The QConnection Base Library

Nguyen Huynh Tri Cuong (RBVH/ECM1)

Mar 10, 2022

CONTENTS:

QConnectBaseLibrary is a connection testing library for Robot Framework. Library will be supported to downloaded from PyPI soon. It provides a mechanism to handle trace log continously receiving from a connection (such as Raw TCP, SSH, Serial, etc.) besides sending data back to the other side. It's especially efficient for monitoring the overflood response trace log from an asynchronous trace systems. It is supporting Python 3.7+ and RobotFramework 3.2+.

CHAPTER

ONE

TABLE OF CONTENTS

CHAPTER

TWO

GETTING STARTED

We have a plan to publish all the sourcecode as OSS in the near future so that you can downloaded from PyPI. For the current period, you can checkout

QConnectBaseLibrary

After checking out the source completely, you can install by running below command inside **robotframework-qconnect-base** directory.

```
python setup.py install
```

CHAPTER

THREE

USAGE

QConnectBase Library support following keywords for testing connection in RobotFramework.

3.1 connect

Use for establishing a connection.

Syntax:

connect `[conn_name] [conn_type] [conn_mode] [conn_conf]` *(All parameters are required to be in order)*

or

connect `conn_name=[conn_name] conn_type=[conn_type] conn_mode=[conn_mode] conn_conf=[conn_conf]` *(All parameters are assigned by name)*

Arguments:

conn_name: Name of the connection.

conn_type: Type of the connection. QConnectBaseLibrary has supported below connection types:

- **TCPIPClient**: Create a Raw TCPIP connection to TCP Server.
- **SSHClient**: Create a client connection to a SSH server.
- **SerialClient**: Create a client connection via Serial Port.

conn_mode: (unused) Mode of a connection type.

conn_conf: Configurations for making a connection. Depend on **conn_type** (Type of Connection), there is a suitable configuration in JSON format as below.

- **TCPIPClient**

```
{
    "address": [server host], # Optional. Default value is
↪"localhost".
    "port": [server port]     # Optional. Default value is 1234.
    "logfile": [Log file path. Possible values: 'nonlog',
↪'console', <user define path>]
 }
```

- **SSHClient**

```
{
    "address" : [server host],  # Optional. Default value is
↪"localhost".
    "port" : [server host],     # Optional. Default value is 22.
    "username" : [username],    # Optional. Default value is
↪"root".
    "password" : [password],    # Optional. Default value is "".
    "authentication" : "password" | "keyfile" | "passwordkeyfile
↪",  # Optional. Default value is "".
    "key_filename" : [filename or list of filenames], #␣
↪Optional. Default value is null.
    "logfile": [Log file path. Possible values: 'nonlog',
↪'console', <user define path>]
 }
```

- **SerialClient**

```
{
    "port" : [comport or null],
    "baudrate" : [Baud rate such as 9600 or 115200 etc.],
    "bytesize" : [Number of data bits. Possible values: 5, 6, 7,
↪ 8],
    "stopbits" : [Number of stop bits. Possible values: 1, 1.5,␣
↪2],
    "parity" : [Enable parity checking. Possible values: 'N', 'E
↪', 'O', 'M', 'S'],
    "rtscts" : [Enable hardware (RTS/CTS) flow control.],
    "xonxoff" : [Enable software flow control.],
    "logfile": [Log file path. Possible values: 'nonlog',
↪'console', <user define path>]
 }
```

3.2 disconnect

Use for disconnect a connection by name.

Syntax:

disconnect conn_name

Arguments:

conn_name: Name of the connection.

3.3 send command

Use for sending a command to the other side of connection.

Syntax:

send command `[conn_name] [command]` *(All parameters are required to be in order)*

or

send command `conn_name=[conn_name] command=[command]` *(All parameters are assigned by name) ##### Arguments*:

- **conn_name**: Name of the connection.
- **command**: Command to be sent.

3.4 verify

Use for verifying a response from the connection if it matched a pattern.

Syntax:

verify `[conn_name] [search_pattern] [timeout] [fetch_block] [eob_pattern] [filter_pattern] [send_cmd]`*(All parameters are required to be in order)*

or

verify `conn_name=[conn_name] search_pattern=[search_pattern] timeout=[timeout] fetch_block=[fetch_block] eob_pattern=[eob_pattern] filter_pattern=[filter_pattern] send_cmd=[send_cmd]` *(All parameters are assigned by name)*

Arguments:

conn_name: Name of the connection.

search_pattern: Regular expression for matching with the response.

timeout: Timeout for waiting response matching pattern.

fetch_block: If this value is true, every response line will be put into a block untill a line match **eob_pattern** pattern.

eob_pattern: Regular expression for matching the endline when using **fetch_block**.

filter_pattern: Regular expression for filtering every line of block when using **fetch_block**.

send_cmd: Command to be sent to the other side of connection and waiting for response.

Return value:

A corresponding match object if it is found.

E.g.

```
${result} = verify  conn_name=SSH_Connection
                    search_pattern=(?<=\s).*([0-9]..).*(command).$
                    send_cmd=*echo This is the 1st test command.*
```

- ${result}[0] will be **"This is the 1st test command."** which is the matched string.
- ${result}[1] will be **"1st"** which is the first captured string.

- ${result}[2] will be **“command”** which is the second captured string.

CHAPTER

FOUR

EXAMPLE

```
*** Settings ***
Documentation    Suite description
Library     QConnectionLibrary.ConnectionManager

*** Test Cases ***
Test SSH Connection
    # Create config for connection.
    ${config_string}=    catenate
    ...  {
    ...   "address": "127.0.0.1",
    ...   "port": 8022,
    ...   "username": "root",
    ...   "password": "",
    ...   "authentication": "password",
    ...   "key_filename": null
    ...  }
    log to console       \nConnecting with configurations:\n${config_string}
    ${config}=             evaluate        json.loads('''${config_string}''')    json

    # Connect to the target with above configurations.
    connect             conn_name=test_ssh
    ...                 conn_type=SSHClient
    ...                 conn_conf=${config}

    # Send command 'cd ..' and 'ls' then wait for the response '.*' pattern.
    send command                conn_name=test_ssh
    ...                         command=cd ..

    ${res}=     verify                  conn_name=test_ssh
    ...                                 search_pattern=.*
    ...                                 send_cmd=ls
    log to console     ${res}

    # Disconnect
    disconnect  test_ssh
```

CHAPTER

FIVE

CONTRIBUTION GUIDELINES

QConnectBaseLibrary is designed for ease of making an extension library. By that way you can take advantage of the QConnectBaseLibrary's infrastructure for handling your own connection protocal. For creating an extension library for QConnectBaseLibrary, please following below steps.

1. Create a library package which have the prefix name is **robotframework-qconnect-***[your specific name]*.
2. Your hadling connection class should be derived from **QConnectionLibrary.connection_base.ConnectionBase** class.
3. In your *Connection Class*, override below attributes and methods:
   - **_CONNECTION_TYPE**: name of your connection type. It will be the input of the conn_type argument when using **connect** keyword. Depend on the type name, the library will detemine the correct connection handling class.
   - **__init__(self, _mode, config)**: in this constructor method, you should:
   - Prepare resource for your connection.
   - Initialize receiver thread by calling **self._init_thread_receiver(cls._socket_instance, mode="")** method.
   - Configure and initialize the lowlevel receiver thread (if it's necessary) as below

```
self._llrecv_thrd_obj = None
 self._llrecv_thrd_term = threading.Event()
 self._init_thrd_llrecv(cls._socket_instance)
```

   - Incase you use the lowlevel receiver thread. You should implement the **thrd_llrecv_from_connection_interface()** method. This method is a mediate layer which will receive the data from connection at the very beginning, do some process then put them in a queue for the **receiver thread** above getting later.
   - Create the queue for this connection (use Queue.Queue).
   - **connect()**: implement the way you use to make your own connection protocol.
   - **_read()**: implement the way to receive data from connection.
   - **_write()**: implement the way to send data via connection.
   - **disconnect()**: implement the way you use to disconnect your own connection protocol.
   - **quit()**: implement the way you use to quit connection and clean resource.

CHAPTER

SIX

CONFIGURE GIT AND CORRECT EOL HANDLING

Here you can find the references for Dealing with line endings.

Every time you press return on your keyboard you're actually inserting an invisible character called a line ending. Historically, different operating systems have handled line endings differently. When you view changes in a file, Git handles line endings in its own way. Since you're collaborating on projects with Git and GitHub, Git might produce unexpected results if, for example, you're working on a Windows machine, and your collaborator has made a change in OS X.

To avoid problems in your diffs, you can configure Git to properly handle line endings. If you are storing the .gitattributes file directly inside of your repository, than you can asure that all EOL are manged by git correctly as defined.

CHAPTER

SEVEN

SOURCECODE DOCUMENTATION

For investigating sourcecode, please refer to QConnectBase library documentation

A detailed documentation of the QConnectBase package can also be found here: QConnectBase.pdf

CHAPTER

EIGHT

FEEDBACK

If you have any problem when using the library or think there is a better solution for any part of the library, I'd love to know it, as this will all help me to improve the library. Please don't hesitate to contact me.

Do share your valuable opinion, I appreciate your honest feedback!

CHAPTER

NINE

ABOUT

9.1 Maintainers

Nguyen Huynh Tri Cuong

9.2 Contributors

Nguyen Huynh Tri Cuong

Thomas Pollerspöck

CHAPTER

TEN

LICENSE

CHAPTER

ELEVEN

QCONNECT BASE LIBRARY'S API!

11.1 QConnectBase package

11.1.1 Module contents

class QConnectBase.connection_manager.**ConnectParam**(***dictionary*)
Bases: *QConnectBase.connection_manager.InputParam*

Class for storing parameters for connect action.

conn_conf = {}

conn_mode = ''

conn_name = 'default_conn'

conn_type = 'TCPIP'

exclude_list = ['conn_conf']

id = 0

class QConnectBase.connection_manager.**ConnectionManager**(**args*, ***kwargs*)
Bases: QConnectBase.utils.Singleton

Class to manage all connections.

LIBRARY_EXTENSION_PREFIX = 'robotframework_qconnect'

LIBRARY_EXTENSION_PREFIX2 = 'QConnection'

ROBOT_AUTO_KEYWORDS = False

ROBOT_LIBRARY_SCOPE = 'GLOBAL'

add_connection(*name*, *conn*)
Add a connection to managed dictionary.

Args: name: connection's name.

conn: connection object.

Returns: None.

connect(**args*, ***kwargs*)
Keyword for making a connection.

Args: args: Non-Keyword Arguments.

kwargs: Keyword Arguments.

Returns: None.

connect_named_args(***kwargs*)
Making a connection with name arguments.

Args: kwargs: Dictionary of arguments.

Returns: None.

connect_unnamed_args(*connection_name*, *connection_type*, *mode*, *config*)
Making a connection.

Args: connection_name: Name of connection.

connection_type: Type of connection.

mode: Connection mode.

config: Configuration for connection.

Returns: None.

disconnect(*connection_name*)
Keyword for disconnecting a connection by name.

Args: connection_name: Name of connection.

Returns: None.

get_connection_by_name(*connection_name*)
Get an exist connection by name.

Args: connection_name: connection's name.

Returns: Connection object.

quit()
Quit connection manager.

Returns: None.

remove_connection(*connection_name*)
Remove a connection by name.

Args: connection_name: connection name.

Returns: None.

send_command(**args*, ***kwargs*)
Keyword for sending command to a connection.

Args: args: Non-Keyword Arguments.

kwargs: Keyword Arguments.

Returns: None.

send_command_named_args(***args*)
Send command to a connection with name arguments.

Args: args: Dictionary of arguments.

Returns: None.

send_command_unnamed_args(*connection_name*, *command*)
Send command to a connection.

Args: connection_name: connection's name.

command: command.

Returns: None.

verify(**args*, ***kwargs*)
Keyword uses to verify a pattern from connection response after sending a command.

Args: args: Non-Keyword Arguments.

kwargs: Keyword Arguments.

Returns: match_res: matched string.

verify_named_args(***kwargs*)
Verify a pattern from connection response after sending a command with named arguments.

Args: kwargs: Dictionary of arguments.

Returns: match_res: matched string.

verify_unnamed_args(*connection_name*, *search_obj*, *timeout=0*, *fetch_block=False*, *eob_pattern='.*'*, *filter_pattern='.*'*, **fct_args*)
Verify a pattern from connection response after sending a command.

Args: connection_name: connection's name.

search_obj: search pattern.

timeout: timeout for waiting result.

fetch_block: use fetch block feature.

end_of_block_pattern: pattern for detecting the end of block.

filter_pattern: line filter pattern.

fct_args: command to be sent.

Returns: match_res: matched string.

class QConnectBase.connection_manager.**InputParam**(***dictionary*)
Bases: QConnectBase.utils.DictToClass

classmethod get_attr_list()

class QConnectBase.connection_manager.**SendCommandParam**(***dictionary*)
Bases: *QConnectBase.connection_manager.InputParam*

Class for storing parameters for send command action.

command = ''

conn_name = 'default_conn'

class QConnectBase.connection_manager.**TestOption**
Bases: object

DLT_OPT = 0

SERIAL_OPT = 2

SSH_OPT = 1

class QConnectBase.connection_manager.**VerifyParam**(***dictionary*)
Bases: *QConnectBase.connection_manager.InputParam*

Class for storing parameters for verify action.

conn_name = 'default_conn'

eob_pattern = '.*'

fetch_block = False

filter_pattern = '.*'

search_pattern = None

send_cmd = ''

timeout = 5

exception QConnectBase.connection_base.**BrokenConnError**
Bases: Exception

class QConnectBase.connection_base.**ConnectionBase**(**args*, ***kwargs*)
Bases: object

Base class for all connection classes.

MAX_LEN_BACKTRACE = 500

RECV_MSGS_POLLING_INTERVAL = 0.005

classmethod **activate_trace_queue**(*search_obj*, *trace_queue*, *use_fetch_block=False*, *end_of_block_pattern='.*'*, *line_filter_pattern=None*)
Activates a trace message filter specified as a regular expression. All matching trace messages are put in the specified queue object.

Args: search_obj : Regular expression all received trace messages are compare to. Can be passed either as a string or a regular expression object. Refer to Python documentation for module 're'.#

trace_queue : A queue object all trace message which matches the regular expression are put in. The using application must assure, that the queue is emptied or deleted.

use_fetch_block : Determine if 'fetch block' feature is used

end_of_block_pattern : The end of block pattern

line_filter_pattern : Regular expression object to filter message line by line.

Returns: <int> : Handle to deactivate the message filter.

check_timeout(*msg*)
>> This method will be override in derived class << Check if responded message come in cls._RESPOND_TIMEOUT or we will raise a timeout event.

Args: msg: Responded message for checking.

Returns: None.

config = None

abstract **connect**(*device*, *files=None*, *test_connection=False*)
>> This method MUST be overridden in derived class << Abstract method for quiting the connection.

Args: device: Determine if it's necessary to disconnect all connections.

files: Determine if it's necessary to disconnect all connections.

test_connection: Determine if it's necessary to disconnect all connections.

Returns: None.

classmethod create_and_activate_trace_queue(*search_element*, *use_fetch_block=False*, *end_of_block_pattern='.*'*, *regex_line_filter_pattern=None*)

Create Queue and assign it to _trace_queue object and activate the queue with the search element.

Args: search_element : Regular expression all received trace messages are compare to. Can be passed either as a string or a regular expression object. Refer to Python documentation for module 're'.#

use_fetch_block : Determine if 'fetch block' feature is used.

end_of_block_pattern : The end of block pattern.

regex_line_filter_pattern : Regular expression object to filter message line by line.

Returns: trq_handle, trace_queue: the handle and search object

classmethod deactivate_and_delete_trace_queue(*trq_handle*, *trace_queue*)

Deactivate trace queue and delete.

Args: trq_handle: Trace queue handle.

trace_queue: Trace queue object.

Returns: None.

classmethod deactivate_trace_queue(*handle*)

Deactivates a trace message filter previously activated by ActivateTraceQ() method.

Args: handle : Integer object returned by ActivateTraceQ() method.

Returns: False : No trace message filter active with the specified handle (i.e. handle is not in use).

True : Trace message filter successfully deleted.

abstract disconnect(*device*)

>> This method MUST be overridden in derived class << Abstract method for disconnecting connection.

Args: device: Device name.

Returns: None.

error_instruction()

Get the error instruction.

Returns: Error instruction string.

classmethod is_precondition_pass()

Check for precondition.

Returns: True if passing the precondition.

False if failing the precondition.

classmethod is_supported_platform()

Check if current platform is supported.

Returns: True if platform is supported.

False if platform is not supported.

post_msg_check(*msg*)

>> This method will be override in derived class << Post-checking message when receiving it from connection.

Args: msg: received message to be checked.

Returns: None.

pre_msg_check(*msg*)
>> This method will be override in derived class << Pre-checking message when receiving it from connection.

Args: msg: received message to be checked.

Returns: None.

abstract quit(*is_disconnect_all=True*)
>> This method MUST be overridden in derived class << Abstract method for quiting the connection.

Args: is_disconnect_all: Determine if it's necessary to disconnect all connections.

Returns: None.

read_obj()
Wrapper method to get the response from connection.

Returns: Responded message.

send_obj(*obj*, *cr=True*)
Wrapper method to send message to a tcp connection.

Args: obj: Data to be sent. cr: Determine if it's necessary to add newline character at the end of command.

Returns: None

supported_devices = []

wait_4_trace(*search_obj*, *timeout=0*, *use_fetch_block=False*, *end_of_block_pattern='.*'*, *filter_pattern='.*'*, **fct_args*)
Suspend the control flow until a Trace message is received which matches to a specified regular expression.

Args: search_obj : Regular expression all received trace messages are compare to. Can be passed either as a string or a regular expression object. Refer to Python documentation for module 're'.

use_fetch_block : Determine if 'fetch block' feature is used.

end_of_block_pattern : The end of block pattern.

filter_pattern : Regular expression object to filter message line by line.

timeout : Optional timeout parameter specified as a floating point number in the unit 'seconds'.

fct_args: Optional list of function arguments passed to be sent.

Returns: None : If no trace message matched to the specified regular expression and a timeout occurred.

<match> : If a trace message has matched to the specified regular expression, a match object is returned as the result. The complete trace message can be accessed by the 'string' attribute of the match object. For access to groups within the regular expression, use the group() method. For more information, refer to Python documentation for module 're'.

wait_4_trace_continuously(*trace_queue*, *timeout=0*, **fct_args*)
Getting trace log continuously without creating a new trace queue.

Args: trace_queue: Queue to store the traces.

timeout: Timeout for waiting a matched log.

fct_args: Arguments to be sent to connection.

Returns: None : If no trace message matched to the specified regular expression and a timeout occurred.

match object : If a trace message has matched to the specified regular expression, a match object is returned as the result. The complete trace message can be accessed by the 'string' attribute of the

match object. For access to groups within the regular expression, use the group() method. For more information, refer to Python documentation for module 're'.

class QConnectBase.qlogger.**ColorFormatter**(*fmt=None*, *datefmt=None*, *style='%'*, *validate=True*)
Bases: logging.Formatter

Custom formatter class for setting log color.

FORMATS = {10: '\x1b[38;21m%(asctime)s [%(threadName)-12.12s] [%(levelname)-5.5s] %(message)s\x1b[0m', 20: '\x1b[38;21m%(asctime)s [%(threadName)-12.12s] [%(levelname)-5.5s] %(message)s\x1b[0m', 30: '\x1b[33;21m%(asctime)s [%(threadName)-12.12s] [%(levelname)-5.5s] %(message)s\x1b[0m', 40: '\x1b[31;21m%(asctime)s [%(threadName)-12.12s] [%(levelname)-5.5s] %(message)s\x1b[0m', 50: '\x1b[31;1m%(asctime)s [%(threadName)-12.12s] [%(levelname)-5.5s] %(message)s\x1b[0m'}

bold_red = '\x1b[31;1m'

format(*record*)
Set the color format for the log.

Args: record: log record.

Returns: Log with color formatter.

grey = '\x1b[38;21m'

red = '\x1b[31;21m'

reset = '\x1b[0m'

yellow = '\x1b[33;21m'

class QConnectBase.qlogger.**QConsoleHandler**(*_config*, *_logger_name*, *_formatter*)
Bases: logging.StreamHandler

Handler class for console log.

static get_config_supported(*config*)
Check if the connection config is supported by this handler.

Args: config: connection configurations.

Returns: True if the config is supported.

False if the config is not supported.

class QConnectBase.qlogger.**QDefaultFileHandler**(*_config*, *logger_name*, *formatter*)
Bases: logging.FileHandler

Handler class for default log file path.

static get_config_supported(*config*)
Check if the connection config is supported by this handler.

Args: config: connection configurations.

Returns: True if the config is supported.

False if the config is not supported.

static get_log_path(*logger_name*)
Get the log file path for this handler.

Args: logger_name: name of the logger.

Returns: Log file path.

class QConnectBase.qlogger.**QFileHandler**(*config*, *_logger_name*, *formatter*)
Bases: logging.FileHandler

Handler class for user defined file in config.

static get_config_supported(*config*)
Check if the connection config is supported by this handler.

Args: config: connection configurations.

Returns: True if the config is supported.

False if the config is not supported.

static get_log_path(*config*)
Get the log file path for this handler.

Args: config: connection configurations.

Returns: Log file path.

class QConnectBase.qlogger.**QLogger**(**args*, ***kwargs*)
Bases: QConnectBase.utils.Singleton

Logger class for QConnect Libraries.

NAME_2_LEVEL_DICT = {'NONE': 51, 'TRACE': 0}

get_logger(*logger_name*)
Get the logger object.

Args: logger_name: Name of the logger.

Returns: Logger object.

set_handler(*config*)
Set handler for logger.

Args: config: connection configurations.

Returns: None if no handler is set. Handler object.

class QConnectBase.tcp.tcp_base.**TCPBase**(**args*, ***kwargs*)
Bases: *QConnectBase.connection_base.ConnectionBase*, object

Base class for a tcp connection.

RECV_MSGS_POLLING_INTERVAL = 0.005

property address
Get connection address.

Returns: Connection address.

close()
Close connection.

Returns: None.

property conn_timeout
Get connection timeout.

Returns: Connection timeout.

connect()
>> Should be override in derived class. Establish the connection.

Returns: None.

disconnect(*device*)
>> Should be override in derived class. Disconnect the connection.

Returns: None.

property is_connected
Get connected state.

Returns: True if connection is connected. False if connection is not connected.

property port
Get connection port.

Returns: Connection port.

quit(*is_disconnect_all=True*)
Quit connection.

Args: is_disconnect_all: Determine if it's necessary for disconnect all connection.

Returns: None.

property socket_instance
Get method of socket_instance property.

Returns: Value of _socket_instance.

property timeout
Get connection timeout value.

Returns: Value of connection timeout.

class QConnectBase.tcp.tcp_base.**TCPBaseClient**
Bases: object

Base class for TCP client.

connect()

class QConnectBase.tcp.tcp_base.**TCPBaseServer**
Bases: object

Base class for TCP server.

accept_connection()
Wrapper method for handling accept action of TCP Server.

Returns: None.

class QConnectBase.tcp.tcp_base.**TCPConfig**(***dictionary*)
Bases: QConnectBase.utils.DictToClass

Class to store configurations for TCP connection.

address = 'localhost'

port = 12345

class QConnectBase.tcp.ssh.ssh_client.**AuthenticationType**
Bases: object

KEYFILE = 'keyfile'

PASSWORD = 'password'

PASSWORDKEYFILE = 'passwordkeyfile'

class QConnectBase.tcp.ssh.ssh_client.**SSHClient**(**args*, ***kwargs*)
Bases: *QConnectBase.tcp.tcp_base.TCPBase*, *QConnectBase.tcp.tcp_base.TCPBaseClient*

SSH client connection class.

close()
Close SSH connection.

Returns:

connect()
Implementation for creating a SSH connection.

Returns: None.

quit()
Quit and stop receiver thread.

Returns: None.

class QConnectBase.tcp.ssh.ssh_client.**SSHConfig**(***dictionary*)
Bases: *QConnectBase.tcp.tcp_base.TCPConfig*

Class to store the configuration for SSH connection.

address = 'localhost'

authentication = 'password'

key_filename = None

password = ''

port = 22

username = 'root'

class QConnectBase.tcp.raw.raw_tcp.**RawTCPBase**(**args*, ***kwargs*)
Bases: *QConnectBase.tcp.tcp_base.TCPBase*

Base class for a raw tcp connection.

class QConnectBase.tcp.raw.raw_tcp.**RawTCPClient**(**args*, ***kwargs*)
Bases: *QConnectBase.tcp.raw.raw_tcp.RawTCPBase*, *QConnectBase.tcp.tcp_base.TCPBaseClient*

Class for a raw tcp connection client.

class QConnectBase.tcp.raw.raw_tcp.**RawTCPServer**(**args*, ***kwargs*)
Bases: *QConnectBase.tcp.raw.raw_tcp.RawTCPBase*, *QConnectBase.tcp.tcp_base.TCPBaseServer*

Class for a raw tcp connection server.

class QConnectBase.serialclient.serial_base.**SerialClient**(**args*, ***kwargs*)
Bases: *QConnectBase.serialclient.serial_base.SerialSocket*

Serial client class.

connect()
Connect to the Serial port.

Returns: None.

class QConnectBase.serialclient.serial_base.**SerialConfig**(***dictionary*)
Bases: QConnectBase.utils.DictToClass

Class to store the configuration for Serial connection.

baudrate = 115200

bytesize = 8

parity = 'N'

port = 'COM1'

rtscts = False

stopbits = 1

xonxoff = False

class QConnectBase.serialclient.serial_base.**SerialSocket**(**args*, ***kwargs*)

Bases: *QConnectBase.connection_base.ConnectionBase*

Class for handling serial connection.

connect()

Connect to serial port.

Returns: None.

disconnect(*_device*)

Disconnect serial port.

Args: _device: unused.

Returns: None.

quit()

Quit serial connection.

Returns: None

PYTHON MODULE INDEX

q