UD002 User and API Reference for the Module serial_port_com

Scope

This document provides user reference on the module *com_lib.serial_port_com*, including design, functionality, implementation details and API reference.

Functional components:

- Function *list_port*()
- Class **SimpleCOM_API**
- Custom exception classes **UT_SerialException** and **UT_SerialTimeoutException**

Design and Functionality

This module implements *atomic* sending and receiving of data over serial port connection using zero-terminated packages - the COBS-encoded bytestrings with the added b'\x00' byte. The supported modes of operation are:

- Asynchronous communication
    - Uni-directional
        - Only sending with the responses being ignored / not claimed, even if any is issued
        - Only listening to an external data provider
    - Bi-directional - sending packages and checking for the incoming data / acquiring the received data in an arbitrary order
- Synchronous communication - bi-directional by definition, when for each sending the response is waited for and reclaimed - the other side MUST send a response to each received data package
- Mixing the synchronous and asynchronous sending and receiving - the other side MUST send a response to each received data package

Note, that in the mixed mode some responses to the asynchronous sendings may be lost depending on the sequence of the send / received requests made, whereas for each synchronous sending the retrieval of the response is guaranteed.

The synchronous send and receive can be performed in the blocking mode or in the timeouted mode. In the blocking mode the control is not returned to the caller until the respose is received, however long it may take. In the timeouted mode the connection is automatically closed and the **UT_SerialTimeoutException** is raised.

Although the actual packages are sent and received as bytestrings, different data types are acceptable as data to be send:

- Bytestring (Python type **bytes**) - no additional data conversion is applied except for the COBS encoding before sending
- Byte arrays (Python type **bytearray**) - simply repacked as bytestrings before encoding

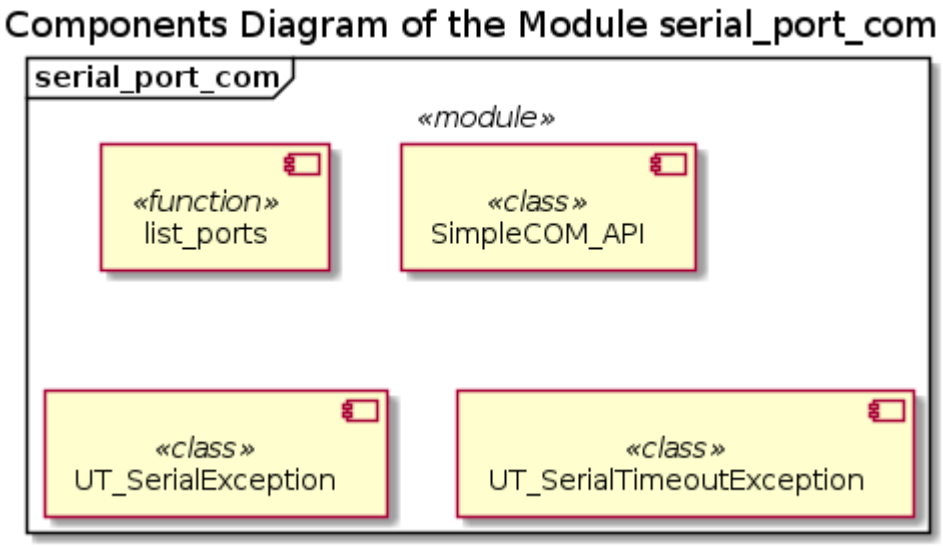
- Unicode strings (Python type **str**) - encoded into a bytestring using 'utf-8' codecs, then COBS encoded
- Any object (class instance) with an instance method *packBytes*(), which is called without an argument and returns a bytestring; that returned bytestring is COBS encoded before sending

The tailing zero-terminator is automatically stripped in the received packages, which are than COBS decoded and returned to the caller. However, a different return data type can be requested:

- **bytearray** (as type, not value) - an instance of the byte array is created from the received bytestring and returned
- **str** (as type, not value) - the bytestring is 'utf-8' decoded into a Unicode string, which is returned
- a user defined class (as **type**, not instance as **object**), which has a class method *unpackBytes*() accepting a single bytestring argument and returning any value, which will be passed to the caller

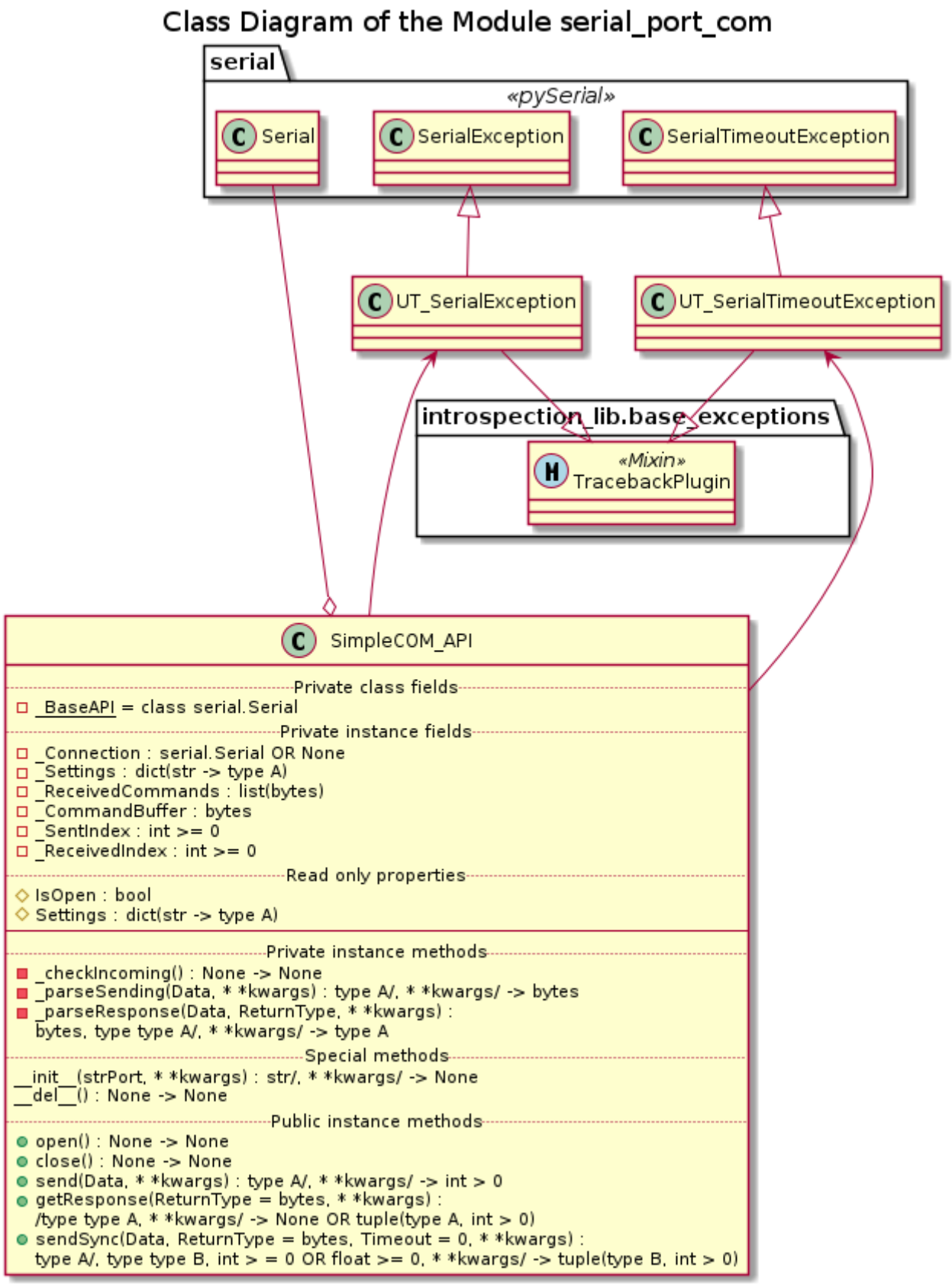
Implementation Details

The components diagram of the module is below.



The function *list_ports*() analizes the data returned by the function *serial.tools.list_ports.comports*() and selects only those entries, for which both vendor and product IDs are not None. The selected entries are returned as a list of 3-elements tuples in the format (port_path: str, VID: int, PID: int).

The next illustration is the class diagram of the module.



The classes **UT_SerialException** and **UT_SerialTimeoutException** sub-class the respective **serial.SerialException** and **serial.SerialTimeoutException** exceptions, and mix them with the **introspection_lib.base_exceptions.TracebackPlugin** class, thus enabling the enhanced traceback analysis functionality via the added read-only property *Traceback*.

The class **SimpleCOM_API** wraps functionality of the class **serial.Serial**. Note, that the said class is stored (by reference, as a type) in the 'private' class attribute *_BaseAPI*, and instantiation of the wrapped class

occurs only upon instantiation of the **SimpleCOM_API** class itself. Therefore, a sub-class of the **SimpleCOM_API** can simply change the value of the *_BaseAPI* attribute to interface another implementation of the serial communication. For instance, in the unit tests made for this module, the mock serial class **com_lib.mock_serial.MockSerial** is interfaced instead of **serial.Serial**. However, the replacement interface must provide minimum compatibility API:

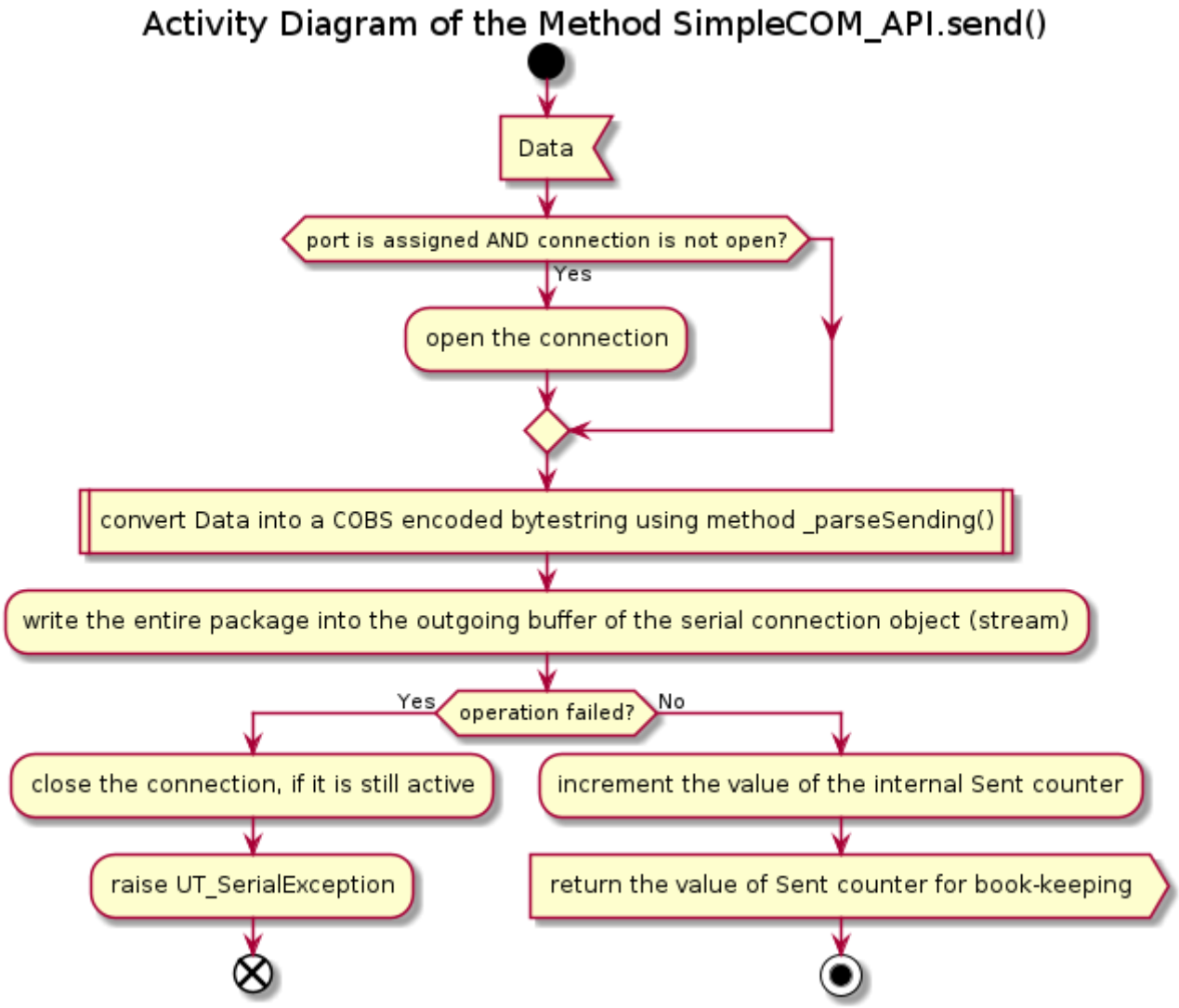
- Readable field / property attributes:
    - *is_open*: **bool**
    - *in_waiting*: **int** >= 0
- Writeable field / property *baudrate*: **int** >= 0
- Instance method *write*() accepting a single bytestring argument
- Instance mehtod *read*() with the signature **int** >0 -> **bytes**

The initialization method of the **SimpleCOM_API** requires a string argument - the path to the port to open, with additional connection settings can be passed as keyword arguments, supported by the initialization method of the **serial.Serial** class. Note, that the passed values for *timeout* (read) and *write_timeout* (write) are ignored and replaced by 0.

The connection is established directly upon instantiation of the class; and the connection is closed by calling the method *close*(). However, the closed connection can be re-openned with the method *open*(). Unlike the standard behaviour of the **serial.Serial** class an attempt to close a closed connection or to open the already open connection is not an error - an exception is not raised, and the request is simply ignored.

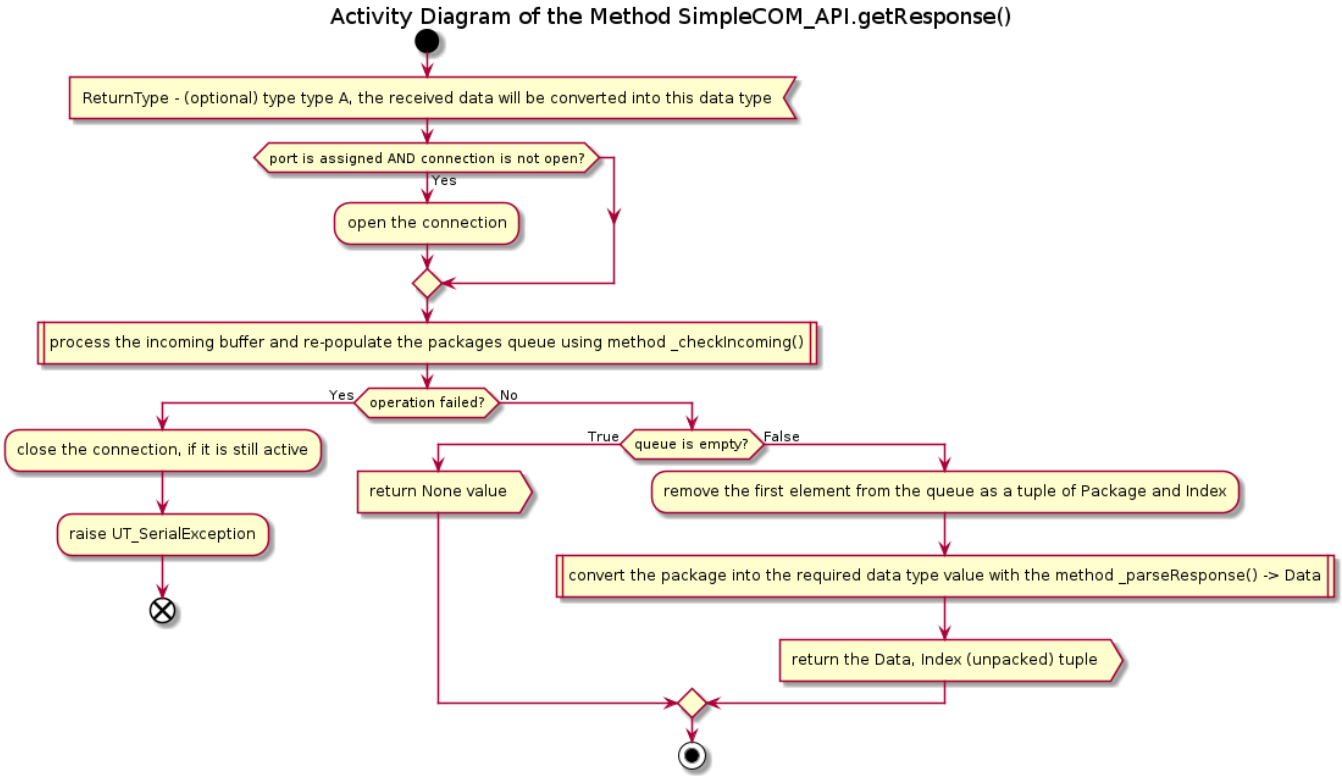
The connection must be in the 'open' state (use read-only property *IsOpen* to check) in order to send and / or receive the data; otherwise **UT_SerialException** is raised.

The method *send*() implements asynchronous sending of a package. It accepts the input data of any of the supported data types (bytestring, byte array, normal string or an instance of an auto-serializable class), converts into a COBS encoded, zero-terminated bytestring package, places it into the outgoing buffer of the serial port, increments and returns the value of the internal counter of the sent packages. Note that if the path to the serial port is properly set but the connection is not open, the method also opens the connection automatically.



The responses to the sent packages are accumulated in the incoming buffer of the serial port handling object, until they are explicitely claimed either by the method *getResponse*() or by the method *sendSync*(). Therefore, the both methods implement *greedy* data retrieval - all packages currently pending in the incoming buffer are retrieved at once and placed into an internal queue of the class. The both methods also implement convertion of the received data into the supported data type value and automatic (re-) openning of the connection.

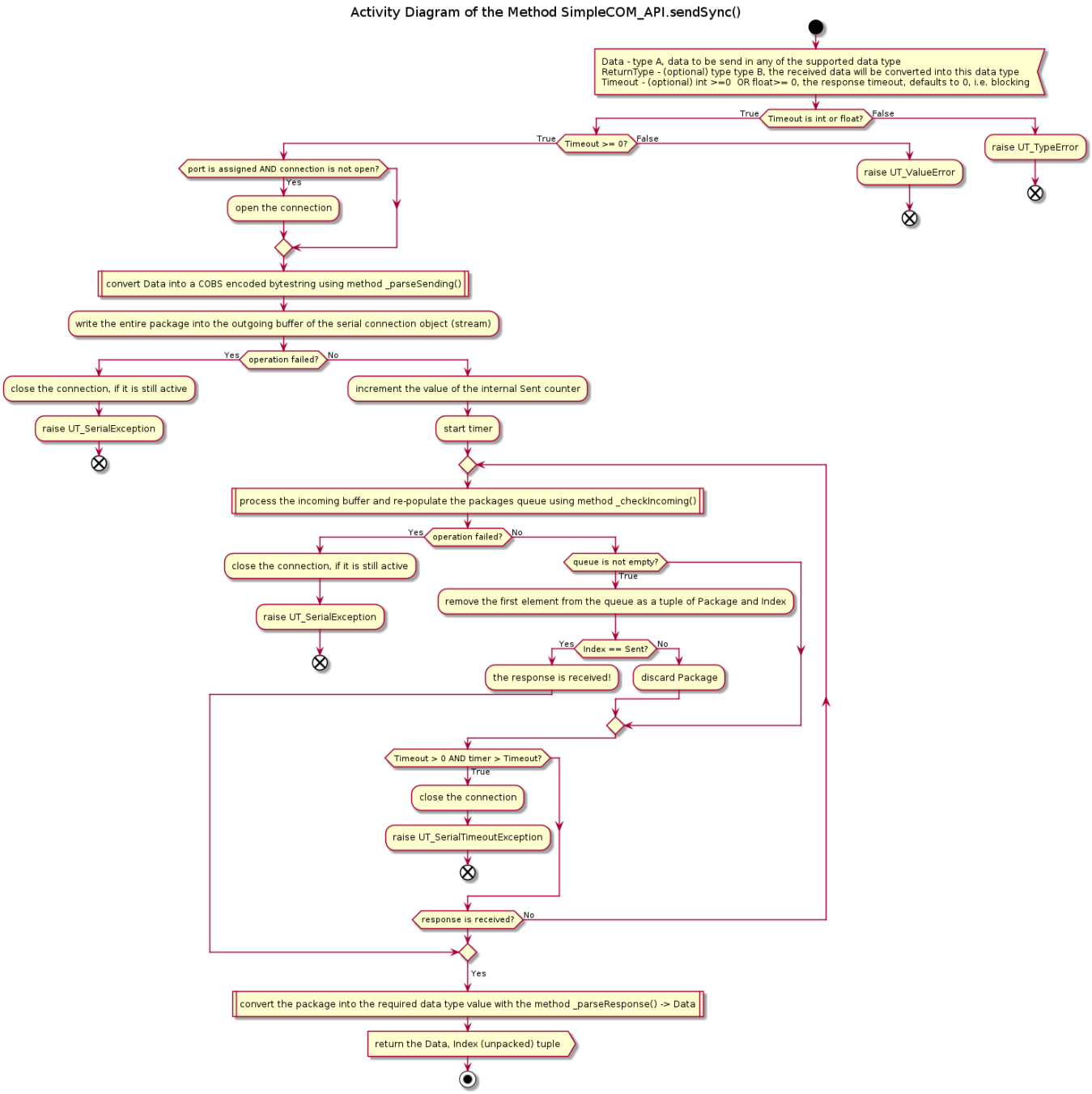
The method *getResponse*() always returns the first package waiting in the queue (together with its received index) or **None** value is the queue is empty; and it doesn't support the *read timeout* functionality - the call is always non-blocking.



The method *sendSync*() implements the synchronous communication - it awaits and returns the response to the sending. It implements two modes:

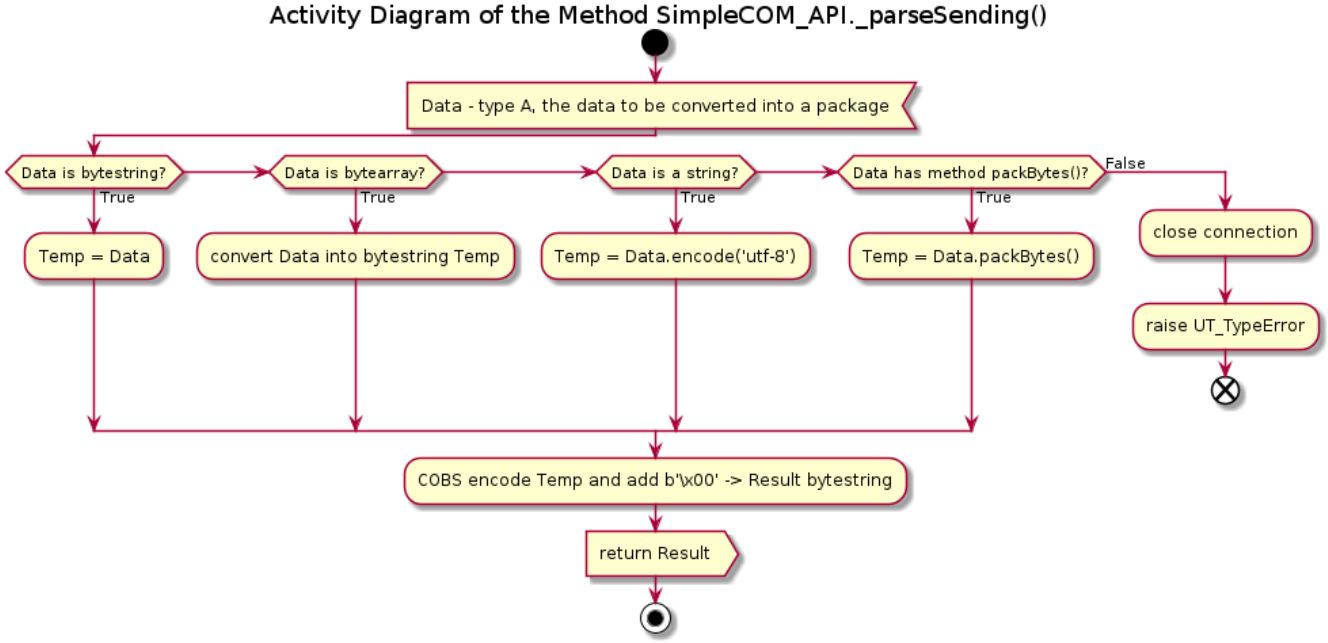
- blocking - when it waits indefinitely until the response is received or disconnection occurs
- timed-out - when it waits until the response is received but no longer than a specified timeout period; if the timeout is reached the connection is closed, and an exception is raised

The synchronous sending and receiving functionality is implemented under the assumption that the connected device always sends a response, thus the method *sendSync*() compares the index of the received package with the index of the last sent package. The received package is considered to be the response to the made sending if the indexes are equal. All yet unclaimed packages recevied earlier are simply discarded.

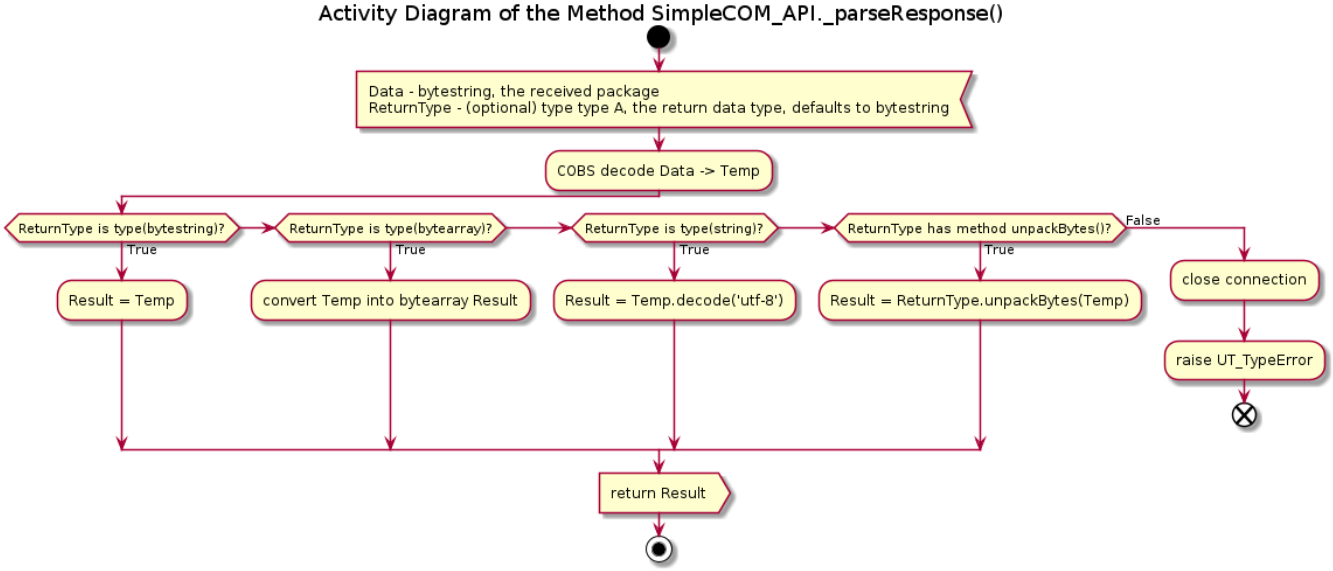


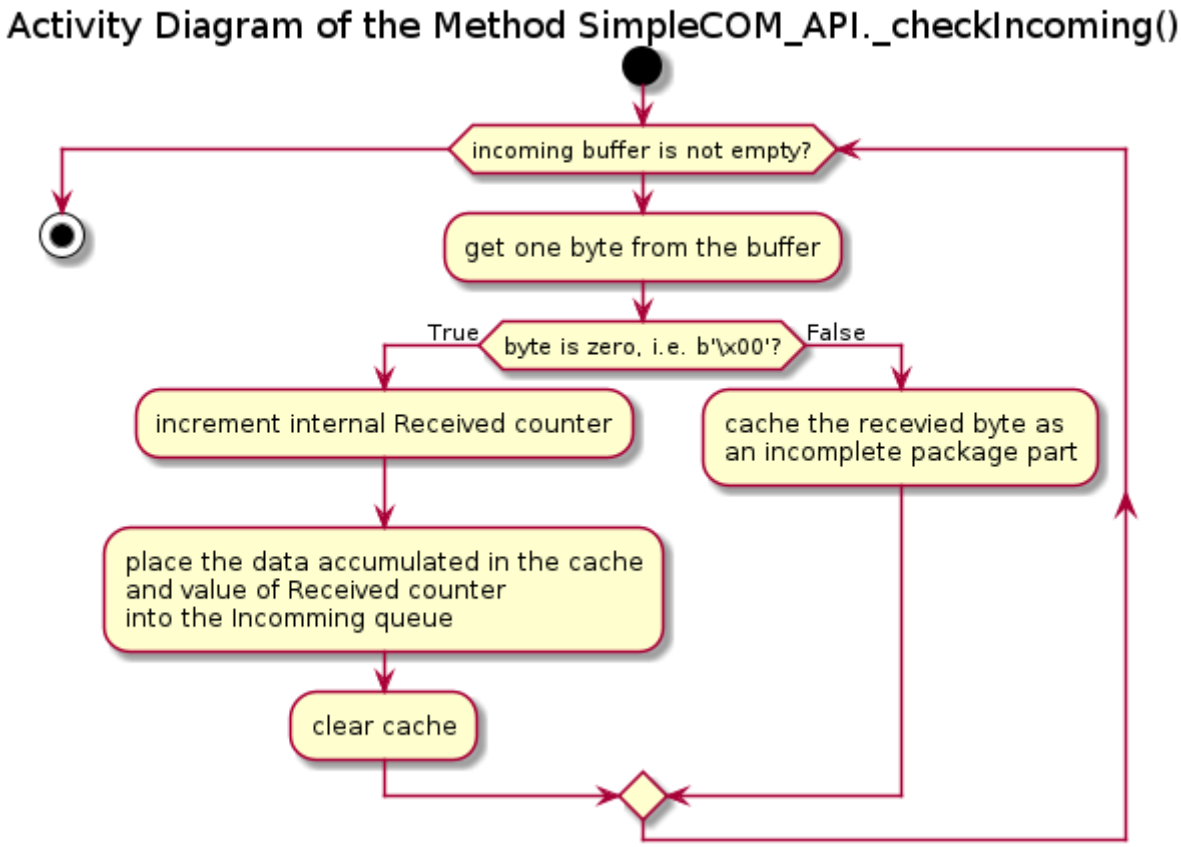
These three public methods implement the process flow logic independent on the input / output data type and the actual structure of the sent and received packages. The sub-classes of the **SimpleCOM_API** are not advised to modify these methods. The tasks of constuction of the packages to be send, retrieval of the recevied packages and data extraction from the received packages are delegated to the 'private' methods *_parseSending*(), *_checkIncoming*() and *_parseResponse*() respectively. The sub-class can re-define these 'private' methods to implement support for the different input / output data types or a different structure of the packages. The activity diagrams of these methods are shown below.

Activity Diagram of the Method SimpleCOM_API._parseSending()



Activity Diagram of the Method SimpleCOM_API._parseResponse()





This design is chosen specifically with the strictly bi-directional, synchronous and uni-directional, asynchronous modes in mind. In the mixed mode the unclaimed responses to the packages sent in the asynchornous mode may be lost due to the finite size of the incoming buffer (**serial.Serial** class from *pySerial* library); and this will break the equality of the indexes based logic of the synchronous sending and receiving method. Thus, the client software must claim the responses frequently in the mixed communication mode, even if they are not processed and simply discarded.

The alternative solution is to sub-class **SimpleCOM_API** and place the port listening and packages queueing functionality of the method *_checkIncoming*() into a function, which will be executed in a separate thread and use the packages queue as the shared object with that class' instance. The method *_checkIncoming*() itself should be re-defined as a stub (doing nothing). Be aware that the threads switching overhead will effectively slow down the data transfer, especially in the cases of high baudrate and small sizes of the sent and received packages.

API

Functions

list_ports()

Signature:

None -> list(tuple(str, int, int))

Returns:

list(tuple(str, int, int)); list of 3-element tuples, where the first element is the port name / path, the second - the vendor ID, and the last - the product ID

Description:

Looks up the connected via USB device, to which the virtual serial port communication can be established.

Classes

SimpleCOM_API

Descritpion

Wrapper class for the **serial.Serial** class - serial port connection API from the library *PySerial*. Implements simple API for asynchronous and synchronous data exchange using zero terminated bytestrings. Keeps track of the number of the sent and received packages. As long as the connected device sends a response to each received package, the both asynchronous and synchronous modes can be mixed.

Requires the path to the port during instantiation. Other connection settings except for the port path, read and write timeouts can be passed as the keyword arguments. The connection settings cannot be changed afterwards, but the port can be closed and re-opened multiple times upon request.

Properties:

- *IsOpen*: (read-only) bool
- *Settings*: (read-only) dict(str -> type A)

Instantiation:

__init__(strPort, **kwargs)

Signature:

str/, **kwargs/ -> None

Args:

- *strPort*: **str**; path to the port to be opened
- *kwargs*: (keyword) type A; any number of the keyword arguments acceptable by the **serial.Serial** class initializator

Raises:

- **UT_SerialException**: the connection cannot be opened, e.g. a device cannot be found or configured
- **UT_TypeError**: port path is not a string, OR any of the keyword arguments is of the improper type
- **UT_ValueError**: any of the keyword arguments is of a proper type, but of an unacceptable value

Description:

Initializer. Additional connection settings, like baudrate, etc. can be passed as keyword arguments, however the values of the keyword arguments port, timeout and write_timeout are replaced by the value of the positional argument, 0 and 0 respectively even if they are present among the keyword arguments. The connection is opened automatically.

If sub-class overrides this method, it must call this 'super' version.

Methods:

open():

Signature:

None -> None

Raises:

UT_SerialException: the connection cannot be opened, e.g. a device cannot be found or configured

Description:

Attempts to open a connection using the stored settings if it is not open at the moment.

The sub-classes should not re-define this particular method.

close():

Signature:

None -> None

Description:

Closes the connection if it is open. Doesn't raise an exception on closing the already closed connection. The cached data is cleared.

The sub-classes should not re-define this particular method

send(Data, **kwargs):

Signature:

type A/, **kwargs/ -> int > 0

Args:

- *Data*: type A; data to be processed and send
- *kwargs*: (keyword) type B; any additional arguments

Returns:

int > 0; the sent package index

Raises:

- **UT_TypeError**: the passed data is of the unsupported type
- **UT_SerialException**: the connection cannot be opened, e.g. a device cannot be found or configured, OR it has been disconnected in the process

Description:

Converts the passed data into a COBS encode bytesting, adds b'\x00' terminator and sends it into the port. The currently supported data types are: Unicode strings, bytestring, byte-arrays and instances of class providing own bytestring packing method *packBytes*().

The method is non-blocking. It exists immediately and returns the sent package index. There is no guarantee that the sending is already finished or even succeeded at this point.

The sub-classes should not re-define this particular method.

getResponse(ReturnType = bytes, **kwargs):

Signature:

/type type A, **kwargs/ -> None OR tuple(type A, int > 0)

Args:

- *ReturnType*: (optional) type type A; the data type, into which the the response should be converted; defaults to bytes
- *kwargs*: (keyword) type B; any additional arguments

Returns:

- **None**: there is no complete package waiting at the moment
- **tuple**(type A, int > 0); the 2-element tuple consisting of the received data converted into the required data type and the received data package index

Raises:

- **UT_TypeError**: the **ReturnType** is unsupported data type
- **UT_SerialException**: the connection cannot be opened, e.g. a device cannot be found or configured, OR it has been disconnected in the process

Description:

Checks the received and unclaimed responces and returns the earliest received response. The bytestring is converted into the requested data type / class instance. The method is not blocking. The currently supported data types are: Unicode strings, bytestring, byte-arrays and classes providing own bytestring unpacking class method *unpackBytes*().

The sub-classes should not re-define this particular method.

sendSync(Data, ReturnType = bytes, Timeout = 0, **kwargs):

Signature:

type A/, type type B, int > = 0 OR float >= 0, **kwargs/ -> tuple(type B, int > 0)

Args:

- *Data*: type A; data to be processed and send
- *ReturnType*: (optional) type type B; the data type, into which the the response should be converted; defaults to bytes

- *Timeout*: (optional) **int** >= 0 OR **float** >= 0; the timeout period; defaults to 0, i.e. blocking call
- *kwargs*: (keyword) type C; any additional arguments

Returns:

tuple(type A, int > 0); the 2-element tuple consisting of the received data converted into the required data type and the received data package index

Raises:

- **UT_TypeError**: the passed data is of the unsupported type
- **UT_SerialException**: the connection cannot be opened, e.g. a device cannot be found or configured, OR it has been disconnected in the process

Description:

Converts the passed data into a COBS encode bytesting, adds b'\x00' terminator and sends it into the port. When it waits until the reply to this sending is received. Unclaimed responses to the previous sendings are discarded in the process. By default (zero timeout) the call is blocking. If a positive timeout is specified, an exception is raised if the response is not received during this time interval. The received bytesting is converted into an instance of the requested data type / class.

The currently supported input data types are: Unicode strings, bytestring, byte-arrays and instances of class providing own bytestring packing method *packBytes*().

The currently supported return data types are: Unicode strings, bytestring, byte-arrays and classes providing own bytestring unpacking class method *unpackBytes*().

The sub-classes should not re-define this particular method.