeDevice`) – Address itself or remote XBee to be data destination.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `ValueError` – If *addr* is *None*.

See also:

XBee64BitAddress
get_dest_address()

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 response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – Digital IO line to sets its value.
- **io_value** (*IOValue*) – IO value to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – IO line to configure.
- **io_mode** (*IOMode*) – IO mode to set to the IO line.

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

See also:

IOLine
IOMode
get_io_configuration()

set_io_sampling_rate(*rate*)
Sets the IO sampling rate of the XBee in seconds. A sample rate of 0 means the IO sampling feature is disabled.

Parameters **rate** (*Integer*) – New IO sampling rate of the XBee in seconds.

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

See also:

get_io_sampling_rate()

set_node_id(*node_id*)
Sets the node identifier ('NI') value of the XBee.

Parameters **node_id** (*String*) – New node identifier ('NI') of the XBee.

Raises

- ValueError – If *node_id* is *None* or its length is greater than 20.
- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.

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

set_pan_id(*value*)

Sets the operating PAN ID of the XBee.

Parameters **value** (*Bytearray*) – New operating PAN ID of the XBee. Must have only 1 or 2 bytes.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

get_pan_id()

set_power_level(*power_level*)

Sets the power level of the XBee.

Parameters **power_level** (*PowerLevel*) – New power level of the XBee.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

PowerLevel
get_power_level()

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*) – IO Line to be assigned.
- **cycle** (*Integer*) – Duty cycle in % to be assigned. Must be between 0 and 100.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – Read timeout in seconds.

update_bluetooth_password(*new_password*)
Changes the Bluetooth password of this XBee with the new one provided.

Note that your device must include Bluetooth Low Energy support.

Parameters **new_password** (*String*) – New Bluetooth password.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

update_device_data_from(*device*)
Updates the current node information with provided data. This is only for internal use.

Parameters **device** (*AbstractXBeeDevice*) – XBee to get the data from.

Returns *True* if the node data has been updated, *False* otherwise.

Return type Boolean

update_firmware(*xml_firmware_file*, *xbee_firmware_file=None*, *bootloader_firmware_file=None*, *timeout=None*, *progress_callback=None*)
Performs a firmware update operation of the XBee.

Parameters

- **xml_firmware_file** (*String*) – Path of the XML file that describes the firmware to upload.
- **xbee_firmware_file** (*String, optional, default=`None`*) – Location of the XBee binary firmware file.
- **bootloader_firmware_file** (*String, optional, default=`None`*) – Location of the bootloader binary firmware file.

- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the update process (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

Raises

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `OperationNotSupportedException` – If XBee does not support firmware update.
- `FirmwareUpdateException` – If there is any error during the firmware update.

write_changes()

Writes configurable parameter values to the non-volatile memory of the XBee so that parameter modifications persist through subsequent resets.

Parameters values remain in the device's memory until overwritten by subsequent use of this method.

If changes are made without writing them, the XBee 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_changes_enabled()* to get its status and *enable_apply_changes()* to enable/disable the option. Method *apply_changes()* can be used in order to manually apply the changes.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

class digi.xbee.devices.**RemoteRaw802Device**(*local_xbee*, *x64bit_addr=None*, *x16bit_addr=None*, *node_id=None*)

Bases: *digi.xbee.devices.RemoteXBeeDevice*

This class represents a remote 802.15.4 XBee.

Class constructor. Instantiates a new *RemoteXBeeDevice* with the provided parameters.

Parameters

- **local_xbee** (*XBeeDevice*) – Local XBee associated with the remote one.
- **x64bit_addr** (*XBee64BitAddress*) – 64-bit address of the remote XBee.
- **x16bit_addr** (*XBee16BitAddress*) – 16-bit address of the remote XBee.
- **node_id** (*String, optional*) – Node identifier of the remote XBee.

Raises `XBeeException` – If the protocol of *local_xbee* is invalid.

See also:

RemoteXBeeDevice
XBee16BitAddress
XBee64BitAddress
XBeeDevice

get_protocol()
Override.

See also:

RemoteXBeeDevice.get_protocol()

set_64bit_addr(*address*)
Sets the 64-bit address of this remote 802.15.4 device.

Parameters **address** (*XBee64BitAddress*) – The 64-bit address to set.

Raises ValueError – If *address* is *None*.

get_ai_status()
Returns the current association status of this XBee. It indicates occurrences of errors during the modem initialization and connection.

Returns

The XBee association indication status.

Return type *AssociationIndicationStatus*

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

apply_changes()
Applies changes via 'AC' command.

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

apply_profile(*profile_path*, *timeout=None*, *progress_callback=None*)
Applies the given XBee profile to the XBee.

Parameters

- **profile_path** (*String*) – Path of the XBee profile file to apply.

- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the apply profile (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to receive progress information. Receives two arguments:
  - The current apply profile task as a String
  - The current apply profile task percentage as an Integer

Raises

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `UpdateProfileException` – If there is any error applying the XBee profile.

determine_protocol(*hardware_version*, *firmware_version*)
Determines the XBee protocol based on the given hardware and firmware versions.

Parameters

- **hardware_version** (*Integer*) – Hardware version to get its protocol.
- **firmware_version** (*Bytearray*) – Firmware version to get its protocol.

Returns

XBee protocol corresponding to the given hardware and firmware versions.

Return type *XBeeProtocol*

disable_bluetooth()
Disables the Bluetooth interface of this XBee.

Note that your device must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

enable_apply_changes(*value*)
Sets apply changes flag.

Parameters value (*Boolean*) – *True* to enable apply changes flag, *False* to disable it.

enable_bluetooth()
Enables the Bluetooth interface of this XBee.

To work with this interface, you must also configure the Bluetooth password if not done previously. Use method *AbstractXBeeDevice.update_bluetooth_password()*.

Note that your XBee must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.

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

execute_command(*parameter*, *value=None*, *apply=None*)

Executes the provided command.

Parameters

- **(String or** (*parameter*) – class: *.ATStringCommand*): AT command to execute.
- **value** (*bytearray, optional, default=`None`*) – Command value (if any).
- **apply** (*Boolean, optional, default=`None`*) – *True* to apply changes in XBee configuration, *False* not to apply them, *None* to use *is_apply_changes_enabled()* returned value.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

AbstractXBeeDevice.get_parameter()
AbstractXBeeDevice.set_parameter()
AbstractXBeeDevice.apply_changes()
AbstractXBeeDevice.write_changes()
AbstractXBeeDevice.is_apply_changes_enabled()
AbstractXBeeDevice.enable_apply_changes()

get_16bit_addr()

Returns the 16-bit address of the XBee.

Returns 16-bit address of the XBee.

Return type *XBee16BitAddress*

See also:

XBee16BitAddress

get_64bit_addr()

Returns the 64-bit address of the XBee.

Returns 64-bit address of the XBee.

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*) – IO line to get its ADC value.

Returns Analog value corresponding to the provided IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If response does not contain the value for the given IO line.

See also:

IOLine
set_io_configuration()

get_api_output_mode()

Deprecated since version 1.3: Use *get_api_output_mode_value()*

Returns the API output mode of the XBee.

The API output mode determines the format of the data through the serial interface of the XBee.

Returns API output mode of the XBee.

Return type *APIOutputMode*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

APIOutputMode

get_api_output_mode_value()

Returns the API output mode of the XBee.

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

Returns the parameter value.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

digi.xbee.models.mode.APIOutputModeBit

get_bluetooth_mac_addr()

Reads and returns the EUI-48 Bluetooth MAC address of this XBee following the format *00112233AABB*.

Note that your device must include Bluetooth Low Energy support.

Returns The Bluetooth MAC address.

Return type String

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

get_comm_iface()

Returns the communication interface of the local XBee associated to the remote one.

Returns

Communication interface of the local XBee associated to the remote one.

Return type `XBeeCommunicationInterface`

See also:

`XBeeCommunicationInterface`

get_current_frame_id()

Returns the last used frame ID.

Returns Last used frame ID.

Return type Integer

get_dest_address()

Returns the 64-bit address of the XBee that is data destination.

Returns 64-bit address of destination XBee.

Return type *XBee64BitAddress*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

XBee64BitAddress
set_dest_address()

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 response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If response does not contain the value for the given IO line.

See also:

IOLine

IOValue
set_io_configuration()

get_file_manager()
Returns the file system manager for the XBee.

Returns The file system manager.

Return type *FileSystemManager*

Raises `FileSystemNotSupportedException` – If the XBee does not support filesystem.

get_firmware_version()
Returns the firmware version of the XBee.

Returns Firmware version of the XBee.

Return type Bytearray

get_hardware_version()
Returns the hardware version of the XBee.

Returns Hardware version of the XBee.

Return type *HardwareVersion*

See also:

HardwareVersion

get_io_configuration(*io_line*)
Returns the configuration of the provided IO line.

Parameters **io_line** (*IOLine*) – IO line to get its configuration.

Returns IO mode of the IO line provided.

Return type *IOMode*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOLine
IOMode
set_io_configuration()

get_io_sampling_rate()

Returns the IO sampling rate of the XBee.

Returns IO sampling rate of XBee.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

set_io_sampling_rate()

get_local_xbee_device()

Returns the local XBee associated to the remote one.

Returns Local XBee.

Return type *XBeeDevice*

get_node_id()

Returns the node identifier ('NI') value of the XBee.

Returns Node identifier ('NI') of the XBee.

Return type String

get_ota_max_block_size()

Returns the maximum number of bytes to send for ota updates.

Returns Maximum ota block size to send.

Return type Integer

get_pan_id()

Returns the operating PAN ID of the XBee.

Returns Operating PAN ID of the XBee.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

set_pan_id()

get_parameter(*parameter*, *parameter_value=None*, *apply=None*)
Override.

See also:

AbstractXBeeDevice.get_parameter()

get_power_level()
Returns the power level of the XBee.

Returns Power level of the XBee.

Return type *PowerLevel*

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

See also:

PowerLevel
set_power_level()

get_pwm_duty_cycle(*io_line*)
Returns the PWM duty cycle in % corresponding to the provided IO line.

Parameters **io_line** (*IOLine*) – IO line to get its PWM duty cycle.

Returns PWM duty cycle of the given IO line.

Return type Integer

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.
- ValueError – If *io_line* has no PWM capability.

See also:

IOLine

get_role()
Gets the XBee role.

Returns the role of the XBee.

Return type *Role*

See also:

Role

get_serial_port()
Returns the serial port of the local XBee associated to the remote one.

Returns

Serial port of the local XBee associated to the remote one.

Return type `XBeeSerialPort`

See also:

`XBeeSerialPort`

get_sync_ops_timeout()
Returns the serial port read timeout.

Returns Serial port read timeout in seconds.

Return type Integer

is_apply_changes_enabled()
Returns whether apply changes flag is enabled.

Returns *True* if apply changes flag is enabled, *False* otherwise.

Return type Boolean

is_device_info_complete()
Returns whether XBee node information is complete.

Returns *True* if node information is complete, *False* otherwise.

Return type Boolean

See also:

AbstractXBeeDevice.read_device_info()

is_remote()
Override method.

See also:

AbstractXBeeDevice.is_remote()

log
Returns the XBee logger.

Returns The XBee device logger.

Return type `Logger`

reachable
Returns whether the XBee is reachable.

Returns *True* if the device is reachable, *False* otherwise.

Return type Boolean

read_device_info(*init=True*, *fire_event=True*)
Updates all instance parameters reading them from the XBee.

Parameters

- **init** (*Boolean, optional, default=`True`*) – If *False* only not initialized parameters are read, all if *True*.
- **fire_event** (*Boolean, optional, default=`True`*) – *True* to throw and update event if any parameter changed, *False* otherwise.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

AbstractXBeeDevice.is_device_info_complete()

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

Returns IO sample read from the XBee.

Return type *IOSample*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOSample

reset()
Override method.

See also:

AbstractXBeeDevice.reset()

scan_counter
Returns the scan counter for this node.

Returns The scan counter for this node.

Return type Integer

set_16bit_addr(*value*)
Sets the 16-bit address of the XBee.

Parameters value (*XBee16BitAddress*) – New 16-bit address of the XBee.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If the protocol is not 802.15.4.

set_api_output_mode(*api_output_mode*)
Deprecated since version 1.3: Use *set_api_output_mode_value()*

Sets the API output mode of the XBee.

Parameters api_output_mode (*APIOutputMode*) – New API output mode.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

APIOutputMode

set_api_output_mode_value(*api_output_mode*)
Sets the API output mode of the XBee.

Parameters **`api_output_mode`** (*`Integer`*) – New API output mode options. Calculate this value using the method `APIOutputModeBit.calculate_api_output_mode_value()` with a set of *`APIOutputModeBit`*.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

`APIOutputModeBit`

`set_dest_address`(*addr*)

Sets the 64-bit address of the XBee that is data destination.

Parameters **`addr`** (*`XBee64BitAddress`* or *`RemoteXBeeDevice`*) – Address itself or remote XBee to be data destination.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `ValueError` – If *addr* is *None*.

See also:

`XBee64BitAddress`
`get_dest_address()`

`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 response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.

- `ATCommandException` – If 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`*) – Digital IO line to sets its value.
- **`io_value`** (*`IOValue`*) – IO value to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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`*) – IO line to configure.
- **`io_mode`** (*`IOMode`*) – IO mode to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`IOLine`
`IOMode`
`get_io_configuration()`

set_io_sampling_rate(*rate*)

Sets the IO sampling rate of the XBee in seconds. A sample rate of 0 means the IO sampling feature is disabled.

Parameters **rate** (*Integer*) – New IO sampling rate of the XBee in seconds.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

get_io_sampling_rate()

set_local_xbee_device(*local_xbee_device*)

This methods associates a *XBeeDevice* to the remote XBee.

Parameters **local_xbee_device** (*XBeeDevice*) – New local XBee associated to the remote one.

See also:

XBeeDevice

set_node_id(*node_id*)

Sets the node identifier ('NI') value of the XBee.

Parameters **node_id** (*String*) – New node identifier ('NI') of the XBee.

Raises

- `ValueError` – If *node_id* is *None* or its length is greater than 20.
- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

set_ota_max_block_size(*size*)

Sets the maximum number of bytes to send for ota updates.

Parameters **size** (*Integer*) – Maximum ota block size to send.

Raises `ValueError` – If size is not between 0 and 255.

set_pan_id(*value*)

Sets the operating PAN ID of the XBee.

Parameters **value** (*Bytearray*) – New operating PAN ID of the XBee. Must have only 1 or 2 bytes.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

get_pan_id()

set_parameter (*parameter*, *value*, *apply=None*)
Override.

See also:

AbstractXBeeDevice.set_parameter()

set_power_level (*power_level*)
Sets the power level of the XBee.

Parameters **power_level** (*PowerLevel*) – New power level of the XBee.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

PowerLevel
get_power_level()

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*) – IO Line to be assigned.
- **cycle** (*Integer*) – Duty cycle in % to be assigned. Must be between 0 and 100.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – Read timeout in seconds.

update_bluetooth_password(*new_password*)
Changes the Bluetooth password of this XBee with the new one provided.

Note that your device must include Bluetooth Low Energy support.

Parameters **new_password** (*String*) – New Bluetooth password.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

update_device_data_from(*device*)
Updates the current node information with provided data. This is only for internal use.

Parameters **device** (*AbstractXBeeDevice*) – XBee to get the data from.

Returns *True* if the node data has been updated, *False* otherwise.

Return type Boolean

update_filesystem_image(*ota_filesystem_file*, *timeout=None*, *progress_callback=None*)
Performs a filesystem image update operation of the device.

Parameters

- **ota_filesystem_file** (*String*) – Location of the OTA filesystem image file.
- **timeout** (*Integer, optional*) – Maximum time to wait for target read operations during the update process.
- **progress_callback** (*Function, optional*) – Function to receive progress information. Receives two arguments:

- The current update task as a String.
- The current update task percentage as an Integer.

Raises

- `XBeeException` – If the device is not open.
- `InvalidOperatingModeException` – If the device operating mode is invalid.
- `FileSystemNotSupportedException` – If the filesystem update is not supported in the XBee.
- `FileSystemException` – If there is any error performing the filesystem update.

update_firmware(*xml_firmware_file*, *xbee_firmware_file=None*, *bootloader_firmware_file=None*, *timeout=None*, *progress_callback=None*)

Performs a firmware update operation of the XBee.

Parameters

- **xml_firmware_file** (*String*) – Path of the XML file that describes the firmware to upload.
- **xbee_firmware_file** (*String, optional, default=`None`*) – Location of the XBee binary firmware file.
- **bootloader_firmware_file** (*String, optional, default=`None`*) – Location of the bootloader binary firmware file.
- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the update process (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

Raises

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `OperationNotSupportedException` – If XBee does not support firmware update.
- `FirmwareUpdateException` – If there is any error during the firmware update.

write_changes()

Writes configurable parameter values to the non-volatile memory of the XBee so that parameter modifications persist through subsequent resets.

Parameters values remain in the device's memory until overwritten by subsequent use of this method.

If changes are made without writing them, the XBee 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_changes_enabled()* to get its status and *enable_apply_changes()* to enable/disable the option. Method *apply_changes()* can be used in order to manually apply the changes.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

class `digi.xbee.devices.`**`RemoteDigiMeshDevice`**(*local_xbee*, *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`** (*`XBeeDevice`*) – Local XBee associated with the remote one.
- **`x64bit_addr`** (*`XBee64BitAddress`*) – 64-bit address of the remote XBee.
- **`node_id`** (*`String, optional`*) – Node identifier of the remote XBee.

Raises `XBeeException` – If the protocol of *local_xbee* is invalid.

See also:

`RemoteXBeeDevice`
`XBee64BitAddress`
`XBeeDevice`

`get_protocol`()

Override.

See also:

`RemoteXBeeDevice.get_protocol()`

`get_neighbors`(*neighbor_cb=None*, *finished_cb=None*, *timeout=None*)

Returns the neighbors of this XBee. If *neighbor_cb* is not defined, the process blocks during the specified timeout.

Parameters

- **`neighbor_cb`** (*`Function, optional, default=`None``*) – Method called when a new neighbor is received. Receives two arguments:
  - The XBee that owns this new neighbor.
  - The new neighbor.
- **`finished_cb`** (*`Function, optional, default=`None``*) – Method to execute when the process finishes. Receives three arguments:
  - The XBee that is searching for its neighbors.
  - A list with the discovered neighbors.

    – An error message if something went wrong.

- **timeout** (*Float, optional, default=`NeighborFinder.DEFAULT_TIMEOUT`*) – The timeout in seconds.

Returns

List of `Neighbor` when *neighbor_cb* is not defined, *None* otherwise (in this case neighbors are received in the callback).

Return type List

Raises `OperationNotSupportedException` – If XBee protocol is not DigiMesh.

See also:

`com.digi.models.zdo.Neighbor`

`apply_changes`()

Applies changes via 'AC' command.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

`apply_profile`(*profile_path*, *timeout=None*, *progress_callback=None*)

Applies the given XBee profile to the XBee.

Parameters

- **profile_path** (*String*) – Path of the XBee profile file to apply.
- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the apply profile (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to receive progress information. Receives two arguments:
    – The current apply profile task as a String
    – The current apply profile task percentage as an Integer

Raises

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `UpdateProfileException` – If there is any error applying the XBee profile.

`determine_protocol`(*hardware_version*, *firmware_version*)

Determines the XBee protocol based on the given hardware and firmware versions.

Parameters

- **hardware_version** (*Integer*) – Hardware version to get its protocol.

- **firmware_version** (*Bytearray*) – Firmware version to get its protocol.

Returns

XBee protocol corresponding to the given hardware and firmware versions.

Return type *XBeeProtocol*

disable_bluetooth()

Disables the Bluetooth interface of this XBee.

Note that your device must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

enable_apply_changes(*value*)

Sets apply changes flag.

Parameters **value** (*Boolean*) – *True* to enable apply changes flag, *False* to disable it.

enable_bluetooth()

Enables the Bluetooth interface of this XBee.

To work with this interface, you must also configure the Bluetooth password if not done previously. Use method *AbstractXBeeDevice.update_bluetooth_password()*.

Note that your XBee must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

execute_command(*parameter*, *value=None*, *apply=None*)

Executes the provided command.

Parameters

- **(String or** (*parameter*) – class: *.ATStringCommand*): AT command to execute.
- **value** (*bytearray, optional, default=`None`*) – Command value (if any).
- **apply** (*Boolean, optional, default=`None`*) – *True* to apply changes in XBee configuration, *False* not to apply them, *None* to use *is_apply_changes_enabled()* returned value.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.

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

See also:

AbstractXBeeDevice.get_parameter()
AbstractXBeeDevice.set_parameter()
AbstractXBeeDevice.apply_changes()
AbstractXBeeDevice.write_changes()
AbstractXBeeDevice.is_apply_changes_enabled()
AbstractXBeeDevice.enable_apply_changes()

get_16bit_addr()
Returns the 16-bit address of the XBee.

Returns 16-bit address of the XBee.

Return type *XBee16BitAddress*

See also:

XBee16BitAddress

get_64bit_addr()
Returns the 64-bit address of the XBee.

Returns 64-bit address of the XBee.

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*) – IO line to get its ADC value.

Returns Analog value corresponding to the provided IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.

- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If response does not contain the value for the given IO line.

See also:

`IOLine`
`set_io_configuration()`

`get_api_output_mode``()`

Deprecated since version 1.3: Use *`get_api_output_mode_value()`*

Returns the API output mode of the XBee.

The API output mode determines the format of the data through the serial interface of the XBee.

Returns API output mode of the XBee.

Return type *`APIOutputMode`*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`APIOutputMode`

`get_api_output_mode_value``()`

Returns the API output mode of the XBee.

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

Returns the parameter value.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

`digi.xbee.models.mode.APIOutputModeBit`

`get_bluetooth_mac_addr``()`
Reads and returns the EUI-48 Bluetooth MAC address of this XBee following the format *00112233AABB*.

Note that your device must include Bluetooth Low Energy support.

Returns The Bluetooth MAC address.

Return type String

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

`get_comm_iface``()`
Returns the communication interface of the local XBee associated to the remote one.

Returns

Communication interface of the local XBee associated to the remote one.

Return type `XBeeCommunicationInterface`

See also:

`XBeeCommunicationInterface`

`get_current_frame_id``()`
Returns the last used frame ID.

Returns Last used frame ID.

Return type Integer

`get_dest_address``()`
Returns the 64-bit address of the XBee that is data destination.

Returns 64-bit address of destination XBee.

Return type *`XBee64BitAddress`*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.

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

See also:

XBee64BitAddress
set_dest_address()

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 response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If response does not contain the value for the given IO line.

See also:

IOLine
IOValue
set_io_configuration()

get_file_manager()

Returns the file system manager for the XBee.

Returns The file system manager.

Return type *FileSystemManager*

Raises `FileSystemNotSupportedException` – If the XBee does not support filesystem.

get_firmware_version()

Returns the firmware version of the XBee.

Returns Firmware version of the XBee.

Return type Bytearray

get_hardware_version()
Returns the hardware version of the XBee.

Returns Hardware version of the XBee.

Return type *HardwareVersion*

See also:

HardwareVersion

get_io_configuration(*io_line*)
Returns the configuration of the provided IO line.

Parameters **io_line** (*IOLine*) – IO line to get its configuration.

Returns IO mode of the IO line provided.

Return type *IOMode*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOLine
IOMode
set_io_configuration()

get_io_sampling_rate()
Returns the IO sampling rate of the XBee.

Returns IO sampling rate of XBee.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

set_io_sampling_rate()

get_local_xbee_device()
Returns the local XBee associated to the remote one.

Returns Local XBee.

Return type *XBeeDevice*

get_node_id()
Returns the node identifier ('NI') value of the XBee.

Returns Node identifier ('NI') of the XBee.

Return type String

get_ota_max_block_size()
Returns the maximum number of bytes to send for ota updates.

Returns Maximum ota block size to send.

Return type Integer

get_pan_id()
Returns the operating PAN ID of the XBee.

Returns Operating PAN ID of the XBee.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

set_pan_id()

get_parameter(*parameter*, *parameter_value=None*, *apply=None*)
Override.

See also:

AbstractXBeeDevice.get_parameter()

get_power_level()
Returns the power level of the XBee.

Returns Power level of the XBee.

Return type *PowerLevel*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`PowerLevel`
`set_power_level()`

`get_pwm_duty_cycle`(*io_line*)
Returns the PWM duty cycle in % corresponding to the provided IO line.

Parameters **`io_line`** (*`IOLine`*) – IO line to get its PWM duty cycle.

Returns PWM duty cycle of the given IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `ValueError` – If *io_line* has no PWM capability.

See also:

`IOLine`

`get_role`()
Gets the XBee role.

Returns the role of the XBee.

Return type *`Role`*

See also:

`Role`

`get_serial_port`()
Returns the serial port of the local XBee associated to the remote one.

Returns

Serial port of the local XBee associated to the remote one.

Return type `XBeeSerialPort`

See also:

`XBeeSerialPort`

`get_sync_ops_timeout``()`
Returns the serial port read timeout.

Returns Serial port read timeout in seconds.

Return type Integer

`is_apply_changes_enabled``()`
Returns whether apply changes flag is enabled.

Returns *True* if apply changes flag is enabled, *False* otherwise.

Return type Boolean

`is_device_info_complete``()`
Returns whether XBee node information is complete.

Returns *True* if node information is complete, *False* otherwise.

Return type Boolean

See also:

`AbstractXBeeDevice.read_device_info()`

`is_remote``()`
Override method.

See also:

`AbstractXBeeDevice.is_remote()`

`log`
Returns the XBee logger.

Returns The XBee device logger.

Return type `Logger`

`reachable`
Returns whether the XBee is reachable.

Returns *True* if the device is reachable, *False* otherwise.

Return type Boolean

`read_device_info``(`*init=True*, *fire_event=True*`)`
Updates all instance parameters reading them from the XBee.

Parameters

- **`init`** (*`Boolean, optional, default=`True``*) – If *False* only not initialized parameters are read, all if *True*.

- **fire_event** (*Boolean, optional, default=`True`*) – *True* to throw and update event if any parameter changed, *False* otherwise.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`AbstractXBeeDevice.is_device_info_complete()`

`read_io_sample`()

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

Returns IO sample read from the XBee.

Return type *`IOSample`*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`IOSample`

`reset`()

Override method.

See also:

`AbstractXBeeDevice.reset()`

`scan_counter`

Returns the scan counter for this node.

Returns The scan counter for this node.

Return type Integer

`set_16bit_addr`(*value*)

Sets the 16-bit address of the XBee.

Parameters **value** (*XBee16BitAddress*) – New 16-bit address of the XBee.

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.
- OperationNotSupportedException – If the protocol is not 802.15.4.

set_api_output_mode(*api_output_mode*)

Deprecated since version 1.3: Use *set_api_output_mode_value()*

Sets the API output mode of the XBee.

Parameters **api_output_mode** (*APIOutputMode*) – New API output mode.

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.
- OperationNotSupportedException – If it is not supported by the current protocol.

See also:

APIOutputMode

set_api_output_mode_value(*api_output_mode*)

Sets the API output mode of the XBee.

Parameters **api_output_mode** (*Integer*) – New API output mode options. Calculate this value using the method APIOutputModeBit.calculate_api_output_mode_value() with a set of *APIOutputModeBit*.

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.
- OperationNotSupportedException – If it is not supported by the current protocol.

See also:

APIOutputModeBit

set_dest_address(*addr*)
Sets the 64-bit address of the XBee that is data destination.

Parameters **addr** (*XBee64BitAddress* or *RemoteXBeeDevice*) – Address itself or remote XBee to be data destination.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `ValueError` – If *addr* is *None*.

See also:

XBee64BitAddress
get_dest_address()

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 response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – Digital IO line to sets its value.
- **io_value** (*IOValue*) – IO value to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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`*) – IO line to configure.
- **`io_mode`** (*`IOMode`*) – IO mode to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`IOLine`
`IOMode`
`get_io_configuration()`

`set_io_sampling_rate`(*rate*)
Sets the IO sampling rate of the XBee in seconds. A sample rate of 0 means the IO sampling feature is disabled.

Parameters **`rate`** (*`Integer`*) – New IO sampling rate of the XBee in seconds.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`get_io_sampling_rate()`

set_local_xbee_device(*local_xbee_device*)
This methods associates a *XBeeDevice* to the remote XBee.

Parameters **local_xbee_device** (*XBeeDevice*) – New local XBee associated to the remote one.

See also:

XBeeDevice

set_node_id(*node_id*)
Sets the node identifier (‘NI‘) value of the XBee.

Parameters **node_id** (*String*) – New node identifier (‘NI’) of the XBee.

Raises

- `ValueError` – If *node_id* is *None* or its length is greater than 20.
- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee’s communication interface is closed.
- `InvalidOperatingModeException` – If the XBee’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

set_ota_max_block_size(*size*)
Sets the maximum number of bytes to send for ota updates.

Parameters **size** (*Integer*) – Maximum ota block size to send.

Raises `ValueError` – If size is not between 0 and 255.

set_pan_id(*value*)
Sets the operating PAN ID of the XBee.

Parameters **value** (*Bytearray*) – New operating PAN ID of the XBee. Must have only 1 or 2 bytes.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee’s communication interface is closed.
- `InvalidOperatingModeException` – If the XBee’s operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

get_pan_id()

set_parameter(*parameter*, *value*, *apply=None*)
Override.

See also:

AbstractXBeeDevice.set_parameter()

set_power_level(*power_level*)

Sets the power level of the XBee.

Parameters **power_level** (*PowerLevel*) – New power level of the XBee.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

PowerLevel
get_power_level()

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*) – IO Line to be assigned.
- **cycle** (*Integer*) – Duty cycle in % to be assigned. Must be between 0 and 100.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – Read timeout in seconds.

update_bluetooth_password(*new_password*)
Changes the Bluetooth password of this XBee with the new one provided.

Note that your device must include Bluetooth Low Energy support.

Parameters **new_password** (*String*) – New Bluetooth password.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

update_device_data_from(*device*)
Updates the current node information with provided data. This is only for internal use.

Parameters **device** (*AbstractXBeeDevice*) – XBee to get the data from.

Returns *True* if the node data has been updated, *False* otherwise.

Return type Boolean

update_filesystem_image(*ota_filesystem_file*, *timeout=None*, *progress_callback=None*)
Performs a filesystem image update operation of the device.

Parameters

- **ota_filesystem_file** (*String*) – Location of the OTA filesystem image file.
- **timeout** (*Integer, optional*) – Maximum time to wait for target read operations during the update process.
- **progress_callback** (*Function, optional*) – Function to receive progress information. Receives two arguments:
  - The current update task as a String.
  - The current update task percentage as an Integer.

Raises

- `XBeeException` – If the device is not open.
- `InvalidOperatingModeException` – If the device operating mode is invalid.
- `FileSystemNotSupportedException` – If the filesystem update is not supported in the XBee.
- `FileSystemException` – If there is any error performing the filesystem update.

update_firmware(*xml_firmware_file*, *xbee_firmware_file=None*, *bootloader_firmware_file=None*, *timeout=None*, *progress_callback=None*)
Performs a firmware update operation of the XBee.

Parameters

- **xml_firmware_file** (*String*) – Path of the XML file that describes the firmware to upload.
- **xbee_firmware_file** (*String, optional, default=`None`*) – Location of the XBee binary firmware file.

- **bootloader_firmware_file** (*String, optional, default=`None`*) – Location of the bootloader binary firmware file.
- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the update process (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

Raises

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `OperationNotSupportedException` – If XBee does not support firmware update.
- `FirmwareUpdateException` – If there is any error during the firmware update.

write_changes()

Writes configurable parameter values to the non-volatile memory of the XBee so that parameter modifications persist through subsequent resets.

Parameters values remain in the device's memory until overwritten by subsequent use of this method.

If changes are made without writing them, the XBee 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_changes_enabled()* to get its status and *enable_apply_changes()* to enable/disable the option. Method *apply_changes()* can be used in order to manually apply the changes.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

class digi.xbee.devices.**RemoteDigiPointDevice**(*local_xbee*, *x64bit_addr=None*, *node_id=None*)

Bases: *digi.xbee.devices.RemoteXBeeDevice*

This class represents a remote DigiPoint XBee.

Class constructor. Instantiates a new *RemoteDigiMeshDevice* with the provided parameters.

Parameters

- **local_xbee** (*XBeeDevice*) – Local XBee associated with the remote one.
- **x64bit_addr** (*XBee64BitAddress*) – 64-bit address of the remote XBee.
- **node_id** (*String, optional*) – Node identifier of the remote XBee.

Raises `XBeeException` – If the protocol of *local_xbee* is invalid.

See also:

RemoteXBeeDevice
XBee64BitAddress
XBeeDevice

get_protocol()

Override.

See also:

RemoteXBeeDevice.get_protocol()

apply_changes()

Applies changes via 'AC' command.

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

apply_profile(*profile_path*, *timeout=None*, *progress_callback=None*)

Applies the given XBee profile to the XBee.

Parameters

- **profile_path** (*String*) – Path of the XBee profile file to apply.
- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the apply profile (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to receive progress information. Receives two arguments:
  - The current apply profile task as a String
  - The current apply profile task percentage as an Integer

Raises

- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- UpdateProfileException – If there is any error applying the XBee profile.

determine_protocol(*hardware_version*, *firmware_version*)

Determines the XBee protocol based on the given hardware and firmware versions.

Parameters

- **hardware_version** (*Integer*) – Hardware version to get its protocol.

- **firmware_version** (*Bytearray*) – Firmware version to get its protocol.

Returns

XBee protocol corresponding to the given hardware and firmware versions.

Return type *XBeeProtocol*

disable_bluetooth()

Disables the Bluetooth interface of this XBee.

Note that your device must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

enable_apply_changes(*value*)

Sets apply changes flag.

Parameters **value** (*Boolean*) – *True* to enable apply changes flag, *False* to disable it.

enable_bluetooth()

Enables the Bluetooth interface of this XBee.

To work with this interface, you must also configure the Bluetooth password if not done previously. Use method *AbstractXBeeDevice.update_bluetooth_password()*.

Note that your XBee must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

execute_command(*parameter*, *value=None*, *apply=None*)

Executes the provided command.

Parameters

- **(String or** (*parameter*) – class: *.ATStringCommand*): AT command to execute.
- **value** (*bytearray, optional, default=`None`*) – Command value (if any).
- **apply** (*Boolean, optional, default=`None`*) – *True* to apply changes in XBee configuration, *False* not to apply them, *None* to use *is_apply_changes_enabled()* returned value.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.

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

See also:

AbstractXBeeDevice.get_parameter()
AbstractXBeeDevice.set_parameter()
AbstractXBeeDevice.apply_changes()
AbstractXBeeDevice.write_changes()
AbstractXBeeDevice.is_apply_changes_enabled()
AbstractXBeeDevice.enable_apply_changes()

get_16bit_addr()
Returns the 16-bit address of the XBee.

Returns 16-bit address of the XBee.

Return type *XBee16BitAddress*

See also:

XBee16BitAddress

get_64bit_addr()
Returns the 64-bit address of the XBee.

Returns 64-bit address of the XBee.

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*) – IO line to get its ADC value.

Returns Analog value corresponding to the provided IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.

- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If response does not contain the value for the given IO line.

See also:

`IOLine`
`set_io_configuration()`

`get_api_output_mode``()`
Deprecated since version 1.3: Use *`get_api_output_mode_value()`*

Returns the API output mode of the XBee.

The API output mode determines the format of the data through the serial interface of the XBee.

Returns API output mode of the XBee.

Return type *`APIOutputMode`*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`APIOutputMode`

`get_api_output_mode_value``()`
Returns the API output mode of the XBee.

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

Returns the parameter value.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

digi.xbee.models.mode.APIOutputModeBit

get_bluetooth_mac_addr()

Reads and returns the EUI-48 Bluetooth MAC address of this XBee following the format *00112233AABB*.

Note that your device must include Bluetooth Low Energy support.

Returns The Bluetooth MAC address.

Return type String

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

get_comm_iface()

Returns the communication interface of the local XBee associated to the remote one.

Returns

Communication interface of the local XBee associated to the remote one.

Return type `XBeeCommunicationInterface`

See also:

`XBeeCommunicationInterface`

get_current_frame_id()

Returns the last used frame ID.

Returns Last used frame ID.

Return type Integer

get_dest_address()

Returns the 64-bit address of the XBee that is data destination.

Returns 64-bit address of destination XBee.

Return type *XBee64BitAddress*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.

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

See also:

XBee64BitAddress
set_dest_address()

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 response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If response does not contain the value for the given IO line.

See also:

IOLine
IOValue
set_io_configuration()

get_file_manager()
Returns the file system manager for the XBee.

Returns The file system manager.

Return type *FileSystemManager*

Raises `FileSystemNotSupportedException` – If the XBee does not support filesystem.

get_firmware_version()
Returns the firmware version of the XBee.

Returns Firmware version of the XBee.

Return type Bytearray

get_hardware_version()
Returns the hardware version of the XBee.

Returns Hardware version of the XBee.

Return type *HardwareVersion*

See also:

HardwareVersion

get_io_configuration(*io_line*)
Returns the configuration of the provided IO line.

Parameters **io_line** (*IOLine*) – IO line to get its configuration.

Returns IO mode of the IO line provided.

Return type *IOMode*

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

See also:

IOLine
IOMode
set_io_configuration()

get_io_sampling_rate()
Returns the IO sampling rate of the XBee.

Returns IO sampling rate of XBee.

Return type Integer

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

See also:

set_io_sampling_rate()

get_local_xbee_device()
Returns the local XBee associated to the remote one.

Returns Local XBee.

Return type *XBeeDevice*

get_node_id()
Returns the node identifier ('NI') value of the XBee.

Returns Node identifier ('NI') of the XBee.

Return type String

get_ota_max_block_size()
Returns the maximum number of bytes to send for ota updates.

Returns Maximum ota block size to send.

Return type Integer

get_pan_id()
Returns the operating PAN ID of the XBee.

Returns Operating PAN ID of the XBee.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

set_pan_id()

get_parameter(*parameter*, *parameter_value=None*, *apply=None*)
Override.

See also:

AbstractXBeeDevice.get_parameter()

get_power_level()
Returns the power level of the XBee.

Returns Power level of the XBee.

Return type *PowerLevel*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`PowerLevel`
`set_power_level()`

`get_pwm_duty_cycle`(*io_line*)
Returns the PWM duty cycle in % corresponding to the provided IO line.

Parameters **`io_line`** (*`IOLine`*) – IO line to get its PWM duty cycle.

Returns PWM duty cycle of the given IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `ValueError` – If *io_line* has no PWM capability.

See also:

`IOLine`

`get_role`()
Gets the XBee role.

Returns the role of the XBee.

Return type *`Role`*

See also:

`Role`

`get_serial_port`()
Returns the serial port of the local XBee associated to the remote one.

Returns

Serial port of the local XBee associated to the remote one.

Return type `XBeeSerialPort`

See also:

`XBeeSerialPort`

`get_sync_ops_timeout`()
Returns the serial port read timeout.

Returns Serial port read timeout in seconds.

Return type Integer

`is_apply_changes_enabled`()
Returns whether apply changes flag is enabled.

Returns *True* if apply changes flag is enabled, *False* otherwise.

Return type Boolean

`is_device_info_complete`()
Returns whether XBee node information is complete.

Returns *True* if node information is complete, *False* otherwise.

Return type Boolean

See also:

`AbstractXBeeDevice.read_device_info()`

`is_remote`()
Override method.

See also:

`AbstractXBeeDevice.is_remote()`

`log`
Returns the XBee logger.

Returns The XBee device logger.

Return type `Logger`

`reachable`
Returns whether the XBee is reachable.

Returns *True* if the device is reachable, *False* otherwise.

Return type Boolean

`read_device_info`(*init=True*, *fire_event=True*)
Updates all instance parameters reading them from the XBee.

Parameters

- **`init`** (*`Boolean, optional, default=`True``*) – If *False* only not initialized parameters are read, all if *True*.

- **fire_event** (*Boolean, optional, default=`True`*) – *True* to throw and update event if any parameter changed, *False* otherwise.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

AbstractXBeeDevice.is_device_info_complete()

read_io_sample()

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

Returns IO sample read from the XBee.

Return type *IOSample*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOSample

reset()

Override method.

See also:

AbstractXBeeDevice.reset()

scan_counter

Returns the scan counter for this node.

Returns The scan counter for this node.

Return type Integer

set_16bit_addr(*value*)

Sets the 16-bit address of the XBee.

Parameters **value** (*XBee16BitAddress*) – New 16-bit address of the XBee.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If the protocol is not 802.15.4.

set_api_output_mode(*api_output_mode*)

Deprecated since version 1.3: Use *set_api_output_mode_value()*

Sets the API output mode of the XBee.

Parameters **api_output_mode** (*APIOutputMode*) – New API output mode.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

APIOutputMode

set_api_output_mode_value(*api_output_mode*)

Sets the API output mode of the XBee.

Parameters **api_output_mode** (*Integer*) – New API output mode options. Calculate this value using the method `APIOutputModeBit.calculate_api_output_mode_value()` with a set of *APIOutputModeBit*.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

APIOutputModeBit

set_dest_address(*addr*)
Sets the 64-bit address of the XBee that is data destination.

Parameters addr (*XBee64BitAddress* or *RemoteXBeeDevice*) – Address itself or remote XBee to be data destination.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `ValueError` – If *addr* is *None*.

See also:

XBee64BitAddress
get_dest_address()

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 response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – Digital IO line to sets its value.
- **io_value** (*IOValue*) – IO value to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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`*) – IO line to configure.
- **`io_mode`** (*`IOMode`*) – IO mode to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`IOLine`
`IOMode`
`get_io_configuration()`

`set_io_sampling_rate`(*rate*)
Sets the IO sampling rate of the XBee in seconds. A sample rate of 0 means the IO sampling feature is disabled.

Parameters **`rate`** (*`Integer`*) – New IO sampling rate of the XBee in seconds.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`get_io_sampling_rate()`

set_local_xbee_device(*local_xbee_device*)

This methods associates a `XBeeDevice` to the remote XBee.

Parameters **local_xbee_device** (`XBeeDevice`) – New local XBee associated to the remote one.

See also:

`XBeeDevice`

set_node_id(*node_id*)

Sets the node identifier ('NI') value of the XBee.

Parameters **node_id** (*String*) – New node identifier ('NI') of the XBee.

Raises

- `ValueError` – If *node_id* is *None* or its length is greater than 20.
- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

set_ota_max_block_size(*size*)

Sets the maximum number of bytes to send for ota updates.

Parameters **size** (*Integer*) – Maximum ota block size to send.

Raises `ValueError` – If size is not between 0 and 255.

set_pan_id(*value*)

Sets the operating PAN ID of the XBee.

Parameters **value** (*Bytearray*) – New operating PAN ID of the XBee. Must have only 1 or 2 bytes.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`get_pan_id()`

set_parameter(*parameter*, *value*, *apply=None*)

Override.

See also:

AbstractXBeeDevice.set_parameter()

set_power_level (*power_level*)

Sets the power level of the XBee.

Parameters **power_level** (*PowerLevel*) – New power level of the XBee.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

PowerLevel
get_power_level()

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*) – IO Line to be assigned.
- **cycle** (*Integer*) – Duty cycle in % to be assigned. Must be between 0 and 100.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – Read timeout in seconds.

update_bluetooth_password(*new_password*)

Changes the Bluetooth password of this XBee with the new one provided.

Note that your device must include Bluetooth Low Energy support.

Parameters **new_password** (*String*) – New Bluetooth password.

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

update_device_data_from(*device*)

Updates the current node information with provided data. This is only for internal use.

Parameters **device** (*AbstractXBeeDevice*) – XBee to get the data from.

Returns *True* if the node data has been updated, *False* otherwise.

Return type Boolean

update_filesystem_image(*ota_filesystem_file*, *timeout=None*, *progress_callback=None*)

Performs a filesystem image update operation of the device.

Parameters

- **ota_filesystem_file** (*String*) – Location of the OTA filesystem image file.
- **timeout** (*Integer, optional*) – Maximum time to wait for target read operations during the update process.
- **progress_callback** (*Function, optional*) – Function to receive progress information. Receives two arguments:
  - The current update task as a String.
  - The current update task percentage as an Integer.

Raises

- XBeeException – If the device is not open.
- InvalidOperatingModeException – If the device operating mode is invalid.
- FileSystemNotSupportedException – If the filesystem update is not supported in the XBee.
- FileSystemException – If there is any error performing the filesystem update.

update_firmware(*xml_firmware_file*, *xbee_firmware_file=None*, *bootloader_firmware_file=None*, *timeout=None*, *progress_callback=None*)

Performs a firmware update operation of the XBee.

Parameters

- **xml_firmware_file** (*String*) – Path of the XML file that describes the firmware to upload.
- **xbee_firmware_file** (*String, optional, default=`None`*) – Location of the XBee binary firmware file.

- **bootloader_firmware_file** (*String, optional, default=`None`*) – Location of the bootloader binary firmware file.
- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the update process (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

Raises

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `OperationNotSupportedException` – If XBee does not support firmware update.
- `FirmwareUpdateException` – If there is any error during the firmware update.

write_changes()

Writes configurable parameter values to the non-volatile memory of the XBee so that parameter modifications persist through subsequent resets.

Parameters values remain in the device's memory until overwritten by subsequent use of this method.

If changes are made without writing them, the XBee 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_changes_enabled()* to get its status and *enable_apply_changes()* to enable/disable the option. Method *apply_changes()* can be used in order to manually apply the changes.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

class digi.xbee.devices.**RemoteZigBeeDevice**(*local_xbee, x64bit_addr=None, x16bit_addr=None, node_id=None*)

Bases: *digi.xbee.devices.RemoteXBeeDevice*

This class represents a remote Zigbee XBee.

Class constructor. Instantiates a new *RemoteDigiMeshDevice* with the provided parameters.

Parameters

- **local_xbee** (*XBeeDevice*) – Local XBee associated with the remote one.
- **x64bit_addr** (*XBee64BitAddress*) – 64-bit address of the remote XBee.
- **x16bit_addr** (*XBee16BitAddress*) – 16-bit address of the remote XBee.
- **node_id** (*String, optional*) – Node identifier of the remote XBee.

Raises `XBeeException` – If the protocol of *local_xbee* is invalid.

See also:

RemoteXBeeDevice
`XBee16BitAddress`
`XBee64BitAddress`
XBeeDevice

parent
Returns the parent of the XBee if it is an end device.

Returns

The parent of the node for end devices, *None* if unknown or if it is not an end device.

Return type *AbstractXBeeDevice*

get_protocol()
Override.

See also:

RemoteXBeeDevice.get_protocol()

is_device_info_complete()
Override.

See also:

AbstractXBeeDevice.is_device_info_complete()

get_ai_status()
Returns the current association status of this XBee. It indicates occurrences of errors during the modem initialization and connection.

Returns

The XBee association indication status.

Return type *AssociationIndicationStatus*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

force_disassociate()

Forces this XBee to immediately disassociate from the network and re-attempt to associate.

Only valid for End Devices.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

get_routes(*route_cb=None*, *finished_cb=None*, *timeout=None*)

Returns the routes of this XBee. If *route_cb* is not defined, the process blocks until the complete routing table is read.

Parameters

- **route_cb** (*Function, optional, default=`None`*) – Method called when a new route is received. Receives two arguments:
  - The XBee that owns this new route.
  - The new route.
- **finished_cb** (*Function, optional, default=`None`*) – Method to execute when the process finishes. Receives three arguments:
  - The XBee that executed the ZDO command.
  - A list with the discovered routes.
  - An error message if something went wrong.
- **timeout** (*Float, optional, default=`RouteTableReader.DEFAULT_TIMEOUT`*) – The ZDO command timeout in seconds.

Returns

List of `Route` when *route_cb* is not defined, *None* otherwise (in this case routes are received in the callback).

Return type List

Raises `OperationNotSupportedException` – If XBee protocol is not Zigbee or Smart Energy.

See also:

```
com.digi.models.zdo.Route
```

get_neighbors(*neighbor_cb=None*, *finished_cb=None*, *timeout=None*)

Returns the neighbors of this XBee. If *neighbor_cb* is not defined, the process blocks until the complete neighbor table is read.

Parameters

- **neighbor_cb** (*Function, optional, default=`None`*) – Method called when a new neighbor is received. Receives two arguments:

  - The XBee that owns this new neighbor.
  - The new neighbor.
- **finished_cb** (*Function, optional, default=`None`*) – Method to execute when the process finishes. Receives three arguments:
  - The XBee that executed the ZDO command.
  - A list with the discovered neighbors.
  - An error message if something went wrong.
- **timeout** (*Float, optional, default=`NeighborTableReader.DEFAULT_TIMEOUT`*) – The ZDO command timeout in seconds.

Returns

List of `Neighbor` when *neighbor_cb* is not defined, *None* otherwise (in this case neighbors are received in the callback).

Return type List

Raises `OperationNotSupportedException` – If XBee protocol is not Zigbee or Smart Energy.

See also:

```
com.digi.models.zdo.Neighbor
```

apply_changes()

Applies changes via 'AC' command.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

apply_profile(*profile_path*, *timeout=None*, *progress_callback=None*)

Applies the given XBee profile to the XBee.

Parameters

- **profile_path** (*String*) – Path of the XBee profile file to apply.
- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the apply profile (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to receive progress information. Receives two arguments:
  - The current apply profile task as a String
  - The current apply profile task percentage as an Integer

Raises

- `XBeeException` – If the XBee's communication interface is closed.

- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `UpdateProfileException` – If there is any error applying the XBee profile.

`determine_protocol`(*hardware_version*, *firmware_version*)

Determines the XBee protocol based on the given hardware and firmware versions.

Parameters

- **`hardware_version`** (*`Integer`*) – Hardware version to get its protocol.
- **`firmware_version`** (*`Bytearray`*) – Firmware version to get its protocol.

Returns

XBee protocol corresponding to the given hardware and firmware versions.

Return type *`XBeeProtocol`*

`disable_bluetooth`()

Disables the Bluetooth interface of this XBee.

Note that your device must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

`enable_apply_changes`(*value*)

Sets apply changes flag.

Parameters **`value`** (*`Boolean`*) – *True* to enable apply changes flag, *False* to disable it.

`enable_bluetooth`()

Enables the Bluetooth interface of this XBee.

To work with this interface, you must also configure the Bluetooth password if not done previously. Use method *`AbstractXBeeDevice.update_bluetooth_password()`*.

Note that your XBee must include Bluetooth Low Energy support.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

`execute_command`(*parameter*, *value=None*, *apply=None*)

Executes the provided command.

Parameters

- **`(String or`** (*`parameter`*) – class: *.ATStringCommand*): AT command to execute.

- **value** (*bytearray, optional, default=`None`*) – Command value (if any).
- **apply** (*Boolean, optional, default=`None`*) – *True* to apply changes in XBee configuration, *False* not to apply them, *None* to use *is_apply_changes_enabled()* returned value.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

AbstractXBeeDevice.get_parameter()
AbstractXBeeDevice.set_parameter()
AbstractXBeeDevice.apply_changes()
AbstractXBeeDevice.write_changes()
AbstractXBeeDevice.is_apply_changes_enabled()
AbstractXBeeDevice.enable_apply_changes()

get_16bit_addr()
Returns the 16-bit address of the XBee.

Returns 16-bit address of the XBee.

Return type *XBee16BitAddress*

See also:

XBee16BitAddress

get_64bit_addr()
Returns the 64-bit address of the XBee.

Returns 64-bit address of the XBee.

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*) – IO line to get its ADC value.

Returns Analog value corresponding to the provided IO line.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If response does not contain the value for the given IO line.

See also:

IOLine
set_io_configuration()

get_api_output_mode()

Deprecated since version 1.3: Use *get_api_output_mode_value()*

Returns the API output mode of the XBee.

The API output mode determines the format of the data through the serial interface of the XBee.

Returns API output mode of the XBee.

Return type *APIOutputMode*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

APIOutputMode

get_api_output_mode_value()

Returns the API output mode of the XBee.

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

Returns the parameter value.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

digi.xbee.models.mode.APIOutputModeBit

get_bluetooth_mac_addr()

Reads and returns the EUI-48 Bluetooth MAC address of this XBee following the format *00112233AABB*.

Note that your device must include Bluetooth Low Energy support.

Returns The Bluetooth MAC address.

Return type String

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

get_comm_iface()

Returns the communication interface of the local XBee associated to the remote one.

Returns

Communication interface of the local XBee associated to the remote one.

Return type `XBeeCommunicationInterface`

See also:

`XBeeCommunicationInterface`

get_current_frame_id()

Returns the last used frame ID.

Returns Last used frame ID.

Return type Integer

get_dest_address()

Returns the 64-bit address of the XBee that is data destination.

Returns 64-bit address of destination XBee.

Return type *XBee64BitAddress*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

XBee64BitAddress
set_dest_address()

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 response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If response does not contain the value for the given IO line.

See also:

IOLine
IOValue
set_io_configuration()

get_file_manager()

Returns the file system manager for the XBee.

Returns The file system manager.

Return type *FileSystemManager*

Raises `FileSystemNotSupportedException` – If the XBee does not support filesystem.

get_firmware_version()
Returns the firmware version of the XBee.

Returns Firmware version of the XBee.

Return type Bytearray

get_hardware_version()
Returns the hardware version of the XBee.

Returns Hardware version of the XBee.

Return type *HardwareVersion*

See also:

HardwareVersion

get_io_configuration(*io_line*)
Returns the configuration of the provided IO line.

Parameters **io_line** (*IOLine*) – IO line to get its configuration.

Returns IO mode of the IO line provided.

Return type *IOMode*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOLine
IOMode
set_io_configuration()

get_io_sampling_rate()
Returns the IO sampling rate of the XBee.

Returns IO sampling rate of XBee.

Return type Integer

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.

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

See also:

set_io_sampling_rate()

get_local_xbee_device()

Returns the local XBee associated to the remote one.

Returns Local XBee.

Return type *XBeeDevice*

get_node_id()

Returns the node identifier ('NI') value of the XBee.

Returns Node identifier ('NI') of the XBee.

Return type String

get_ota_max_block_size()

Returns the maximum number of bytes to send for ota updates.

Returns Maximum ota block size to send.

Return type Integer

get_pan_id()

Returns the operating PAN ID of the XBee.

Returns Operating PAN ID of the XBee.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

set_pan_id()

get_parameter(*parameter*, *parameter_value=None*, *apply=None*)

Override.

See also:

AbstractXBeeDevice.get_parameter()

get_power_level()

Returns the power level of the XBee.

Returns Power level of the XBee.

Return type *PowerLevel*

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

See also:

PowerLevel

set_power_level()

get_pwm_duty_cycle(*io_line*)

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

Parameters **io_line** (*IOLine*) – IO line to get its PWM duty cycle.

Returns PWM duty cycle of the given IO line.

Return type Integer

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.
- ValueError – If *io_line* has no PWM capability.

See also:

IOLine

get_role()

Gets the XBee role.

Returns the role of the XBee.

Return type *Role*

See also:

Role

get_serial_port()
Returns the serial port of the local XBee associated to the remote one.

Returns

Serial port of the local XBee associated to the remote one.

Return type `XBeeSerialPort`

See also:

`XBeeSerialPort`

get_sync_ops_timeout()
Returns the serial port read timeout.

Returns Serial port read timeout in seconds.

Return type Integer

is_apply_changes_enabled()
Returns whether apply changes flag is enabled.

Returns *True* if apply changes flag is enabled, *False* otherwise.

Return type Boolean

is_remote()
Override method.

See also:

`AbstractXBeeDevice.is_remote()`

log
Returns the XBee logger.

Returns The XBee device logger.

Return type `Logger`

reachable
Returns whether the XBee is reachable.

Returns *True* if the device is reachable, *False* otherwise.

Return type Boolean

read_device_info(*init=True*, *fire_event=True*)
Updates all instance parameters reading them from the XBee.

Parameters

- **init** (*Boolean, optional, default=`True`*) – If *False* only not initialized parameters are read, all if *True*.
- **fire_event** (*Boolean, optional, default=`True`*) – *True* to throw and update event if any parameter changed, *False* otherwise.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`AbstractXBeeDevice.is_device_info_complete()`

`read_io_sample`()

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

Returns IO sample read from the XBee.

Return type *`IOSample`*

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

`IOSample`

`reset`()

Override method.

See also:

`AbstractXBeeDevice.reset()`

`scan_counter`

Returns the scan counter for this node.

Returns The scan counter for this node.

Return type Integer

`set_16bit_addr`(*value*)

Sets the 16-bit address of the XBee.

Parameters **`value`** (*`XBee16BitAddress`*) – New 16-bit address of the XBee.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If the protocol is not 802.15.4.

`set_api_output_mode`(*api_output_mode*)

Deprecated since version 1.3: Use *`set_api_output_mode_value()`*

Sets the API output mode of the XBee.

Parameters **`api_output_mode`** (*`APIOutputMode`*) – New API output mode.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

`APIOutputMode`

`set_api_output_mode_value`(*api_output_mode*)

Sets the API output mode of the XBee.

Parameters **`api_output_mode`** (*`Integer`*) – New API output mode options. Calculate this value using the method `APIOutputModeBit.calculate_api_output_mode_value()` with a set of *`APIOutputModeBit`*.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `OperationNotSupportedException` – If it is not supported by the current protocol.

See also:

`APIOutputModeBit`

set_dest_address(*addr*)

Sets the 64-bit address of the XBee that is data destination.

Parameters addr (*XBee64BitAddress* or *RemoteXBeeDevice*) – Address itself or remote XBee to be data destination.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.
- `ValueError` – If *addr* is *None*.

See also:

XBee64BitAddress
get_dest_address()

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 response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – Digital IO line to sets its value.
- **io_value** (*IOValue*) – IO value to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.

- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – IO line to configure.
- **io_mode** (*IOMode*) – IO mode to set to the IO line.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

IOLine
IOMode
get_io_configuration()

set_io_sampling_rate(*rate*)
Sets the IO sampling rate of the XBee in seconds. A sample rate of 0 means the IO sampling feature is disabled.

Parameters **rate** (*Integer*) – New IO sampling rate of the XBee in seconds.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

get_io_sampling_rate()

set_local_xbee_device(*local_xbee_device*)
This methods associates a *XBeeDevice* to the remote XBee.

Parameters **local_xbee_device** (*XBeeDevice*) – New local XBee associated to the remote one.

See also:

XBeeDevice

set_node_id(*node_id*)
Sets the node identifier ('NI') value of the XBee.

Parameters **node_id** (*String*) – New node identifier ('NI') of the XBee.

Raises

- `ValueError` – If *node_id* is *None* or its length is greater than 20.
- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

set_ota_max_block_size(*size*)
Sets the maximum number of bytes to send for ota updates.

Parameters **size** (*Integer*) – Maximum ota block size to send.

Raises `ValueError` – If size is not between 0 and 255.

set_pan_id(*value*)
Sets the operating PAN ID of the XBee.

Parameters **value** (*Bytearray*) – New operating PAN ID of the XBee. Must have only 1 or 2 bytes.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

get_pan_id()

set_parameter(*parameter*, *value*, *apply=None*)
Override.

See also:

AbstractXBeeDevice.set_parameter()

set_power_level(*power_level*)
Sets the power level of the XBee.

Parameters **power_level** (*PowerLevel*) – New power level of the XBee.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

PowerLevel
get_power_level()

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*) – IO Line to be assigned.
- **cycle** (*Integer*) – Duty cycle in % to be assigned. Must be between 0 and 100.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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*) – Read timeout in seconds.

update_bluetooth_password(*new_password*)

Changes the Bluetooth password of this XBee with the new one provided.

Note that your device must include Bluetooth Low Energy support.

Parameters **new_password** (*String*) – New Bluetooth password.

Raises

- TimeoutException – If response is not received before the read timeout expires.
- XBeeException – If the XBee's communication interface is closed.
- InvalidOperatingModeException – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- ATCommandException – If response is not as expected.

update_device_data_from(*device*)

Updates the current node information with provided data. This is only for internal use.

Parameters **device** (*AbstractXBeeDevice*) – XBee to get the data from.

Returns *True* if the node data has been updated, *False* otherwise.

Return type Boolean

update_filesystem_image(*ota_filesystem_file*, *timeout=None*, *progress_callback=None*)

Performs a filesystem image update operation of the device.

Parameters

- **ota_filesystem_file** (*String*) – Location of the OTA filesystem image file.
- **timeout** (*Integer, optional*) – Maximum time to wait for target read operations during the update process.
- **progress_callback** (*Function, optional*) – Function to receive progress information. Receives two arguments:
  - The current update task as a String.
  - The current update task percentage as an Integer.

Raises

- XBeeException – If the device is not open.
- InvalidOperatingModeException – If the device operating mode is invalid.
- FileSystemNotSupportedException – If the filesystem update is not supported in the XBee.
- FileSystemException – If there is any error performing the filesystem update.

update_firmware(*xml_firmware_file*, *xbee_firmware_file=None*, *bootloader_firmware_file=None*, *timeout=None*, *progress_callback=None*)

Performs a firmware update operation of the XBee.

Parameters

- **xml_firmware_file** (*String*) – Path of the XML file that describes the firmware to upload.
- **xbee_firmware_file** (*String, optional, default=`None`*) – Location of the XBee binary firmware file.

- **bootloader_firmware_file** (*String, optional, default=`None`*) – Location of the bootloader binary firmware file.
- **timeout** (*Integer, optional, default=`None`*) – Maximum time to wait for target read operations during the update process (seconds).
- **progress_callback** (*Function, optional, default=`None`*) – Function to to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

Raises

- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `OperationNotSupportedException` – If XBee does not support firmware update.
- `FirmwareUpdateException` – If there is any error during the firmware update.

write_changes()

Writes configurable parameter values to the non-volatile memory of the XBee so that parameter modifications persist through subsequent resets.

Parameters values remain in the device's memory until overwritten by subsequent use of this method.

If changes are made without writing them, the XBee 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_changes_enabled()`* to get its status and *`enable_apply_changes()`* to enable/disable the option. Method *`apply_changes()`* can be used in order to manually apply the changes.

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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`*) – Local XBee 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.

DEFAULT_TIME_BETWEEN_SCANS = 10
Default time (in seconds) to wait before starting a new scan.

MIN_TIME_BETWEEN_SCANS = 0
Low limit for the time (in seconds) to wait before starting a new scan.

MAX_TIME_BETWEEN_SCANS = 259200
High limit for the time (in seconds) to wait before starting a new scan.

DEFAULT_TIME_BETWEEN_REQUESTS = 5
Default time (in seconds) to wait between node neighbors requests.

MIN_TIME_BETWEEN_REQUESTS = 0
Low limit for the time (in seconds) to wait between node neighbors requests.

MAX_TIME_BETWEEN_REQUESTS = 600
High limit for the time (in seconds) to wait between node neighbors requests.

SCAN_TIL_CANCEL = 0
The neighbor discovery process continues until is manually stopped.

scan_counter
Returns the scan counter.

Returns The scan counter.

Return type Integer

start_discovery_process(*deep=False*, *n_deep_scans=1*)
Starts the discovery process. This method is not blocking.

This process can discover node neighbors and connections, or only nodes:

- Deep discovery: Network nodes and connections between them (including quality) are discovered.

  The discovery process will be running the number of scans configured in *n_deep_scans*. A scan is considered the process of discovering the full network. If there are more than one number of scans configured, after finishing one another is started, until *n_deep_scans* is satisfied.

  See `set_deep_discovery_options()` to establish the way the network discovery process is performed.

- No deep discovery: Only network nodes are discovered.

  The discovery process will be running until the configured timeout expires or, in case of 802.15.4, until the 'end' packet is read.

  It may occur that, after timeout expiration, there are nodes that continue sending discovery responses to the local XBee. In this case, these nodes will not be added to the network.

In 802.15.4, both (deep and no deep discovery) are the same and none discover the node connections or their quality. The difference is the possibility of running more than one scan using a deep discovery.

Parameters

- **deep** (*Boolean, optional, default=`False`*) – *True* for a deep network scan, looking for neighbors and their connections, *False* otherwise.
- **n_deep_scans** (*Integer, optional, default=1*) – Number of scans to perform before automatically stopping the discovery process. *SCAN_TIL_CANCEL* means the process will not be automatically stopped. Only applicable if *deep=True*.

See also:

XBeeNetwork.add_device_discovered_callback()
XBeeNetwork.add_discovery_process_finished_callback()
XBeeNetwork.del_device_discovered_callback()
XBeeNetwork.del_discovery_process_finished_callback()
XBeeNetwork.get_deep_discovery_options()
XBeeNetwork.set_deep_discovery_options()
XBeeNetwork.get_deep_discovery_timeouts()
XBeeNetwork.set_deep_discovery_timeouts()
XBeeNetwork.get_discovery_options()
XBeeNetwork.set_discovery_options()
XBeeNetwork.get_discovery_timeout()
XBeeNetwork.set_discovery_timeout()

stop_discovery_process()
Stops the discovery process if it is running.

Note that some DigiMesh/DigiPoint devices are blocked until the discovery time configured ('NT' parameter) has elapsed, so, when trying to get/set any parameter during the discovery process, a TimeoutException is raised.

discover_device(*node_id*)
Blocking method. Discovers and reports the first remote XBee that matches the supplied identifier.

Parameters **node_id** (*String*) – Node identifier of the node to discover.

Returns

Discovered remote XBee, *None* if the timeout expires and the node was not found.

Return type *RemoteXBeeDevice*

See also:

XBeeNetwork.get_discovery_options()
XBeeNetwork.set_discovery_options()
XBeeNetwork.get_discovery_timeout()
XBeeNetwork.set_discovery_timeout()

discover_devices(*device_id_list*)
Blocking method. Attempts to discover a list of nodes and add them to the current network.

This method does not guarantee that all nodes of *device_id_list* will be found, even if they exist physically. This depends on the node discovery operation and timeout.

Parameters **device_id_list** (*List*) – List of device IDs to discover.

Returns

List with the discovered nodes. It may not contain all nodes specified in *device_id_list*.

Return type List

See also:

XBeeNetwork.get_discovery_options()
XBeeNetwork.set_discovery_options()
XBeeNetwork.get_discovery_timeout()
XBeeNetwork.set_discovery_timeout()

`is_discovery_running``()`
Returns whether the discovery process is running.

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 a new XBee node is added to the list after the execution of this method, this new XBee is not 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.

Returns

***True* if there is at least one node in the network,** *False* otherwise.

Return type Boolean

`get_number_devices``()`
Returns the number of nodes in the network.

Returns Number of nodes in the network.

Return type Integer

`export`(*dir_path=None*, *name=None*, *desc=None*)
Exports this network to the given file path.

If the provided path already exists the file is removed.

Params:

dir_path (String, optional, default='None'): Absolute path of the directory to export the network. It should not include the file name. If not defined home directory is used.

name (String, optional, default='None'): Network human readable name. desc (String, optional, default='None'): Network description.

Returns

Tuple with result (0: success, 1: failure) and string (exported file path if success, error string otherwise).

Return type Tuple (Integer, String)

add_network_modified_callback(*callback*)
Adds a callback for the event *NetworkModified*.

Parameters **callback** (*Function*) – The callback. Receives three arguments.

- The event type as a *NetworkEventType*.
- The reason of the event as a *NetworkEventReason*.
- The node added, updated or removed from the network as a *XBeeDevice* or *RemoteXBeeDevice*.

See also:

XBeeNetwork.del_network_modified_callback()

add_device_discovered_callback(*callback*)
Adds a callback for the event *DeviceDiscovered*.

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

- The discovered remote XBee as a *RemoteXBeeDevice*.

See also:

XBeeNetwork.del_device_discovered_callback()
XBeeNetwork.add_discovery_process_finished_callback()
XBeeNetwork.del_discovery_process_finished_callback()

add_init_discovery_scan_callback(*callback*)
Adds a callback for the event *InitDiscoveryScan*.

Parameters **callback** (*Function*) – The callback. Receives two arguments.

- Number of scan to start (starting with 1).
- Total number of scans.

See also:

XBeeNetwork.del_init_discovery_scan_callback()

add_end_discovery_scan_callback(*callback*)
Adds a callback for the event *EndDiscoveryScan*.

Parameters **callback** (*Function*) – The callback. Receives two arguments.

- Number of scan that has finished (starting with 1).
- Total number of scans.

See also:

XBeeNetwork.del_end_discovery_scan_callback()

add_discovery_process_finished_callback(*callback*)

Adds a callback for the event *DiscoveryProcessFinished*.

Parameters **callback** (*Function*) – The callback. Receives two arguments.

- The event code as an *NetworkDiscoveryStatus*.
- (Optional) A description of the discovery process as a string.

See also:

XBeeNetwork.del_discovery_process_finished_callback()
XBeeNetwork.add_device_discovered_callback()
XBeeNetwork.del_device_discovered_callback()

add_packet_received_from_callback(*node*, *callback*)

Adds a callback to listen to any received packet from the provided node.

Parameters

- **node** (*RemoteXBeeDevice*) – The node to listen for frames.
- **callback** (*Function*) – The callback. Receives two arguments.
  - The received packet as a *XBeeAPIPacket*.
  - The remote XBee who sent the packet as a *RemoteXBeeDevice*.

See also:

XBeeNetwork.del_packet_received_from_callback()

del_network_modified_callback(*callback*)

Deletes a callback for the callback list of *NetworkModified*.

Parameters **callback** (*Function*) – The callback to delete.

See also:

XBeeNetwork.add_network_modified_callback()

del_device_discovered_callback(*callback*)

Deletes a callback for the callback list of *DeviceDiscovered* event.

Parameters **callback** (*Function*) – The callback to delete.

See also:

XBeeNetwork.add_device_discovered_callback()
XBeeNetwork.add_discovery_process_finished_callback()
XBeeNetwork.del_discovery_process_finished_callback()

del_init_discovery_scan_callback(*callback*)

Deletes a callback for the callback list of *InitDiscoveryScan*.

Parameters **callback** (*Function*) – The callback to delete.

See also:

XBeeNetwork.add_init_discovery_scan_callback()

del_end_discovery_scan_callback(*callback*)

Deletes a callback for the callback list of *EndDiscoveryScan*.

Parameters **callback** (*Function*) – The callback to delete.

See also:

XBeeNetwork.add_end_discovery_scan_callback()

del_discovery_process_finished_callback(*callback*)

Deletes a callback for the callback list of *DiscoveryProcessFinished* event.

Parameters **callback** (*Function*) – The callback to delete.

See also:

XBeeNetwork.add_discovery_process_finished_callback()
XBeeNetwork.add_device_discovered_callback()
XBeeNetwork.del_device_discovered_callback()

del_packet_received_from_callback(*node*, *callb=None*)

Deletes a received packet callback from the provided node.

Parameters

- **node** (*RemoteXBeeDevice*) – The node to listen for frames.
- **callb** (*Function, optional, default=`None`*) – The callback to delete, *None* to delete all.

See also:

XBeeNetwork.add_packet_received_from_callback()

clear()

Removes all remote XBee nodes from the network.

get_discovery_options()

Returns the network discovery process options.

Returns Discovery options value.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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 response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

See also:

DiscoveryOptions

get_deep_discovery_options()

Returns the deep discovery process options.

Returns

(*NeighborDiscoveryMode*, Boolean): Tuple containing:

- **mode (*NeighborDiscoveryMode*): Neighbor discovery** mode, the way to perform the network discovery process.
- **remove_nodes (Boolean): *True* to remove nodes from the** network if they were not discovered in the last scan, *False* otherwise.

Return type Tuple

See also:

digi.xbee.models.mode.NeighborDiscoveryMode
XBeeNetwork.set_deep_discovery_timeouts()
XBeeNetwork.start_discovery_process()

set_deep_discovery_options(*deep_mode=<NeighborDiscoveryMode.CASCADE: (0, 'Cascade')>, del_not_discovered_nodes_in_last_scan=False*)

Configures the deep discovery options with the given values. These options are only applicable for "deep" discovery (see *start_discovery_process()*)

Parameters

- **deep_mode** (*NeighborDiscoveryMode*, optional, default='NeighborDiscoveryMode.CASCADE') – Neighbor discovery mode, the way to perform the network discovery process.
- **del_not_discovered_nodes_in_last_scan** (*Boolean, optional, default=`False`*) – *True* to remove nodes from the network if they were not discovered in the last scan.

See also:

digi.xbee.models.mode.NeighborDiscoveryMode
XBeeNetwork.get_deep_discovery_timeouts()
XBeeNetwork.start_discovery_process()

get_discovery_timeout()

Returns the network discovery timeout.

Returns Network discovery timeout.

Return type Float

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

set_discovery_timeout(*discovery_timeout*)

Sets the discovery network timeout.

Parameters **discovery_timeout** (*Float*) – Timeout in seconds.

Raises

- `ValueError` – If *discovery_timeout* is not between the allowed minimum and maximum values.
- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

get_deep_discovery_timeouts()

Gets deep discovery network timeouts. These timeouts are only applicable for "deep" discovery (see *start_discovery_process()*)

Returns

Tuple containing:

- **node_timeout (Float): Maximum duration in seconds of the** discovery process per node. This is used to find neighbors of a node. This timeout is highly dependent on the nature of the network:
  - It should be greater than the highest 'NT' (Node Discovery Timeout) of your network.
  - And include enough time to let the message propagate depending on the sleep cycle of your network nodes.
- **time_bw_nodes (Float): Time to wait between node neighbors** requests. Use this setting not to saturate your network:
  - For 'Cascade', the number of seconds to wait after completion of the neighbor discovery process of the previous node.
  - For 'Flood', the minimum time to wait between each node's neighbor requests.
- **time_bw_scans (Float): Time to wait before starting a new** network scan.

Return type Tuple (Float, Float, Float)

See also:

XBeeNetwork.set_deep_discovery_timeouts()

XBeeNetwork.start_discovery_process()

set_deep_discovery_timeouts(*node_timeout=None*, *time_bw_requests=None*, *time_bw_scans=None*)

Sets deep discovery network timeouts. These timeouts are only applicable for "deep" discovery (see *start_discovery_process()*)

node_timeout (Float, optional, default='None'): Maximum duration in seconds of the discovery process used to find neighbors of a node. If *None* already configured timeouts are used.

time_bw_requests (Float, optional, default='DEFAULT_TIME_BETWEEN_REQUESTS'): Time to wait between node neighbors requests. It must be between *MIN_TIME_BETWEEN_REQUESTS* and *MAX_TIME_BETWEEN_REQUESTS* seconds inclusive. Use this setting not to saturate your network:

- For 'Cascade', the number of seconds to wait after completion of the neighbor discovery process of the previous node.
- For 'Flood', the minimum time to wait between each node's neighbor requests.

time_bw_scans (Float, optional, default='DEFAULT_TIME_BETWEEN_SCANS'): Time to wait before starting a new network scan. It must be between *MIN_TIME_BETWEEN_SCANS* and *MAX_TIME_BETWEEN_SCANS* seconds inclusive.

Raises `ValueError` – if *node_timeout*, *time_bw_requests* or *time_bw_scans* are not between their corresponding limits.

See also:

XBeeNetwork.get_deep_discovery_timeouts()

XBeeNetwork.start_discovery_process()

classmethod get_nt_limits(*protocol*)
Returns a tuple with the minimum and maximum values for the 'NT' value depending on the protocol.

Returns

Minimum value in seconds, maximum value in seconds.

Return type Tuple (Float, Float)

is_node_in_network(*node*)
Checks if the provided node is in the network or if it is the local XBee.

Parameters **node** (*AbstractXBeeDevice*) – The node to check.

Returns *True* if the node is in the network, *False* otherwise.

Return type Boolean

Raises `ValueError` – If *node* is *None*.

get_device_by_64(*x64bit_addr*)
Returns the XBee in the network whose 64-bit address matches the given one.

Parameters **x64bit_addr** (`XBee64BitAddress`) – 64-bit address of the node to retrieve.

Returns XBee in the network or *None* if not found.

Return type *AbstractXBeeDevice*

Raises `ValueError` – If *x64bit_addr* is *None* or unknown.

get_device_by_16(*x16bit_addr*)
Returns the XBee in the network whose 16-bit address matches the given one.

Parameters **x16bit_addr** (`XBee16BitAddress`) – 16-bit address of the node to retrieve.

Returns XBee in the network or *Non* if not found.

Return type *AbstractXBeeDevice*

Raises `ValueError` – If *x16bit_addr* is *None* or unknown.

get_device_by_node_id(*node_id*)
Returns the XBee in the network whose node identifier matches the given one.

Parameters **node_id** (*String*) – Node identifier of the node to retrieve.

Returns XBee in the network or *None* if not found.

Return type *AbstractXBeeDevice*

Raises `ValueError` – If *node_id* is *None*.

add_if_not_exist(*x64bit_addr=None*, *x16bit_addr=None*, *node_id=None*)
Adds an XBee with the provided information if it does not exist in the current network.

If the XBee already exists, its data is updated with the provided information.

If no valid address is provided (*x64bit_addr*, *x16bit_addr*), *None* is returned.

Parameters

- **x64bit_addr** (`XBee64BitAddress`, optional, default='None') – 64-bit address.

- **x16bit_addr** (`XBee16BitAddress`, optional, default='None') – 16-bit address.
- **node_id** (*`String, optional, default=`None``*) – Node identifier.

Returns

the remote XBee with the updated information. If the XBee was not in the list yet, this method returns the given XBee without changes.

Return type *`AbstractXBeeDevice`*

`add_remote`(*remote_xbee*)

Adds the provided remote XBee to the network if it is not in yet.

If the XBee is already in the network, its data is updated with the information of the provided XBee that are not *None*.

Parameters **remote_xbee** (*`RemoteXBeeDevice`*) – Remote XBee to add.

Returns

Provided XBee with updated data. If the XBee was not in the list, it returns it without changes.

Return type *`RemoteXBeeDevice`*

`add_remotes`(*remote_xbees*)

Adds a list of remote XBee nodes to the network.

If any node in the list is already in the network, its data is updated with the information of the corresponding XBee in the list.

Parameters **remote_xbees** (*`List`*) – List of *`RemoteXBeeDevice`* to add.

`remove_device`(*remote_xbee*)

Removes the provided remote XBee from the network.

Parameters **remote_xbee** (*`RemoteXBeeDevice`*) – Remote XBee to remove.

Raises `ValueError` – If the provided *remote_xbee* 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 discovered.

Returns

Callback for generic devices discovery process, callback for discovery specific XBee ops.

Return type Tuple (Function, Function)

`get_connections`()

Returns a copy of the XBee network connections.

A deep discover must be performed to get the connections between network nodes.

If a new connection is added to the list after the execution of this method, this new connection is not added to the list returned by this method.

Returns A copy of the list of *`Connection`* for the network.

Return type List

See also:

XBeeNetwork.get_node_connections()
XBeeNetwork.start_discovery_process()

get_node_connections(*node*)
Returns the network connections with one of their ends *node*.

A deep discover must be performed to get the connections between network nodes.

If a new connection is added to the list after the execution of this method, this new connection is not added to the list returned by this method.

Parameters **node** (*AbstractXBeeDevice*) – The node to get its connections.

Returns List of *Connection* with *node* end.

Return type List

See also:

XBeeNetwork.get_connections()
XBeeNetwork.start_discovery_process()

class digi.xbee.devices.**ZigBeeNetwork**(*device*)
Bases: *digi.xbee.devices.XBeeNetwork*

This class represents a Zigbee network.

The network allows the discovery of remote nodes in the same network as the local one and stores them.

Class constructor. Instantiates a new *ZigBeeNetwork*.
Parameters **device** (*ZigBeeDevice*) – Local Zigbee node 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 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 two arguments.

- The event code as an *NetworkDiscoveryStatus*.
- (Optional) A description of the discovery process as a string.

See also:

XBeeNetwork.del_discovery_process_finished_callback()
XBeeNetwork.add_device_discovered_callback()
XBeeNetwork.del_device_discovered_callback()

add_end_discovery_scan_callback(*callback*)
Adds a callback for the event *EndDiscoveryScan*.

Parameters **callback** (*Function*) – The callback. Receives two arguments.

- Number of scan that has finished (starting with 1).
- Total number of scans.

See also:

XBeeNetwork.del_end_discovery_scan_callback()

add_if_not_exist(*x64bit_addr=None*, *x16bit_addr=None*, *node_id=None*)
Adds an XBee with the provided information if it does not exist in the current network.

If the XBee already exists, its data is updated with the provided information.

If no valid address is provided (*x64bit_addr*, *x16bit_addr*), *None* is returned.

Parameters

- **x64bit_addr** (XBee64BitAddress, optional, default='None') – 64-bit address.
- **x16bit_addr** (XBee16BitAddress, optional, default='None') – 16-bit address.
- **node_id** (*String, optional, default=`None`*) – Node identifier.

Returns

the remote XBee with the updated information. If the XBee was not in the list yet, this method returns the given XBee without changes.

Return type *AbstractXBeeDevice*

add_init_discovery_scan_callback(*callback*)
Adds a callback for the event *InitDiscoveryScan*.

Parameters **callback** (*Function*) – The callback. Receives two arguments.

- Number of scan to start (starting with 1).
- Total number of scans.

See also:

XBeeNetwork.del_init_discovery_scan_callback()

add_network_modified_callback(*callback*)
Adds a callback for the event *NetworkModified*.

Parameters **callback** (*Function*) – The callback. Receives three arguments.

- The event type as a `NetworkEventType`.
- The reason of the event as a `NetworkEventReason`.
- The node added, updated or removed from the network as a `XBeeDevice` or `RemoteXBeeDevice`.

See also:

`XBeeNetwork.del_network_modified_callback()`

add_packet_received_from_callback(*node*, *callback*)
Adds a callback to listen to any received packet from the provided node.

Parameters

- **node** (`RemoteXBeeDevice`) – The node to listen for frames.
- **callback** (`Function`) – The callback. Receives two arguments.
  - The received packet as a `XBeeAPIPacket`.
  - The remote XBee who sent the packet as a `RemoteXBeeDevice`.

See also:

`XBeeNetwork.del_packet_received_from_callback()`

add_remote(*remote_xbee*)
Adds the provided remote XBee to the network if it is not in yet.

If the XBee is already in the network, its data is updated with the information of the provided XBee that are not *None*.

Parameters **remote_xbee** (`RemoteXBeeDevice`) – Remote XBee to add.

Returns

Provided XBee with updated data. If the XBee was not in the list, it returns it without changes.

Return type `RemoteXBeeDevice`

add_remotes(*remote_xbees*)
Adds a list of remote XBee nodes to the network.

If any node in the list is already in the network, its data is updated with the information of the corresponding XBee in the list.

Parameters **remote_xbees** (`List`) – List of `RemoteXBeeDevice` to add.

clear()
Removes all remote XBee nodes 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.

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.

See also:

XBeeNetwork.add_discovery_process_finished_callback()
XBeeNetwork.add_device_discovered_callback()
XBeeNetwork.del_device_discovered_callback()

del_end_discovery_scan_callback(*callback*)
Deletes a callback for the callback list of *EndDiscoveryScan*.

Parameters **callback** (*Function*) – The callback to delete.

See also:

XBeeNetwork.add_end_discovery_scan_callback()

del_init_discovery_scan_callback(*callback*)
Deletes a callback for the callback list of *InitDiscoveryScan*.

Parameters **callback** (*Function*) – The callback to delete.

See also:

XBeeNetwork.add_init_discovery_scan_callback()

del_network_modified_callback(*callback*)
Deletes a callback for the callback list of *NetworkModified*.

Parameters **callback** (*Function*) – The callback to delete.

See also:

XBeeNetwork.add_network_modified_callback()

del_packet_received_from_callback(*node*, *callb=None*)
Deletes a received packet callback from the provided node.

Parameters

- **node** (*RemoteXBeeDevice*) – The node to listen for frames.

- **callb** (*Function, optional, default=`None`*) – The callback to delete, *None* to delete all.

See also:

XBeeNetwork.add_packet_received_from_callback()

discover_device(*node_id*)

Blocking method. Discovers and reports the first remote XBee that matches the supplied identifier.

Parameters **node_id** (*String*) – Node identifier of the node to discover.

Returns

Discovered remote XBee, *None* if the timeout expires and the node was not found.

Return type *RemoteXBeeDevice*

See also:

XBeeNetwork.get_discovery_options()
XBeeNetwork.set_discovery_options()
XBeeNetwork.get_discovery_timeout()
XBeeNetwork.set_discovery_timeout()

discover_devices(*device_id_list*)

Blocking method. Attempts to discover a list of nodes and add them to the current network.

This method does not guarantee that all nodes of *device_id_list* will be found, even if they exist physically. This depends on the node discovery operation and timeout.

Parameters **device_id_list** (*List*) – List of device IDs to discover.

Returns

List with the discovered nodes. It may not contain all nodes specified in *device_id_list*.

Return type List

See also:

XBeeNetwork.get_discovery_options()
XBeeNetwork.set_discovery_options()
XBeeNetwork.get_discovery_timeout()
XBeeNetwork.set_discovery_timeout()

export(*dir_path=None*, *name=None*, *desc=None*)

Exports this network to the given file path.

If the provided path already exists the file is removed.

Params:

dir_path (String, optional, default='None'): Absolute path of the directory to export the network. It should not include the file name. If not defined home directory is used.

name (String, optional, default='None'): Network human readable name. desc (String, optional, default='None'): Network description.

Returns

Tuple with result (0: success, 1: failure) and string (exported file path if success, error string otherwise).

Return type Tuple (Integer, String)

get_connections()

Returns a copy of the XBee network connections.

A deep discover must be performed to get the connections between network nodes.

If a new connection is added to the list after the execution of this method, this new connection is not added to the list returned by this method.

Returns A copy of the list of `Connection` for the network.

Return type List

See also:

XBeeNetwork.get_node_connections()
XBeeNetwork.start_discovery_process()

get_deep_discovery_options()

Returns the deep discovery process options.

Returns

(*NeighborDiscoveryMode*, Boolean): Tuple containing:

- **mode (*NeighborDiscoveryMode*): Neighbor discovery** mode, the way to perform the network discovery process.
- **remove_nodes (Boolean): *True* to remove nodes from the** network if they were not discovered in the last scan, *False* otherwise.

Return type Tuple

See also:

digi.xbee.models.mode.NeighborDiscoveryMode
XBeeNetwork.set_deep_discovery_timeouts()
XBeeNetwork.start_discovery_process()

get_deep_discovery_timeouts()

Gets deep discovery network timeouts. These timeouts are only applicable for "deep" discovery (see *start_discovery_process()*)

Returns

Tuple containing:

- **node_timeout (Float): Maximum duration in seconds of the** discovery process per node. This is used to find neighbors of a node. This timeout is highly dependent on the nature of the network:
  - It should be greater than the highest 'NT' (Node Discovery Timeout) of your network.
  - And include enough time to let the message propagate depending on the sleep cycle of your network nodes.
- **time_bw_nodes (Float): Time to wait between node neighbors** requests. Use this setting not to saturate your network:
  - For 'Cascade', the number of seconds to wait after completion of the neighbor discovery process of the previous node.
  - For 'Flood', the minimum time to wait between each node's neighbor requests.
- **time_bw_scans (Float): Time to wait before starting a new** network scan.

Return type Tuple (Float, Float, Float)

See also:

XBeeNetwork.set_deep_discovery_timeouts()
XBeeNetwork.start_discovery_process()

get_device_by_16(*x16bit_addr*)
Returns the XBee in the network whose 16-bit address matches the given one.

Parameters **x16bit_addr** (XBee16BitAddress) – 16-bit address of the node to retrieve.

Returns XBee in the network or *Non* if not found.

Return type *AbstractXBeeDevice*

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

get_device_by_64(*x64bit_addr*)
Returns the XBee in the network whose 64-bit address matches the given one.

Parameters **x64bit_addr** (XBee64BitAddress) – 64-bit address of the node to retrieve.

Returns XBee in the network or *None* if not found.

Return type *AbstractXBeeDevice*

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

get_device_by_node_id(*node_id*)
Returns the XBee in the network whose node identifier matches the given one.

Parameters **node_id** (*String*) – Node identifier of the node to retrieve.

Returns XBee in the network or *None* if not found.

Return type *AbstractXBeeDevice*

Raises ValueError – If *node_id* is *None*.

get_devices()

Returns a copy of the XBee devices list of the network.

If a new XBee node is added to the list after the execution of this method, this new XBee is not 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 discovered.

Returns

Callback for generic devices discovery process, callback for discovery specific XBee ops.

Return type Tuple (Function, Function)

get_discovery_options()

Returns the network discovery process options.

Returns Discovery options value.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

get_discovery_timeout()

Returns the network discovery timeout.

Returns Network discovery timeout.

Return type Float

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

get_node_connections(*node*)

Returns the network connections with one of their ends *node*.

A deep discover must be performed to get the connections between network nodes.

If a new connection is added to the list after the execution of this method, this new connection is not added to the list returned by this method.

Parameters **`node`** (*`AbstractXBeeDevice`*) – The node to get its connections.

Returns List of `Connection` with *node* end.

Return type List

See also:

`XBeeNetwork.get_connections()`

`XBeeNetwork.start_discovery_process()`

classmethod get_nt_limits(*protocol*)

Returns a tuple with the minimum and maximum values for the 'NT' value depending on the protocol.

Returns

Minimum value in seconds, maximum value in seconds.

Return type Tuple (Float, Float)

get_number_devices()

Returns the number of nodes in the network.

Returns Number of nodes in the network.

Return type Integer

has_devices()

Returns whether there is any device in the network.

Returns

***True* if there is at least one node in the network,** *False* otherwise.

Return type Boolean

is_discovery_running()

Returns whether the discovery process is running.

Returns *True* if the discovery process is running, *False* otherwise.

Return type Boolean

is_node_in_network(*node*)

Checks if the provided node is in the network or if it is the local XBee.

Parameters **`node`** (*`AbstractXBeeDevice`*) – The node to check.

Returns *True* if the node is in the network, *False* otherwise.

Return type Boolean

Raises `ValueError` – If *node* is *None*.

remove_device(*remote_xbee*)

Removes the provided remote XBee from the network.

Parameters **`remote_xbee`** (*`RemoteXBeeDevice`*) – Remote XBee to remove.

Raises `ValueError` – If the provided *remote_xbee* is not in the network.

scan_counter
Returns the scan counter.

Returns The scan counter.

Return type Integer

set_deep_discovery_options(*deep_mode=<NeighborDiscoveryMode.CASCADE: (0, 'Cascade')>, del_not_discovered_nodes_in_last_scan=False*)
Configures the deep discovery options with the given values. These options are only applicable for "deep" discovery (see `start_discovery_process()`)

Parameters

- **deep_mode** (`NeighborDiscoveryMode`, optional, default='NeighborDiscoveryMode.CASCADE') – Neighbor discovery mode, the way to perform the network discovery process.
- **del_not_discovered_nodes_in_last_scan** (*`Boolean, optional, default=`False``*) – *True* to remove nodes from the network if they were not discovered in the last scan.

See also:

`digi.xbee.models.mode.NeighborDiscoveryMode`
`XBeeNetwork.get_deep_discovery_timeouts()`
`XBeeNetwork.start_discovery_process()`

set_deep_discovery_timeouts(*node_timeout=None, time_bw_requests=None, time_bw_scans=None*)
Sets deep discovery network timeouts. These timeouts are only applicable for "deep" discovery (see `start_discovery_process()`)

node_timeout (Float, optional, default='None'): Maximum duration in seconds of the discovery process used to find neighbors of a node. If *None* already configured timeouts are used.

time_bw_requests (Float, optional, default='DEFAULT_TIME_BETWEEN_REQUESTS'): Time to wait between node neighbors requests. It must be between `MIN_TIME_BETWEEN_REQUESTS` and `MAX_TIME_BETWEEN_REQUESTS` seconds inclusive. Use this setting not to saturate your network:

- For 'Cascade', the number of seconds to wait after completion of the neighbor discovery process of the previous node.
- For 'Flood', the minimum time to wait between each node's neighbor requests.

time_bw_scans (Float, optional, default='DEFAULT_TIME_BETWEEN_SCANS'): Time to wait before starting a new network scan. It must be between `MIN_TIME_BETWEEN_SCANS` and `MAX_TIME_BETWEEN_SCANS` seconds inclusive.

Raises `ValueError` – if *node_timeout*, *time_bw_requests* or *time_bw_scans* are not between their corresponding limits.

See also:

`XBeeNetwork.get_deep_discovery_timeouts()`
`XBeeNetwork.start_discovery_process()`

set_discovery_options(*options*)

Configures the discovery options (*NO* parameter) with the given value.

Parameters **options** (Set of *DiscoveryOptions*) – New discovery options, empty set to clear the options.

Raises

- `ValueError` – If *options* is *None*.
- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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

- `ValueError` – If *discovery_timeout* is not between the allowed minimum and maximum values.
- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

start_discovery_process(*deep=False*, *n_deep_scans=1*)

Starts the discovery process. This method is not blocking.

This process can discover node neighbors and connections, or only nodes:

- Deep discovery: Network nodes and connections between them (including quality) are discovered.

  The discovery process will be running the number of scans configured in *n_deep_scans*. A scan is considered the process of discovering the full network. If there are more than one number of scans configured, after finishing one another is started, until *n_deep_scans* is satisfied.

  See *set_deep_discovery_options()* to establish the way the network discovery process is performed.

- No deep discovery: Only network nodes are discovered.

The discovery process will be running until the configured timeout expires or, in case of 802.15.4, until the 'end' packet is read.

It may occur that, after timeout expiration, there are nodes that continue sending discovery responses to the local XBee. In this case, these nodes will not be added to the network.

In 802.15.4, both (deep and no deep discovery) are the same and none discover the node connections or their quality. The difference is the possibility of running more than one scan using a deep discovery.

Parameters

- **`deep`** (*`Boolean, optional, default=`False``*) – *True* for a deep network scan, looking for neighbors and their connections, *False* otherwise.
- **`n_deep_scans`** (*`Integer, optional, default=1`*) – Number of scans to perform before automatically stopping the discovery process. `SCAN_TIL_CANCEL` means the process will not be automatically stopped. Only applicable if *deep=True*.

See also:

`XBeeNetwork.add_device_discovered_callback()`
`XBeeNetwork.add_discovery_process_finished_callback()`
`XBeeNetwork.del_device_discovered_callback()`
`XBeeNetwork.del_discovery_process_finished_callback()`
`XBeeNetwork.get_deep_discovery_options()`
`XBeeNetwork.set_deep_discovery_options()`
`XBeeNetwork.get_deep_discovery_timeouts()`
`XBeeNetwork.set_deep_discovery_timeouts()`
`XBeeNetwork.get_discovery_options()`
`XBeeNetwork.set_discovery_options()`
`XBeeNetwork.get_discovery_timeout()`
`XBeeNetwork.set_discovery_timeout()`

`stop_discovery_process``()`

Stops the discovery process if it is running.

Note that some DigiMesh/DigiPoint devices are blocked until the discovery time configured ('NT' parameter) has elapsed, so, when trying to get/set any parameter during the discovery process, a TimeoutException is raised.

`class` `digi.xbee.devices.`**`Raw802Network`**`(`*xbee_device*`)`

Bases: *`digi.xbee.devices.XBeeNetwork`*

This class represents an 802.15.4 network.

The network allows the discovery of remote nodes in the same network as the local one and stores them.

Class constructor. Instantiates a new *XBeeNetwork*.

Parameters **`xbee_device`** (*`XBeeDevice`*) – Local XBee to get the network from.

Raises `ValueError` – If *xbee_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 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 two arguments.

- The event code as an *NetworkDiscoveryStatus*.
- (Optional) A description of the discovery process as a string.

See also:

XBeeNetwork.del_discovery_process_finished_callback()
XBeeNetwork.add_device_discovered_callback()
XBeeNetwork.del_device_discovered_callback()

add_end_discovery_scan_callback(*callback*)
Adds a callback for the event *EndDiscoveryScan*.

Parameters **callback** (*Function*) – The callback. Receives two arguments.

- Number of scan that has finished (starting with 1).
- Total number of scans.

See also:

XBeeNetwork.del_end_discovery_scan_callback()

add_if_not_exist(*x64bit_addr=None*, *x16bit_addr=None*, *node_id=None*)
Adds an XBee with the provided information if it does not exist in the current network.

If the XBee already exists, its data is updated with the provided information.

If no valid address is provided (*x64bit_addr*, *x16bit_addr*), *None* is returned.

Parameters

- **x64bit_addr** (XBee64BitAddress, optional, default='None') – 64-bit address.
- **x16bit_addr** (XBee16BitAddress, optional, default='None') – 16-bit address.
- **node_id** (*String, optional, default=`None`*) – Node identifier.

Returns

the remote XBee with the updated information. If the XBee was not in the list yet, this method returns the given XBee without changes.

Return type *AbstractXBeeDevice*

add_init_discovery_scan_callback(*callback*)

Adds a callback for the event *InitDiscoveryScan*.

Parameters **callback** (*Function*) – The callback. Receives two arguments.

- Number of scan to start (starting with 1).
- Total number of scans.

See also:

XBeeNetwork.del_init_discovery_scan_callback()

add_network_modified_callback(*callback*)

Adds a callback for the event *NetworkModified*.

Parameters **callback** (*Function*) – The callback. Receives three arguments.

- The event type as a *NetworkEventType*.
- The reason of the event as a *NetworkEventReason*.
- The node added, updated or removed from the network as a *XBeeDevice* or *RemoteXBeeDevice*.

See also:

XBeeNetwork.del_network_modified_callback()

add_packet_received_from_callback(*node*, *callback*)

Adds a callback to listen to any received packet from the provided node.

Parameters

- **node** (*RemoteXBeeDevice*) – The node to listen for frames.
- **callback** (*Function*) – The callback. Receives two arguments.
  - The received packet as a *XBeeAPIPacket*.
  - The remote XBee who sent the packet as a *RemoteXBeeDevice*.

See also:

XBeeNetwork.del_packet_received_from_callback()

add_remote(*remote_xbee*)

Adds the provided remote XBee to the network if it is not in yet.

If the XBee is already in the network, its data is updated with the information of the provided XBee that are not *None*.

Parameters **remote_xbee** (*RemoteXBeeDevice*) – Remote XBee to add.

Returns

Provided XBee with updated data. If the XBee was not in the list, it returns it without changes.

Return type *RemoteXBeeDevice*

add_remotes(*remote_xbees*)

Adds a list of remote XBee nodes to the network.

If any node in the list is already in the network, its data is updated with the information of the corresponding XBee in the list.

Parameters **remote_xbees** (*List*) – List of *RemoteXBeeDevice* to add.

clear()

Removes all remote XBee nodes 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.

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.

See also:

XBeeNetwork.add_discovery_process_finished_callback()
XBeeNetwork.add_device_discovered_callback()
XBeeNetwork.del_device_discovered_callback()

del_end_discovery_scan_callback(*callback*)

Deletes a callback for the callback list of *EndDiscoveryScan*.

Parameters **callback** (*Function*) – The callback to delete.

See also:

XBeeNetwork.add_end_discovery_scan_callback()

del_init_discovery_scan_callback(*callback*)

Deletes a callback for the callback list of *InitDiscoveryScan*.

Parameters **callback** (*Function*) – The callback to delete.

See also:

XBeeNetwork.add_init_discovery_scan_callback()

del_network_modified_callback(*callback*)

Deletes a callback for the callback list of *NetworkModified*.

Parameters **callback** (*Function*) – The callback to delete.

See also:

XBeeNetwork.add_network_modified_callback()

del_packet_received_from_callback(*node*, *callb=None*)

Deletes a received packet callback from the provided node.

Parameters

- **node** (*RemoteXBeeDevice*) – The node to listen for frames.
- **callb** (*Function, optional, default=`None`*) – The callback to delete, *None* to delete all.

See also:

XBeeNetwork.add_packet_received_from_callback()

discover_device(*node_id*)

Blocking method. Discovers and reports the first remote XBee that matches the supplied identifier.

Parameters **node_id** (*String*) – Node identifier of the node to discover.

Returns

Discovered remote XBee, *None* if the timeout expires and the node was not found.

Return type *RemoteXBeeDevice*

See also:

XBeeNetwork.get_discovery_options()
XBeeNetwork.set_discovery_options()
XBeeNetwork.get_discovery_timeout()
XBeeNetwork.set_discovery_timeout()

discover_devices(*device_id_list*)

Blocking method. Attempts to discover a list of nodes and add them to the current network.

This method does not guarantee that all nodes of *device_id_list* will be found, even if they exist physically. This depends on the node discovery operation and timeout.

Parameters **device_id_list** (*List*) – List of device IDs to discover.

Returns

List with the discovered nodes. It may not contain all nodes specified in *device_id_list*.

Return type List

See also:

XBeeNetwork.get_discovery_options()
XBeeNetwork.set_discovery_options()
XBeeNetwork.get_discovery_timeout()
XBeeNetwork.set_discovery_timeout()

export (*dir_path=None*, *name=None*, *desc=None*)
Exports this network to the given file path.

If the provided path already exists the file is removed.

Params:

dir_path (String, optional, default='None'): Absolute path of the directory to export the network. It should not include the file name. If not defined home directory is used.

name (String, optional, default='None'): Network human readable name. desc (String, optional, default='None'): Network description.

Returns

Tuple with result (0: success, 1: failure) and string (exported file path if success, error string otherwise).

Return type Tuple (Integer, String)

get_connections ()
Returns a copy of the XBee network connections.

A deep discover must be performed to get the connections between network nodes.

If a new connection is added to the list after the execution of this method, this new connection is not added to the list returned by this method.

Returns A copy of the list of `Connection` for the network.

Return type List

See also:

XBeeNetwork.get_node_connections()
XBeeNetwork.start_discovery_process()

get_deep_discovery_options ()
Returns the deep discovery process options.

Returns

(*NeighborDiscoveryMode*, Boolean): Tuple containing:

- **mode (*NeighborDiscoveryMode*): Neighbor discovery** mode, the way to perform the network discovery process.
- **remove_nodes (Boolean): *True* to remove nodes from the** network if they were not discovered in the last scan, *False* otherwise.

Return type Tuple

See also:

digi.xbee.models.mode.NeighborDiscoveryMode

XBeeNetwork.set_deep_discovery_timeouts()

XBeeNetwork.start_discovery_process()

get_deep_discovery_timeouts()

Gets deep discovery network timeouts. These timeouts are only applicable for "deep" discovery (see *start_discovery_process()*)

Returns

Tuple containing:

- **node_timeout (Float): Maximum duration in seconds of the** discovery process per node. This is used to find neighbors of a node. This timeout is highly dependent on the nature of the network:
  - It should be greater than the highest 'NT' (Node Discovery Timeout) of your network.
  - And include enough time to let the message propagate depending on the sleep cycle of your network nodes.
- **time_bw_nodes (Float): Time to wait between node neighbors** requests. Use this setting not to saturate your network:
  - For 'Cascade', the number of seconds to wait after completion of the neighbor discovery process of the previous node.
  - For 'Flood', the minimum time to wait between each node's neighbor requests.
- **time_bw_scans (Float): Time to wait before starting a new** network scan.

Return type Tuple (Float, Float, Float)

See also:

XBeeNetwork.set_deep_discovery_timeouts()

XBeeNetwork.start_discovery_process()

get_device_by_16(*x16bit_addr*)

Returns the XBee in the network whose 16-bit address matches the given one.

Parameters **x16bit_addr** (XBee16BitAddress) – 16-bit address of the node to retrieve.

Returns XBee in the network or *Non* if not found.

Return type *AbstractXBeeDevice*

Raises `ValueError` – If *x16bit_addr* is *None* or unknown.

get_device_by_64(*x64bit_addr*)
Returns the XBee in the network whose 64-bit address matches the given one.

Parameters **x64bit_addr** (`XBee64BitAddress`) – 64-bit address of the node to retrieve.

Returns XBee in the network or *None* if not found.

Return type *AbstractXBeeDevice*

Raises `ValueError` – If *x64bit_addr* is *None* or unknown.

get_device_by_node_id(*node_id*)
Returns the XBee in the network whose node identifier matches the given one.

Parameters **node_id** (*String*) – Node identifier of the node to retrieve.

Returns XBee in the network or *None* if not found.

Return type *AbstractXBeeDevice*

Raises `ValueError` – If *node_id* is *None*.

get_devices()
Returns a copy of the XBee devices list of the network.

If a new XBee node is added to the list after the execution of this method, this new XBee is not 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 discovered.

Returns

Callback for generic devices discovery process, callback for discovery specific XBee ops.

Return type Tuple (Function, Function)

get_discovery_options()
Returns the network discovery process options.

Returns Discovery options value.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

get_discovery_timeout()

Returns the network discovery timeout.

Returns Network discovery timeout.

Return type Float

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

get_node_connections(*node*)

Returns the network connections with one of their ends *node*.

A deep discover must be performed to get the connections between network nodes.

If a new connection is added to the list after the execution of this method, this new connection is not added to the list returned by this method.

Parameters **node** (*AbstractXBeeDevice*) – The node to get its connections.

Returns List of `Connection` with *node* end.

Return type List

See also:

XBeeNetwork.get_connections()

XBeeNetwork.start_discovery_process()

classmethod get_nt_limits(*protocol*)

Returns a tuple with the minimum and maximum values for the 'NT' value depending on the protocol.

Returns

Minimum value in seconds, maximum value in seconds.

Return type Tuple (Float, Float)

get_number_devices()

Returns the number of nodes in the network.

Returns Number of nodes in the network.

Return type Integer

has_devices()

Returns whether there is any device in the network.

Returns

***True* if there is at least one node in the network,** *False* otherwise.

Return type Boolean

is_discovery_running()

Returns whether the discovery process is running.

Returns *True* if the discovery process is running, *False* otherwise.

Return type Boolean

is_node_in_network(*node*)

Checks if the provided node is in the network or if it is the local XBee.

Parameters **node** (*AbstractXBeeDevice*) – The node to check.

Returns *True* if the node is in the network, *False* otherwise.

Return type Boolean

Raises `ValueError` – If *node* is *None*.

remove_device(*remote_xbee*)

Removes the provided remote XBee from the network.

Parameters **remote_xbee** (*RemoteXBeeDevice*) – Remote XBee to remove.

Raises `ValueError` – If the provided *remote_xbee* is not in the network.

scan_counter

Returns the scan counter.

Returns The scan counter.

Return type Integer

set_deep_discovery_options(*deep_mode=<NeighborDiscoveryMode.CASCADE: (0, 'Cascade')>*, *del_not_discovered_nodes_in_last_scan=False*)

Configures the deep discovery options with the given values. These options are only applicable for "deep" discovery (see `start_discovery_process()`)

Parameters

- **deep_mode** (*NeighborDiscoveryMode*, optional, default='NeighborDiscoveryMode.CASCADE') – Neighbor discovery mode, the way to perform the network discovery process.
- **del_not_discovered_nodes_in_last_scan** (*Boolean, optional, default=`False`*) – *True* to remove nodes from the network if they were not discovered in the last scan.

See also:

digi.xbee.models.mode.NeighborDiscoveryMode

XBeeNetwork.get_deep_discovery_timeouts()

XBeeNetwork.start_discovery_process()

set_deep_discovery_timeouts(*node_timeout=None*, *time_bw_requests=None*, *time_bw_scans=None*)

Sets deep discovery network timeouts. These timeouts are only applicable for "deep" discovery (see `start_discovery_process()`)

node_timeout (Float, optional, default='None'): Maximum duration in seconds of the discovery process used to find neighbors of a node. If *None* already configured timeouts are used.

time_bw_requests (Float, optional, default='DEFAULT_TIME_BETWEEN_REQUESTS'): Time to wait between node neighbors requests. It must be between `MIN_TIME_BETWEEN_REQUESTS` and `MAX_TIME_BETWEEN_REQUESTS` seconds inclusive. Use this setting not to saturate your network:

- For 'Cascade', the number of seconds to wait after completion of the neighbor discovery process of the previous node.
- For 'Flood', the minimum time to wait between each node's neighbor requests.

time_bw_scans (Float, optional, default='DEFAULT_TIME_BETWEEN_SCANS'): Time to wait before starting a new network scan. It must be between `MIN_TIME_BETWEEN_SCANS` and `MAX_TIME_BETWEEN_SCANS` seconds inclusive.

Raises `ValueError` – if *node_timeout*, *time_bw_requests* or *time_bw_scans* are not between their corresponding limits.

See also:

`XBeeNetwork.get_deep_discovery_timeouts()`

`XBeeNetwork.start_discovery_process()`

`set_discovery_options`(*options*)

Configures the discovery options (*NO* parameter) with the given value.

Parameters **`options`** (Set of *`DiscoveryOptions`*) – New discovery options, empty set to clear the options.

Raises

- `ValueError` – If *options* is *None*.
- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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

- `ValueError` – If *discovery_timeout* is not between the allowed minimum and maximum values.

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

`start_discovery_process`(*deep=False*, *n_deep_scans=1*)

Starts the discovery process. This method is not blocking.

This process can discover node neighbors and connections, or only nodes:

- Deep discovery: Network nodes and connections between them (including quality) are discovered.

  The discovery process will be running the number of scans configured in *n_deep_scans*. A scan is considered the process of discovering the full network. If there are more than one number of scans configured, after finishing one another is started, until *n_deep_scans* is satisfied.

  See *`set_deep_discovery_options()`* to establish the way the network discovery process is performed.

- No deep discovery: Only network nodes are discovered.

  The discovery process will be running until the configured timeout expires or, in case of 802.15.4, until the 'end' packet is read.

  It may occur that, after timeout expiration, there are nodes that continue sending discovery responses to the local XBee. In this case, these nodes will not be added to the network.

In 802.15.4, both (deep and no deep discovery) are the same and none discover the node connections or their quality. The difference is the possibility of running more than one scan using a deep discovery.

Parameters

- **deep** (*Boolean, optional, default=`False`*) – *True* for a deep network scan, looking for neighbors and their connections, *False* otherwise.
- **n_deep_scans** (*Integer, optional, default=1*) – Number of scans to perform before automatically stopping the discovery process. `SCAN_TIL_CANCEL` means the process will not be automatically stopped. Only applicable if *deep=True*.

See also:

`XBeeNetwork.add_device_discovered_callback()`

`XBeeNetwork.add_discovery_process_finished_callback()`

`XBeeNetwork.del_device_discovered_callback()`

`XBeeNetwork.del_discovery_process_finished_callback()`

`XBeeNetwork.get_deep_discovery_options()`

`XBeeNetwork.set_deep_discovery_options()`

`XBeeNetwork.get_deep_discovery_timeouts()`

`XBeeNetwork.set_deep_discovery_timeouts()`

`XBeeNetwork.get_discovery_options()`

`XBeeNetwork.set_discovery_options()`

`XBeeNetwork.get_discovery_timeout()`

XBeeNetwork.set_discovery_timeout()

stop_discovery_process()

Stops the discovery process if it is running.

Note that some DigiMesh/DigiPoint devices are blocked until the discovery time configured ('NT' parameter) has elapsed, so, when trying to get/set any parameter during the discovery process, a TimeoutException is raised.

class digi.xbee.devices.**DigiMeshNetwork**(*device*)

Bases: *digi.xbee.devices.XBeeNetwork*

This class represents a DigiMesh network.

The network allows the discovery of remote nodes in the same network as the local one and stores them.

Class constructor. Instantiates a new *DigiMeshNetwork*.

Parameters **device** (*DigiMeshDevice*) – Local DigiMesh node 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 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 two arguments.

- The event code as an *NetworkDiscoveryStatus*.
- (Optional) A description of the discovery process as a string.

See also:

XBeeNetwork.del_discovery_process_finished_callback()
XBeeNetwork.add_device_discovered_callback()
XBeeNetwork.del_device_discovered_callback()

add_end_discovery_scan_callback(*callback*)

Adds a callback for the event *EndDiscoveryScan*.

Parameters **callback** (*Function*) – The callback. Receives two arguments.

- Number of scan that has finished (starting with 1).
- Total number of scans.

See also:

XBeeNetwork.del_end_discovery_scan_callback()

add_if_not_exist(*x64bit_addr=None*, *x16bit_addr=None*, *node_id=None*)

Adds an XBee with the provided information if it does not exist in the current network.

If the XBee already exists, its data is updated with the provided information.

If no valid address is provided (*x64bit_addr*, *x16bit_addr*), *None* is returned.

Parameters

- **x64bit_addr** (`XBee64BitAddress`, optional, default='None') – 64-bit address.
- **x16bit_addr** (`XBee16BitAddress`, optional, default='None') – 16-bit address.
- **node_id** (*String, optional, default=`None`*) – Node identifier.

Returns

the remote XBee with the updated information. If the XBee was not in the list yet, this method returns the given XBee without changes.

Return type *AbstractXBeeDevice*

add_init_discovery_scan_callback(*callback*)

Adds a callback for the event *InitDiscoveryScan*.

Parameters **callback** (*Function*) – The callback. Receives two arguments.

- Number of scan to start (starting with 1).
- Total number of scans.

See also:

XBeeNetwork.del_init_discovery_scan_callback()

add_network_modified_callback(*callback*)

Adds a callback for the event *NetworkModified*.

Parameters **callback** (*Function*) – The callback. Receives three arguments.

- The event type as a *NetworkEventType*.
- The reason of the event as a *NetworkEventReason*.
- The node added, updated or removed from the network as a *XBeeDevice* or *RemoteXBeeDevice*.

See also:

XBeeNetwork.del_network_modified_callback()

`add_packet_received_from_callback`(*node*, *callback*)

Adds a callback to listen to any received packet from the provided node.

Parameters

- **`node`** (`RemoteXBeeDevice`) – The node to listen for frames.
- **`callback`** (`Function`) – The callback. Receives two arguments.
  - The received packet as a `XBeeAPIPacket`.
  - The remote XBee who sent the packet as a `RemoteXBeeDevice`.

See also:

`XBeeNetwork.del_packet_received_from_callback()`

`add_remote`(*remote_xbee*)

Adds the provided remote XBee to the network if it is not in yet.

If the XBee is already in the network, its data is updated with the information of the provided XBee that are not *None*.

Parameters **`remote_xbee`** (`RemoteXBeeDevice`) – Remote XBee to add.

Returns

Provided XBee with updated data. If the XBee was not in the list, it returns it without changes.

Return type `RemoteXBeeDevice`

`add_remotes`(*remote_xbees*)

Adds a list of remote XBee nodes to the network.

If any node in the list is already in the network, its data is updated with the information of the corresponding XBee in the list.

Parameters **`remote_xbees`** (`List`) – List of `RemoteXBeeDevice` to add.

`clear`()

Removes all remote XBee nodes 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.

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.

See also:

XBeeNetwork.add_discovery_process_finished_callback()
XBeeNetwork.add_device_discovered_callback()
XBeeNetwork.del_device_discovered_callback()

del_end_discovery_scan_callback(*callback*)
Deletes a callback for the callback list of *EndDiscoveryScan*.

Parameters **callback** (*Function*) – The callback to delete.

See also:

XBeeNetwork.add_end_discovery_scan_callback()

del_init_discovery_scan_callback(*callback*)
Deletes a callback for the callback list of *InitDiscoveryScan*.

Parameters **callback** (*Function*) – The callback to delete.

See also:

XBeeNetwork.add_init_discovery_scan_callback()

del_network_modified_callback(*callback*)
Deletes a callback for the callback list of *NetworkModified*.

Parameters **callback** (*Function*) – The callback to delete.

See also:

XBeeNetwork.add_network_modified_callback()

del_packet_received_from_callback(*node*, *callb=None*)
Deletes a received packet callback from the provided node.

Parameters

- **node** (*RemoteXBeeDevice*) – The node to listen for frames.
- **callb** (*Function, optional, default=`None`*) – The callback to delete, *None* to delete all.

See also:

XBeeNetwork.add_packet_received_from_callback()

discover_device(*node_id*)
Blocking method. Discovers and reports the first remote XBee that matches the supplied identifier.

Parameters **node_id** (*String*) – Node identifier of the node to discover.

Returns

Discovered remote XBee, *None* if the timeout expires and the node was not found.

Return type `RemoteXBeeDevice`

See also:

`XBeeNetwork.get_discovery_options()`
`XBeeNetwork.set_discovery_options()`
`XBeeNetwork.get_discovery_timeout()`
`XBeeNetwork.set_discovery_timeout()`

`discover_devices`(*device_id_list*)

Blocking method. Attempts to discover a list of nodes and add them to the current network.

This method does not guarantee that all nodes of *device_id_list* will be found, even if they exist physically. This depends on the node discovery operation and timeout.

Parameters **`device_id_list`** (*`List`*) – List of device IDs to discover.

Returns

List with the discovered nodes. It may not contain all nodes specified in *device_id_list*.

Return type List

See also:

`XBeeNetwork.get_discovery_options()`
`XBeeNetwork.set_discovery_options()`
`XBeeNetwork.get_discovery_timeout()`
`XBeeNetwork.set_discovery_timeout()`

`export`(*dir_path=None*, *name=None*, *desc=None*)

Exports this network to the given file path.

If the provided path already exists the file is removed.

Params:

dir_path (String, optional, default=‘None‘): Absolute path of the directory to export the network. It should not include the file name. If not defined home directory is used.

name (String, optional, default=‘None‘): Network human readable name. desc (String, optional, default=‘None‘): Network description.

Returns

Tuple with result (0: success, 1: failure) and string (exported file path if success, error string otherwise).

Return type Tuple (Integer, String)

`get_connections`()

Returns a copy of the XBee network connections.

A deep discover must be performed to get the connections between network nodes.

If a new connection is added to the list after the execution of this method, this new connection is not added to the list returned by this method.

Returns A copy of the list of `Connection` for the network.

Return type List

See also:

`XBeeNetwork.get_node_connections()`

`XBeeNetwork.start_discovery_process()`

`get_deep_discovery_options``()`

Returns the deep discovery process options.

Returns

(`NeighborDiscoveryMode`, Boolean): Tuple containing:

- **mode (`NeighborDiscoveryMode`): Neighbor discovery** mode, the way to perform the network discovery process.
- **remove_nodes (Boolean): *True* to remove nodes from the** network if they were not discovered in the last scan, *False* otherwise.

Return type Tuple

See also:

`digi.xbee.models.mode.NeighborDiscoveryMode`

`XBeeNetwork.set_deep_discovery_timeouts()`

`XBeeNetwork.start_discovery_process()`

`get_deep_discovery_timeouts``()`

Gets deep discovery network timeouts. These timeouts are only applicable for "deep" discovery (see *`start_discovery_process()`*)

Returns

Tuple containing:

- **node_timeout (Float): Maximum duration in seconds of the** discovery process per node. This is used to find neighbors of a node. This timeout is highly dependent on the nature of the network:
  - It should be greater than the highest 'NT' (Node Discovery Timeout) of your network.
  - And include enough time to let the message propagate depending on the sleep cycle of your network nodes.
- **time_bw_nodes (Float): Time to wait between node neighbors** requests. Use this setting not to saturate your network:
  - For 'Cascade', the number of seconds to wait after completion of the neighbor discovery process of the previous node.

  – For 'Flood', the minimum time to wait between each node's neighbor requests.

- **time_bw_scans (Float): Time to wait before starting a new** network scan.

Return type Tuple (Float, Float, Float)

See also:

XBeeNetwork.set_deep_discovery_timeouts()
XBeeNetwork.start_discovery_process()

get_device_by_16(*x16bit_addr*)
Returns the XBee in the network whose 16-bit address matches the given one.

Parameters **x16bit_addr** (XBee16BitAddress) – 16-bit address of the node to retrieve.

Returns XBee in the network or *Non* if not found.

Return type *AbstractXBeeDevice*

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

get_device_by_64(*x64bit_addr*)
Returns the XBee in the network whose 64-bit address matches the given one.

Parameters **x64bit_addr** (XBee64BitAddress) – 64-bit address of the node to retrieve.

Returns XBee in the network or *None* if not found.

Return type *AbstractXBeeDevice*

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

get_device_by_node_id(*node_id*)
Returns the XBee in the network whose node identifier matches the given one.

Parameters **node_id** (*String*) – Node identifier of the node to retrieve.

Returns XBee in the network or *None* if not found.

Return type *AbstractXBeeDevice*

Raises ValueError – If *node_id* is *None*.

get_devices()
Returns a copy of the XBee devices list of the network.

If a new XBee node is added to the list after the execution of this method, this new XBee is not 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 discovered.

Returns

Callback for generic devices discovery process, callback for discovery specific XBee ops.

Return type Tuple (Function, Function)

get_discovery_options()

Returns the network discovery process options.

Returns Discovery options value.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

get_discovery_timeout()

Returns the network discovery timeout.

Returns Network discovery timeout.

Return type Float

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

get_node_connections(*node*)

Returns the network connections with one of their ends *node*.

A deep discover must be performed to get the connections between network nodes.

If a new connection is added to the list after the execution of this method, this new connection is not added to the list returned by this method.

Parameters **node** (*AbstractXBeeDevice*) – The node to get its connections.

Returns List of *Connection* with *node* end.

Return type List

See also:

XBeeNetwork.get_connections()

XBeeNetwork.start_discovery_process()

classmethod get_nt_limits(*protocol*)
Returns a tuple with the minimum and maximum values for the 'NT' value depending on the protocol.

Returns

Minimum value in seconds, maximum value in seconds.

Return type Tuple (Float, Float)

get_number_devices()
Returns the number of nodes in the network.

Returns Number of nodes in the network.

Return type Integer

has_devices()
Returns whether there is any device in the network.

Returns

***True* if there is at least one node in the network,** *False* otherwise.

Return type Boolean

is_discovery_running()
Returns whether the discovery process is running.

Returns *True* if the discovery process is running, *False* otherwise.

Return type Boolean

is_node_in_network(*node*)
Checks if the provided node is in the network or if it is the local XBee.

Parameters **node** (*AbstractXBeeDevice*) – The node to check.

Returns *True* if the node is in the network, *False* otherwise.

Return type Boolean

Raises `ValueError` – If *node* is *None*.

remove_device(*remote_xbee*)
Removes the provided remote XBee from the network.

Parameters **remote_xbee** (*RemoteXBeeDevice*) – Remote XBee to remove.

Raises `ValueError` – If the provided *remote_xbee* is not in the network.

scan_counter
Returns the scan counter.

Returns The scan counter.

Return type Integer

set_deep_discovery_options(*deep_mode=<NeighborDiscoveryMode.CASCADE: (0, 'Cascade')>, del_not_discovered_nodes_in_last_scan=False*)
Configures the deep discovery options with the given values. These options are only applicable for "deep" discovery (see `start_discovery_process()`)

Parameters

- **deep_mode** (*NeighborDiscoveryMode*, optional, default='NeighborDiscoveryMode.CASCADE') – Neighbor discovery mode, the way to perform the network discovery process.

- **del_not_discovered_nodes_in_last_scan** (*Boolean, optional, default=`False`*) – *True* to remove nodes from the network if they were not discovered in the last scan.

See also:

digi.xbee.models.mode.NeighborDiscoveryMode
XBeeNetwork.get_deep_discovery_timeouts()
XBeeNetwork.start_discovery_process()

set_deep_discovery_timeouts(*node_timeout=None*, *time_bw_requests=None*, *time_bw_scans=None*)
Sets deep discovery network timeouts. These timeouts are only applicable for "deep" discovery (see *start_discovery_process()*)

node_timeout (Float, optional, default='None'): Maximum duration in seconds of the discovery process used to find neighbors of a node. If *None* already configured timeouts are used.

time_bw_requests (Float, optional, default='DEFAULT_TIME_BETWEEN_REQUESTS'): Time to wait between node neighbors requests. It must be between `MIN_TIME_BETWEEN_REQUESTS` and `MAX_TIME_BETWEEN_REQUESTS` seconds inclusive. Use this setting not to saturate your network:

- For 'Cascade', the number of seconds to wait after completion of the neighbor discovery process of the previous node.
- For 'Flood', the minimum time to wait between each node's neighbor requests.

time_bw_scans (Float, optional, default='DEFAULT_TIME_BETWEEN_SCANS'): Time to wait before starting a new network scan. It must be between `MIN_TIME_BETWEEN_SCANS` and `MAX_TIME_BETWEEN_SCANS` seconds inclusive.

Raises `ValueError` – if *node_timeout*, *time_bw_requests* or *time_bw_scans* are not between their corresponding limits.

See also:

XBeeNetwork.get_deep_discovery_timeouts()
XBeeNetwork.start_discovery_process()

set_discovery_options(*options*)
Configures the discovery options (*NO* parameter) with the given value.

Parameters **options** (Set of *DiscoveryOptions*) – New discovery options, empty set to clear the options.

Raises

- `ValueError` – If *options* is *None*.
- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.

- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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

- `ValueError` – If *discovery_timeout* is not between the allowed minimum and maximum values.
- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

`start_discovery_process`(*deep=False*, *n_deep_scans=1*)

Starts the discovery process. This method is not blocking.

This process can discover node neighbors and connections, or only nodes:

- Deep discovery: Network nodes and connections between them (including quality) are discovered.

  The discovery process will be running the number of scans configured in *n_deep_scans*. A scan is considered the process of discovering the full network. If there are more than one number of scans configured, after finishing one another is started, until *n_deep_scans* is satisfied.

  See *`set_deep_discovery_options()`* to establish the way the network discovery process is performed.

- No deep discovery: Only network nodes are discovered.

  The discovery process will be running until the configured timeout expires or, in case of 802.15.4, until the 'end' packet is read.

  It may occur that, after timeout expiration, there are nodes that continue sending discovery responses to the local XBee. In this case, these nodes will not be added to the network.

In 802.15.4, both (deep and no deep discovery) are the same and none discover the node connections or their quality. The difference is the possibility of running more than one scan using a deep discovery.

Parameters

- **`deep`** (*`Boolean, optional, default=`False``*) – *True* for a deep network scan, looking for neighbors and their connections, *False* otherwise.

- **n_deep_scans** (*Integer, optional, default=1*) – Number of scans to perform before automatically stopping the discovery process. SCAN_TIL_CANCEL means the process will not be automatically stopped. Only applicable if *deep=True*.

See also:

XBeeNetwork.add_device_discovered_callback()
XBeeNetwork.add_discovery_process_finished_callback()
XBeeNetwork.del_device_discovered_callback()
XBeeNetwork.del_discovery_process_finished_callback()
XBeeNetwork.get_deep_discovery_options()
XBeeNetwork.set_deep_discovery_options()
XBeeNetwork.get_deep_discovery_timeouts()
XBeeNetwork.set_deep_discovery_timeouts()
XBeeNetwork.get_discovery_options()
XBeeNetwork.set_discovery_options()
XBeeNetwork.get_discovery_timeout()
XBeeNetwork.set_discovery_timeout()

stop_discovery_process()
Stops the discovery process if it is running.

Note that some DigiMesh/DigiPoint devices are blocked until the discovery time configured ('NT' parameter) has elapsed, so, when trying to get/set any parameter during the discovery process, a TimeoutException is raised.

class digi.xbee.devices.**DigiPointNetwork**(*xbee_device*)
Bases: *digi.xbee.devices.XBeeNetwork*

This class represents a DigiPoint network.

The network allows the discovery of remote nodes in the same network as the local one and stores them.

Class constructor. Instantiates a new *XBeeNetwork*.

Parameters **xbee_device** (*XBeeDevice*) – Local XBee to get the network from.

Raises ValueError – If *xbee_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 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 two arguments.

- The event code as an *`NetworkDiscoveryStatus`*.
- (Optional) A description of the discovery process as a string.

See also:

`XBeeNetwork.del_discovery_process_finished_callback()`
`XBeeNetwork.add_device_discovered_callback()`
`XBeeNetwork.del_device_discovered_callback()`

`add_end_discovery_scan_callback`(*callback*)
Adds a callback for the event *`EndDiscoveryScan`*.

Parameters **`callback`** (*`Function`*) – The callback. Receives two arguments.

- Number of scan that has finished (starting with 1).
- Total number of scans.

See also:

`XBeeNetwork.del_end_discovery_scan_callback()`

`add_if_not_exist`(*x64bit_addr=None*, *x16bit_addr=None*, *node_id=None*)
Adds an XBee with the provided information if it does not exist in the current network.

If the XBee already exists, its data is updated with the provided information.

If no valid address is provided (*x64bit_addr*, *x16bit_addr*), *None* is returned.

Parameters

- **`x64bit_addr`** (`XBee64BitAddress`, optional, default='None') – 64-bit address.
- **`x16bit_addr`** (`XBee16BitAddress`, optional, default='None') – 16-bit address.
- **`node_id`** (*`String, optional, default=`None``*) – Node identifier.

Returns

the remote XBee with the updated information. If the XBee was not in the list yet, this method returns the given XBee without changes.

Return type *`AbstractXBeeDevice`*

`add_init_discovery_scan_callback`(*callback*)
Adds a callback for the event *`InitDiscoveryScan`*.

Parameters **`callback`** (*`Function`*) – The callback. Receives two arguments.

- Number of scan to start (starting with 1).
- Total number of scans.

See also:

XBeeNetwork.del_init_discovery_scan_callback()

add_network_modified_callback(*callback*)

Adds a callback for the event *NetworkModified*.

Parameters **callback** (*Function*) – The callback. Receives three arguments.

- The event type as a *NetworkEventType*.
- The reason of the event as a *NetworkEventReason*.
- The node added, updated or removed from the network as a *XBeeDevice* or *RemoteXBeeDevice*.

See also:

XBeeNetwork.del_network_modified_callback()

add_packet_received_from_callback(*node*, *callback*)

Adds a callback to listen to any received packet from the provided node.

Parameters

- **node** (*RemoteXBeeDevice*) – The node to listen for frames.
- **callback** (*Function*) – The callback. Receives two arguments.
  - The received packet as a *XBeeAPIPacket*.
  - The remote XBee who sent the packet as a *RemoteXBeeDevice*.

See also:

XBeeNetwork.del_packet_received_from_callback()

add_remote(*remote_xbee*)

Adds the provided remote XBee to the network if it is not in yet.

If the XBee is already in the network, its data is updated with the information of the provided XBee that are not *None*.

Parameters **remote_xbee** (*RemoteXBeeDevice*) – Remote XBee to add.

Returns

Provided XBee with updated data. If the XBee was not in the list, it returns it without changes.

Return type *RemoteXBeeDevice*

add_remotes(*remote_xbees*)

Adds a list of remote XBee nodes to the network.

If any node in the list is already in the network, its data is updated with the information of the corresponding XBee in the list.

Parameters **remote_xbees** (*List*) – List of *RemoteXBeeDevice* to add.

clear()
Removes all remote XBee nodes 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.

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.

See also:

XBeeNetwork.add_discovery_process_finished_callback()
XBeeNetwork.add_device_discovered_callback()
XBeeNetwork.del_device_discovered_callback()

del_end_discovery_scan_callback(*callback*)
Deletes a callback for the callback list of *EndDiscoveryScan*.

Parameters **callback** (*Function*) – The callback to delete.

See also:

XBeeNetwork.add_end_discovery_scan_callback()

del_init_discovery_scan_callback(*callback*)
Deletes a callback for the callback list of *InitDiscoveryScan*.

Parameters **callback** (*Function*) – The callback to delete.

See also:

XBeeNetwork.add_init_discovery_scan_callback()

del_network_modified_callback(*callback*)
Deletes a callback for the callback list of *NetworkModified*.

Parameters **callback** (*Function*) – The callback to delete.

See also:

XBeeNetwork.add_network_modified_callback()

del_packet_received_from_callback(*node*, *callb=None*)
Deletes a received packet callback from the provided node.

Parameters

- **node** (*RemoteXBeeDevice*) – The node to listen for frames.
- **callb** (*Function, optional, default=`None`*) – The callback to delete, *None* to delete all.

See also:

XBeeNetwork.add_packet_received_from_callback()

discover_device(*node_id*)
Blocking method. Discovers and reports the first remote XBee that matches the supplied identifier.

Parameters **node_id** (*String*) – Node identifier of the node to discover.

Returns

Discovered remote XBee, *None* if the timeout expires and the node was not found.

Return type *RemoteXBeeDevice*

See also:

XBeeNetwork.get_discovery_options()
XBeeNetwork.set_discovery_options()
XBeeNetwork.get_discovery_timeout()
XBeeNetwork.set_discovery_timeout()

discover_devices(*device_id_list*)
Blocking method. Attempts to discover a list of nodes and add them to the current network.

This method does not guarantee that all nodes of *device_id_list* will be found, even if they exist physically. This depends on the node discovery operation and timeout.

Parameters **device_id_list** (*List*) – List of device IDs to discover.

Returns

List with the discovered nodes. It may not contain all nodes specified in *device_id_list*.

Return type List

See also:

XBeeNetwork.get_discovery_options()
XBeeNetwork.set_discovery_options()
XBeeNetwork.get_discovery_timeout()

XBeeNetwork.set_discovery_timeout()

export(*dir_path=None*, *name=None*, *desc=None*)

Exports this network to the given file path.

If the provided path already exists the file is removed.

Params:

dir_path (String, optional, default='None'): Absolute path of the directory to export the network. It should not include the file name. If not defined home directory is used.

name (String, optional, default='None'): Network human readable name. desc (String, optional, default='None'): Network description.

Returns

Tuple with result (0: success, 1: failure) and string (exported file path if success, error string otherwise).

Return type Tuple (Integer, String)

get_connections()

Returns a copy of the XBee network connections.

A deep discover must be performed to get the connections between network nodes.

If a new connection is added to the list after the execution of this method, this new connection is not added to the list returned by this method.

Returns A copy of the list of `Connection` for the network.

Return type List

See also:

XBeeNetwork.get_node_connections()
XBeeNetwork.start_discovery_process()

get_deep_discovery_options()

Returns the deep discovery process options.

Returns

(NeighborDiscoveryMode, Boolean): Tuple containing:

- **mode (NeighborDiscoveryMode): Neighbor discovery** mode, the way to perform the network discovery process.
- **remove_nodes (Boolean): *True* to remove nodes from the** network if they were not discovered in the last scan, *False* otherwise.

Return type Tuple

See also:

digi.xbee.models.mode.NeighborDiscoveryMode
XBeeNetwork.set_deep_discovery_timeouts()

XBeeNetwork.start_discovery_process()

get_deep_discovery_timeouts()

Gets deep discovery network timeouts. These timeouts are only applicable for "deep" discovery (see *start_discovery_process()*)

Returns

Tuple containing:

- **node_timeout (Float): Maximum duration in seconds of the** discovery process per node. This is used to find neighbors of a node. This timeout is highly dependent on the nature of the network:
  - It should be greater than the highest 'NT' (Node Discovery Timeout) of your network.
  - And include enough time to let the message propagate depending on the sleep cycle of your network nodes.
- **time_bw_nodes (Float): Time to wait between node neighbors** requests. Use this setting not to saturate your network:
  - For 'Cascade', the number of seconds to wait after completion of the neighbor discovery process of the previous node.
  - For 'Flood', the minimum time to wait between each node's neighbor requests.
- **time_bw_scans (Float): Time to wait before starting a new** network scan.

Return type Tuple (Float, Float, Float)

See also:

XBeeNetwork.set_deep_discovery_timeouts()
XBeeNetwork.start_discovery_process()

get_device_by_16(*x16bit_addr*)

Returns the XBee in the network whose 16-bit address matches the given one.

Parameters x16bit_addr (`XBee16BitAddress`) – 16-bit address of the node to retrieve.

Returns XBee in the network or *Non* if not found.

Return type *AbstractXBeeDevice*

Raises `ValueError` – If *x16bit_addr* is *None* or unknown.

get_device_by_64(*x64bit_addr*)

Returns the XBee in the network whose 64-bit address matches the given one.

Parameters x64bit_addr (`XBee64BitAddress`) – 64-bit address of the node to retrieve.

Returns XBee in the network or *None* if not found.

Return type *AbstractXBeeDevice*

Raises `ValueError` – If *x64bit_addr* is *None* or unknown.

get_device_by_node_id(*node_id*)

Returns the XBee in the network whose node identifier matches the given one.

Parameters **node_id** (*String*) – Node identifier of the node to retrieve.

Returns XBee in the network or *None* if not found.

Return type *AbstractXBeeDevice*

Raises `ValueError` – If *node_id* is *None*.

get_devices()

Returns a copy of the XBee devices list of the network.

If a new XBee node is added to the list after the execution of this method, this new XBee is not 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 discovered.

Returns

Callback for generic devices discovery process, callback for discovery specific XBee ops.

Return type Tuple (Function, Function)

get_discovery_options()

Returns the network discovery process options.

Returns Discovery options value.

Return type Bytearray

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

get_discovery_timeout()

Returns the network discovery timeout.

Returns Network discovery timeout.

Return type Float

Raises

- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.

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

`get_node_connections`(*node*)

Returns the network connections with one of their ends *node*.

A deep discover must be performed to get the connections between network nodes.

If a new connection is added to the list after the execution of this method, this new connection is not added to the list returned by this method.

Parameters **`node`** (*`AbstractXBeeDevice`*) – The node to get its connections.

Returns List of *`Connection`* with *node* end.

Return type List

See also:

`XBeeNetwork.get_connections()`

`XBeeNetwork.start_discovery_process()`

`classmethod get_nt_limits`(*protocol*)

Returns a tuple with the minimum and maximum values for the 'NT' value depending on the protocol.

Returns

Minimum value in seconds, maximum value in seconds.

Return type Tuple (Float, Float)

`get_number_devices`()

Returns the number of nodes in the network.

Returns Number of nodes in the network.

Return type Integer

`has_devices`()

Returns whether there is any device in the network.

Returns

***True* if there is at least one node in the network,** *False* otherwise.

Return type Boolean

`is_discovery_running`()

Returns whether the discovery process is running.

Returns *True* if the discovery process is running, *False* otherwise.

Return type Boolean

`is_node_in_network`(*node*)

Checks if the provided node is in the network or if it is the local XBee.

Parameters **`node`** (*`AbstractXBeeDevice`*) – The node to check.

Returns *True* if the node is in the network, *False* otherwise.

Return type Boolean

Raises `ValueError` – If *node* is *None*.

remove_device(*remote_xbee*)
Removes the provided remote XBee from the network.

Parameters **remote_xbee** (*RemoteXBeeDevice*) – Remote XBee to remove.

Raises `ValueError` – If the provided *remote_xbee* is not in the network.

scan_counter
Returns the scan counter.

Returns The scan counter.

Return type Integer

set_deep_discovery_options(*deep_mode=<NeighborDiscoveryMode.CASCADE: (0, 'Cascade')>*, *del_not_discovered_nodes_in_last_scan=False*)
Configures the deep discovery options with the given values. These options are only applicable for "deep" discovery (see *start_discovery_process()*)

Parameters

- **deep_mode** (*NeighborDiscoveryMode*, optional, default='NeighborDiscoveryMode.CASCADE') – Neighbor discovery mode, the way to perform the network discovery process.
- **del_not_discovered_nodes_in_last_scan** (*Boolean, optional, default=`False`*) – *True* to remove nodes from the network if they were not discovered in the last scan.

See also:

digi.xbee.models.mode.NeighborDiscoveryMode

XBeeNetwork.get_deep_discovery_timeouts()

XBeeNetwork.start_discovery_process()

set_deep_discovery_timeouts(*node_timeout=None*, *time_bw_requests=None*, *time_bw_scans=None*)
Sets deep discovery network timeouts. These timeouts are only applicable for "deep" discovery (see *start_discovery_process()*)

node_timeout (Float, optional, default='None'): Maximum duration in seconds of the discovery process used to find neighbors of a node. If *None* already configured timeouts are used.

time_bw_requests (Float, optional, default='DEFAULT_TIME_BETWEEN_REQUESTS'): Time to wait between node neighbors requests. It must be between `MIN_TIME_BETWEEN_REQUESTS` and `MAX_TIME_BETWEEN_REQUESTS` seconds inclusive. Use this setting not to saturate your network:

- For 'Cascade', the number of seconds to wait after completion of the neighbor discovery process of the previous node.
- For 'Flood', the minimum time to wait between each node's neighbor requests.

time_bw_scans (Float, optional, default='DEFAULT_TIME_BETWEEN_SCANS'): Time to wait before starting a new network scan. It must be between `MIN_TIME_BETWEEN_SCANS` and `MAX_TIME_BETWEEN_SCANS` seconds inclusive.

Raises `ValueError` – if *node_timeout*, *time_bw_requests* or *time_bw_scans* are not between their corresponding limits.

See also:

XBeeNetwork.get_deep_discovery_timeouts()
XBeeNetwork.start_discovery_process()

set_discovery_options(*options*)
Configures the discovery options (*NO* parameter) with the given value.

Parameters **options** (Set of *DiscoveryOptions*) – New discovery options, empty set to clear the options.

Raises

- `ValueError` – If *options* is *None*.
- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If 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

- `ValueError` – If *discovery_timeout* is not between the allowed minimum and maximum values.
- `TimeoutException` – If response is not received before the read timeout expires.
- `XBeeException` – If the XBee's communication interface is closed.
- `InvalidOperatingModeException` – If the XBee's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `ATCommandException` – If response is not as expected.

start_discovery_process(*deep=False*, *n_deep_scans=1*)
Starts the discovery process. This method is not blocking.

This process can discover node neighbors and connections, or only nodes:

- Deep discovery: Network nodes and connections between them (including quality) are discovered.

  The discovery process will be running the number of scans configured in *n_deep_scans*. A scan is considered the process of discovering the full network. If there are more than one number of scans configured, after finishing one another is started, until *n_deep_scans* is satisfied.

  See `set_deep_discovery_options()` to establish the way the network discovery process is performed.

- No deep discovery: Only network nodes are discovered.

  The discovery process will be running until the configured timeout expires or, in case of 802.15.4, until the 'end' packet is read.

  It may occur that, after timeout expiration, there are nodes that continue sending discovery responses to the local XBee. In this case, these nodes will not be added to the network.

In 802.15.4, both (deep and no deep discovery) are the same and none discover the node connections or their quality. The difference is the possibility of running more than one scan using a deep discovery.

Parameters

- **deep** (*Boolean, optional, default=`False`*) – *True* for a deep network scan, looking for neighbors and their connections, *False* otherwise.
- **n_deep_scans** (*Integer, optional, default=1*) – Number of scans to perform before automatically stopping the discovery process. `SCAN_TIL_CANCEL` means the process will not be automatically stopped. Only applicable if *deep=True*.

See also:

`XBeeNetwork.add_device_discovered_callback()`
`XBeeNetwork.add_discovery_process_finished_callback()`
`XBeeNetwork.del_device_discovered_callback()`
`XBeeNetwork.del_discovery_process_finished_callback()`
`XBeeNetwork.get_deep_discovery_options()`
`XBeeNetwork.set_deep_discovery_options()`
`XBeeNetwork.get_deep_discovery_timeouts()`
`XBeeNetwork.set_deep_discovery_timeouts()`
`XBeeNetwork.get_discovery_options()`
`XBeeNetwork.set_discovery_options()`
`XBeeNetwork.get_discovery_timeout()`
`XBeeNetwork.set_discovery_timeout()`

stop_discovery_process()

Stops the discovery process if it is running.

Note that some DigiMesh/DigiPoint devices are blocked until the discovery time configured ('NT' parameter) has elapsed, so, when trying to get/set any parameter during the discovery process, a TimeoutException is raised.

class digi.xbee.devices.**NetworkEventType**(*code*, *description*)

Bases: `enum.Enum`

Enumerates the different network event types.

Values:

NetworkEventType.ADD = (0, 'XBee added to the network')

NetworkEventType.DEL = (1, 'XBee removed from the network')

NetworkEventType.UPDATE = (2, 'XBee in the network updated')

NetworkEventType.CLEAR = (3, 'Network cleared')

`code`

Returns the code of the *NetworkEventType* element.

Returns Integer: Code of the *NetworkEventType* element.

`description`

Returns the description of the *NetworkEventType* element.

Returns Description of the *NetworkEventType* element.

Return type String

class `digi.xbee.devices.`**`NetworkEventReason`**(*code*, *description*)

Bases: `enum.Enum`

Enumerates the different network event reasons.

Values:

NetworkEventReason.DISCOVERED = (0, 'Discovered XBee')

NetworkEventReason.NEIGHBOR = (1, 'Discovered as XBee neighbor')

NetworkEventReason.RECEIVED_MSG = (2, 'Received message from XBee')

NetworkEventReason.MANUAL = (3, 'Manual modification')

NetworkEventReason.ROUTE = (4, 'Hop of a network route')

NetworkEventReason.READ_INFO = (5, 'Read XBee information')

NetworkEventReason.FIRMWARE_UPDATE = (6, 'The firmware of the device was updated')

`code`

Returns the code of the *NetworkEventReason* element.

Returns Code of the *NetworkEventReason* element.

Return type Integer

`description`

Returns the description of the *NetworkEventReason* element.

Returns Description of the *NetworkEventReason* element.

Return type String

class `digi.xbee.devices.`**`LinkQuality`**(*lq=None*, *is_rssi=False*)

Bases: `object`

This class represents the link quality of a connection. It can be a LQI (Link Quality Index) for Zigbee devices, or RSSI (Received Signal Strength Indicator) for the rest.

Class constructor. Instantiates a new *LinkQuality*.

Parameters

- **lq** (*Integer, optional, default=`UNKNOWN`*) – Link quality.
- **is_rssi** (*Boolean, optional, default=`False`*) – *True* to specify the value is a RSSI, *False* for LQI.

UNKNOWN = <digi.xbee.devices.LinkQuality object>
Unknown link quality.

UNKNOWN_VALUE = -9999
Unknown link quality value.

lq
Returns the link quality value.

Returns The link quality value.

Return type Integer

is_rssi
Returns whether this is a RSSI value.

Returns *True* if this is an RSSI value, *False* for LQI.

Return type Boolean

class digi.xbee.devices.**Connection**(*node_a, node_b, lq_a2b=None, lq_b2a=None, status_a2b=None, status_b2a=None*)
Bases: `object`

This class represents a generic connection between two nodes in a XBee network. It contains the source and destination nodes, the link quality of the connection between them and its status.

Class constructor. Instantiates a new *Connection*.

Parameters

- **node_a** (*AbstractXBeeDevice*) – One of the connection ends.
- **node_b** (*AbstractXBeeDevice*) – The other connection end.
- **lq_a2b** (*LinkQuality* or Integer, optional, default='None') – Link quality for the connection node_a -> node_b. If not specified *LinkQuality.UNKNOWN* is used.
- **lq_b2a** (*LinkQuality* or Integer, optional, default='None') – Link quality for the connection node_b -> node_a. If not specified *LinkQuality.UNKNOWN* is used.
- **status_a2b** (*digi.xbee.models.zdo.RouteStatus*, optional, default='None') – The status for the connection node_a -> node_b. If not specified *RouteStatus.UNKNOWN* is used.
- **status_b2a** (*digi.xbee.models.zdo.RouteStatus*, optional, default='None') – The status for the connection node_b -> node_a. If not specified *RouteStatus.UNKNOWN* is used.

Raises `ValueError` – If *node_a* or *node_b* is *None*.

See also:

AbstractXBeeDevice
LinkQuality
digi.xbee.models.zdo.RouteStatus

node_a
Returns the node A of this connection.

Returns The node A.

Return type *AbstractXBeeDevice*

See also:

AbstractXBeeDevice

node_b

Returns the node B of this connection.

Returns The node B.

Return type *AbstractXBeeDevice*

See also:

AbstractXBeeDevice

lq_a2b

Returns the link quality of the connection from node A to node B.

Returns Link quality for the connection A -> B.

Return type *LinkQuality*

See also:

LinkQuality

lq_b2a

Returns the link quality of the connection from node B to node A.

Returns Link quality for the connection B -> A.

Return type *LinkQuality*

See also:

LinkQuality

status_a2b

Returns the status of this connection from node A to node B.

Returns The status for A -> B connection.

Return type *RouteStatus*

See also:

digi.xbee.models.zdo.RouteStatus

status_b2a

Returns the status of this connection from node B to node A.

Returns The status for B -> A connection.

Return type *RouteStatus*

See also:

digi.xbee.models.zdo.RouteStatus

scan_counter_a2b

Returns the scan counter for this connection, discovered by its A node.

Returns The scan counter for this connection, discovered by its A node.

Return type Integer

scan_counter_b2a

Returns the scan counter for this connection, discovered by its B node.

Returns The scan counter for this connection, discovered by its B node.

Return type Integer

digi.xbee.exception module

exception digi.xbee.exception.**XBeeException**

Bases: Exception

Generic XBee API exception. This class and its subclasses indicate conditions that an application might want to catch.

All functionality of this class is the inherited of Exception.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception digi.xbee.exception.**CommunicationException**

Bases: *digi.xbee.exception.XBeeException*

This exception will be thrown when any problem related to the communication with the XBee device occurs.

All functionality of this class is the inherited of Exception.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception digi.xbee.exception.**ATCommandException**(*message='There was a problem sending the AT command packet.'*, *cmd_status=None*)

Bases: *digi.xbee.exception.CommunicationException*

This exception will be thrown when a response of a packet is not success or OK.

All functionality of this class is the inherited of Exception.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception digi.xbee.exception.**ConnectionException**

Bases: *digi.xbee.exception.XBeeException*

This exception will be thrown when any problem related to the connection with the XBee device occurs.

All functionality of this class is the inherited of Exception.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception digi.xbee.exception.**XBeeDeviceException**

Bases: *digi.xbee.exception.XBeeException*

This exception will be thrown when any problem related to the XBee device occurs.

All functionality of this class is the inherited of Exception.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception digi.xbee.exception.**InvalidConfigurationException**(*message='The configuration used to open the interface is invalid.'*)

Bases: *digi.xbee.exception.ConnectionException*

This exception will be thrown when trying to open an interface with an invalid configuration.

All functionality of this class is the inherited of Exception.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception digi.xbee.exception.**InvalidOperatingModeException**(*message=None*, *op_mode=None*)

Bases: *digi.xbee.exception.ConnectionException*

This exception will be thrown if the operating mode is different than *OperatingMode.API_MODE* and *OperatingMode.API_MODE*

All functionality of this class is the inherited of Exception.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception digi.xbee.exception.**InvalidPacketException**(*message='The XBee API packet is not properly formed.'*)

Bases: *digi.xbee.exception.CommunicationException*

This exception will be thrown when there is an error parsing an API packet from the input stream.

All functionality of this class is the inherited of Exception.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception digi.xbee.exception.**OperationNotSupportedException**(*message='The requested operation is not supported by either the connection interface or the XBee device.'*)

Bases: *digi.xbee.exception.XBeeDeviceException*

This exception will be thrown when the operation performed is not supported by the XBee device.

All functionality of this class is the inherited of Exception.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception digi.xbee.exception.**TimeoutException**(*message='There was a timeout while executing the requested operation.'*)
Bases: *digi.xbee.exception.CommunicationException*

This exception will be thrown when performing synchronous operations and the configured time expires.

All functionality of this class is the inherited of Exception.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception digi.xbee.exception.**TransmitException**(*message='There was a problem with a transmitted packet response (status not ok)', transmit_status=None*)
Bases: *digi.xbee.exception.CommunicationException*

This exception will be thrown when receiving a transmit status different than *TransmitStatus.SUCCESS* after sending an XBee API packet.

All functionality of this class is the inherited of Exception.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception digi.xbee.exception.**XBeeSocketException**(*message='There was a socket error', status=None*)
Bases: *digi.xbee.exception.XBeeException*

This exception will be thrown when there is an error performing any socket operation.

All functionality of this class is the inherited of Exception.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception digi.xbee.exception.**FirmwareUpdateException**
Bases: *digi.xbee.exception.XBeeException*

This exception will be thrown when any problem related to the firmware update process of the XBee device occurs.

All functionality of this class is the inherited of Exception.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception digi.xbee.exception.**RecoveryException**
Bases: *digi.xbee.exception.XBeeException*

This exception will be thrown when any problem related to the auto-recovery process of the XBee device occurs.

All functionality of this class is the inherited of Exception.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

digi.xbee.filesystem module

class digi.xbee.filesystem.**FileSystemElement**(*name*, *path=None*, *is_dir=False*, *size=0*, *is_secure=False*)

Bases: `object`

Class used to represent XBee file system elements (files and directories).

Class constructor. Instantiates a new *FileSystemElement* object with the given parameters.

Parameters

- **name** (*String or bytearray*) – Name of the file system element.
- **path** (*String or bytearray, optional, default=`None`*) – Absolute path of the element.
- **is_dir** (*Boolean, optional, default=`True`*) – *True* if the element is a directory, *False* for a file.
- **size** (*Integer, optional, default=0*) – Element size in bytes. Only for files.
- **is_secure** (*Boolean, optional, default=`False`*) – *True* for a secure element, *False* otherwise.

Raises `ValueError` – If any of the parameters are invalid.

name

Returns the file system element name.

Returns File system element name.

Return type String

path

Returns the file system element absolute path.

Returns File system element absolute path.

Return type String

is_dir

Returns whether the file system element is a directory.

Returns *True* for a directory, *False* otherwise.

Return type Boolean

size

Returns the size in bytes of the element.

Returns The size in bytes of the file, 0 for a directory.

Return type Integer

size_pretty

Returns a human readable size (e.g., 1K 234M 2G).

Returns Human readable size.

Return type String

is_secure

Returns whether the element is secure.

Returns *True* for a secure element, *False* otherwise.

Return type Boolean

static from_data(*name*, *size*, *flags*, *path=None*)
Creates a file element from its name and the bytearray with info and size.

Parameters

- **name** (*String or bytearray*) – The name of the element to create.
- **size** (*Bytearray*) – Byte array containing file size.
- **flags** (*Integer*) – Integer with file system element information.
- **path** (*String or bytearray, optional, default=`None`*) – The absolute path of the element (without its name).

Returns The new file system element.

Return type *FileSystemElement*

exception digi.xbee.filesystem.**FileSystemException**(*message*, *fs_status=None*)
Bases: *digi.xbee.exception.XBeeException*
This exception will be thrown when any problem related with the XBee file system occurs.
All functionality of this class is the inherited from Exception.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception digi.xbee.filesystem.**FileSystemNotSupportedException**(*message*, *fs_status=None*)
Bases: *digi.xbee.filesystem.FileSystemException*

This exception will be thrown when the file system feature is not supported in the device.

All functionality of this class is the inherited from Exception.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class digi.xbee.filesystem.**FileProcess**(*f_mng*, *file*, *timeout*)
Bases: `object`

This class represents a file process.

Class constructor. Instantiates a new `_FileProcess` object with the provided parameters.
Parameters

- **(class** (*f_mng*) – *.FileSystemManager*): The file system manager.
- **file** (*FileSystemElement* or String) – File or its absolute path.
- **timeout** (*Float*) – Timeout in seconds.

running
Returns if this file command is running.

Returns *True* if it is running, *False* otherwise.

Return type Boolean

status
Returns the status code.

Returns The status.

Return type Integer

block_size

Returns the size of the block for this file operation.

Returns Size of the block for this file operation.

Return type Integer

class digi.xbee.filesystem.**FileSystemManager**(*xbee*)

Bases: object

Helper class used to manage local or remote XBee file system.

Class constructor. Instantiates a new *FileSystemManager* with the given parameters.

Parameters **xbee** (*AbstractXBeeDevice*) – XBee to manage its file system.

Raises *FileSystemNotSupportedException* – If the XBee does not support filesystem.

xbee

Returns the XBee of this file system manager.

Returns XBee to manage its file system.

Return type *AbstractXBeeDevice*

np_value

The 'NP' parameter value of the local XBee.

Returns The 'NP' value.

Return type Integer

get_root()

Returns the root directory.

Returns The root directory.

Return type *FileSystemElement*

Raises *FileSystemException* – If there is any error performing the operation or the function is not supported.

make_directory(*dir_path*, *base=None*, *mk_parents=True*, *timeout=20*)

Creates the provided directory.

Parameters

- **dir_path** (*String*) – Path of the new directory to create. It is relative to the directory specify in base.
- **base** (*FileSystemElement*, optional, default='None) – Base directory. If not specify it refers to '/flash'.
- **mk_parents** (*Boolean, optional, default=`True`*) – *True* to make parent directories as needed, *False* otherwise.
- **timeout** (*Float, optional, default=`DEFAULT_TIMEOUT`*) – Maximum number of seconds to wait for the operation completion. If *mk_parents* this is the timeout per directory creation.

Returns List of *FileSystemElement* created directories.

Return type List

Raises

- *FileSystemException* – If there is any error performing the operation or the function is not supported.

- `ValueError` – If any of the parameters is invalid.

`list_directory`(*directory=None*, *timeout=20*)

Lists the contents of the given directory.

Parameters

- **`directory`** (*`FileSystemElement`* or String) – Directory to list or its absolute path.
- **`timeout`** (*`Float, optional, default=`DEFAULT_TIMEOUT``*) – Maximum number of seconds to wait for the operation completion.

Returns

List of *:class:*.FilesystemElement' objects contained in the given directory, empty list if status is not 0.

Return type List

Raises

- *`FileSystemException`* – If there is any error performing the operation or the function is not supported.
- `ValueError` – If any of the parameters is invalid.

`remove`(*entry*, *rm_children=True*, *timeout=20*)

Removes the given file system entry.

All files in a directory must be deleted before removing the directory. On XBee 3 802.15.4, DigiMesh, and Zigbee, deleted files are marked as unusable space unless they are at the "end" of the file system (most-recently created). On these products, deleting a file triggers recovery of any deleted file space at the end of the file system, and can lead to a delayed response.

Parameters

- **`entry`** (*`FileSystemElement`* or String) – File system entry to remove or its absolute path.
- **`rm_children`** (*`Boolean, optional, default=`True``*) – *True* to remove directory children if they exist, *False* otherwise.
- **`timeout`** (*`Float, optional, default=`DEFAULT_TIMEOUT``*) – Maximum number of seconds to wait for the operation completion.

Raises

- *`FileSystemException`* – If there is any error performing the operation or the function is not supported.
- `ValueError` – If any of the parameters is invalid.

`read_file`(*file*, *offset=0*, *progress_cb=None*)

Reads from the provided file starting at the given offset. If there is no progress callback the function blocks until the required amount of bytes is read.

Parameters

- **`file`** (*`FileSystemElement`* or String) – File to read or its absolute path.
- **`offset`** (*`Integer, optional, default=0`*) – File offset to start reading.
- **`progress_cb`** (*`Function, optional, default=`None``*) – Function called when new data is read. Receives four arguments:

- The chunk of data read as byte array.
- The progress percentage as float.
- The total size of the file.
- The status when process finishes.

Returns The process to read data from the file.

Return type `FileProcess`

Raises

- `FileSystemException` – If there is any error performing the operation and *progress_cb* is *None*.
- `ValueError` – If any of the parameters is invalid.

See also:

`get_file()`

write_file (*file*, *offset=0*, *secure=False*, *options=None*, *progress_cb=None*)
Writes to the provided file the data starting at the given offset. The function blocks until the all data is written.

Parameters

- **file** (`FileSystemElement` or String) – File to write or its absolute path.
- **offset** (*Integer, optional, default=0*) – File offset to start writing.
- **secure** (*Boolean, optional, default=`False`*) – *True* to store the file securely (no read access), *False* otherwise.
- **options** (*Dictionary, optional*) – Other write options as list: *exclusive*, *truncate*, *append*.
- **progress_cb** (*Function, optional, default=`None`*) – Function call when data is written. Receives three arguments:
  - The amount of bytes written (for each chunk).
  - The progress percentage as float.
  - The status when process finishes.

Raises

- `FileSystemException` – If there is any error performing the operation and *progress_cb* is *None*.
- `ValueError` – If any of the parameters is invalid.

See also:

`put_file()`

get_file (*src*, *dest*, *progress_cb=None*)
Downloads the given XBee file in the specified destination path.

Parameters

- **src** (*FileSystemElement* or String) – File to download or its absolute path.
- **dest** (*String*) – The absolute path of the destination file.
- **progress_cb** (*Function, optional*) – Function call when data is being downloaded. Receives three arguments:
  - The progress percentage as float.
  - Destination file path.
  - Source file path.

Raises

- *FileSystemException* – If there is any error performing the operation and *progress_cb* is *None*.
- `ValueError` – If any of the parameters is invalid.

put_file (*src*, *dest*, *secure=False*, *overwrite=False*, *mk_parents=True*, *progress_cb=None*)

Uploads the given file to the specified destination path of the XBee.

Parameters

- **src** (*String*) – Absolute path of the file to upload.
- **dest** (*FileSystemElement* or String) – The file in the XBee or its absolute path.
- **secure** (*Boolean, optional, default=`False`*) – *True* if the file should be stored securely, *False* otherwise.
- **overwrite** (*Boolean, optional, default=`False`*) – *True* to overwrite the file if it exists, *False* otherwise.
- **mk_parents** (*Boolean, optional, default=`True`*) – *True* to make parent directories as needed, *False* otherwise.
- **progress_cb** (*Function, optional*) – Function call when data is being uploaded. Receives two arguments:
  - The progress percentage as float.
  - Destination file path.
  - Source file path.

Returns The new created file.

Return type *FileSystemElement*

Raises

- *FileSystemException* – If there is any error performing the operation and *progress_cb* is *None*.
- `ValueError` – If any of the parameters is invalid.

put_dir (*src*, *dest='/flash'*, *verify=True*, *progress_cb=None*)

Uploads the given source directory contents into the given destination directory in the XBee.

Parameters

- **src** (*String*) – Local directory to upload its contents.

- **dest** (*FileSystemElement* or String) – The destination dir in the XBee or its absolute path. Defaults to '/flash'.
- **verify** (*Boolean, optional, default=`True`*) – *True* to check the hash of the uploaded content.
- **progress_cb** (*Function, optional*) – Function call when data is being uploaded. Receives three argument:
  - The progress percentage as float.
  - Destination file path.
  - The absolute path of the local being uploaded as string.

Raises

- *FileSystemException* – If there is any error performing the operation and *progress_cb* is *None*.
- ValueError – If any of the parameters is invalid.

get_file_hash (*file*, *timeout=20*)

Returns the SHA256 hash of the given file.

Parameters

- **file** (*FileSystemElement* or String) – File to get its hash or its absolute path.
- **timeout** (*Float, optional, default=`DEFAULT_TIMEOUT`*) – Maximum number of seconds to wait for the operation completion.

Returns SHA256 hash of the given file.

Return type Bytearray

Raises

- *FileSystemException* – If there is any error performing the operation or the function is not supported.
- ValueError – If any of the parameters is invalid.

move (*source*, *dest*, *timeout=20*)

Moves the given source element to the given destination path.

Parameters

- **source** (*FileSystemElement* or String) – Source entry to move.
- **dest** (*String*) – Destination path of the element to move.
- **timeout** (*Float, optional, default=`DEFAULT_TIMEOUT`*) – Maximum number of seconds to wait for the operation completion.

Raises

- *FileSystemException* – If there is any error performing the operation or the function is not supported.
- ValueError – If any of the parameters is invalid.

get_volume_info (*vol='/flash'*, *timeout=20*)

Returns the file system volume information. Currently '/flash' is the only supported value.

Parameters

- **vol** (FileSystemElement`or String, optional, default=/flash‘) – Volume name.
- **timeout** (*Float, optional, default=`DEFAULT_TIMEOUT`*) – Maximum number of seconds to wait for the operation completion.

Returns Collection of pair values describing volume information.

Return type Dictionary

Raises

- *FileSystemException* – If there is any error performing the operation or the function is not supported.
- ValueError – If any of the parameters is invalid.

See also:

FSCommandStatus

format (*vol='/flash', timeout=30*)

Formats provided volume. Currently '/flash' is the only supported value. Formatting the file system takes time, and any other requests will fail until it completes and sends a response.

Parameters

- **vol** (FileSystemElement`or String, optional, default=/flash‘) – Volume name.
- **timeout** (*Float, optional, default=`DEFAULT_FORMAT_TIMEOUT`*) – Maximum number Of seconds to wait for the operation completion.

Returns Collection of pair values describing volume information.

Return type Dictionary

Raises

- *FileSystemException* – If there is any error performing the operation or the function is not supported.
- ValueError – If any of the parameters is invalid.

See also:

FSCommandStatus

pget_path_id (*dir_path, path_id=0, timeout=20*)

Returns the directory path id of the given path. Returned directory path id expires if not referenced in 2 minutes.

Parameters

- **dir_path** (*String*) – Path of the directory to get its id. It is relative to the directory path id.
- **path_id** (*Integer, optional, default=0*) – Directory path id. 0 for the root directory.

- **timeout** (*Float, optional, default=`DEFAULT_TIMEOUT`*) – Maximum number of seconds to wait for the operation completion.

Returns

Status of the file system command execution, new directory path id (-1 if status is not 0) and its absolute path (empty if status is not 0). The full path may be *None* or empty if it is too long and exceeds the communication frames length.

Return type Tuple (Integer, Integer, String)

Raises

- *FileSystemException* – If there is any error performing the operation or the function is not supported.
- `ValueError` – If any of the parameters is invalid.

See also:

FSCommandStatus

pmake_directory(*dir_path*, *path_id=0*, *timeout=20*)

Creates the provided directory. Parent directories of the one to be created must exist. Separate requests must be dane to make intermediate directories.

Parameters

- **dir_path** (*String*) – Path of the new directory to create. It is relative to the directory path id.
- **path_id** (*Integer, optional, default=0*) – Directory path id. 0 for the root directory.
- **timeout** (*Float, optional, default=`DEFAULT_TIMEOUT`*) – Maximum number of seconds to wait for the operation completion. If *mk_parents* this is the timeout per directory creation.

Returns

Status of the file system command execution (see *FSCommandStatus*).

Return type Integer

Raises

- *FileSystemException* – If there is any error performing the operation or the function is not supported.
- `ValueError` – If any of the parameters is invalid.

See also:

FSCommandStatus

plist_directory(*dir_path*, *path_id=0*, *timeout=20*)

Lists the contents of the given directory.

Parameters

- **dir_path** (*String*) – Path of the directory to list. It is relative to the directory path id.
- **path_id** (*Integer, optional, default=0*) – Directory path id. 0 for the root directory.
- **timeout** (*Float, optional, default=`DEFAULT_TIMEOUT`*) – Maximum number of seconds to wait for the operation completion.

Returns

Status of the file system command execution and a list of *:class:*.FilesystemElement‘ objects contained in the given directory, empty list if status is not 0.

Return type Tuple (Integer, List)

Raises

- *FileSystemException* – If there is any error performing the operation or the function is not supported.
- ValueError – If any of the parameters is invalid.

See also:

FSCommandStatus

premove (*entry_path*, *path_id=0*, *timeout=20*)

Removes the given file system entry.

All files in a directory must be deleted before removing the directory. On XBee 3 802.15.4, DigiMesh, and Zigbee, deleted files are marked as as unusable space unless they are at the “end” of the file system (most-recently created). On these products, deleting a file triggers recovery of any deleted file space at the end of the file system, and can lead to a delayed response.

Parameters

- **entry_path** (*String*) – Path of the entry to remove. It is relative to the directory path id.
- **path_id** (*Integer, optional, default=0*) – Directory path id. 0 for the root directory.
- **timeout** (*Float, optional, default=`DEFAULT_TIMEOUT`*) – Maximum number of seconds to wait for the operation completion.

Returns

Status of the file system command execution (see *FSCommandStatus*).

Return type Integer

Raises

- *FileSystemException* – If there is any error performing the operation or the function is not supported.
- ValueError – If any of the parameters is invalid.

See also:

`FSCommandStatus`

popen_file (*file_path*, *path_id=0*, *options=<FileOpenRequestOption.READ: 4>*, *timeout=20*)

Open a file for reading and/or writing. Use the *FileOpenRequestOption.SECURE* (0x80) bitmask for options to upload a write-only file (one that cannot be downloaded or viewed), useful for protecting files on the device. Returned file id expires if not referenced in 2 minutes.

Parameters

- **file_path** (*String*) – Path of the file to open. It is relative to the directory path id.
- **path_id** (*Integer, optional, default=0*) – Directory path id. 0 for the root directory.
- **options** (*Integer, optional, default=`FileOpenRequestOption.READ`*) – Bitmask that specifies the options to open the file. It defaults to *FileOpenRequestOption.READ* which means open for reading. See `FileOpenRequestOption` for more options.
- **timeout** (*Float, optional, default=`DEFAULT_TIMEOUT`*) – Maximum number of seconds to wait for the operation completion.

Returns

Status of the file system command execution (see `FSCommandStatus`), the file id to use in later requests, and the size of the file (in bytes), 0xFFFFFFFF if unknown.

Return type Tuple (Integer, Integer, Integer)

Raises

- `FileSystemException` – If there is any error performing the operation or the function is not supported.
- `ValueError` – If any of the parameters is invalid.

See also:

`FileOpenRequestOption`
`FSCommandStatus`
`pclose_file()`

pclose_file (*file_id*, *timeout=20*)

Closes an open file and releases its file handle.

Parameters

- **file_id** (*Integer*) – File id returned when opening.
- **timeout** (*Float, optional, default=`DEFAULT_TIMEOUT`*) – Maximum number of seconds to wait for the operation completion.

Returns

Status of the file system command execution (see `FSCommandStatus`).

Return type Integer

Raises

- *FileSystemException* – If there is any error performing the operation or the function is not supported.
- ValueError – If any of the parameters is invalid.

See also:

FSCommandStatus
popen_file()

pread_file (*file_id*, *offset=-1*, *size=-1*, *timeout=20*)
Reads from the provided file the given amount of bytes starting at the given offset. The file must be opened for reading first.

Parameters

- **file_id** (*Integer*) – File id returned when opening.
- **offset** (*Integer, optional, default=-1*) – File offset to start reading. -1 to use current position.
- **size** (*Integer, optional, default=-1*) – Number of bytes to read. -1 to read as many as possible.
- **timeout** (*Float, optional, default=`DEFAULT_TIMEOUT`*) – Maximum number of seconds to wait for the operation completion.

Returns

Status of the file system command execution (see *FSCommandStatus*), the file id, the offset of the read data, and the read data.

Return type Tuple (Integer, Integer, Integer, Bytearray)

Raises

- *FileSystemException* – If there is any error performing the operation or the function is not supported.
- ValueError – If any of the parameters is invalid.

See also:

FSCommandStatus
popen_file()

pwrite_file (*file_id*, *data*, *offset=-1*, *timeout=20*)
Writes to the provided file the given data bytes starting at the given offset. The file must be opened for writing first.

Parameters

- **file_id** (*Integer*) – File id returned when opening.
- **data** (*Bytearray, bytes or String*) – Data to write.
- **offset** (*Integer, optional, default=-1*) – File offset to start writing. -1 to use current position.

- **timeout** (*Float, optional, default=`DEFAULT_TIMEOUT`*) – Maximum number of seconds to wait for the operation completion.

Returns

Status of the file system command execution (see *FSCommandStatus*), the file id, and the current offset after writing.

Return type Tuple (Integer, Integer, Integer)

Raises

- *FileSystemException* – If there is any error performing the operation or the function is not supported.
- `ValueError` – If any of the parameters is invalid.

See also:

FSCommandStatus
popen_file()

pget_file_hash (*file_path*, *path_id=0*, *timeout=20*)
Returns the SHA256 hash of the given file.

Parameters

- **file_path** (*String*) – Path of the file to get its hash. It is relative to the directory path id.
- **path_id** (*Integer, optional, default=0*) – Directory path id. 0 for the root directory.
- **timeout** (*Float, optional, default=`DEFAULT_TIMEOUT`*) – Maximum number of seconds to wait for the operation completion.

Returns

Status of the file system command execution and SHA256 hash of the given file (empty bytearray if status is not 0).

Return type Tuple (Integer, Bytearray)

Raises

- *FileSystemException* – If there is any error performing the operation or the function is not supported.
- `ValueError` – If any of the parameters is invalid.

See also:

FSCommandStatus

prename (*current_path*, *new_path*, *path_id=0*, *timeout=20*)
Rename provided file.

Parameters

- **current_path** (*String*) – Current path name. It is relative to the directory path id.
- **new_path** (*String*) – New name. It is relative to the directory path id.
- **path_id** (*Integer, optional, default=0*) – Directory path id. 0 for the root directory.
- **timeout** (*Float, optional, default=`DEFAULT_TIMEOUT`*) – Maximum number of seconds to wait for the operation completion.

Returns

Status of the file system command execution (see *FSCommandStatus*).

Return type Integer

Raises

- *FileSystemException* – If there is any error performing the operation or the function is not supported.
- ValueError – If any of the parameters is invalid.

See also:

FSCommandStatus

prelease_path_id(*path_id*, *timeout=20*)

Releases the provided directory path id.

Parameters

- **path_id** (*Integer*) – Directory path id to release.
- **timeout** (*Float, optional, default=`DEFAULT_TIMEOUT`*) – Maximum number of seconds to wait for the operation completion.

Returns Status of the file system command execution.

Return type Integer

Raises

- *FileSystemException* – If there is any error performing the operation or the function is not supported.
- ValueError – If any of the parameters is invalid.

See also:

FSCommandStatus

class digi.xbee.filesystem.**LocalXBeeFileSystemManager**(*xbee_device*)

Bases: object

Helper class used to manage the local XBee file system.

Class constructor. Instantiates a new *LocalXBeeFileSystemManager* with the given parameters.

Parameters **xbee_device** (*XBeeDevice*) – The local XBee to manage its file system.

is_connected
Returns whether the file system manager is connected or not.

Returns

True **if the file system manager is connected,** ***False*** otherwise.

Return type Boolean

connect()
Connects the file system manager.

Raises

- *FileSystemException* – If there is any error connecting the file system manager.
- *FileSystemNotSupportedException* – If the device does not support filesystem feature.

disconnect()
Disconnects the file system manager and restores the device connection.

Raises `XBeeException` – If there is any error restoring the XBee connection.

get_current_directory()
Returns the current device directory.

Returns Current device directory.

Return type String

Raises *FileSystemException* – If there is any error getting the current directory or the function is not supported.

change_directory(*directory*)
Changes the current device working directory to the given one.

Parameters **directory** (*String*) – New directory to change to.

Returns Current device working directory after the directory change.

Return type String

Raises *FileSystemException* – If there is any error changing the current directory or the function is not supported.

make_directory(*directory*)
Creates the provided directory.

Parameters **directory** (*String*) – New directory to create.

Raises *FileSystemException* – If there is any error creating the directory or the function is not supported.

list_directory(*directory=None*)
Lists the contents of the given directory.

Parameters **directory** (*String, optional*) – the directory to list its contents. If not provided, the current directory contents are listed.

Returns

list of ***:class:*****.FilesystemElement‘ objects contained in** the given (or current) directory.

Return type List

Raises `FileSystemException` – if there is any error listing the directory contents or the function is not supported.

remove_element (*element_path*)

Removes the given file system element path.

Parameters **element_path** (`String`) – Path of the file system element to remove.

Raises `FileSystemException` – If there is any error removing the element or the function is not supported.

move_element (*source_path*, *dest_path*)

Moves the given source element to the given destination path.

Parameters

- **source_path** (`String`) – Source path of the element to move.
- **dest_path** (`String`) – Destination path of the element to move.

Raises `FileSystemException` – If there is any error moving the element or the function is not supported.

put_file (*source_path*, *dest_path*, *secure=False*, *progress_callback=None*)

Transfers the given file in the specified destination path of the XBee.

Parameters

- **source_path** (`String`) – the path of the file to transfer.
- **dest_path** (`String`) – the destination path to put the file in.
- **secure** (`Boolean, optional, default=`False``) – *True* if the file should be stored securely, *False* otherwise.
- **progress_callback** (`Function, optional`) – Function to execute to receive progress information. Takes the following arguments:
  - The progress percentage as integer.

Raises `FileSystemException` – If there is any error transferring the file or the function is not supported.

put_dir (*source_dir*, *dest_dir=None*, *progress_callback=None*)

Uploads the given source directory contents into the given destination directory in the device.

Parameters

- **source_dir** (`String`) – Local directory to upload its contents.
- **dest_dir** (`String, optional`) – Remote directory to upload the contents to. Defaults to current directory.
- **progress_callback** (`Function, optional`) – Function to execute to receive progress information. Takes the following arguments:
  - The file being uploaded as string.
  - The progress percentage as integer.

Raises `FileSystemException` – If there is any error uploading the directory or the function is not supported.

get_file (*source_path*, *dest_path*, *progress_callback=None*)

Downloads the given XBee device file in the specified destination path.

Parameters

- **source_path** (*String*) – Path of the XBee device file to download.
- **dest_path** (*String*) – Destination path to store the file in.
- **progress_callback** (*Function, optional*) – Function to execute to receive progress information. Takes the following arguments:
  - The progress percentage as integer.

Raises *FileSystemException* – If there is any error downloading the file or the function is not supported.

format_filesystem()

Formats the device file system.

Raises *FileSystemException* – If there is any error formatting the file system.

get_usage_information()

Returns the file system usage information.

Returns Collection of pair values describing the usage information.

Return type Dictionary

Raises *FileSystemException* – If there is any error retrieving the file system usage information.

get_file_hash(*file_path*)

Returns the SHA256 hash of the given file path.

Parameters **file_path** (*String*) – Path of the file to get its hash.

Returns SHA256 hash of the given file path.

Return type String

Raises *FileSystemException* – If there is any error retrieving the file hash.

digi.xbee.filesystem.**update_remote_filesystem_image**(*remote_device*, *ota_filesystem_file*, *max_block_size=0*, *timeout=None*, *progress_callback=None*, *_prepare=True*)

Performs a remote filesystem update operation in the given target.

Parameters

- **remote_device** (*RemoteXBeeDevice*) – Remote XBee to update its filesystem image.
- **ota_filesystem_file** (*String*) – Path of the OTA filesystem file to upload.
- **max_block_size** (*Integer, optional*) – Maximum size of the ota block to send.
- **timeout** (*Integer, optional*) – Timeout to wait for remote frame requests.
- **progress_callback** (*Function, optional*) – Function to execute to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

Raises

- `FileSystemNotSupportedException` – If the target does not support filesystem update.
- `FileSystemException` – If there is any error updating the remote filesystem image.

`digi.xbee.filesystem.`**`check_fs_support`**(*xbee*, *min_fw_vers=None*, *max_fw_vers=None*)

Checks if filesystem API feature is supported.

Parameters

- **`xbee`** (`:AbstractXBeeDevice`) – The XBee to check.
- **`min_fw_vers`** (*`Dictionary, optional, default=`None``*) – A dictionary with protocol as key, and minimum firmware version with filesystem support as value.
- **`max_fw_vers`** (*`Dictionary, optional, default=`None``*) – A dictionary with protocol as key, and maximum firmware version with filesystem support as value.

Returns *True* if filesystem is supported, *False* otherwise.

Return type Boolean

`digi.xbee.filesystem.`**`get_local_file_hash`**(*local_path*)

Returns the SHA256 hash of the given local file.

Parameters **`local_path`** (*`String`*) – Absolute path of the file to get its hash.

Returns SHA256 hash of the given file.

Return type Bytearray

digi.xbee.firmware module

`class` `digi.xbee.firmware.`**`UpdateConfigurer`**(*node*, *timeout=None*, *callback=None*)

Bases: `object`

For internal use only. Helper class used to prepare nodes and/or network for an update.

Class constructor. Instantiates a new `UpdateConfigurer` with the given parameters.

Parameters

- **`node`** (`AbstractXBeeDevice`) – Target being updated.
- **`timeout`** (*`Float, optional, default=`None``*) – Operations timeout.
- **`callback`** (*`Function`*) – Function to notify about the progress.

`sync_sleep`

Returns whether node is part of a DigiMesh synchronous sleeping network.

Returns *True* if it synchronous sleeps, *False* otherwise.

Return type Boolean

`prepare_total`

Returns the total work for update preparation step.

Returns Total prepare work.

Return type Integer

`restore_total`

Returns the total work for update restoration step.

Returns Total restore work.

Return type Integer

prepare_for_update (*prepare_node=True*, *prepare_net=True*, *restore_later=True*)

Prepares the node for an update process.

Parameters

- **prepare_node** (*Boolean, optional, default=`True`*) – *True* to prepare the node.
- **prepare_net** (*Boolean, optional, default=`True`*) – *True* to prepare the network.
- **restore_later** (*Boolean, optional, default=`True`*) – *True* to restore node original values when finish the update process.

restore_after_update (*restore_settings=True*, *port_settings=None*)

Restores the node after an update process.

Parameters

- **restore_settings** (*Boolean, optional, default=`True`*) – *True* to restore stored settings, *False* otherwise.
- **port_settings** (*Dictionary, optional, default=`None`*) – Dictionary with the new serial port configuration, *None* for remote node or if the serial config has not changed.

static exec_at_cmd (*func*, *node*, *cmd*, *value=None*, *retries=5*, *apply=False*)

Reads the given parameter from the XBee with the given number of retries.

Parameters

- **func** (*Function*) – Function to execute.
- **node** (*AbstractXBeeDevice*) – XBee to get/set parameter.
- **(String or** (*cmd*) – class: *ATStringCommand*): Parameter to get/set.
- **value** (*Bytearray, optional, default=`None`*) – Value to set.
- **retries** (*Integer, optional, default=5*) – Number of retries to perform.
- **apply** (*Boolean, optional, default=`False`*) – *True* to apply.

Returns Read parameter value.

Return type Bytearray

Raises `XBeeException` – If the value could be get/set after the retries.

progress_cb (*task*, *done=0*)

If a callback was provided in the constructor, notifies it with the provided task and the corresponding percentage.

Parameters

- **task** (*String*) – The task to inform about, it must be *TASK_PREPARE* or *TASK_RESTORE*.
- **done** (*Integer, optional, default=0*) – Total amount of done job. If 0, it is increased by one.

Returns Total work done for the task.

Return type Integer

`digi.xbee.firmware.`**`update_local_firmware`**(*target*, *xml_fw_file*, *xbee_firmware_file=None*, *bootloader_firmware_file=None*, *timeout=None*, *progress_callback=None*)

Performs a local firmware update operation in the given target.

Parameters

- **`target`** (String or `XBeeDevice`) – Target of the firmware upload operation. String: serial port identifier. `XBeeDevice`: XBee to upload its firmware.
- **`xml_fw_file`** (*`String`*) – Path of the XML file that describes the firmware.
- **`xbee_firmware_file`** (*`String, optional`*) – Location of the XBee binary firmware file.
- **`bootloader_firmware_file`** (*`String, optional`*) – Location of the bootloader binary firmware file.
- **`timeout`** (*`Integer, optional`*) – Serial port read data timeout.
- **`progress_callback`** (*`Function, optional`*) – Function to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

Raises `FirmwareUpdateException` – If there is any error performing the firmware update.

`digi.xbee.firmware.`**`update_remote_firmware`**(*remote*, *xml_fw_file*, *firmware_file=None*, *bootloader_file=None*, *max_block_size=0*, *timeout=None*, *progress_callback=None*, *_prepare=True*)

Performs a remote firmware update operation in the given target.

Parameters

- **`remote`** (`RemoteXBeeDevice`) – Remote XBee to upload.
- **`xml_fw_file`** (*`String`*) – Path of the XML file that describes the firmware.
- **`firmware_file`** (*`String, optional`*) – Path of the binary firmware file.
- **`bootloader_file`** (*`String, optional`*) – Path of the bootloader firmware file.
- **`max_block_size`** (*`Integer, optional`*) – Maximum size of the ota block to send.
- **`timeout`** (*`Integer, optional`*) – Timeout to wait for remote frame requests.
- **`progress_callback`** (*`Function, optional`*) – Function to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

Raises `FirmwareUpdateException` – if there is any error performing the remote firmware update.

`digi.xbee.firmware.`**`update_remote_filesystem`**(*remote*, *ota_fs_file*, *max_block_size=0*, *timeout=None*, *progress_callback=None*, *_prepare=True*)

Performs a remote filesystem update operation in the given target.

Parameters

- **remote** (*RemoteXBeeDevice*) – Remote XBee to update its filesystem.
- **ota_fs_file** (*String*) – Path of the OTA filesystem image file.
- **max_block_size** (*Integer, optional*) – Maximum size of the ota block to send.
- **timeout** (*Integer, optional*) – Timeout to wait for remote frame requests.
- **progress_callback** (*Function, optional*) – Function to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

Raises `FirmwareUpdateException` – If there is any error updating the remote filesystem image.

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’)

description
Returns the description of the IOLine element.

Returns The description of the IOLine element.

Return type String

index
Returns the index of the IOLine element.

Returns The index of the IOLine element.

Return type Integer

at_command
Returns the AT command of the IOLine element.

Returns The AT command of the IOLine element.

Return type String

pwm_at_command
Returns the PWM AT command associated to the IOLine element.

Returns

The PWM AT command associated to the IO line, *None* if the IO line does not have a PWM AT command associated.

Return type String

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

class digi.xbee.io.**IOValue**(*code*)
Bases: `enum.Enum`
Enumerates the possible values of a `IOLine` configured as digital I/O.

Values:
IOValue.LOW = 4
IOValue.HIGH = 5

code
Returns the code of the IOValue element.

Returns The code of the IOValue element.

Return type String

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:

```
bit 0 =  DIO01
bit 1 =  DIO10
bit 2 =  DIO20
bit 3 =  DIO31
bit 4 =  DIO40
bit 5 =  DIO51
bit 6 =  DIO60
bit 7 =  DIO70
bit 8 =  DIO80
bit 9 =  AD00
bit 10 = AD11
bit 11 = AD21
bit 12 = AD30
bit 13 = AD40
bit 14 = AD50
bit 15 = NA0

Example: mask of 0x0C29 means DIO0, DIO3, DIO5, AD1 and AD2 enabled.
0 0 0 0 1 1 0 0 0 0 1 0 1 0 0 1
```

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

```
bit 0 =  DIO0AD0
bit 1 =  DIO1AD1
bit 2 =  DIO2AD2
bit 3 =  DIO3AD3
bit 4 =  DIO4AD4
bit 5 =  DIO5AD5ASSOC
bit 6 =  DIO6RTS
bit 7 =  DIO7CTS
bit 8 =  DIO8DTRSLEEP_RQ
bit 9 =  DIO9ON_SLEEP
bit 10 = DIO10PWM0RSSI
bit 11 = DIO11PWM1
bit 12 = DIO12CD
bit 13 = DIO13
bit 14 = DIO14
bit 15 = NA

Example: mask of 0x040B means DIO0, DIO1, DIO2, DIO3 and DIO10 enabled.
0 0 0 0 0 1 0 0 0 0 0 0 1 0 1 1
```

Analog Channel Mask Indicates which lines are configured as ADC. Each bit in the analog channel mask corresponds to one ADC line on the module.

```
bit 0 = AD0DIO0
bit 1 = AD1DIO1
bit 2 = AD2DIO2
bit 3 = AD3DIO3
bit 4 = AD4DIO4
```

(continues on next page)

(continued from previous page)

```
bit 5 = AD5DIO5ASSOC
bit 6 = NA
bit 7 = Supply Voltage Value

Example: mask of 0x03 means AD0, and AD1 enabled.
0 0 0 0 0 0 1 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

digital_hsb_mask

Returns the High Significant Byte (HSB) of the digital mask.

Returns The HSB of the digital mask.

Return type Integer

digital_lsb_mask

Returns the Low Significant Byte (HSB) of the digital mask.

Returns The LSB of the digital mask.

Return type Integer

digital_mask

Returns the combined (HSB + LSB) of the digital mask.

Returns The digital mask.

Return type Integer

digital_values

Returns the digital values map.

To verify if this sample contains a valid digital values, use the method *IOSample.has_digital_values()*.

Returns The digital values map.

Return type Dictionary

analog_mask

Returns the analog mask.

Returns the analog mask.

Return type Integer

analog_values

Returns the analog values map.

To verify if this sample contains a valid analog values, use the method *IOSample.has_analog_values()*.

Returns The analog values map.

Return type Dictionary

`power_supply_value`
Returns the value of the power supply voltage.

To verify if this sample contains the power supply voltage, use the method *`IOSample.has_power_supply_value()`*.

Returns

The power supply value, *None* if the sample does not contain power supply value.

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 th 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 {@code 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

class digi.xbee.io.**IOMode**
Bases: enum.Enum

Enumerates the different Input/Output modes that an IO line can be configured with.

DISABLED = 0
Disabled

SPECIAL_FUNCTIONALITY = 1
Firmware special functionality

PWM = 2
PWM output

ADC = 2
Analog to Digital Converter

DIGITAL_IN = 3
Digital input

DIGITAL_OUT_LOW = 4
Digital output, Low

DIGITAL_OUT_HIGH = 5
Digital output, High

I2C_FUNCTIONALITY = 6
I2C functionality

digi.xbee.profile module

class digi.xbee.profile.**FirmwareBaudrate**(*index*, *baudrate*)

Bases: enum.Enum

This class lists the available firmware baudrate options for XBee Profiles.

Inherited properties:

name (String): The name of this *FirmwareBaudrate*.
value (Integer): The ID of this *FirmwareBaudrate*.

Values:

FirmwareBaudrate.BD_1200 = (0, 1200)
FirmwareBaudrate.BD_2400 = (1, 2400)
FirmwareBaudrate.BD_4800 = (2, 4800)
FirmwareBaudrate.BD_9600 = (3, 9600)
FirmwareBaudrate.BD_19200 = (4, 19200)
FirmwareBaudrate.BD_38400 = (5, 38400)
FirmwareBaudrate.BD_57600 = (6, 57600)
FirmwareBaudrate.BD_115200 = (7, 115200)
FirmwareBaudrate.BD_230400 = (8, 230400)
FirmwareBaudrate.BD_460800 = (9, 460800)
FirmwareBaudrate.BD_921600 = (10, 921600)

index

Returns the index of the *FirmwareBaudrate* element.

Returns Index of the *FirmwareBaudrate* element.

Return type Integer

baudrate

Returns the baudrate of the *FirmwareBaudrate* element.

Returns Baudrate of the *FirmwareBaudrate* element.

Return type Integer

class digi.xbee.profile.**FirmwareParity**(*index*, *parity*)

Bases: enum.Enum

This class lists the available firmware parity options for XBee Profiles.

Inherited properties:

name (String): The name of this *FirmwareParity*.
value (Integer): The ID of this *FirmwareParity*.

Values:

FirmwareParity.NONE = (0, <sphinx.ext.autodoc.importer._MockObject object at 0x7f02d6d0c5d0>)
FirmwareParity.EVEN = (1, <sphinx.ext.autodoc.importer._MockObject object at 0x7f02d6a48310>)

FirmwareParity.ODD = (2, <sphinx.ext.autodoc.importer._MockObject object at 0x7f02d6e04850>)
FirmwareParity.MARK = (3, <sphinx.ext.autodoc.importer._MockObject object at 0x7f02d5e20790>)
FirmwareParity.SPACE = (4, <sphinx.ext.autodoc.importer._MockObject object at 0x7f02d5e20b10>)

index
Returns the index of the *FirmwareParity* element.

Returns Index of the *FirmwareParity* element.

Return type Integer

parity
Returns the parity of the *FirmwareParity* element.

Returns Parity of the *FirmwareParity* element.

Return type String

class digi.xbee.profile.**FirmwareStopbits**(*index*, *stop_bits*)
Bases: `enum.Enum`
This class lists the available firmware stop bits options for XBee Profiles.

Inherited properties:
name (String): The name of this *FirmwareStopbits*.
value (Integer): The ID of this *FirmwareStopbits*.

Values:
FirmwareStopbits.SB_1 = (0, <sphinx.ext.autodoc.importer._MockObject object at 0x7f02d6013d50>)
FirmwareStopbits.SB_2 = (1, <sphinx.ext.autodoc.importer._MockObject object at 0x7f02d6013390>)
FirmwareStopbits.SB_1_5 = (2, <sphinx.ext.autodoc.importer._MockObject object at 0x7f02d6111650>)

index
Returns the index of the *FirmwareStopbits* element.

Returns Index of the *FirmwareStopbits* element.

Return type Integer

stop_bits
Returns the stop bits of the *FirmwareStopbits* element.

Returns Stop bits of the *FirmwareStopbits* element.

Return type Float

class digi.xbee.profile.**FlashFirmwareOption**(*code*, *description*)
Bases: `enum.Enum`
This class lists the available flash firmware options for XBee Profiles.

Inherited properties:

name (String): The name of this *FlashFirmwareOption*.
value (Integer): The ID of this *FlashFirmwareOption*.

Values:

FlashFirmwareOption.FLASH_ALWAYS = (0, 'Flash always')
FlashFirmwareOption.FLASH_DIFFERENT = (1, 'Flash firmware if it is different')
FlashFirmwareOption.DONT_FLASH = (2, 'Do not flash firmware')

`code`
Returns the code of the *FlashFirmwareOption* element.

Returns Code of the *FlashFirmwareOption* element.

Return type Integer

`description`
Returns the description of the *FlashFirmwareOption* element.

Returns Description of the *FlashFirmwareOption* element.

Return type String

class `digi.xbee.profile.`**`XBeeSettingType`**(*tag*, *description*)
Bases: `enum.Enum`
This class lists the available firmware setting types.

Inherited properties:
name (String): The name of this *XBeeSettingType*.
value (Integer): The ID of this *XBeeSettingType*.

Values:

XBeeSettingType.NUMBER = ('number', 'Number')
XBeeSettingType.COMBO = ('combo', 'Combo')
XBeeSettingType.TEXT = ('text', 'Text')
XBeeSettingType.BUTTON = ('button', 'Button')
XBeeSettingType.NO_TYPE = ('none', 'No type')

`tag`
Returns the tag of the *XBeeSettingType* element.

Returns Tag of the *XBeeSettingType* element.

Return type String

`description`
Returns the description of the *XBeeSettingType* element.

Returns Description of the *XBeeSettingType* element.

Return type String

class `digi.xbee.profile.`**`XBeeSettingFormat`**(*tag*, *description*)

Bases: `enum.Enum`

This class lists the available text firmware setting formats.

Inherited properties:

name (String): The name of this *XBeeSettingFormat*.

value (Integer): The ID of this *XBeeSettingFormat*.

Values:

XBeeSettingFormat.HEX = ('HEX', 'Hexadecimal')

XBeeSettingFormat.ASCII = ('ASCII', 'ASCII')

XBeeSettingFormat.IPV4 = ('IPV4', 'IPv4')

XBeeSettingFormat.IPV6 = ('IPV6', 'IPv6')

XBeeSettingFormat.PHONE = ('PHONE', 'phone')

XBeeSettingFormat.NO_FORMAT = ('none', 'No format')

`tag`

Returns the tag of the *XBeeSettingFormat* element.

Returns Tag of the *XBeeSettingFormat* element.

Return type String

`description`

Returns the description of the *XBeeSettingFormat* element.

Returns Description of the *XBeeSettingFormat* element.

Return type String

class `digi.xbee.profile.`**`XBeeProfileSetting`**(*name*, *setting_type*, *setting_format*, *value*)

Bases: `object`

This class represents an XBee profile setting and provides information like the setting name, type, format and value.

Class constructor. Instantiates a new *`XBeeProfileSetting`* with the given parameters.

Parameters

- **`name`** (*`String`*) – Setting name.
- **`setting_type`** (*`XBeeSettingType`*) – Setting type.
- **`setting_format`** (*`XBeeSettingType`*) – Setting format.
- **`value`** (*`String`*) – Setting value.

`name`

Returns the XBee setting name.

Returns XBee setting name.

Return type String

`type`

Returns the XBee setting type.

Returns XBee setting type.

Return type *XBeeSettingType*

format
Returns the XBee setting format.

Returns XBee setting format.

Return type *XBeeSettingFormat*

value
Returns the XBee setting value as string.

Returns XBee setting value as string.

Return type String

bytearray_value
Returns the XBee setting value as bytearray to be set in the device.

Returns XBee setting value as bytearray to be set in the device.

Return type Bytearray

exception digi.xbee.profile.**ReadProfileException**
Bases: *digi.xbee.exception.XBeeException*

This exception will be thrown when any problem reading the XBee profile occurs.

All functionality of this class is the inherited from Exception.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception digi.xbee.profile.**UpdateProfileException**
Bases: *digi.xbee.exception.XBeeException*

This exception will be thrown when any problem updating the XBee profile into a device occurs.

All functionality of this class is the inherited from Exception.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class digi.xbee.profile.**XBeeProfile**(*profile_file*)
Bases: object

Helper class used to manage serial port break line in a parallel thread.

Class constructor. Instantiates a new *XBeeProfile* with the given parameters.
Parameters **profile_file** (*String*) – Path of the '.xpro' profile file.

Raises

- ProfileReadException – If there is any error reading the profile file.
- ValueError – If the provided profile file is not valid

open()
Opens the profile so its components are accessible from properties *firmware_description_file*, *file_system_path*, *remote_file_system_image*, and *bootloader_file*.

The user is responsible for closing the profile when done with it.

Raises ProfileReadException – If there is any error opening the profile.

See also:

close()
is_open()

close()
Closes the profile. Its components are no more accessible.

See also:

open()
is_open()

is_open()
Returns *True* if the profile is opened, *False* otherwise.

See also:

open()
close()

get_setting_default_value(*setting_name*)
Returns the default value of the given firmware setting.

Parameters **setting_name** (String or *ATStringCommand*) – Name of the setting to retrieve its default value.

Returns

Default value of the setting, *None* if the setting is not found or it has no default value.

Return type String

profile_file
Returns the profile file.

Returns Profile file.

Return type String

version
Returns the profile version.

Returns Profile version.

Return type String

flash_firmware_option
Returns the profile flash firmware option.

Returns Profile flash firmware option.

Return type *FlashFirmwareOption*

See also:

FlashFirmwareOption

description

Returns the profile description.

Returns Profile description.

Return type String

reset_settings

Returns whether the settings of the XBee will be reset before applying the profile ones or not.

Returns

True **if the settings of the XBee will be reset before** applying the profile ones, *False* otherwise.

Return type Boolean

has_local_filesystem

Returns whether the profile has local filesystem information or not.

Returns

True **if the profile has local filesystem information,** *False* otherwise.

Return type Boolean

has_remote_filesystem

Returns whether the profile has remote filesystem information or not.

Returns

True **if the profile has remote filesystem information,** *False* otherwise.

Return type Boolean

has_filesystem

Returns whether the profile has filesystem information (local or remote) or not.

Returns

True **if the profile has filesystem information (local or** remote), *False* otherwise.

Return type Boolean

has_local_firmware_files

Returns whether the profile has local firmware binaries.

Returns

True **if the profile has local firmware files,** *False* otherwise.

Return type Boolean

has_remote_firmware_files

Returns whether the profile has remote firmware binaries.

Returns

True **if the profile has remote firmware files,** *False* otherwise.

Return type Boolean

has_firmware_files

Returns whether the profile has firmware binaries (local or remote).

Returns

***True* if the profile has local or remote firmware files,** *False* otherwise.

Return type Boolean

profile_settings
Returns all the firmware settings that the profile configures.

Returns

List with all the firmware settings that the profile configures (`XBeeProfileSetting`).

Return type Dict

firmware_version
Returns the compatible firmware version of the profile.

Returns Compatible firmware version of the profile.

Return type Integer

hardware_version
Returns the compatible hardware version of the profile.

Returns Compatible hardware version of the profile.

Return type Integer

compatibility_number
Returns the compatibility number of the profile.

Returns The compatibility number, *None* if not defined.

Return type Integer

region_lock
Returns the region lock of the profile.

Returns The region lock, *None* if not defined.

Return type Integer

profile_description_file
Returns the path of the profile description file.

Returns Path of the profile description file.

Return type String

firmware_description_file
Returns the path of the profile firmware description file.

Returns Path of the profile firmware description file.

Return type String

file_system_path
Returns the profile file system path. *None* until the profile is extracted.

Returns Path of the profile file system directory.

Return type String

remote_file_system_image
Returns the path of the remote OTA file system image. *None* until the profile is extracted.

Returns Path of the remote OTA file system image.

Return type String

bootloader_file
Returns the profile bootloader file path. *None* until the profile is extracted.

Returns Path of the profile bootloader file.

Return type String

protocol
Returns the profile XBee protocol.

Returns Profile XBee protocol.

Return type *XBeeProtocol*

digi.xbee.profile.**apply_xbee_profile**(*target*, *profile_path*, *timeout=None*, *progress_callback=None*)
Applies the given XBee profile into the given XBee. If a serial port is provided as *target*, the XBee profile must include the firmware binaries, that are always programmed. In this case, a restore defaults is also performed before applying settings in the profile (no matter if the profile is configured to do so or not). If the value of 'AP' (operating mode) in the profile is not an API mode or it is not defined, XBee is configured to use API 1.

Parameters

- **target** (String or *AbstractXBeeDevice*) – Target to apply profile to. String: serial port identifier. *AbstractXBeeDevice*: XBee to apply the profile.
- **profile_path** (*String*) – path of the XBee profile file to apply.
- **timeout** (*Integer, optional*) – Maximum time to wait for target read operations during the apply profile.
- **progress_callback** (*Function, optional*) – Function to execute to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

Raises

- ValueError – If the XBee profile or the XBee device is not valid.
- *UpdateProfileException* – If there is any error during the update XBee profile operation.

digi.xbee.reader module

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*): 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.**PacketReceivedFrom**
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*): Received packet.
2. sender (*RemoteXBeeDevice*): Remote XBee who sent the packet.

See also:

RemoteXBeeDevice
XBeeAPIPacket
XBeeEvent

append()
Append object to the end of the list.

clear()
Remove all items from list.

copy()
Return a shallow copy of the list.

count()
Return number of occurrences of value.

extend()
Extend list by appending elements from the iterable.

index()
Return first index of value.

Raises ValueError if the value is not present.

insert()
Insert object before index.

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

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

remove()
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse()
Reverse *IN PLACE*.

sort()
Stable sort *IN PLACE*.

class digi.xbee.reader.**DataReceived**
Bases: *digi.xbee.reader.XBeeEvent*

This event is fired when an XBee receives data.
The callbacks for handle this events will receive the following arguments:

1. message (*XBeeMessage*): Message containing the data received, the sender and the time.

See also:

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*): 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*): Received IO sample.
2. sender (*RemoteXBeeDevice*): Remote XBee who sent the packet.
3. time (Integer): the time in which the packet was received.

See also:

IOSample
RemoteXBeeDevice
XBeeEvent

append()
Append object to the end of the list.

clear()
Remove all items from list.

copy()
Return a shallow copy of the list.

count()
Return number of occurrences of value.

extend()
Extend list by appending elements from the iterable.

index()
Return first index of value.

Raises ValueError if the value is not present.

insert()
Insert object before index.

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

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

remove()
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse()
Reverse *IN PLACE*.

sort()
Stable sort *IN PLACE*.

class digi.xbee.reader.**NetworkModified**
Bases: *digi.xbee.reader.XBeeEvent*

This event is fired when the network is being modified by the addition of a new node, an existing node information is updated, a node removal, or when the network items are cleared.
The callbacks that handle this event will receive the following arguments:

1. event_type (*digi.xbee.devices.NetworkEventType*): Network event type.
2. reason (*digi.xbee.devices.NetworkEventReason*): Reason of the event.
3. node (*digi.xbee.devices.XBeeDevice* or *digi.xbee.devices.RemoteXBeeDevice*): Node added, updated or removed from the network.

See also:

digi.xbee.devices.NetworkEventReason
digi.xbee.devices.NetworkEventType
digi.xbee.devices.RemoteXBeeDevice
digi.xbee.devices.XBeeDevice
XBeeEvent

append()
Append object to the end of the list.

clear()
Remove all items from list.

copy()
Return a shallow copy of the list.

count()
Return number of occurrences of value.

extend()
Extend list by appending elements from the iterable.

index()
Return first index of value.

Raises ValueError if the value is not present.

insert()
Insert object before index.

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

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

remove()
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse()
Reverse *IN PLACE*.

sort()
Stable sort *IN PLACE*.

class digi.xbee.reader.**DeviceDiscovered**
Bases: *digi.xbee.reader.XBeeEvent*

This event is fired when an XBee discovers another remote XBee during a discovering operation.
The callbacks that handle this event will receive the following arguments:

1. discovered_device (*RemoteXBeeDevice*): Discovered remote XBee.

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*): Network discovery status.
2. description (String, optional): Description of the discovery status.

See also:

NetworkDiscoveryStatus

XBeeEvent

append()
Append object to the end of the list.

clear()
Remove all items from list.

copy()
Return a shallow copy of the list.

count()
Return number of occurrences of value.

extend()
Extend list by appending elements from the iterable.

index()
Return first index of value.

Raises ValueError if the value is not present.

insert()
Insert object before index.

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

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

remove()
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse()
Reverse *IN PLACE*.

sort()
Stable sort *IN PLACE*.

class digi.xbee.reader.**ExplicitDataReceived**
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 received data, 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, source and destination ports, 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.

Raises ValueError if the value is not present.

insert()
Insert object before index.

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

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

remove()
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse()
Reverse *IN PLACE*.

sort()
Stable sort *IN PLACE*.

class digi.xbee.reader.**SocketStateReceived**
Bases: *digi.xbee.reader.XBeeEvent*

This event is fired when an XBee receives a socket state packet.
The callbacks to handle these events will receive the following arguments:

1. socket_id (Integer): Socket ID for state reported.
2. state (*SocketState*): Received state.

See also:

XBeeEvent

append()
Append object to the end of the list.

clear()
Remove all items from list.

copy()
Return a shallow copy of the list.

count()
Return number of occurrences of value.

extend()
Extend list by appending elements from the iterable.

index()
Return first index of value.

Raises ValueError if the value is not present.

insert()
Insert object before index.

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

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

remove()
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse()
Reverse *IN PLACE*.

sort()
Stable sort *IN PLACE*.

class digi.xbee.reader.**SocketDataReceived**
Bases: *digi.xbee.reader.XBeeEvent*

This event is fired when an XBee receives a socket receive data packet.
The callbacks to handle these events will receive the following arguments:

1. socket_id (Integer): ID of the socket that received the data.
2. payload (Bytearray): Received data.

See also:

XBeeEvent

append()
Append object to the end of the list.

clear()
Remove all items from list.

copy()
Return a shallow copy of the list.

count()
Return number of occurrences of value.

extend()
Extend list by appending elements from the iterable.

index()
Return first index of value.

Raises ValueError if the value is not present.

insert()
Insert object before index.

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

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

remove()
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse()
Reverse *IN PLACE*.

sort()
Stable sort *IN PLACE*.

class digi.xbee.reader.**SocketDataReceivedFrom**
Bases: *digi.xbee.reader.XBeeEvent*

This event is fired when an XBee receives a socket receive from data packet.
The callbacks to handle these events will receive the following arguments:

1. socket_id (Integer): ID of the socket that received the data.
2. **address (Tuple): Pair (host, port) of the source address where** host is a string representing an IPv4 address like '100.50.200.5', and port is an integer.
3. payload (Bytearray): Received data.

See also:

XBeeEvent

append()
Append object to the end of the list.

clear()
Remove all items from list.

copy()
Return a shallow copy of the list.

count()
Return number of occurrences of value.

extend()
Extend list by appending elements from the iterable.

index()
Return first index of value.

Raises ValueError if the value is not present.

insert()
Insert object before index.

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

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

remove()
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse()
Reverse *IN PLACE*.

sort()
Stable sort *IN PLACE*.

class digi.xbee.reader.**RouteRecordIndicatorReceived**
Bases: `digi.xbee.reader.XBeeEvent`

This event is fired when a route record packet is received.
The callbacks to handle these events will receive the following arguments:

1. **Source (`RemoteXBeeDevice`): Remote node that sent the** route record.
2. **Hops (List): List of intermediate hops 16-bit addresses from closest** to source (who sent the route record) to closest to destination (`XBee16BitAddress`).

See also:

`XBeeEvent`

append()
Append object to the end of the list.

clear()
Remove all items from list.

copy()
Return a shallow copy of the list.

count()
Return number of occurrences of value.

extend()
Extend list by appending elements from the iterable.

index()
Return first index of value.

Raises ValueError if the value is not present.

insert()
: Insert object before index.

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

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

remove()
: Remove first occurrence of value.

  Raises ValueError if the value is not present.

reverse()
: Reverse *IN PLACE*.

sort()
: Stable sort *IN PLACE*.

class digi.xbee.reader.**RouteInformationReceived**

Bases: *digi.xbee.reader.XBeeEvent*

This event is fired when a route information packet is received.

The callbacks to handle these events will receive the following arguments:

1. Source event (Integer): Source event (0x11: NACK, 0x12: Trace route)
2. **Timestamp (Integer): System timer value on the node generating** this package. The timestamp is in microseconds.
3. ACK timeout count (Integer): Number of MAC ACK timeouts that occur.
4. **TX blocked count (Integer): Number of times the transmissions was** blocked due to reception in progress.
5. **Destination address (*XBee64BitAddress*): 64-bit address of** the final destination node.
6. **Source address (*XBee64BitAddress*): 64-bit address of** the source node.
7. **Responder address (*XBee64BitAddress*): 64-bit address of** of the node that generates this packet after it sends (or attempts to send) the packet to the next hop (successor node)
8. **Successor address (*XBee64BitAddress*): 64-bit address of** of the next node after the responder in the route towards the destination.

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.**RouteReceived**
Bases: *digi.xbee.reader.XBeeEvent*

This event is fired when a route is received.
The callbacks to handle these events will receive the following arguments:

1. source (*XBeeDevice*): Local node.
2. destination (*RemoteXBeeDevice*): Remote node.
3. **hops (List): List of intermediate hops from source node to** closest to destination (*RemoteXBeeDevice*).

See also:

XBeeEvent

append()
Append object to the end of the list.

clear()
Remove all items from list.

copy()
Return a shallow copy of the list.

count()
Return number of occurrences of value.

extend()
Extend list by appending elements from the iterable.

index()
Return first index of value.

Raises ValueError if the value is not present.

insert()
Insert object before index.

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

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

remove()
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse()
Reverse *IN PLACE*.

sort()
Stable sort *IN PLACE*.

class digi.xbee.reader.**InitDiscoveryScan**
Bases: *digi.xbee.reader.XBeeEvent*

This event is fired when a new network discovery scan is about to start.
The callbacks to handle these events will receive the following arguments:

1. Number of scan to start (starting with 1).
2. Total number of scans.

See also:

XBeeEvent

append()
Append object to the end of the list.

clear()
Remove all items from list.

copy()
Return a shallow copy of the list.

count()
Return number of occurrences of value.

extend()
Extend list by appending elements from the iterable.

index()
Return first index of value.

Raises ValueError if the value is not present.

insert()
Insert object before index.

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

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

remove()
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse()
Reverse *IN PLACE*.

sort()
Stable sort *IN PLACE*.

class digi.xbee.reader.**EndDiscoveryScan**
Bases: *digi.xbee.reader.XBeeEvent*

This event is fired when a network discovery scan has just finished.
The callbacks to handle these events will receive the following arguments:

1. Number of scan that has finished (starting with 1).
2. Total number of scans.

See also:

XBeeEvent

append()
Append object to the end of the list.

clear()
Remove all items from list.

copy()
Return a shallow copy of the list.

count()
Return number of occurrences of value.

extend()
Extend list by appending elements from the iterable.

index()
Return first index of value.

Raises ValueError if the value is not present.

insert()
Insert object before index.

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

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

remove()
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse()
Reverse *IN PLACE*.

sort()
Stable sort *IN PLACE*.

class digi.xbee.reader.**FileSystemFrameReceived**

Bases: *digi.xbee.reader.XBeeEvent*

This event is fired when a file system packet is received.

The callbacks to handle these events will receive the following arguments:

1. Source (*AbstractXBeeDevice*): Node that sent the file system frame.
2. Frame id (Integer): Received frame id.
3. Command (*FSCmd*): File system command.
4. Status (:class: *.FSCommandStatus*): Status code.
5. Receive options (Integer): Bitfield indicating receive options. See *ReceiveOptions*.

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.**PacketListener**(*comm_iface*, *xbee_device*, *queue_max_size=None*)

Bases: threading.Thread

This class represents a packet listener, which is a thread that's always listening for incoming packets to the XBee.

When it receives a packet, this class throws an event depending on which packet it is. You can add your own callbacks for this events via certain class methods. This callbacks must have a certain header, see each event documentation.

This class has fields that are events. Its recommended to use only the append() and remove() method on them, or -= and += operators. If you do something more with them, it's for your own risk.

Here are the parameters which will be received by the event callbacks, depending on which event it is in each case:

The following parameters are passed via **kwargs to event callbacks of:

1. **PacketReceived:** 1.1 received_packet (`XBeeAPIPacket`): Received packet.
2. **DataReceived**

   2.1 message (`XBeeMessage`): Message containing the data received, the sender and the time.
3. **ModemStatusReceived** 3.1 modem_status (`ModemStatus`): Modem status received.

Class constructor. Instantiates a new `PacketListener` object with the provided parameters.

Parameters

- **`comm_iface`** (`XBeeCommunicationInterface`) – Hardware interface to listen to.
- **`xbee_device`** (`XBeeDevice`) – XBee that is the listener owner.
- **`queue_max_size`** (`Integer`) – Maximum size of the XBee queue.

`daemon`

A boolean value indicating whether this thread is a daemon thread.

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when only daemon threads are left.

`wait_until_started`(*timeout=None*)

Blocks until the thread has fully started. If already started, returns immediately.

Parameters **`timeout`** (`Float`) – Timeout for the operation in seconds.

`run`()

This is the method that will be executing for listening packets.

For each packet, it will execute the proper callbacks.

`stop`()

Stops listening.

`is_running`()

Returns whether this instance is running or not.

Returns *True* if this instance is running, *False* otherwise.

Return type Boolean

`get_queue`()

Returns the packets queue.

Returns Packets queue.

Return type `XBeeQueue`

get_data_queue()
Returns the data packets queue.

Returns Data packets queue.

Return type *XBeeQueue*

get_explicit_queue()
Returns the explicit packets queue.

Returns Explicit packets queue.

Return type *XBeeQueue*

get_ip_queue()
Returns the IP packets queue.

Returns IP packets queue.

Return type *XBeeQueue*

add_packet_received_callback(*callback*)
Adds a callback for the event *PacketReceived*.

Parameters **callback** (*Function or List of functions*) – Callback. Receives one argument.

- The received packet as a *XBeeAPIPacket*

add_packet_received_from_callback(*callback*)
Adds a callback for the event *PacketReceivedFrom*.

Parameters **callback** (*Function or List of functions*) – Callback. Receives two arguments.

- The received packet as a *XBeeAPIPacket*
- The remote XBee device who has sent the packet as a *RemoteXBeeDevice*

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

Parameters **callback** (*Function or List of functions*) – Callback. Receives one argument.

- The data received as an *XBeeMessage*

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

Parameters **callback** (*Function or List of functions*) – Callback. Receives one argument.

- The modem status as a *ModemStatus*

add_io_sample_received_callback(*callback*)
Adds a callback for the event *IOSampleReceived*.

Parameters **callback** (*Function or List of functions*) – Callback. Receives three arguments.

- The received IO sample as an *IOSample*
- The remote XBee device who has sent the packet as a *RemoteXBeeDevice*
- The time in which the packet was received as an Integer

add_explicit_data_received_callback(*callback*)
Adds a callback for the event *ExplicitDataReceived*.

Parameters **callback** (*Function or List of functions*) – Callback. Receives one argument.

- The explicit data received as an *ExplicitXBeeMessage*

add_ip_data_received_callback(*callback*)
Adds a callback for the event *IPDataReceived*.

Parameters **callback** (*Function or List of functions*) – Callback. Receives one argument.

- The data received as an *IPMessage*

add_sms_received_callback(*callback*)
Adds a callback for the event *SMSReceived*.

Parameters **callback** (*Function or List of functions*) – Callback. Receives one argument.

- The data received as an *SMSMessage*

add_user_data_relay_received_callback(*callback*)
Adds a callback for the event *RelayDataReceived*.

Parameters **callback** (*Function or List of functions*) – Callback. Receives one argument.

- The data received as a *UserDataRelayMessage*

add_bluetooth_data_received_callback(*callback*)
Adds a callback for the event *BluetoothDataReceived*.

Parameters **callback** (*Function or List of functions*) – Callback. Receives one argument.

- The data received as a Bytearray

add_micropython_data_received_callback(*callback*)
Adds a callback for the event *MicroPythonDataReceived*.

Parameters **callback** (*Function or List of functions*) – Callback. Receives one argument.

- The data received as a Bytearray

add_socket_state_received_callback(*callback*)
Adds a callback for the event *SocketStateReceived*.

Parameters **callback** (*Function or List of functions*) – Callback. Receives two arguments.

- The socket ID as an Integer.
- The state received as a *SocketState*

add_socket_data_received_callback(*callback*)
Adds a callback for the event *SocketDataReceived*.

Parameters **callback** (*Function or List of functions*) – Callback. Receives two arguments.

- The socket ID as an Integer.

- The status received as a *SocketStatus*

add_socket_data_received_from_callback(*callback*)
Adds a callback for the event *SocketDataReceivedFrom*.

Parameters callback (*Function or List of functions*) – Callback. Receives three arguments.

- The socket ID as an Integer.
- A pair (host, port) of the source address where host is a string representing an IPv4 address like '100.50.200.5', and port is an integer.
- The status received as a *SocketStatus*

add_route_record_received_callback(*callback*)
Adds a callback for the event *RouteRecordIndicatorReceived*.

Parameters callback (*Function or List of functions*) – Callback. Receives two arguments.

- **Source (*RemoteXBeeDevice*): Remote node that sent** the route record.
- **Hops (List): List of intermediate hops 16-bit addresses from** closest to source (who sent the route record) to closest to destination.

add_route_info_received_callback(*callback*)
Adds a callback for the event *RouteInformationReceived*.

Parameters callback (*Function or List of functions*) – Callback. Receives eight arguments.

- Source event (Integer): Source event (0x11: NACK, 0x12: Trace route)
- Timestamp (Integer): System timer value on the node generating this package. The timestamp is in microseconds.
- ACK timeout count (Integer): Number of MAC ACK timeouts that occur.
- TX blocked count (Integer): Number of times the transmissions was blocked due to reception in progress.
- Destination address (*XBee64BitAddress*): 64-bit address of the final destination node.
- Source address (*XBee64BitAddress*): 64-bit address of the source node.
- Responder address (*XBee64BitAddress*): 64-bit address of the node that generated this packet after it sent (or attempted to send) the packet to the next hop (successor node)
- Successor address (*XBee64BitAddress*): 64-bit address of the next node after the responder in the route towards the destination.

add_fs_frame_received_callback(*callback*)
Adds a callback for the event *FileSystemFrameReceived*.

Parameters callback (*Function or List of functions*) – Callback. Receives four arguments.

- **Source (*AbstractXBeeDevice*): Node that sent the** file system frame.
- Frame id (Integer): Received frame id.
- Command (*FSCmd*): File system command.

- Receive options (Integer): Bitfield indicating receive options. See `ReceiveOptions`.

del_packet_received_callback(*callback*)
Deletes a callback for the callback list of `PacketReceived` event.

Parameters **callback** (`Function`) – Callback to delete.

Raises `ValueError` – If *callback* is not in the callback list of `PacketReceived` event.

del_packet_received_from_callback(*callback*)
Deletes a callback for the callback list of `PacketReceivedFrom` event.

Parameters **callback** (`Function`) – Callback to delete.

Raises `ValueError` – If *callback* is not in the callback list of `PacketReceivedFrom` event.

del_data_received_callback(*callback*)
Deletes a callback for the callback list of `DataReceived` event.

Parameters **callback** (`Function`) – 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`) – 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`) – 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`) – 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`) – 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`) – 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`) – 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`) – 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`) – Callback to delete.

Raises `ValueError` – If *callback* is not in the callback list of `MicroPythonDataReceived` event.

del_socket_state_received_callback(*callback*)
Deletes a callback for the callback list of `SocketStateReceived` event.

Parameters **callback** (`Function`) – Callback to delete.

Raises `ValueError` – If *callback* is not in the callback list of `SocketStateReceived` event.

del_socket_data_received_callback(*callback*)
Deletes a callback for the callback list of `SocketDataReceived` event.

Parameters **callback** (`Function`) – Callback to delete.

Raises `ValueError` – If *callback* is not in the callback list of `SocketDataReceived` event.

del_socket_data_received_from_callback(*callback*)
Deletes a callback for the callback list of `SocketDataReceivedFrom` event.

Parameters **callback** (`Function`) – Callback to delete.

Raises `ValueError` – If *callback* is not in the callback list of `SocketDataReceivedFrom` event.

del_route_record_received_callback(*callback*)
Deletes a callback for the callback list of `RouteRecordIndicatorReceived` event.

Parameters **callback** (`Function`) – Callback to delete.

Raises `ValueError` – If *callback* is not in the callback list of `RouteRecordIndicatorReceived` event.

del_route_info_callback(*callback*)
Deletes a callback for the callback list of `RouteInformationReceived` event.

Parameters **callback** (`Function`) – Callback to delete.

Raises `ValueError` – If *callback* is not in the callback list of `RouteInformationReceived` event.

del_fs_frame_received_callback(*callback*)
Deletes a callback for the callback list of `FileSystemFrameReceived` event.

Parameters **callback** (`Function`) – Callback to delete.

Raises `ValueError` – If *callback* is not in the callback list of `FileSystemFrameReceived` event.

get_packet_received_callbacks()
Returns the list of registered callbacks for received packets.

Returns List of *PacketReceived* events.

Return type List

get_packet_received_from_callbacks()
Returns the list of registered callbacks for received packets.

Returns List of *PacketReceivedFrom* events.

Return type List

get_data_received_callbacks()
Returns the list of registered callbacks for received data.

Returns List of *DataReceived* events.

Return type List

get_modem_status_received_callbacks()
Returns the list of registered callbacks for received modem status.

Returns List of *ModemStatusReceived* events.

Return type List

get_io_sample_received_callbacks()
Returns the list of registered callbacks for received IO samples.

Returns List of *IOSampleReceived* events.

Return type List

get_explicit_data_received_callbacks()
Returns the list of registered callbacks for received explicit data.

Returns List of *ExplicitDataReceived* events.

Return type List

get_ip_data_received_callbacks()
Returns the list of registered callbacks for received IP data.

Returns List of *IPDataReceived* events.

Return type List

get_sms_received_callbacks()
Returns the list of registered callbacks for received SMS.

Returns List of *SMSReceived* events.

Return type List

get_user_data_relay_received_callbacks()
Returns the list of registered callbacks for received user data relay.

Returns List of *RelayDataReceived* events.

Return type List

get_bluetooth_data_received_callbacks()
Returns the list of registered callbacks for received Bluetooth data.

Returns List of *BluetoothDataReceived* events.

Return type List

get_micropython_data_received_callbacks()
Returns the list of registered callbacks for received MicroPython data.

Returns List of *MicroPythonDataReceived* events.

Return type List

get_socket_state_received_callbacks()
Returns the list of registered callbacks for received socket state.

Returns List of *SocketStateReceived* events.

Return type List

get_socket_data_received_callbacks()
Returns the list of registered callbacks for received socket data.

Returns List of *SocketDataReceived* events.

Return type List

get_socket_data_received_from_callbacks()
Returns the list of registered callbacks for received socket data from.

Returns List of *SocketDataReceivedFrom* events.

Return type List

get_route_record_received_callbacks()
Returns the list of registered callbacks for received route records.

Returns List of *RouteRecordIndicatorReceived* events.

Return type List

get_route_info_callbacks()
Returns the list of registered callbacks for received route information packets.

Returns List of *RouteInformationReceived* events.

Return type List

get_fs_frame_received_callbacks()
Returns the list of registered callbacks for received file system packets.

Returns List of *FileSystemFrameReceived* events.

Return type List

ident
Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

isAlive()
Return whether the thread is alive.

This method is deprecated, use is_alive() instead.

is_alive()
Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

join(*timeout=None*)

Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call is_alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

name

A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

start()

Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

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 **maxsize** (*Integer, optional, default=10*) – Maximum size of the queue.

get(*block=True*, *timeout=None*)

Returns the first element of the queue if there is some element ready before timeout expires, in case of the timeout is not *None*.

If timeout is *None*, this method is non-blocking. In this case, if there is not 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

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 is not any packet available before the timeout expires.

get_by_remote(*remote*, *timeout=None*)

Returns the first element of the queue that had been sent by *remote*, if there is some in the specified timeout.

If timeout is *None*, this method is non-blocking. In this case, if there is not any packet sent by *remote* in the queue, it returns *None*, otherwise it returns an `XBeeAPIPacket`.

Parameters

- **remote** (`RemoteXBeeDevice`) – Remote XBee to get its first element from queue.
- **timeout** (*Integer, optional, default=`None`*) – 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 is not any packet available that was sent by *remote* before the timeout expires.

get_by_ip(*ip_addr*, *timeout=None*)

Returns the first IP data packet from the queue whose IP address matches the provided address.

If timeout is *None*, this method is non-blocking. In this case, if there is not any packet sent by *ip_addr* in the queue, it returns *None*, otherwise it returns an `XBeeAPIPacket`.

Parameters

- **ip_addr** (`ipaddress.IPv4Address`) – IP address to look for in the list of packets.
- **timeout** (*Integer, optional, default=`None`*) – 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 is not any packet available that was sent by *ip_addr* before the timeout expires.

empty()

Return True if the queue is empty, False otherwise (not reliable!).

This method is likely to be removed at some point. Use qsize() == 0 as a direct substitute, but be aware that either approach risks a race condition where a queue can grow before the result of empty() or qsize() can be used.

To create code that needs to wait for all queued tasks to be completed, the preferred technique is to use the join() method.

full()

Return True if the queue is full, False otherwise (not reliable!).

This method is likely to be removed at some point. Use qsize() >= n as a direct substitute, but be aware that either approach risks a race condition where a queue can shrink before the result of full() or qsize() can be used.

get_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 is not any received packet with the provided frame ID in the queue, it returns *None*, otherwise it returns an `XBeeAPIPacket`.

Parameters

- **frame_id** (*Integer*) – Frame ID to look for in the list of packets.
- **timeout** (*Integer, optional, default=`None`*) – 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 is not any packet available that matches the provided frame ID before the timeout expires.

get_nowait()

Remove and return an item from the queue without blocking.

Only get an item if one is immediately available. Otherwise raise the Empty exception.

join()

Blocks until all items in the Queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls task_done() to indicate the item was retrieved and all work on it is complete.

When the count of unfinished tasks drops to zero, join() unblocks.

put(*item*, *block=True*, *timeout=None*)

Put an item into the queue.

If optional args 'block' is true and 'timeout' is None (the default), block if necessary until a free slot is available. If 'timeout' is a non-negative number, it blocks at most 'timeout' seconds and raises the Full exception if no free slot was available within that time. Otherwise ('block' is false), put an item on the queue if a free slot is immediately available, else raise the Full exception ('timeout' is ignored in that case).

put_nowait(*item*)

Put an item into the queue without blocking.

Only enqueue the item if a free slot is immediately available. Otherwise raise the Full exception.

qsize()

Return the approximate size of the queue (not reliable!).

task_done()

Indicate that a formerly enqueued task is complete.

Used by Queue consumer threads. For each get() used to fetch a task, a subsequent call to task_done() tells the queue that the processing on the task is complete.

If a join() is currently blocking, it will resume when all items have been processed (meaning that a task_done() call was received for every item that had been put() into the queue).

Raises a ValueError if called more times than there were items placed in the queue.

flush()
Clears the queue.

digi.xbee.recovery module

digi.xbee.recovery.**recover_device**(*target*)
Recovers the XBee from an unknown state and leaves if configured for normal operations.

Parameters **target** (String or *XBeeDevice*) – Target of the recovery operation.

Raises RecoveryException – If there is any error performing the recovery action.

digi.xbee.recovery.**enter_at_command_mode**(*port*)
Attempts to put this device in AT Command mode.

Parameters **port** – The serial port where the XBee is connected to.

Returns

True* if the XBee has entered in AT command mode, *False otherwise.

Return type Boolean

Raises

- SerialTimeoutException – If there is any error trying to write to the serial port.
- InvalidOperatingModeException – If the XBee is in API mode.

digi.xbee.sender module

class digi.xbee.sender.**PacketSender**(*xbee*)
Bases: object

Class to send XBee packets.

Class constructor. Instantiates a new *PacketSender* object with the provided parameters.

Parameters **xbee** (*XBeeDevice*) – The XBee.

send_packet(*packet*)
Sends a packet to the XBee. The packet to send is escaped depending on the current operating mode.

Parameters **packet** (*XBeePacket*) – The packet 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 the XBee device's communication interface is closed.

See also:

XBeePacket

is_op_mode_valid(*value*)
Returns *True* if the provided value is a valid operating mode for the library.

Parameters **value** (*Bytearray*) – The value to check.

Returns *True* for a valid value, *False* otherwise.

Return type Boolean

`at_response_received_cb`(*response*)
Callback to deal with AT command responses and update the corresponding node. Only for internal use.

Parameters **(**(*`response`*) – class: *.XBeeAPIPacket*): The received API packet.

`class` `digi.xbee.sender.`**`SyncRequestSender`**(*xbee*, *packet_to_send*, *timeout*)
Bases: `object`

Class to synchronously send XBee packets. This means after sending the packet it waits for its response, if the package includes a frame ID, otherwise it does not wait.

Class constructor. Instantiates a new *`SyncRequestSender`* object with the provided parameters.

Parameters

- **`xbee`** (*`XBeeDevice`*) – The local XBee to send the packet.
- **`packet_to_send`** (*`XBeePacket`*) – The packet to transmit.
- **`timeout`** (*`Integer`*) – Number of seconds to wait. -1 to wait indefinitely.

`send`()
Sends the packet and waits for its corresponding response.

Returns Received response packet.

Return type *`XBeePacket`*

Raises

- `InvalidOperatingModeException` – If the XBee device's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TimeoutException` – If the response is not received in the configured timeout.
- `XBeeException` – If the XBee device's communication interface is closed.

See also:

`XBeePacket`

`xbee`
Returns the local XBee to send the packet.

Returns Local XBee device.

Return type *`XBeeDevice`*

`packet`
Returns the packet to send.

Returns Packet to send.

Return type *`XBeePacket`*

`timeout`
Returns the maximum number of seconds to wait for a response.

Returns Timeout to wait for a response.

Return type Integer

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`, `digi.xbee.comm_interface.XBeeCommunicationInterface`

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

It also introduces a minor change in its behaviour: the serial port is not automatically open when instantiated, only when calling open().

See also:

_PySerial: https://github.com/pyserial/pyserial

Class constructor. Instantiates a new *XBeeSerialPort* object with the given port parameters.

Parameters

- **baud_rate** (*Integer*) – Serial port baud rate.
- **port** (*String*) – Serial port name to use.
- **data_bits** (*Integer, optional, default=8*) – Serial data bits.
- **stop_bits** (*Float, optional, default=1*) – sSerial stop bits.
- **parity** (*Char, optional, default=`N`*) – Parity. Default to 'N' (None).
- **flow_control** (*Integer, optional, default=`None`*) – Flow control.
- **timeout** (*Integer, optional, default=0.1*) – Read timeout (seconds).

See also:

_PySerial: https://github.com/pyserial/pyserial

is_interface_open

Returns whether the underlying hardware communication interface is active.

Returns Boolean. *True* if the interface is active, *False* otherwise.

write_frame(*frame*)

Writes an XBee frame to the underlying hardware interface.

Subclasses may throw specific exceptions to signal implementation specific hardware errors.

Parameters **frame** (*Bytearray*) – The XBee API frame packet to write. If the bytearray does not correctly represent an XBee frame, the behaviour is undefined.

read_byte()
: Synchronous. Reads one byte from serial port.

  Returns The read byte.

  Return type Integer

  Raises `TimeoutException` – If there is no bytes ins serial port buffer.

read_bytes(*num_bytes*)
: Synchronous. Reads the specified number of bytes from the serial port.

  Parameters **num_bytes** (*Integer*) – the number of bytes to read.

  Returns the read bytes.

  Return type Bytearray

  Raises `TimeoutException` – if the number of bytes read is less than *num_bytes*.

quit_reading()
: Makes the thread (if any) blocking on wait_for_frame return.

  If a thread was blocked on wait_for_frame, this method blocks (for a maximum of 'timeout' seconds) until the blocked thread is resumed.

wait_for_frame(*operating_mode*)
: Reads the next packet. Starts to read when finds the start delimiter. The last byte read is the checksum.

  If there is something in the COM buffer after the start delimiter, this method discards it.

  If the method can't read a complete and correct packet, it will return *None*.

  Parameters **operating_mode** (*OperatingMode*) – The operating mode in which the packet should be read.

  Returns

  The read packet as bytearray if a packet is read, *None* otherwise.

  Return type Bytearray

read_existing()
: Asynchronous. Reads all bytes in the serial port buffer. May read 0 bytes.

  Returns The bytes read.

  Return type Bytearray

get_read_timeout()
: Returns the serial port read timeout.

  Returns Read timeout in seconds.

  Return type Integer

set_read_timeout(*read_timeout*)
: Sets the serial port read timeout in seconds.

  Parameters **read_timeout** (*Integer*) – The new serial port read timeout in seconds.

set_baudrate(*new_baudrate*)
: Changes the serial port baudrate.

  Parameters **new_baudrate** (*Integer*) – The new baudrate to set.

purge_port()
: Purges the serial port by cleaning the input and output buffers.

apply_profile(*xbee*, *profile_path*, *timeout=None*, *progress_callback=None*)
Applies the given XBee profile to the XBee device.

Parameters

- **xbee** (*AbstractXBeeDevice*) – Local or remote XBee node to be updated.
- **profile_path** (*String*) – Path of the XBee profile file to apply.
- **timeout** (*Integer, optional*) – Maximum time to wait for target read operations during the apply profile.
- **progress_callback** (*Function, optional*) – Function to execute to receive progress information. Receives two arguments:
  - The current apply profile task as a String
  - The current apply profile task percentage as an Integer

Raises

- `XBeeException` – If the local XBee is not open.
- `InvalidOperatingModeException` – If the local XBee operating mode is invalid.
- `UpdateProfileException` – If there is any error applying the XBee profile.
- `OperationNotSupportedException` – If XBee profiles are not supported in the XBee.

close()
Terminates the underlying hardware communication interface.

Subclasses may throw specific exceptions to signal implementation specific hardware errors.

get_local_xbee_info()
Returns a tuple with the local XBee information.

This is used when opening the local XBee. If this information is provided, it is used as internal XBee data, if not provided, the data is requested to the XBee.

Returns

Tuple with local XBee information: operation mode (int), hardware version (int), firmware version (int), 64-bit address (string), 16-bit address (string), node identifier (string), and role (int).

Return type Tuple

get_network(*local_xbee*)
Returns the XBeeNetwork object associated to the XBeeDevice associated to this *XBeeCommunicationInterface*.

Some *XBeeCommunicationInterface implementations may need to handle the 'XBeeNetwork* associated to the *XBeeDevice* themselves. If that is the case, a implementation-specific XBeeNetwork object that complains to the generic *XBeeNetwork* class will be returned. Otherwise, this method returns *None* and the associated *XBeeNetwork* is handled as for a serial-connected *XBeeDevice*.

Parameters **local_xbee** (*XBeeDevice*) – The local XBee device.

Returns

class: .*XBeeNetwork*: *None* if the XBeeNetwork should handled as usual, otherwise a *XBeeNetwork* object.

open()
Establishes the underlying hardware communication interface.

Subclasses may throw specific exceptions to signal implementation specific errors.

supports_apply_profile()
Returns if the interface supports the apply profile feature.

Returns *True* if it is supported, *False* otherwise.

Return type Boolean

supports_update_firmware()
Returns if the interface supports the firmware update feature.

Returns *True* if it is supported, *False* otherwise.

Return type Boolean

timeout
Returns the read timeout.

Returns Read timeout in seconds.

Return type Integer

update_firmware(*xbee*, *xml_fw_file*, *xbee_fw_file=None*, *bootloader_fw_file=None*, *timeout=None*, *progress_callback=None*)
Performs a firmware update operation of the provided XBee.

Parameters

- **xbee** (*AbstractXBeeDevice*) – Local or remote XBee node to be updated.
- **xml_fw_file** (*String*) – Path of the XML file that describes the firmware to upload.
- **xbee_fw_file** (*String, optional*) – Location of the XBee binary firmware file.
- **bootloader_fw_file** (*String, optional*) – Location of the bootloader binary firmware file.
- **timeout** (*Integer, optional*) – Maximum time to wait for target read operations during the update process.
- **progress_callback** (*Function, optional*) – Function to execute to receive progress information. Receives two arguments:
  - The current update task as a String
  - The current update task percentage as an Integer

Raises

- XBeeException – If the local XBee is not open.
- InvalidOperatingModeException – If the local XBee operating mode is invalid.
- OperationNotSupportedException – If the firmware update is not supported in the XBee.
- FirmwareUpdateException – If there is any error performing the firmware update.

digi.xbee.xsocket module

class digi.xbee.xsocket.**socket**(*xbee_device*, *ip_protocol=<IPProtocol.TCP: (1, 'TCP')>*)
Bases: `object`

This class represents an XBee socket and provides methods to create, connect, bind and close a socket, as well as send and receive data with it.

Class constructor. Instantiates a new XBee socket object for the given XBee device.

Parameters

- **xbee_device** (*XBeeDevice*) – XBee device of the socket.
- **ip_protocol** (*IPProtocol*) – protocol of the socket.

Raises

- `ValueError` – if *xbee_device* is *None* or if *xbee_device* is not an instance of *CellularDevice*.
- `ValueError` – if *ip_protocol* is *None*.
- `XBeeException` – if the connection with the XBee device is not open.

connect(*address*)
Connects to a remote socket at the given address.

Parameters **address** (*Tuple*) – A pair *(host, port)* where *host* is the domain name or string representation of an IPv4 and *port* is the numeric port value.

Raises

- `TimeoutException` – If the connect response is not received in the configured timeout.
- `ValueError` – If *address* is *None* or not a pair *(host, port)*.
- `ValueError` – If *port* is less than 1 or greater than 65535.
- `XBeeException` – If the connection with the XBee device is not open.
- `XBeeSocketException` – If the connect status is not *SUCCESS*.

bind(*address*)
Binds the socket to the given address. The socket must not already be bound.

Parameters **address** (*Tuple*) – A pair *(host, port)* where *host* is the local interface (not used) and *port* is the numeric port value.

Raises

- `TimeoutException` – If the bind response is not received in the configured timeout.
- `ValueError` – If *address* is *None* or not a pair *(host, port)*.
- `ValueError` – If *port* is less than 1 or greater than 65535.
- `XBeeException` – If the connection with the XBee device is not open.
- `XBeeSocketException` – If the bind status is not *SUCCESS*.
- `XBeeSocketException` – If the socket is already bound.

listen(*backlog=1*)
Enables a server to accept connections.

Parameters **backlog** (*Integer, optional*) – The number of unaccepted connections that the system will allow before refusing new connections. If specified, it must be at least 0 (if it is lower, it is set to 0).

Raises `XBeeSocketException` – If the socket is not bound.

accept()

Accepts a connection. The socket must be bound to an address and listening for connections.

Returns

A pair *(conn, address)* where *conn* is a new socket object usable to send and receive data on the connection, and *address* is a pair *(host, port)* with the address bound to the socket on the other end of the connection.

Return type Tuple

Raises

- `XBeeException` – If the connection with the XBee device is not open.
- `XBeeSocketException` – If the socket is not bound or not listening.

gettimeout()

Returns the configured socket timeout in seconds.

Returns The configured timeout in seconds.

Return type Integer

settimeout(*timeout*)

Sets the socket timeout in seconds.

Parameters **timeout** (*Integer*) – The new socket timeout in seconds.

getblocking()

Returns whether the socket is in blocking mode or not.

Returns *True* if the socket is in blocking mode, *False* otherwise.

Return type Boolean

setblocking(*flag*)

Sets the socket in blocking or non-blocking mode.

Parameters **flag** (*Boolean*) – *True* to set the socket in blocking mode, *False* to set it in no blocking mode and configure the timeout with the default value (*5* seconds).

recv(*bufsize*)

Receives data from the socket.

Parameters **bufsize** (*Integer*) – The maximum amount of data to be received at once.

Returns The data received.

Return type Bytearray

Raises `ValueError` – If *bufsize* is less than *1*.

recvfrom(*bufsize*)

Receives data from the socket.

Parameters **bufsize** (*Integer*) – The maximum amount of data to be received at once.

Returns

Pair containing the data received (Bytearray) and the address of the socket sending the data. The address is also a pair *(host, port)* where *host* is the string representation of an IPv4 and *port* is the numeric port value.

Return type Tuple (Bytearray, Tuple)

Raises `ValueError` – If *bufsize* is less than *1*.

send(*data*)

Sends data to the socket and returns the number of bytes sent. The socket must be connected to a remote socket. Applications are responsible for checking that all data has been sent; if only some of the data was transmitted, the application needs to attempt delivery of the remaining data.

Parameters **data** (*Bytearray*) – The data to send.

Returns The number of bytes sent.

Return type Integer

Raises

- `ValueError` – If the data to send is *None*.
- `ValueError` – If the number of bytes to send is *0*.
- `XBeeException` – If the connection with the XBee device is not open.
- `XBeeSocketException` – If the socket is not valid.
- `XBeeSocketException` – If the socket is not open.

sendall(*data*)

Sends data to the socket. The socket must be connected to a remote socket. Unlike *send()*, this method continues to send data from bytes until either all data has been sent or an error occurs. *None* is returned on success. On error, an exception is raised, and there is no way to determine how much data, if any, was successfully sent.

Parameters **data** (*Bytearray*) – The data to send.

Raises

- `TimeoutException` – If the send status response is not received in the configured timeout.
- `ValueError` – If the data to send is *None*.
- `ValueError` – If the number of bytes to send is *0*.
- `XBeeException` – If the connection with the XBee device is not open.
- `XBeeSocketException` – If the socket is not valid.
- `XBeeSocketException` – If the send status is not *SUCCESS*.
- `XBeeSocketException` – If the socket is not open.

sendto(*data*, *address*)

Sends data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by *address*.

Parameters

- **data** (*Bytearray*) – The data to send.
- **address** (*Tuple*) – The address of the destination socket. It must be a pair *(host, port)* where *host* is the domain name or string representation of an IPv4 and *port* is the numeric port value.

Returns The number of bytes sent.

Return type Integer

Raises

- `TimeoutException` – If the send status response is not received in the configured timeout.
- `ValueError` – If the data to send is *None*.
- `ValueError` – If the number of bytes to send is *0*.
- `XBeeException` – If the connection with the XBee device is not open.
- `XBeeSocketException` – If the socket is already open.
- `XBeeSocketException` – If the send status is not *SUCCESS*.

close()

Closes the socket.

Raises

- `TimeoutException` – If the close response is not received in the configured timeout.
- `XBeeException` – If the connection with the XBee device is not open.
- `XBeeSocketException` – If the close status is not *SUCCESS*.

setsocketopt(*option*, *value*)

Sets the value of the given socket option.

Parameters

- **option** (*SocketOption*) – The socket option to set its value.
- **value** (*Bytearray*) – The new value of the socket option.

Raises

- `TimeoutException` – If the socket option response is not received in the configured timeout.
- `ValueError` – If the option to set is *None*.
- `ValueError` – If the value of the option is *None*.
- `XBeeException` – If the connection with the XBee device is not open.
- `XBeeSocketException` – If the socket option response status is not *SUCCESS*.

getsocketopt(*option*)

Returns the value of the given socket option.

Parameters **option** (*SocketOption*) – The socket option to get its value.

Returns The value of the socket option.

Return type Bytearray

Raises

- `TimeoutException` – If the socket option response is not received in the configured timeout.
- `ValueError` – If the option to set is *None*.

- `XBeeException` – If the connection with the XBee device is not open.
- `XBeeSocketException` – If the socket option response status is not *SUCCESS*.

add_socket_state_callback(*callback*)

Adds a callback for the event *digi.xbee.reader.SocketStateReceived*.

Parameters **callback** (*Function*) – The callback. Receives two arguments.

- The socket ID as an Integer.
- The state received as a *SocketState*

del_socket_state_callback(*callback*)

Deletes a callback for the callback list of *digi.xbee.reader.SocketStateReceived* event.

Parameters **callback** (*Function*) – The callback to delete.

Raises `ValueError` – If *callback* is not in the callback list of *digi.xbee.reader.SocketStateReceived* event.

get_sock_info()

Returns the information of this socket.

Returns The socket information.

Return type *SocketInfo*

Raises

- `InvalidOperatingModeException` – If the XBee device's operating mode is not API or ESCAPED API. This method only checks the cached value of the operating mode.
- `TimeoutException` – If the response is not received before the read timeout expires.
- `XBeeException` – If the XBee device's communication interface is closed.

See also:

SocketInfo

is_connected

Returns whether the socket is connected or not.

Returns *True* if the socket is connected *False* otherwise.

Return type Boolean

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

CHAPTER 4

License

Python Module Index

Index