

QConnectBase

v. 1.2.1

Nguyen Huynh Tri Cuong

06.11.2025

Contents

1	Introduction	1
2	Description	2
2.1	User interface	2
2.1.1	Keyword overview	2
2.1.2	Library import	2
2.1.3	Usecases	3
2.2	Verify examples	6
2.2.1	Verify a single message (1)	6
2.2.2	Verify a single message (2)	6
2.2.3	Verify a block of messages (1)	7
2.2.4	Verify a block of messages (2)	8
2.2.5	Verify a block of messages (3)	9
2.2.6	Verify a block of messages (4)	10
2.3	=== REWORKED ===	11
2.4	Connection types	12
2.5	Usage	12
2.5.1	connect	12
2.5.2	disconnect	13
2.5.3	send command	13
2.5.4	transfer file	14
2.5.5	transfer item	14
2.5.6	verify	14
2.6	Example	18
2.7	Auxiliary Utilities	20
2.7.1	Introduction to RMQSignal	20
2.7.2	Prerequisites	20
2.7.3	Library Keywords and Usage	21
2.7.4	Example of Inter-Process Communication	23
2.8	Contribution Guidelines	23
2.9	Configure Git and correct EOL handling	24
2.10	Sourcecode Documentation	24
2.11	Feedback	24
2.12	About	24
2.12.1	Maintainers	24
2.12.2	Contributors	24
2.13	License	24

3	<code>__init__.py</code>	26
3.1	Class: <code>ConnectionManager</code>	26
4	<code>connection_base.py</code>	27
4.1	Class: <code>BrokenConnError</code>	27
4.2	Class: <code>ConnectionBase</code>	27
4.2.1	Method: <code>is_supported_platform</code>	27
4.2.2	Method: <code>is_precondition_pass</code>	27
4.2.3	Method: <code>get_connection_type</code>	27
4.2.4	Method: <code>error_instruction</code>	28
4.2.5	Method: <code>quit</code>	28
4.2.6	Method: <code>connect</code>	28
4.2.7	Method: <code>disconnect</code>	28
4.2.8	Method: <code>send_obj</code>	29
4.2.9	Method: <code>read_obj</code>	29
4.2.10	Method: <code>wait_4_trace</code>	29
4.2.11	Method: <code>wait_4_trace_continuously</code>	30
4.2.12	Method: <code>create_and_activate_trace_queue</code>	30
4.2.13	Method: <code>deactivate_and_delete_trace_queue</code>	31
4.2.14	Method: <code>activate_trace_queue</code>	31
4.2.15	Method: <code>deactivate_trace_queue</code>	32
4.2.16	Method: <code>check_timeout</code>	32
4.2.17	Method: <code>pre_msg_check</code>	32
4.2.18	Method: <code>post_msg_check</code>	32
5	<code>connection_manager.py</code>	33
5.1	Class: <code>InputParam</code>	33
5.1.1	Method: <code>get_attr_list</code>	33
5.2	Class: <code>ConnectParam</code>	33
5.3	Class: <code>SendCommandParam</code>	33
5.4	Class: <code>VerifyParam</code>	33
5.5	Class: <code>ConnectionManager</code>	33
5.5.1	Method: <code>end_suite</code>	34
5.5.2	Method: <code>quit</code>	34
5.5.3	Method: <code>add_connection</code>	34
5.5.4	Method: <code>remove_connection</code>	34
5.5.5	Method: <code>get_connection_by_name</code>	34
5.5.6	Keyword: <code>disconnect</code>	35
5.5.7	Keyword: <code>connect</code>	35
5.5.8	Keyword: <code>send_command</code>	36
5.5.9	Keyword: <code>transfer_file</code>	36
5.5.10	Keyword: <code>transfer_item</code>	36
5.5.11	Keyword: <code>execute_script</code>	37
5.5.12	Keyword: <code>set_default_verify_timeout</code>	37
5.5.13	Keyword: <code>set_default_emergency_timeout</code>	38
5.5.14	Keyword: <code>verify</code>	38
5.6	Class: <code>TestOption</code>	39

6	constants.py	40
6.1	Class: SocketType	40
6.2	Class: String	40
7	rabbitmq_client.py	41
7.1	Class: RabbitmqClientConfig	41
7.2	Class: RabbitmqClient	41
7.2.1	Method: on_response	41
7.2.2	Method: connect	41
7.2.3	Method: close	41
7.2.4	Method: quit	41
7.3	Class: RMQSignal	41
7.3.1	Method: send_signal	42
7.3.2	Method: unset_signal_receiver_name	42
7.3.3	Method: set_signal_receiver_name	42
7.3.4	Method: consume_channel	42
7.3.5	Method: wait_for_signal	43
7.3.6	Method: wait_for_signals	43
8	qlogger.py	45
8.1	Class: ColorFormatter	45
8.1.1	Method: format	45
8.2	Class: QFileHandler	45
8.2.1	Method: get_log_path	45
8.2.2	Method: get_config_supported	46
8.3	Class: QDefaultFileHandler	46
8.3.1	Method: get_log_path	46
8.3.2	Method: get_config_supported	46
8.4	Class: QConsoleHandler	47
8.4.1	Method: get_config_supported	47
8.5	Class: QLogger	47
8.5.1	Method: get_logger	47
8.5.2	Method: set_handler	47
9	serial_base.py	49
9.1	Class: SerialConfig	49
9.2	Class: SerialSocket	49
9.2.1	Method: connect	49
9.2.2	Method: disconnect	49
9.2.3	Method: quit	49
9.3	Class: SerialClient	50
9.3.1	Method: connect	50
10	raw_tcp.py	51
10.1	Class: RawTCPBase	51
10.2	Class: RawTCPServer	51
10.3	Class: RawTCPClient	51

11 ssh_client.py	52
11.1 Class: AuthenticationType	52
11.2 Class: SSHConfig	52
11.3 Class: SSHClient	52
11.3.1 Method: connect	52
11.3.2 Method: transfer_file	52
11.3.3 Method: transfer_item	53
11.3.4 Method: close	53
11.3.5 Method: quit	53
12 tcp_base.py	54
12.1 Class: TCPConfig	54
12.1.1 Method: validate	54
12.2 Class: TCPBase	54
12.2.1 Method: close	54
12.2.2 Method: quit	54
12.2.3 Method: connect	54
12.2.4 Method: disconnect	55
12.3 Class: TCPBaseServer	55
12.3.1 Method: accept_connection	55
12.3.2 Method: connect	55
12.3.3 Method: disconnect	55
12.4 Class: TCPBaseClient	55
12.4.1 Method: connect	55
12.4.2 Method: disconnect	55
13 utils.py	56
13.1 Function: has_capturing_groups	56
13.2 Class: DictToClass	56
13.2.1 Method: validate	56
13.3 Class: Utils	56
13.3.1 Method: get_all_descendant_classes	56
13.3.2 Method: get_all_sub_classes	57
13.3.3 Method: is_valid_host	57
13.3.4 Method: execute_command	57
13.3.5 Method: kill_process	57
13.3.6 Method: caller_name	57
13.3.7 Method: load_library	57
13.3.8 Method: is_ascii_or_unicode	58
13.4 Class: Job	58
13.4.1 Method: stop	58
13.4.2 Method: run	58
13.5 Class: ResultType	58
13.6 Class: ResponseMessage	58
13.6.1 Method: get_json	58
13.6.2 Method: get_data	58
13.6.3 Method: create_from_string	58

14 Appendix	59
15 History	60

Chapter 1

Introduction

The **QConnectBase** package provides **Robot Framework** keywords for interacting with devices, servers, or services through various communication protocols such as TCP/IP, SSH, or Serial. It allows users to establish multiple connections, send commands, and easily verify responses.

Key features

- Support for multiple simultaneous connections.
- Built-in keywords for sending and receiving data.
- Protocol abstraction for TCP/IP, SSH, and Serial.
- Flexible connection management and session handling.
- Extensible architecture for custom protocols or behaviors.

Developers can extend the **QConnectBase** code to support additional connection types or custom behaviors, ensuring the library remains flexible and adaptable to diverse automation needs.

The sources of the **QConnectBase** are available in [GitHub](#).

Information about how to install **QConnectBase** can be found in the [README](#).

Chapter 2

Description

2.1 User interface

2.1.1 Keyword overview

QConnectBase contains the following keywords:

- `connect`
Establishes a connection to a specified host.
- `disconnect`
Terminates the connection to a host.
- `send_command`
Sends a command to the connected host.
- `verify`
Verifies the answer received from the connected host. Optionally sends a command (usually to trigger the answer).
- `transfer_item`
Sends an item to the connected host. Receives items from the connected host also.
- `transfer_file`
Deprecated: Use `transfer_item` instead!

2.1.2 Library import

The keywords can be accessed with the following library import:

```
Library      QConnectBase.ConnectionManager  WITH NAME  conn_manager
```

We recommend to shorten the `QConnectBase` library name by using the `WITH NAME` statement.

2.1.3 Usecases

QConnectBase supports various types of connections. This results in different `connect` parameters. The following examples refer to the connection type `TCPIPClient`.

The `connect` parameters for the connection type `TCPIPClient` are defined as follows:

```
*** Variables ***

${HOST}=    localhost
${PORT}=    ${1234}

&{TCPIPClientParams}    conn_type=TCPIPClient
...                    address=${HOST}
...                    port=${PORT}
...                    logfile=./tcp_ip_incoming.log
```

The logfile parameter is optional. It contains the path and the name of a file in which **QConnectBase** logs all incoming messages.

————— ★ —————

Connect to a host

Required are a name of the connection together with the connection parameters.

```
conn_manager.connect    conn_name=my_test_connection
...                    conn_conf=${TCPIPClientParams}
```

In case of errors, **QConnectBase** terminates the test execution. If you want to keep full control over the test flow, you'll need to use the Robot Framework BuiltIn keyword `run_keyword_and_ignore_error` (*the same for all other keywords also*).

```
${status}    ${result}=    run_keyword_and_ignore_error    conn_manager.connect    ↵
    ↵ conn_name=my_test_connection                                ...
    ↵ conn_conf=${TCPIPClientParams}
```

`status` can be one of `PASS`, `FAIL`. `result` is either `None` or contains an error message.

A full description of the keyword interface can be found [here](#).

————— ★ —————

Terminate a connection

```
conn_manager.disconnect    conn_name=my_test_connection
```

The connection must be established before, using `connect`.

A full description of the keyword interface can be found [here](#).

————— ★ —————

Send a command

Send a command to the connected host.

```
conn_manager.send_command    conn_name=my_test_connection    <the command to send>
```

The connection must be established before, using `connect`.

A full description of the keyword interface can be found [here](#).

————— ★ —————

Verify an answer

Verify an answer received from connected host. Optionally send a command (usually to trigger the answer).

`verify` distinguishes between the verification of a single incoming message and a block of several incoming messages. The verification occurs by checking whether an incoming message or a block of messages matches one or more than one pattern. `verify` patterns are regular expressions. Several different patterns are supported. Some allow matching groups, while others do not.

This results in the following possibilities:

1. Verify a single incoming message.
 - (a) The pattern does not contain matching groups:
In case of a match, `verify` returns `PASS` (but not the entire message), otherwise `FAIL`.
 - (b) The pattern contain one or more than one matching group:
In case of a match, `verify` returns a list of substrings captured from the incoming message, otherwise `None`.
2. Verify a block of several incoming messages.

In case of a block is verified, the pattern is interpreted as *multiline* pattern by the regex engine. This multiline pattern is applied to all messages of the block **in one single step**. This means: No iteration of a list of messages happens!

 - (a) The pattern does not contain matching groups:
In case of a match, `verify` returns `PASS` (but not the entire block of messages), otherwise `FAIL`.
 - (b) The pattern contain one or more than one matching group:
In case of a match, `verify` returns a list of substrings captured from the entire block of messages, otherwise `None`.

The `verify` keyword has the following prototype:

```
verify    conn_name      < the name of the connection >
          search_pattern  < the regular expression used to verify the message(s) >
          timeout         < timeout while waiting for messages within a match_try >
          match_try       < number of attempts to match the search_pattern >
          fetch_block     < whether to wait for a single message or for a block >
          eob_pattern     < pattern to define the end of a block >
          filter_pattern  < pattern to make a preselection of block messages >
          send_cmd        < command sent to the connected host >
          kwargs          < additional parameter >
```

The boolean parameter `fetch_block` is the main switch to distinguish between waiting for a single message or for a block of messages. The default is `False` (= wait for a single message only).

The regex parameter `search_pattern`

- decides if the received message is the desired one or not (if `fetch_block` is `False`),
- decides if all received messages of a block are the desired ones or not (if `fetch_block` is `True`).

`verify` waits within `timeout` `match_try` times. Every single `match_try` causes `verify` to send `send_cmd` again (if defined).

Block detection

A block is a list of incoming messages and is used if a certain pattern (the `search_pattern`) is not expected to match a single message only but several messages instead. Therefore, the `search_pattern` is a multiline pattern. A block of messages has a begin and an end.

- The block detection starts with the execution of the `verify` keyword (= *moment in time*)
- The block detection ends when an incoming message matches a pattern named `eob_pattern` (= *event driven*).

Like the `search_pattern`, the `eob_pattern` is a regular expression - but must not contain matching groups. It's a simple yes/no decision. The `eob_pattern` does not extract any content.

All messages belonging to a block, are stored within an internal message queue intermediately. After the block is detected (match of `eob_pattern`), the list of messages within this queue is converted to a single string with hard coded line ending characters (`\r\n`).

```
"message1\r\nmessage2\r\nmessage3\r\nmessage4"
```

And exactly this is finally the string on which the `search_pattern` is applied.

Preselection of incoming messages

Up to now, we have assumed that during block detection, every incoming message is temporarily stored in the message queue. But that doesn't necessarily have to be the case. If many of the incoming messages are irrelevant for later content analysis, it is possible to place a filter in front that only allows messages with the desired content to pass through.

This filter is called `filter_pattern`. Optionally this filter can be used to make a preselection of messages. If the `filter_pattern` is not defined, all incoming messages will be accepted and stored in the message queue.

Like the `eob_pattern`, the `filter_pattern` is a regular expression - and must also not contain matching groups. It's a simple yes/no decision. The `filter_pattern` does not extract any content.

Consider: If you use backslashes within regular expressions defined in Robot Framework code, you have to mask the backslashes: `\\` !

A full description of the keyword interface can be found [here](#).

————— ★ —————

2.2 Verify examples

Because the `verify` keyword provides a lot of parameters, this section contains some examples that demonstrate the usage.

2.2.1 Verify a single message (1)

Verify a single message without matching groups.

Use case:

- Outgoing message: `TEST-MESSAGE`
- Received message: `TEST-MESSAGE ACK`
- `fetch_block` is `False` (default)
- No matching groups within the `search_pattern`
- `search_pattern` matches the received message

Code:

```
${result}=    conn_manager.verify    conn_name=TEST-CONNECTION
              ...                  search_pattern=TEST-MESSAGE\\sACK
              ...                  send_cmd=TEST-MESSAGE
```

Consider: In Robot Framework code, backslashes need to be masked: `\\` !

Result:

Because the `search_pattern` does not contain matching groups, the result is `None` .

```
INFO - Search pattern 'TEST-MESSAGE\\sACK' matched (try 1) on connection 'TEST-CONNECTION'
INFO - ${result} = None
```

————— ★ —————

2.2.2 Verify a single message (2)

Verify a single message with matching groups.

Use case:

- Outgoing message: `TEST-MESSAGE-1`
- Received message: `TEST-MESSAGE-1 ACK`
- `fetch_block` is `False` (default)
- `search_pattern` contains matching groups
- `search_pattern` matches the received message

Code:

```
conn_manager.verify    conn_name=TEST-CONNECTION
...                  search_pattern=TEST-MESSAGE-(\\d)\\s(\\w+)
...                  send_cmd=TEST-MESSAGE-1
```

Result:

Because `search_pattern` contains matching groups, the result is a list containing the captured substrings from incoming message.

```
INFO - Search pattern 'TEST-MESSAGE-(\d)\s(\w+)' matched (try 1) on connection ↵
      ↪ 'TEST-CONNECTION'
INFO - ${result} = ['1', 'ACK']
```

The result list can be iterated as follows:

```
FOR    ${element}    IN    @{result}
  log    element: '${element}'
END
```

————— ★ —————

2.2.3 Verify a block of messages (1)

Verify a block of messages without filtering and without `search_pattern`.

Use case:

- Outgoing message: `FETCHBLOCK-1`
- Received list of messages:

```
transmission started
transmission point 1 passed
any noise message
transmission point 2 passed
any noise message
transmission point 3 passed
any noise message
transmission ended with code '0'
```

- `fetch_block` is `True`
- No `filter_pattern` defined
- No `search_pattern` defined

Code:

```
${result}=    conn_manager.verify    conn_name=TEST-CONNECTION
              ...                    eob_pattern=transmission\\sended
              ...                    fetch_block=${True}
              ...                    send_cmd=FETCHBLOCK-1
              ...                    timeout=10
```

In this example it requires eight seconds to receive all messages that belongs to the block. Therefore, it is required to define a `timeout` also (here: 10 seconds). The default value (1 second) would cause a verify timeout.

Result:

- Because the `filter_pattern` is not defined, all incoming messages are stored intermediately in the internal message queue.
- Because the `search_pattern` is not defined, the default value (`' .* '`) becomes active. Therefore, we will have a match in every case.

```
INFO - Search pattern '.*' matched (try 1) on connection 'TEST-CONNECTION'
INFO - ${result} = None
```

————— ★ —————

2.2.4 Verify a block of messages (2)

Verify a block of messages without filtering - but with `search_pattern` without matching groups.

Use case:

- Outgoing message: `FETCHBLOCK-1`
- Received list of messages:

```
transmission started
transmission point 1 passed
any noise message
transmission point 2 passed
any noise message
transmission point 3 passed
any noise message
transmission ended with code '0'
```

- `fetch_block` is `True`
- No `filter_pattern` defined
- No matching groups within the `search_pattern`
- `search_pattern` matches the received block of messages

Code:

```
${result}=      conn_manager.verify      conn_name=TEST-CONNECTION
...              ...                      eob_pattern=transmission\\sended
...              ...                      search_pattern=transmission
...              ...                      fetch_block=${True}
...              ...                      send_cmd=FETCHBLOCK-1
...              ...                      timeout=10
```

Result:

- Because the `filter_pattern` is not defined, all incoming messages are stored intermediately in the internal message queue.
- Because the `search_pattern` does not contain matching groups, the result is `None` .

```
INFO - Search pattern 'transmission' matched (try 1) on connection 'TEST-CONNECTION'
INFO - ${result} = None
```

————— ★ —————

2.2.5 Verify a block of messages (3)

Verify a block of messages without filtering - but with `search_pattern` containing matching groups.

Use case:

- Outgoing message: `FETCHBLOCK-1`
- Received list of messages:

```
transmission started
transmission point 1 passed
any noise message
transmission point 2 passed
any noise message
transmission point 3 passed
any noise message
transmission ended with code '0'
```

- `fetch_block` is `True`
- No `filter_pattern` defined
- `search_pattern` contains matching groups
- `search_pattern` matches the received block of messages

Code:

```

${result}=      conn_manager.verify      conn_name=TEST-CONNECTION
                ...                      eob_pattern=transmission\\sended
                ...                      ↩
↪ search_pattern=^transmission\\sstarted\\r\\n(.*)\\r\\ntransmission\\sended ↩
↪ with\\scode\\s'(\d+)'
                ...                      fetch_block=${True}
                ...                      send_cmd=FETCHBLOCK-1
                ...                      timeout=10

```

In this example, the `search_pattern` is used to capture two parts of the content:

- the entire message content between `transmission started` and `transmission ended`,
- the exit code at end of the last message.

Result:

- Because the `filter_pattern` is not defined, all incoming messages are stored intermediately in the internal message queue.
- Because the `search_pattern` contains matching groups, the result is a list of all captured message parts.

```

INFO - Search pattern ↩
↪ '^transmission\sstarted\r\n(.*)\r\ntransmission\sended\s'with\scode\s'(\d+)' ↩
↪ matched (try 1) on connection 'TEST-CONNECTION'
INFO - ${result} = ['transmission point 1 passed\r\nany noise message\r\ntransmission ↩
↪ point 2 passed\r\nany noise message\r\ntransmission point 3 passed\r\nany noise ↩
↪ message', '0']

```

Remember: The `search_pattern` is a multiline pattern! Therefore, the first element of the result list is a single string that is a concatenation of all affected messages from the message queue (with hard coded line endings).

2.2.6 Verify a block of messages (4)

Verify a block of messages with `search_pattern` containing matching groups and with `filter_pattern` defined.

Use case:

- Outgoing message: `FETCHBLOCK-1`
- Received list of messages:

```
transmission started
transmission point 1 passed
any noise message
transmission point 2 passed
any noise message
transmission point 3 passed
any noise message
transmission ended with code '0'
```

- `fetch_block` is `True`
- `filter_pattern` defined
- `search_pattern` contains matching groups
- `search_pattern` matches the received block of messages

Code:

```

${result}=      conn_manager.verify      conn_name=TEST-CONNECTION
                ...                      eob_pattern=transmission\\sended
                ...                      filter_pattern=transmission
                ...                      ↩
↪ search_pattern=^transmission\\sstarted\\r\\n(.*)\\r\\ntransmission\\sended ↪
↪ with\\scode\\s' (\\d+) '
                ...                      fetch_block=${True}
                ...                      send_cmd=FETCHBLOCK-1
                ...                      timeout=10

```

In this example, the `search_pattern` is used to capture two parts of the content:

- the entire message content between `transmission started` and `transmission ended`,
- the exit code at end of the last message.

But we only want to have the `transmission point` messages within our results. We are not interested in all `noise` messages. Therefore, we use the `filter_pattern` to collect the `transmission` messages only.

Result:

- Because the `filter_pattern` is defined, the result only contains messages that matches this filter.
- Because the `search_pattern` contains matching groups, the result is a list of all captured message parts.

```

INFO - Search pattern ↩
↪ '^transmission\\sstarted\\r\\n(.*)\\r\\ntransmission\\sended\\swith\\scode\\s' (\\d+) ' ' ↪
↪ matched (try 1) on connection 'TEST-CONNECTION'
INFO - ${result} = ['transmission point 1 passed\\r\\ntransmission point 2 ↪
↪ passed\\r\\ntransmission point 3 passed', '0']

```

Now the result only contains the `transmission point` messages together with the exit code.

Remember: The `search_pattern` is a multiline pattern! Therefore, the first element of the result list is a single string that is a concatenation of all affected messages from the message queue (with hard coded line endings).

2.3 === REWORKED ===

2.4 Connection types

TODO

2.5 Usage

QConnectBase Library support following keywords for testing connection in RobotFramework.

2.5.1 connect

Use for establishing a connection.

Syntax:

```
connect conn_name=[conn_name]    conn_conf=[conn_conf]
```

Notice

Only `conn_name` and `conn_conf` are now required.



Legacy syntax with 4 arguments is still supported for backward compatibility

```
connect    conn_name=[conn_name]    conn_type=[conn_type]    ↔  
↔ conn_conf=[conn_conf]    conn_mode=[conn_mode]
```

Arguments:

- **conn_name**: Name of the connection.
- **conn_conf**: A dictionary containing configurations for the connection.
It must include **conn_type**, and optionally **conn_mode** and other connection-specific fields (depending on type).
This replaces the need to pass **conn_type** and **conn_mode** as separate arguments.
Each **conn_type** requires a specific structure in **conn_conf**.

Below are examples for each supported type:

- **TCPIPClient**: Create a Raw TCPIP connection to TCP Server.

```
{
  "conn_type": "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**: Create a client connection to a SSH server.

```
{
  "conn_type": "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**: Create a client connection via Serial Port.

```
{
  "conn_type": "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] ]
}
```

2.5.2 disconnect

Use for disconnect a connection by name.

Syntax:

disconnect conn_name

Arguments:

conn_name: Name of the connection.

2.5.3 send command

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

Syntax:

send **command** conn_name=[conn_name] command=[command] [argument
name]=[argument value]

Arguments:

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

2.5.4 transfer file

Use for transferring file from local to remote and vice versa. (Only available for SSHClient connection type)

Syntax:

```
transfer file conn_name=[conn_name] src=[source] dest=[destination]
type=[transfer type]
```

Arguments:

conn_name: Name of the connection.
src: Your source file's path.
dest: The destination path on the remote side.
type: 'put' to send from local to remote, 'get' to bring it back.

2.5.5 transfer item

Use for transferring item from local to remote and vice versa. (Only available for SSHClient connection type)

Syntax:

```
transfer item conn_name=[conn_name] src=[source] dest=[destination]
type=[transfer type]
```

Arguments:

conn_name: Name of the connection.
src: Your source item's path.
dest: The destination path on the remote side.
type: 'put' to send from local to remote, 'get' to bring it back.

2.5.6 verify

The keyword `verify` verifies a single message (or several messages) received from the connected server. Several pattern are available to filter the messages. Optionally, a message can be sent.

Prototype:

```
verify(conn_name: str,
       search_pattern: str=".*",
       timeout: int=1,
       match_try: int=1,
       fetch_block: bool=False,
       eob_pattern: str=".*",
       filter_pattern: str=".*",
       send_cmd: str="",
       kwargs: dict) -> dict
```

Arguments:

- **conn_name:**
Name of the connection.
- **send_cmd:**

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

Default value: empty string `''`.

- **search_pattern:**

Expectation expressed as a **regular expression pattern** (more robust than a plain string comparison). It will match:

- a single line by default (`fetch_block` not used)
- multiple lines if `fetch_block` is enabled

Default value: `.*`, which means "match any character (`.`) repeated zero or more times (`*`)". In practice, this will always match the response, regardless of its content, unless you specify a stricter pattern.

- **timeout:**

Timeout (in seconds) for each `match_try` (if used) while waiting for the response to match the pattern.

Default value: `5`.

- **match_try:**

Number of times to try matching the pattern.

Default value: `1`.

- **fetch_block:**

If `True`, every response line will be put into a block until a line match `eob_pattern` pattern.

Default value: `False`.

- **eob_pattern:**

Applicable only when `fetch_block` is `True`.

Regular expression for matching the endline when using `fetch_block`.

Default value: `.*`.

- **filter_pattern:**

Applicable only when `fetch_block` is `True`.

Regular expression for filtering every line to put into the block of response when using `fetch_block`.

Default value: `.*`.

Return value:

A list of captured string from search.

Example

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

- `${result}[0]` will be `1st`, i.e. the first *captured string* defined in the pattern `([0-9]..)`.
- `${result}[1]` will be `command`, i.e. the second *captured string* defined in the pattern `(command)`.

The `${result}` can be iterated in `FOR` loop as below example:

```
FOR    ${item}    IN    &{result}
Log To Console    Captured string: ${item}
END
```

The following diagrams illustrate the internal flow of the `verify` keyword in `QConnectBase`, including handling of `fetch_block`, `filter_pattern`, `eob_pattern`, `timeout`, and `match_try`.

The sequence diagram shows how messages are read from the connection in a separate thread, and how the `verify` keyword processes them depending on the keyword arguments.

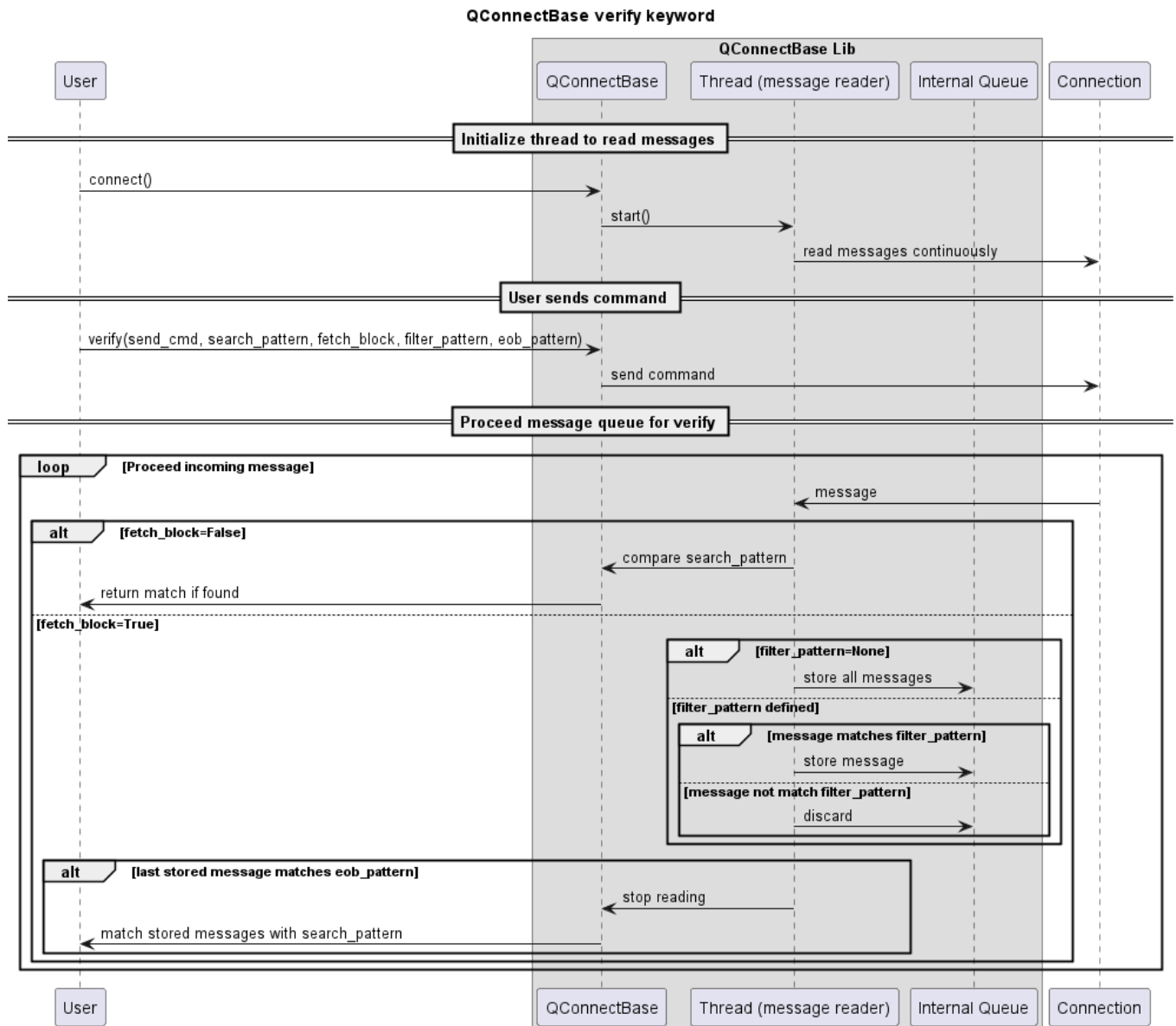


Figure 2.1: Sequence to process messages

The activity diagram illustrates the decision flow of the `verify` keyword. It shows how messages are read, optionally filtered, collected into a block, and checked against `search_pattern`. Each `match_try` has its own timeout, and the keyword stops immediately if a match is found.

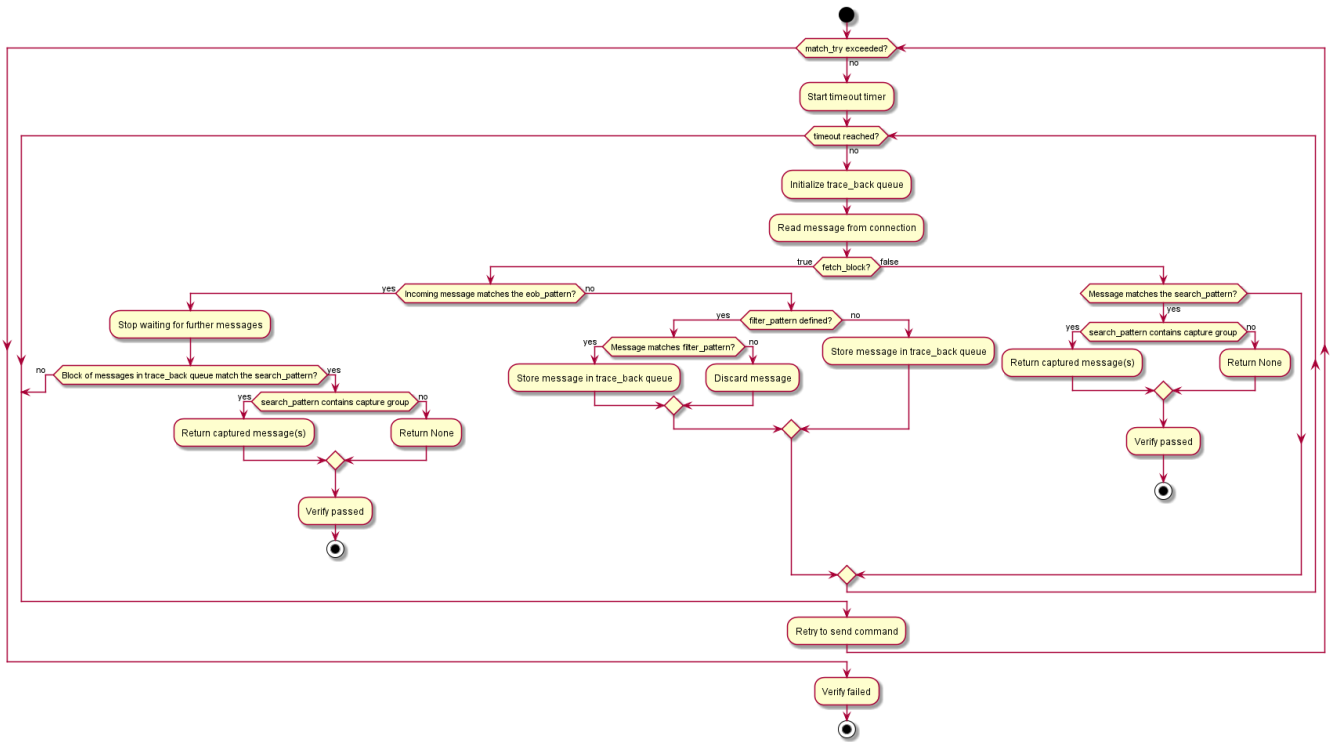


Figure 2.2: Verify workflow

2.6 Example

```

*** Settings ***
Documentation      Suite description
Library           QConnectBase.ConnectionManager

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

    ${res}=              Verify      conn_name=${CONNECTION_NAME}
    ...                  send_cmd=echo ~^START~^;ls -la;echo ~^END$?~^~\r\n
    ...                  search_pattern=~^~START~^~\r\n([\s\S]*)~^~END(\\d+)~^~
    ...                  fetch_block=True
    ...                  eob_pattern=^~^~END(\\d*)~^~

    Log To Console      ${res}[1]

    # Copy a local file to the SFTP server
    transfer file        conn_name=test_ssh
    ...                  src=${src_put}
    ...                  dest=${dest_put}
    ...                  type=put

    # Copy a remote item from the SFTP server to the local host
    transfer item        conn_name=test_ssh
    ...                  src=${src_get}
    ...                  dest=${dest_get}
    ...                  type=get

    # Disconnect
    disconnect    test_ssh

```

Listing 2.1: Robot code example

Explanation:

In the example above, we will establish an SSH connection to the remote host 127.0.0.1:8002 and then navigate back one directory level to list all files/folders within that directory.

To achieve this, we utilize the following steps:

1. First, we use the `send command` keyword, which is designed to send commands to the server and immediately return without waiting for a response. In this case, we send the command "cd .." to the remote host.
2. Next, we employ the `verify` keyword, specifically designed for verifying the response from the server after sending a command. The command sent is specified by the `send_cmd` argument (here, it is the "ls -la" command). We use a regular expression defined by the `search_pattern` argument to verify the response and extract the list of

files/folders after executing the "ls -la" command.

However, by default, if no additional arguments are provided, only the first line of the response after running the command will be captured. To address this limitation, we introduce two additional arguments: **fetch_block** and **eob_pattern**.

When **fetch_block** is set to "True", the **Verify** keyword will capture all lines returned from the server until it encounters a line that matches the pattern defined by **eob_pattern** (End of Block pattern). This extended functionality ensures that we can effectively capture multi-line returns from the server.

To handle variable-length output, the **send_cmd** parameter encapsulates the command between "START" and "END\$?". By setting **fetch_block** to "True" and specifying the **eob_pattern**, we can accurately capture the entire list returned by the server between "START" and "END\$?".

Example: Verifying service startup logs with **fetch_block**

Use case: This use case demonstrates how to use **verify** with **fetch_block** and **eob_pattern** to monitor the startup of a Linux systemd service. When restarting a service, the log usually contains multiple lines: some indicate that the service is starting, some show internal information, and finally, one line confirms that the service has been started successfully. By using **fetch_block**, all these lines can be collected into a block until the end-of-block pattern (**eob_pattern**) is reached.

Command under test:

```
systemctl restart cron; journalctl -u cron --since "now"
```

Sample log output:

```
Sep 12 15:25:01 myhost systemd[1]: Stopping Regular background program processing daemon...
Sep 12 15:25:01 myhost systemd[1]: Starting Regular background program processing daemon...
Sep 12 15:25:01 myhost cron[5678]: (CRON) INFO (pidfile fd = 3)
Sep 12 15:25:01 myhost cron[5678]: (CRON) INFO (Running @reboot jobs)
Sep 12 15:25:01 myhost systemd[1]: Started Regular background program processing daemon.
```

Robot Framework usage:

```

${result} = verify  conn_name=SSH_Connection
...                fetch_block=True
...                search_pattern=( [^\r\n]*Starting Regular background program ↵
↵ processing daemon.*)
...                eob_pattern=.*Started Regular background program processing daemon.*
...                send_cmd=systemctl restart cron; journalctl -u cron --since "now"
```

Returned block:

```
Sep 12 15:25:01 myhost systemd[1]: Starting Regular background program processing daemon...
Sep 12 15:25:01 myhost cron[5678]: (CRON) INFO (pidfile fd = 3)
Sep 12 15:25:01 myhost cron[5678]: (CRON) INFO (Running @reboot jobs)
Sep 12 15:25:01 myhost systemd[1]: Started Regular background program processing daemon.
```

Notes:

- **search_pattern** defines the entry point (*Starting...*).
- **eob_pattern** defines the stopping point (*Started...*).
- If the service fails to start (no *Started...* line appears), the keyword will eventually timeout, which makes this a convenient way to verify service startup behavior in automated tests.

Example: Filtering service startup logs with **fetch_block** and **filter_pattern**

Use case: When a Linux device boots, it sends a large block of startup logs through the connection (e.g., COM port). We only care about one particular service ("systemd-networkd"). By using **fetch_block** together with **filter_pattern**, we can extract only the relevant messages. Finally, **eob_pattern** is used to stop reading logs as soon as the service reports a successful startup.

Important Note:

`filter_pattern` filters the incoming lines, keeping only those relevant to the service of interest. `eob_pattern` is applied **after filtering**, meaning it must match one of the filtered lines. If it does not, the keyword will not stop until timeout occurs. This ensures the block stops at the correct line related to the service and avoids including unrelated logs.

```
${result} = verify conn_name=COM_Connection
...               fetch_block=True
...               search_pattern=(.*)
...               filter_pattern=.*systemd-networkd.*
...               eob_pattern=.*eth0: Gained carrier.*
...               send_cmd=dmesg --follow
```

Connection log (sample):

```
[ 1.234567] systemd[1]: Starting udev Kernel Device Manager...
[ 1.456789] systemd[1]: Started udev Kernel Device Manager.
[ 2.123456] systemd[1]: Starting Network Service...
[ 2.223344] systemd-networkd[120]: eth0: link is up
[ 2.223355] systemd-networkd[120]: eth0: Gained carrier
[ 2.345678] systemd[1]: Started Network Service.
[ 2.456789] systemd[1]: Starting Login Service...
[ 2.567890] systemd[1]: Started Login Service.
```

Returned block after applying `filter_pattern` and stopping at `eob_pattern` :

```
[ 2.223344] systemd-networkd[120]: eth0: link is up
[ 2.223355] systemd-networkd[120]: eth0: Gained carrier
```

Explanation:

- `fetch_block=True` collects all filtered lines until `eob_pattern` is matched.
- `filter_pattern=.*systemd-networkd.*` ensures only logs from the relevant service are included.
- `eob_pattern=.*eth0: Gained carrier.*` stops collection when the filtered line matches the pattern.
- Lines from other services (e.g., udev, Login Service) are automatically excluded from the block.

2.7 Auxiliary Utilities

2.7.1 Introduction to RMQSignal

The *RMQSignal* class serves as an extension to the core functionalities, providing support for Inter-Process Communication (IPC) based on the RabbitMQ infrastructure. This utility module is designed to facilitate the exchange of signals accompanied by payloads between various processes, enhancing the overall communication mechanism.

2.7.2 Prerequisites

For Windows:

To use this library, installing RabbitMQ is required. For RabbitMQ installation on Windows, Earlang needs to be installed beforehand.

For user convenience, we have created a .bat file to install all necessary components. Users simply need to run the file `QConnectBase/tools/setup_rabbitmq.bat` to install the entire infrastructure required for these Auxiliary Utilities.

For Ubuntu:

1. Update the system. First, ensure your system is up to date by running the command:

```
sudo apt update && sudo apt upgrade -y
```

2. Add the required repositories. Add the official RabbitMQ signing key and repository by running the following commands:

```
sudo apt install curl gnupg -y
curl -fsSL https://packages.rabbitmq.com/gpg | sudo apt-key add -
sudo add-apt-repository 'deb https://dl.bintray.com/rabbitmq/debian focal main'
```

3. Install RabbitMQ Server. Now, proceed with installing RabbitMQ:

```
sudo apt update && sudo apt install rabbitmq-server -y
```

4. Enable and start the RabbitMQ Service. After installation, enable and start the RabbitMQ service:

```
sudo systemctl enable rabbitmq-server
sudo systemctl start rabbitmq-server
```

2.7.3 Library Keywords and Usage

The keywords provided by this library offer a range of functionalities designed to streamline various tasks and processes. Below are some of the key functionalities along with their usage:

Send Signal Keyword

Usage: Used to send a signal to other processes.

Syntax: Send Signal [signal_name] [payloads] [Receiver]

Arguments:

- `signal_name` (string): Defines the name of the signal.
- `payloads`: Specifies the payloads of the signal.
- `Receiver` (optional, default is None): Specifies the receiver. If defined, the signal will be sent only to that specified receiver. If not defined, it sends a broadcast to all receivers.

Wait For Signal Keyword

Usage: Used to wait for a specific signal within a timeout.

Syntax: Wait For Signal [signal_name] [timeout]

Arguments:

- `signal_name` (string): Defines the name of the signal to wait for.
- `timeout` (optional, default is 10): Specifies the timeout in seconds to wait for the specific signal.

Return: Returns the payloads of the received signal, if received within the timeout.

Wait For Signals Keyword

Usage: Used to wait for multiple specific signals within a timeout.

Syntax: Wait For Signals [signal_names] [timeout]

Arguments:

- `signal_names` (list of strings): Defines the list of signal names to wait for.
- `timeout` (optional, default is 10): Specifies the timeout in seconds to wait for the full set of specific signals.

Return: Returns a list of payloads of the signals in the same order if the full set of signals is received within the timeout.

Set Signal Receiver Name Keyword

Usage: Used to set a specific name for the Signal Receiver, which defines a unique name for the current process. Other processes can send signals directly to the current process using this receiver name.

Syntax: `Set Signal Receiver Name [receiver] [force]`

Arguments:

- `receiver` (string): Specifies the name to set for the current process signal receiver.
- `force` (optional, default is True): Determines whether to force-create the signal receiver. If `force` is set to False, an exception will be raised if any process has already defined this name as a signal receiver.

Unset Signal Receiver Name Keyword

Usage: Used to unset the current Signal Receiver name of the current process.

Syntax: `Unset Signal Receiver Name`

2.7.4 Example of Inter-Process Communication

Content of ProcessA.robot file:

```
*** Settings ***
Library           QConnectBase.message.RMQSignal

*** Test Cases ***
Test signal
    Log To Console      Process A start
    Send Signal         HelloFromProcA      Hello, I'm A           receiverB
    Sleep               1s
    Send Signal         Hello2FromProcA      Hello 2nd time, I'm A           receiverB
    ${res}=             Wait For Signal      HelloFromProcB      ${20}
    Log To Console      Get resp signal from ProcB: ${res}
```

Content of ProcessB.robot file:

```
*** Settings ***
Library           QConnectBase.message.RMQSignal

*** Test Cases ***
Receiving Multiple Signals
    Log To Console      message
    Set Signal Receiver Name      receiverB
    ${list_of_signal} =      Create List      HelloFromProcA      Hello2FromProcAfull
    ${res}=                 Wait For Signals      ${list_of_signal}      ${15}
    Sleep                   2s
    Log To Console          Get signal from ProcA: ${res}
    Send Signal              HelloFromProcB      Hello A, I'm B. Got your signal:${res}
```

Explanation: In the above test cases, two separate Robot Framework test files, ProcessA.robot, and ProcessB.robot are used for inter-process communication.

In ProcessB.robot, the test case 'Receiving Multiple Signals' starts by setting up a signal receiver 'receiverB'. It then waits for a set of signals (HelloFromProcA and Hello2FromProcA) from ProcessA. Once both signals are received within 15 seconds, it logs to console the received signals' payloads and waits for 2 seconds before sending a response signal 'HelloFromProcB' to ProcessA.

In ProcessA.robot, the test case 'Sending Multiple Signals' sends HelloFromProcA signal to 'receiverB' in ProcessB. After a 1-second pause, it sends another signal 'Hello2FromProcA' to 'receiverB'. Then, it waits for a response, expecting signal 'HelloFromProcB' from ProcessB within 20 seconds. Upon receiving signal, it logs the received payloads to console.

2.8 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 protocol. 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 handling connection class should be derived from **QConnectBase.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 determine 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)
```
- In case you use the lowlevel receiver thread for receiving data byte by byte. 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.

2.9 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, then you can assure that all EOL are managed by git correctly as defined.

2.10 Sourcecode Documentation

For investigating sourcecode, please refer to [QConnectBase library documentation](#)

2.11 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!

2.12 About

2.12.1 Maintainers

[Nguyen Huynh Tri Cuong](#)

2.12.2 Contributors

[Nguyen Huynh Tri Cuong](#)

[Thomas Pollerspöck](#)

2.13 License

Copyright 2020-2023 Robert Bosch GmbH

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Chapter 3

`__init__.py`

3.1 Class: `ConnectionManager`

Imported by:

```
from QConnectBase.__init__ import ConnectionManager
```

Class to manage all connections.

Chapter 4

connection_base.py

4.1 Class: BrokenConnError

Imported by:

```
from QConnectBase.connection_base import BrokenConnError
```

4.2 Class: ConnectionBase

Imported by:

```
from QConnectBase.connection_base import ConnectionBase
```

Base class for all connection classes.

4.2.1 Method: is_supported_platform

Check if current platform is supported.

Returns:

/ Type: bool /
True if platform is supported.
False if platform is not supported.

4.2.2 Method: is_precondition_pass

Check for precondition.

Returns:

/ Type: bool /
True if passing the precondition.
False if failing the precondition.

4.2.3 Method: get_connection_type

Get the connection type.

Returns:

/ Type: str /
The connection type.

4.2.4 Method: error_instruction

Get the error instruction.

Returns:

/ Type: str /
Error instruction string.

4.2.5 Method: quit

>> This method MUST be overridden in derived class <<

Abstract method for quitting the connection.

Arguments:

- `is_disconnect_all`
/ Condition: optional / Type: bool /
Determine if it's necessary to disconnect all connections.

Returns:

(no returns)

4.2.6 Method: connect

>> This method MUST be overridden in derived class <<

Abstract method for quitting the connection.

Arguments:

- `device`
/ Condition: required / Type: str /
Device name.
- `files`
/ Condition: optional / Type: list /
Trace file list if using dlt connection.
- `test_connection`
/ Condition: optional / Type: bool /
Determine if it's necessary for testing the connection.

Returns:

(no returns)

4.2.7 Method: disconnect

>> This method MUST be overridden in derived class <<

Abstract method for disconnecting connection.

Arguments:

- `device`
/ Condition: required / Type: str /
Device's name.

Returns:

(no returns)

4.2.8 Method: send_obj

Wrapper method to send message to a tcp connection.

Arguments:

- `send_cmd`
/ *Condition*: required / *Type*: str /
Data to be sent.
- `cr`
/ *Condition*: optional / *Type*: str /
Determine if it's necessary to add newline character at the end of command.

Returns:

(no returns)

4.2.9 Method: read_obj

Wrapper method to get the response from connection.

Returns:

- `msg`
/ *Type*: str /
Responded message.

4.2.10 Method: wait_4_trace

Suspend the control flow until a Trace message is received which matches to a specified regular expression.

Arguments:

- `search_obj`
/ *Condition*: required / *Type*: str /
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`
/ *Condition*: optional / *Type*: bool / *Default*: False /
Determine if 'fetch block' feature is used.
- `end_of_block_pattern`
/ *Condition*: optional / *Type*: str / *Default*: None /
The end of block pattern.
- `filter_pattern`
/ *Condition*: optional / *Type*: str / *Default*: '.*' /
Pattern to filter message line by line.
- `timeout`
/ *Condition*: optional / *Type*: int / *Default*: 0 /
Timeout parameter specified as a floating point number in the unit 'seconds'.
- `fct_args`
/ *Condition*: optional / *Type*: Tuple / *Default*: None /
List of function arguments passed to be sent.

Returns:

- `match`
/ *Type*: `re.Match` /

If no trace message matched to the specified regular expression and a timeout occurred, return `None`.

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'`.

4.2.11 Method: `wait_4_trace_continuously`

Getting trace log continuously without creating a new trace queue.

Arguments:

- `trace_queue`
/ *Condition*: required / *Type*: `Queue` /
Queue to store the traces.
- `timeout`
/ *Condition*: optional / *Type*: `int` / *Default*: 0 /
Timeout for waiting a matched log.
- `fct_args`
/ *Condition*: optional / *Type*: `Tuple` / *Default*: `None` /
Arguments to be sent to connection.

Returns:

- `None`
/ *Type*: `None` /
If no trace message matched to the specified regular expression and a timeout occurred.
- `match`
/ *Type*: `re.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'`.

4.2.12 Method: `create_and_activate_trace_queue`

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

Arguments:

- `search_element`
/ *Condition*: required / *Type*: `str` /
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`
/ *Condition*: optional / *Type*: `bool` / *Default*: `False` /
Determine if 'fetch block' feature is used.
- `end_of_block_pattern`
/ *Condition*: optional / *Type*: `str` / *Default*: `None` /
The end of block pattern.

- `regex.line_filter_pattern`
/ *Condition*: optional / *Type*: `re.Pattern` / *Default*: `None` /
Regular expression object to filter message line by line.

Returns:

- `trq_handle, trace_queue`
/ *Type*: tuple /
The handle and search object

4.2.13 Method: deactivate_and_delete_trace_queue

Deactivate trace queue and delete.

Arguments:

- `trq_handle`
/ *Condition*: required / *Type*: `int` /
Trace queue handle.
- `trace_queue`
/ *Condition*: required / *Type*: `Queue` /
Trace queue object.

Returns:

(no returns)

4.2.14 Method: activate_trace_queue

Activates a trace message filter specified as a regular expression. All matching trace messages are put in the specified queue object.

Arguments:

- `search_obj`
/ *Condition*: required / *Type*: `str` /
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`
/ *Condition*: required / *Type*: `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`
/ *Condition*: optional / *Type*: `bool` / *Default*: `False` /
Determine if 'fetch block' feature is used.
- `end_of_block_pattern`
/ *Condition*: optional / *Type*: `str` / *Default*: `None` /
The end of block pattern.
- `line_filter_pattern`
/ *Condition*: optional / *Type*: `re.Pattern` / *Default*: `None` /
Regular expression object to filter message line by line.

Returns:

- `handle_id`
/ *Type*: `int` /
Handle to deactivate the message filter.

4.2.15 Method: deactivate_trace_queue

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

Arguments:

- handle
/ Condition: required / Type: int /
Integer object returned by ActivateTraceQ() method.

Returns:

* is_success

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

True : Trace message filter successfully deleted.

4.2.16 Method: check_timeout

>> This method will be override in derived class <<

Check if responded message come in cls.RESPOND_TIMEOUT or we will raise a timeout event.

Arguments:

- timeout
/ Condition: required / Type: int /
Timeout in seconds.

Returns:

(no returns)

4.2.17 Method: pre_msg_check

>> This method will be override in derived class <<

Pre-checking message when receiving it from connection.

Arguments:

- msg
/ Condition: required / Type: str /
Received message to be checked.

Returns:

(no returns)

4.2.18 Method: post_msg_check

>> This method should be overridden in the derived class <<

Post-checking message when receiving it from connection.

Arguments:

- msg
/ Condition: required / Type: str /
Received message to be checked.

Returns:

(no returns)

Chapter 5

connection_manager.py

5.1 Class: InputParam

Imported by:

```
from QConnectBase.connection_manager import InputParam
```

5.1.1 Method: get_attr_list

5.2 Class: ConnectParam

Imported by:

```
from QConnectBase.connection_manager import ConnectParam
```

Class for storing parameters for connect action.

5.3 Class: SendCommandParam

Imported by:

```
from QConnectBase.connection_manager import SendCommandParam
```

Class for storing parameters for send command action.

5.4 Class: VerifyParam

Imported by:

```
from QConnectBase.connection_manager import VerifyParam
```

Class for storing parameters for verify action.

5.5 Class: ConnectionManager

Imported by:

```
from QConnectBase.connection_manager import ConnectionManager
```

Class to manage all connections.

5.5.1 Method: end_suite

5.5.2 Method: quit

Quit connection manager.

Returns:

(no returns)

5.5.3 Method: add_connection

Add a connection to managed dictionary.

Arguments:

- name
/ *Condition*: required / *Type*: str /
Connection's name.
- conn
/ *Condition*: required / *Type*: socket.socket /
Connection object.

Returns:

(no returns)

5.5.4 Method: remove_connection

Remove a connection by name.

Arguments:

- conn_name
/ *Condition*: required / *Type*: str /
Connection's name.

Returns:

(no returns)

5.5.5 Method: get_connection_by_name

Get an exist connection by name.

Arguments:

- conn_name
/ *Condition*: required / *Type*: str /
Connection's name.

Returns:

- conn
/ *Type*: socket.socket /
Connection object.

5.5.6 Keyword: disconnect

Keyword for disconnecting a connection by name.

Arguments:

- `conn_name`
/ *Condition*: required / *Type*: str /
Connection's name.

Returns:

(no returns)

5.5.7 Keyword: connect

Making a connection.

Arguments:

- `conn_name`
/ *Condition*: required / *Type*: str /
Name of connection. It can be specified in `conn_conf` dictionary.
- `conn_conf`
/ *Condition*: optional / *Type*: dictionary / *Default*: None /
A dictionary containing configurations for the connection.
It must include `conn_type`, and optionally `conn_mode` and other connection-specific fields (depending on type).
Example `conn_conf` for `TCPIPClient`:

```
` {   "conn_type": "TCPIPClient",   "address": [server host], # Optional. Default
value is "localhost".   "port": [server port]           # Optional. Default value is
1234. }`
```
- `conn_type` (deprecated)
/ *Condition*: optional / *Type*: str / *Default*: 'TCPIPClient' /
Type of connection. It can be specified in `conn_conf` dictionary.
Supported connection types:
 - `TCPIPClient`: Create a Raw TCP/IP connection to TCP Server.
 - `SSHClient`: Create a client connection to a SSH server.
 - `SerialClient`: Create a client connection via Serial Port.

In addition to the connection types listed above, other types are also available through classes inheriting from `QConnectBase`.

- `conn_mode` (deprecated)
/ *Condition*: optional / *Type*: str / *Default*: '' /
Connection mode. It can be specified in `conn_conf` dictionary.

Returns:

(no returns)

5.5.8 Keyword: send_command

Send command to a connection.

Arguments:

- `conn_name`
/ *Condition*: required / *Type*: str /
Name of connection.
- `command`
/ *Condition*: required / *Type*: str /
Command to be sent.
- `kwargs`
/ *Condition*: optional / *Type*: dict / *Default*: {} /
The optional arguments depend on the connection type used in the 'connect' keyword.

Returns:

(no returns)

5.5.9 Keyword: transfer_file

DEPRECATED!! Use keyword transfer_item instead. Transfer file from local to remote and vice versa.

Arguments:

- `conn_name`
/ *Condition*: required / *Type*: str /
Name of connection.
- `src`
/ *Condition*: required / *Type*: str /
Source file path.
- `dest`
/ *Condition*: required / *Type*: str /
Destination file path.
- `type`
/ *Condition*: required / *Type*: str /
Transfer file type.

 'get' - Copy a remote file from the SFTP server to the local host.
 'put' - Copy a local file to the SFTP server.

Returns:

(no returns)

5.5.10 Keyword: transfer_item

Transfer item from local to remote and vice versa.

Arguments:

- `conn_name`
/ *Condition*: required / *Type*: str /
Name of connection.

- `src`
/ *Condition*: required / *Type*: str /
Source item path.
- `dest`
/ *Condition*: required / *Type*: str /
Destination item path.
- `type`
/ *Condition*: required / *Type*: str /
Transfer item type.

 'get' - Copy a remote item from the SFTP server to the local host.
 'put' - Copy a local item to the SFTP server.

Returns:

(no returns)

5.5.11 Keyword: `execute_script`

Executes a script file by sending commands to a device through the provided connection.

Arguments:

- `conn_name`
/ *Condition*: required / *Type*: str /
Name of connection.
- `script_path`
/ *Condition*: required / *Type*: str /
Script file path.

Returns:

(no returns)

5.5.12 Keyword: `set_default_verify_timeout`

Set the default verify timeout value for the connection.

Supports flexible input formats such as: - Duration with units (e.g. '1h 10s', '2m30s', '500ms') - HH:MM:SS format (e.g. '01:00:10' for 1 hour, 0 minutes, 10 seconds) - Plain numeric values (e.g. '42') interpreted as seconds

Arguments:

- `time_str`
/ *Condition*: required / *Type*: str or float or int /

A string representing the duration. Units supported include:

- h for hours
- m for minutes (or ms for milliseconds)
- s for seconds
- ms for milliseconds

If no unit is specified, the value is interpreted as seconds.

5.5.13 Keyword: `set_default_emergency_timeout`

Set the default emergency timeout value for the connection.

Supports flexible input formats such as: - Duration with units (e.g. '1h 10s', '2m30s', '500ms') - HH:MM:SS format (e.g. '01:00:10' for 1 hour, 0 minutes, 10 seconds) - Plain numeric values (e.g. '42') interpreted as seconds

Arguments:

- `time_out`
/ *Condition*: required / *Type*: str or float or int /

A string representing the duration. Units supported include:

- h for hours
- m for minutes (or ms for milliseconds)
- s for seconds
- ms for milliseconds

If no unit is specified, the value is interpreted as seconds.

5.5.14 Keyword: `verify`

Verify a pattern from connection response after sending a command.

Arguments:

- `conn_name`
/ *Condition*: required / *Type*: str /
Name of connection.
- `search_pattern`
/ *Condition*: optional / *Type*: str / *Default*: .* /
Expectation expressed as a **regular expression pattern** (more robust than a plain string comparison).
It will match:
 - a single line by default (`fetch_block` not used)
 - multiple lines if `fetch_block` is enabled

Default value: `.*`, which means "match any character (.) repeated zero or more times (*)".

In practice, this will always match the response, regardless of its content, unless you specify a stricter pattern.

- `timeout`
/ *Condition*: optional / *Type*: float / *Default*: 5 /
Timeout parameter specified as a floating point number in the unit 'seconds'.
- `match_try`
/ *Condition*: optional / *Type*: int / *Default*: 1 /
Number of time for trying to match the pattern.
- `fetch_block`
/ *Condition*: optional / *Type*: bool / *Default*: False /
Determine if 'fetch block' feature is used.
If True, every single line of received message will be put into a block until a line match `eob_pattern` pattern.
- `eob_pattern`
/ *Condition*: optional / *Type*: str / *Default*: None /
Applicable only when `fetch_block` is True.
Regular expression for matching the endline when using `fetch_block`.

- `filter_pattern`
/ Condition: optional / Type: str / Default: '.' /*
 Applicable only when `fetch_block` is `True`.
 Regular expression for filtering every line to put into the block of response when using `fetch_block`.
- `send_cmd`
/ Condition: optional / Type: str / Default: '' /
 Command to be sent.
- `kwargs`
/ Condition: optional / Type: Dict / Default: None /
 The optional arguments depend on the connection type used in the 'connect' keyword.
 Supported options:

Connection Type	Argument	Explanation
Winapp	<code>element_def</code>	Definition for detecting GUI item: <i>Type: str / Default: ''</i>

Returns:

- `res`
/ Type: list /
 List of captured string from `search_pattern`
 For example, if `search_pattern` is `(?<=\s).*([0-9]...)*(command).$`, and the response from connection is `This is the 1st test command.`, then the returned list will be `['1st', 'command']`.
 Thus:
 - `{res}[0]` will be **1st**, i.e. the first *captured string* defined in the pattern `([0-9]...)`.
 - `{res}[1]` will be **command**, i.e. the second *captured string* defined in the pattern `(command)`.

5.6 Class: TestOption

Imported by:

```
from QConnectBase.connection_manager import TestOption
```

Chapter 6

constants.py

6.1 Class: SocketType

Imported by:

```
from QConnectBase.constants import SocketType
```

6.2 Class: String

Imported by:

```
from QConnectBase.constants import String
```

Chapter 7

rabbitmq_client.py

7.1 Class: RabbitmqClientConfig

Imported by:

```
from QConnectBase.message.rabbitmq_client import RabbitmqClientConfig
```

Class to store the configuration for SSH connection.

7.2 Class: RabbitmqClient

Imported by:

```
from QConnectBase.message.rabbitmq_client import RabbitmqClient
```

Rabbitmq client connection class.

7.2.1 Method: on_response

7.2.2 Method: connect

Implementation for creating a rabbitmq connection.

Returns:

(no returns)

7.2.3 Method: close

Close rabbitmq connection.

Returns:

(no returns)

7.2.4 Method: quit

Quit and stop receiver thread.

Returns:

(no returns)

7.3 Class: RMQSignal

Imported by:

```
from QConnectBase.message.rabbitmq-client import RMQSignal
```

RMQSignal class.

7.3.1 Method: send_signal

Send sinal to other processes.

Arguments:

- `signal_name`
/ *Condition*: required / *Type*: str /
Signal to be sent.
- `payload`
/ *Condition*: required / *Type*: str /
Payloads for signal.
- `receiver`
/ *Condition*: optional / *Type*: str / *Default*: None /
Specific the signal receiver to send signal to.

Returns:

(no returns)

7.3.2 Method: unset_signal_receiver_name

Unset siganl receiver.

Returns:

(no returns)

7.3.3 Method: set_signal_receiver_name

Set the signal receiver to be received signal.

Arguments:

- `receiver`
/ *Condition*: optional / *Type*: str / *Default*: " /
Name a signal receiver to receive signal.
- `force`
/ *Condition*: optional / *Type*: bool / *Default*: True /
Force create the signal receiver (delete the exist signal receiver with the same name).

Returns:

(no returns)

7.3.4 Method: consume_channel

Consume the message from specific queue.

Arguments:

- `exchange`
/ *Condition*: required / *Type*: str /
Name of the exchange.

- `queue_name`
/ *Condition*: required / *Type*: str /
Name of the queue.
- `routing_key`
/ *Condition*: required / *Type*: str /
Routing key string.
- `stop_event`
/ *Condition*: required / *Type*: Event /
Event to notify stopping consuming.
- `signal_name`
/ *Condition*: required / *Type*: str or list /
Name of the signal to be wait for.
- `messages`
/ *Condition*: required / *Type*: list or dict /
Storage for received messages.
- `queue_delete`
/ *Condition*: optional / *Type*: bool / *Default*: False /
Determine if we should delete the queue at the end.

Returns:*(no returns)***7.3.5 Method: wait_for_signal**

Wait for specific signal in timeout.

Arguments:

- `signal_name`
/ *Condition*: optional / *Type*: str /
Name of the signal to wait for.
- `timeout`
/ *Condition*: optional / *Type*: int / *Default*: 10 /
Timeout for waitting the signal. Default is 10 seconds.

Returns:

/ *Type*: str /
Payloads of the waitting signal if received.

7.3.6 Method: wait_for_signals

Wait for multiple specific signals in timeout.

Arguments:

- `signal_name`
/ *Condition*: optional / *Type*: list /
List of the signals to wait for.

- `timeout`
/ *Condition*: optional / *Type*: int / *Default*: 10 /
Timeout for waiting the signal. Default is 10 seconds.

Returns:

/ *Type*: str /
List of payloads of the waiting signals if received.

Chapter 8

qlogger.py

8.1 Class: ColorFormatter

Imported by:

```
from QConnectBase.qlogger import ColorFormatter
```

Custom formatter class for setting log color.

8.1.1 Method: format

Set the color format for the log.

Arguments:

- record
/ *Condition:* required / *Type:* str /
Log record.

Returns:

/ *Type:* logging.Formatter /
Log with color formatter.

8.2 Class: QFileHandler

Imported by:

```
from QConnectBase.qlogger import QFileHandler
```

Handler class for user defined file in config.

8.2.1 Method: get_log_path

Get the log file path for this handler.

Arguments:

- config
/ *Condition:* required / *Type:* DictToClass /
Connection configurations.

Returns:

/ Type: str /

Log file path.

8.2.2 Method: get_config_supported

Check if the connection config is supported by this handler.

Arguments:

- `config`
/ Condition: required / Type: DictToClass /
Connection configurations.

Returns:

/ Type: bool /

True if the config is supported.

False if the config is not supported.

8.3 Class: QDefaultFileHandler

Imported by:

```
from QConnectBase.qlogger import QDefaultFileHandler
```

Handler class for default log file path.

8.3.1 Method: get_log_path

Get the log file path for this handler.

Arguments:

- `logger_name`
/ Condition: required / Type: str /
Name of the logger.

Returns:

/ Type: str /

Log file path.

8.3.2 Method: get_config_supported

Check if the connection config is supported by this handler.

Arguments:

- `config`
/ Condition: required / Type: DictToClass /
Connection configurations.

Returns:

/ Type: bool /

True if the config is supported.

False if the config is not supported.

8.4 Class: QConsoleHandler

Imported by:

```
from QConnectBase.qlogger import QConsoleHandler
```

Handler class for console log.

8.4.1 Method: get_config_supported

Check if the connection config is supported by this handler.

Arguments:

- `config`
/ *Condition*: required / *Type*: DictToClass /
Connection configurations.

Returns:

/ *Type*: bool /
True if the config is supported.
False if the config is not supported.

8.5 Class: QLogger

Imported by:

```
from QConnectBase.qlogger import QLogger
```

Logger class for QConnect Libraries.

8.5.1 Method: get_logger

Get the logger object.

Arguments:

- `logger_name`
/ *Condition*: required / *Type*: str /
Name of the logger.

Returns:

- `logger`
/ *Type*: Logger /
Logger object. .

8.5.2 Method: set_handler

Set handler for logger.

Arguments:

- `config`
/ *Condition*: required / *Type*: DictToClass /
Connection configurations.

Returns:

- `handler_ins`
/ *Type*: `logging.handler` /
None if no handler is set.
Handler object.

Chapter 9

serial_base.py

9.1 Class: SerialConfig

Imported by:

```
from QConnectBase.serialclient.serialbase import SerialConfig
```

Class to store the configuration for Serial connection.

9.2 Class: SerialSocket

Imported by:

```
from QConnectBase.serialclient.serialbase import SerialSocket
```

Class for handling serial connection.

9.2.1 Method: connect

Connect to serial port.

Returns:

(no returns)

9.2.2 Method: disconnect

Disconnect serial port.

Arguments:

- `_device`
/ *Condition:* required / *Type:* str /
Unused

Returns:

(no returns)

9.2.3 Method: quit

Quit serial connection.

Returns:

(no returns)

9.3 Class: SerialClient

Imported by:

```
from QConnectBase.serialclient.serialbase import SerialClient
```

Serial client class.

9.3.1 Method: connect

Connect to the Serial port.

Returns:

(no returns)

Chapter 10

raw_tcp.py

10.1 Class: RawTCPBase

Imported by:

```
from QConnectBase.tcp.raw.raw_tcp import RawTCPBase
```

Base class for a raw tcp connection.

10.2 Class: RawTCPServer

Imported by:

```
from QConnectBase.tcp.raw.raw_tcp import RawTCPServer
```

Class for a raw tcp connection server.

10.3 Class: RawTCPClient

Imported by:

```
from QConnectBase.tcp.raw.raw_tcp import RawTCPClient
```

Class for a raw tcp connection client.

Chapter 11

ssh_client.py

11.1 Class: AuthenticationType

Imported by:

```
from QConnectBase.tcp.ssh.ssh_client import AuthenticationType
```

11.2 Class: SSHConfig

Imported by:

```
from QConnectBase.tcp.ssh.ssh_client import SSHConfig
```

Class to store the configuration for SSH connection.

11.3 Class: SSHClient

Imported by:

```
from QConnectBase.tcp.ssh.ssh_client import SSHClient
```

SSH client connection class.

11.3.1 Method: connect

Implementation for creating a SSH connection.

Returns:

(no returns)

11.3.2 Method: transfer_file

DEPRECATED!! Use keyword transfer_item instead. Transfer file from local to remote and vice versa.

Arguments:

- src
/ *Condition:* required / *Type:* str /
Source file path.
- dest
/ *Condition:* required / *Type:* str /
Destination file path.

- `transfer_type`
/ *Condition*: required / *Type*: str /
Transfer file type.

'get' - Copy a remote file from the SFTP server to the local host
'put' - Copy a local file to the SFTP server

Returns:

(no returns)

11.3.3 Method: transfer_item

Transfer item from local to remote and vice versa.

Arguments:

- `src`
/ *Condition*: required / *Type*: str /
Source item path.
- `dest`
/ *Condition*: required / *Type*: str /
Destination item path.
- `transfer_type`
/ *Condition*: required / *Type*: str /
Transfer item type.

'get' - Copy a remote item from the SFTP server to the local host
'put' - Copy a local item to the SFTP server

Returns:

(no returns)

11.3.4 Method: close

Close SSH connection.

Returns:

(no returns)

11.3.5 Method: quit

Quit and stop receiver thread.

Returns:

(no returns)

Chapter 12

tcp_base.py

12.1 Class: TCPConfig

Imported by:

```
from QConnectBase.tcp.tcp_base import TCPConfig
```

Class to store configurations for TCP connection.

12.1.1 Method: validate

12.2 Class: TCPBase

Imported by:

```
from QConnectBase.tcp.tcp_base import TCPBase
```

Base class for a tcp connection.

12.2.1 Method: close

Close connection.

Returns:

(no returns)

12.2.2 Method: quit

Quit connection.

Arguments:

- `is_disconnect_all`
/ *Condition*: required / *Type*: bool /
Determine if it's necessary for disconnect all connection.

Returns:

(no returns)

12.2.3 Method: connect

>> Should be override in derived class.

Establish the connection.

Returns:

(no returns)

12.2.4 Method: disconnect

>> Should be override in derived class.

Disconnect the connection.

Returns:

(no returns)

12.3 Class: TCPBaseServer

Imported by:

```
from QConnectBase.tcp.tcp_base import TCPBaseServer
```

Base class for TCP server.

12.3.1 Method: accept_connection

Wrapper method for handling accept action of TCP Server.

Returns:

(no returns)

12.3.2 Method: connect

12.3.3 Method: disconnect

12.4 Class: TCPBaseClient

Imported by:

```
from QConnectBase.tcp.tcp_base import TCPBaseClient
```

Base class for TCP client.

12.4.1 Method: connect

12.4.2 Method: disconnect

Chapter 13

utils.py

13.1 Function: has_capturing_groups

Checks whether a pattern contains capturing groups qconnectbase-utils-validate-regex-pattern =====

Checks whether the given pattern is valid qconnectbase-utils-singleton =====

Imported by:

```
from QConnectBase.utils import Singleton
```

Class to implement Singleton Design Pattern. This class is used to derive the TTFisClientReal as only a single instance of this class is allowed.

Disabled pyLint Messages: R0903: Too few public methods (%s/%s) Used when class has too few public methods, so be sure it's really worth it.

This base class implements the Singleton Design Pattern required for the TTFisClientReal. Adding further methods does not make sense.

13.2 Class: DictToClass

Imported by:

```
from QConnectBase.utils import DictToClass
```

Class for converting dictionary to class object.

13.2.1 Method: validate

13.3 Class: Utils

Imported by:

```
from QConnectBase.utils import Utils
```

Class to implement utilities for supporting development.

13.3.1 Method: get_all_descendant_classes

Get all descendant classes of a class

Arguments: cls: Input class for finding descendants.

Returns:

/ Type: list /
 Array of descendant classes.

13.3.2 Method: get_all_sub_classes

Get all children classes of a class

Arguments:

- `cls`
/ Condition: required / Type: class /
 Input class for finding children.

Returns:

/ Type: list /
 Array of children classes.

13.3.3 Method: is_valid_host**13.3.4 Method: execute_command****13.3.5 Method: kill_process****13.3.6 Method: caller_name**

Get a name of a caller in the format module.class.method

Arguments:

- `skip`
/ Condition: required / Type: int /

Specifies how many levels of stack to skip while getting caller name. `skip=1` means "who calls me", `skip=2` "who calls my caller" etc.

Returns:

/ Type: str /
 An empty string is returned if skipped levels exceed stack height

13.3.7 Method: load_library

Load native library depend on the calling convention.

Arguments:

- `path`
/ Condition: required / Type: str /
 Library path.
- `is_stdcall`
/ Condition: optional / Type: bool / Default: True /
 Determine if the library's calling convention is stdcall or cdecl.

Returns:

Loaded library object.

13.3.8 Method: is_ascii_or_unicode

Check if the string is ascii or unicode

Arguments: str_check: string for checking codecs: encoding type list

Returns:

/ Type: bool /

True : if checked string is ascii or unicode

False : if checked string is not ascii or unicode

13.4 Class: Job

Imported by:

```
from QConnectBase.utils import Job
```

13.4.1 Method: stop

13.4.2 Method: run

13.5 Class: ResultType

Imported by:

```
from QConnectBase.utils import ResultType
```

Result Types.

13.6 Class: ResponseMessage

Imported by:

```
from QConnectBase.utils import ResponseMessage
```

Response message class

13.6.1 Method: get_json

Convert response message to json

Returns:

Response message in json format

13.6.2 Method: get_data

Get string data result

Returns:

String result

13.6.3 Method: create_from_string

Chapter 14

Appendix

About this package:

Table 14.1: Package setup

Setup parameter	Value
Name	QConnectBase
Version	1.2.1
Date	06.11.2025
Description	Robot Framework test library for TCP, SSH, serial connection
Package URL	robotframework-qconnect-base
Author	Nguyen Huynh Tri Cuong
Email	cuong.nguyenhuyhtri@vn.bosch.com
Language	Programming Language :: Python :: 3
License	License :: OSI Approved :: Apache Software License
OS	Operating System :: OS Independent
Python required	>=3.0
Development status	Development Status :: 4 - Beta
Intended audience	Intended Audience :: Developers
Topic	Topic :: Software Development

Chapter 15

History

1.1.0	07/2022
<i>Initial version</i>	
1.1.5	06/2025
<i>Modified timeout handling including <code>verify_timeout</code> and <code>emergency_timeout</code>.</i>	
<i>Added support for time string parsing and new keywords: <code>set_default_verify_timeout</code>, <code>set_default_emergency_timeout</code>.</i>	
<i>Made <code>search_pattern</code> an optional parameter.</i>	
<i>Improved error handling when disconnecting non-existent connections.</i>	
<i>Fixed issues causing infinite wait on SSH disconnect and other minor bugs.</i>	
<i>Ensured compatibility with Python 3.11.</i>	
<i>Updated log format and improved documentation.</i>	
<i>Applied self-test maintenance and improved connection configuration handling.</i>	
1.1.6	07/2025
<i>Added support item transfer for SSH connection</i>	
1.1.7	08/2025
- Enhanced the input parameter of keyword <code>connect</code>. - Improved logging message and levels. - Improved singleton implementation. - Updated documentation to give more detail about parameters of <code>verify</code> keyword.	
1.1.8	09/2025
- Modified the response of <code>verify</code> keyword to return list of captured strings. - Updated to raise <code>UNKNOWN</code> instead of <code>ERROR</code> when providing invalid <code>connection_type</code>. - Fixed issue that <code>Broken Connection</code> is not detected on <code>Linux</code>.	
1.1.9	10/2025
<i>Improved the logging messages</i>	
1.1.10	10/2025
<i>Changed logic of <code>verify</code> keyword to check the <code>eob_pattern</code> before applying <code>filter_pattern</code></i>	
1.2.0	11/2025
- Updated the default value of the <code>eob_pattern</code> parameter in the <code>verify</code> keyword to <code>None</code>. - Improved the <code>verify</code> keyword to validate all input regex patterns. - Updated the <code>connect</code> keyword to make <code>conn_name</code> a required parameter.	

1.2.1	11/2025
<i>Fixed exception chaining to avoid 'During handling of the above exception' warning in debug log file.</i>	