RobotFramework_DBus

v. 0.1.0

TODO

22.05.2023

Contents

Chapter 1

Introduction

1.1 Introduction

RobotFramework_DBus is a Python extension library for Robot Framework. It is designed to support the automation testing of DBus services. The library provides a set of keywords that enable testing of DBus methods and signals using both synchronous and asynchronous mechanisms.

Built upon the Python *dasbus* library, **RobotFramework_DBus** simplifies the process of interacting with DBus services and facilitates efficient testing of their functionality. It offers a comprehensive set of features to streamline the testing process and ensure the reliability of DBus-based applications.

1.2 Key Features

- Integration with Robot Framework: **RobotFramework_DBus** seamlessly integrates with Robot Framework, allowing you to leverage its powerful test automation capabilities for DBus services.
- DBus Service Automation: With **RobotFramework_DBus**, you can easily automate the testing of DBus services. It provides a range of keywords specifically designed for this purpose.
- Testing DBus Methods and Signals: The library enables you to test both DBus methods and signals. You can validate the behavior of DBus methods and monitor the emission of signals during the testing process.
- Synchronous and Asynchronous Testing: **RobotFramework_DBus** supports both synchronous and asynchronous testing approaches. You can choose the appropriate mechanism based on your testing requirements.

Whether you are testing a single DBus service or a complex system of interconnected services, **RobotFramework_DBus** offers a reliable and efficient solution for DBus automation testing. It empowers you to ensure the quality and robustness of your DBus-based applications.

Chapter 2

Description

RobotFramework_DBus

2.1 Prerequisites

Before using **RobotFramework_DBus**, make sure you have the following libraries installed:

- **pycairo**: This library provides Python bindings for the Cairo graphics library. It is used for rendering graphics and creating visual elements in your applications.
- **PyGObject**: PyGObject is a Python package that provides bindings for the GObject library. It allows you to use GObject-based libraries, such as GTK+, in Python applications.
- **dasbus**: **RobotFramework_DBus** is built upon the *dasbus* library, which is a Pythonic D-Bus library. Make sure you have *dasbus* installed before using **Your Library Name**.
- **pyinstaller**: PyInstaller is a tool used to package Python applications into standalone executables. If you plan to distribute your application as an executable, you will need to have *pyinstaller* installed.

Ensure that the above libraries are installed and properly configured in your Python environment before using **robotframework-dbus**. This will ensure the smooth functioning and compatibility of the library with your system.

2.2 Getting Started

You can checkout all robotframework-dbus sourcecode from the GitHub.

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

```
python setup.py install
```

2.3 Usage

RobotFramework_DBus Library support following keywords for testing connection in RobotFramework.

2.3.1 connect

Use for establishing a connection.

Syntax:

connect `conn_name=[conn_name] namespace=[namespace] object_path=[object path] mode=[test mode] host=[remote host] port=[remote port]`

Arguments:

conn_name: The name or identifier of the connection instance used to interact with the DBus service. This parameter is optional and can be used to uniquely identify a specific connection when multiple connections are established. If not provided, a default connection will be used.
Default is 'default_conn'.

namespace: The namespace of the DBus service. This identifies the specific service or group of services. It is used to differentiate between different service instances. The namespace should be a string that uniquely identifies the service.
E.g. namespace=org.example.HelloWorld

object_path: The object path of the DBus service. This identifies the specific object within the service that the action will be performed on. The object path should be a string that follows the DBus object path naming convention. It typically consists of a hierarchical structure separated by slashes (/).
E.g. object_path=/org/example/HelloWorld

mode: The mode of testing the DBus service. Possible values are 'local' or 'remote'. 'local' indicates testing on the current system, while 'remote' indicates testing on a remote system.
Default is 'local'.

host: The IP address or hostname of the remote system where the DBus agent is running. This parameter is applicable only if 'mode' is set to 'remote'.
Default is 'localhost'.

2.3.2 disconnect

Use for disconnecting from the DBus service by connection name.

Syntax:

disconnect `conn_name`

Arguments:

conn_name: The name or identifier of the connection instance to disconnect from. This parameter is optional and can be used to specify a specific connection to disconnect. If the connection name is 'ALL', all connections will be disconnected.

2.3.3 set signal received handler

Use to set a signal received handler for a specific DBus connection and signal.

Syntax:

set signal received handler `conn_name=[conn_name] signal=[signal name] handler=[keyword to handle signal emitted event]`

Arguments:

conn_name: The name or identifier of the connection instance used to interact with the DBus service. This parameter is optional and can be used to uniquely identify a specific connection when multiple connections are established. If not provided, a default connection will be used.
Default is 'default_conn'.

signal: The name of the DBus signal to be set emitted handler.

handler: The robotframework keyword to handle the received signal. The handler should accept the necessary parameters based on the signal being handled.

2.3.4 unset signal received handler

Use to unset a signal received handler for a specific signal.

Syntax:

unset signal received handler `conn_name=[conn_name] signal=[signal name] handler=[keyword to handle signal emitted event]`

Arguments:

conn_name: The name or identifier of the connection instance used to interact with the DBus service. This parameter is optional and can be used to uniquely identify a specific connection when multiple connections are established. If not provided, a default connection will be used. Default is 'default_conn'.

signal: The name of the DBus signal to be unset emitted handler.

handler: The robotframework keyword which is handling the signal emitted event.

2.3.5 register signal

Use to register a DBus signal or signals to be monitored for a specific connection.

Syntax:

register signal `conn_name=[conn_name] signal=[signal name]`

Arguments:

conn_name: The name or identifier of the connection instance used to interact with the DBus service. This parameter is optional and can be used to uniquely identify a specific connection when multiple connections are established. If not provided, a default connection will be used. Default is 'default_conn'.

signal: The name of the DBus signal(s) to register. It can be a single signal name as a string, or multiple signal names joined by ','. For example: "signal1,signal2,signal3".

2.3.6 call dbus method

Use to call a DBus method with the specified method name and input arguments.

Syntax:

call dbus method `conn_name=[conn_name] signal=[signal name] handler=[keyword to handle signal emitted event]`

Arguments:

conn_name: The name or identifier of the connection instance used to interact with the DBus service. This parameter is optional and can be used to uniquely identify a specific connection when multiple connections are established. If not provided, a default connection will be used. Default is 'default_conn'.

method_name: The name of the DBus method to be called.

args: Input arguments to be passed to the method.

Return value:

Return from called method.

2.3.7 call dbus method with keyword args

Use to call a DBus method with the specified method name and input keyword arguments.

Syntax:

call dbus method with keyword args `conn_name=[conn_name] method_name=[DBus method name] [keyword parameter name 1]=[keyword parameter value 1] [keyword parameter name 2]=[keyword parameter value 2]...`

Arguments:

conn_name: The name or identifier of the connection instance used to interact with the DBus service. This parameter is optional and can be used to uniquely identify a specific connection when multiple connections are established. If not provided, a default connection will be used. Default is 'default_conn'.

method_name: The name of the DBus method to be called.

kwargs: Input keyword arguments to be passed to the method.

Return value:

Return from called method.

2.3.8 wait for signal

Use to wait for a specific DBus signal to be received within a specified timeout period.

Syntax:

wait for signal `conn_name=[conn_name] signal=[signal name] timeout=[timeout]`

Arguments:

conn_name: The name or identifier of the connection instance used to interact with the DBus service. This parameter is optional and can be used to uniquely identify a specific connection when multiple connections are established. If not provided, a default connection will be used. Default is 'default_conn'.

signal: The name of the DBus signal to wait for.

timeout: The maximum time (in seconds) to wait for the signal.

Return value:

The signal payloads.

2.4 Remote testing

After installing the library, you can find the DBus Client Agent at `/opt/rfwaio/python39/install/bin/`.

For remote testing, follow these steps:

1. Start the DBus Agent on the remote system by the following command:
   dbus_client_agent [-h] [–host HOST] [–port PORT]
   The DBus Client Agent supports the following command-line arguments:

   `--host` (str, optional) The host where the agent is running. Default is `0.0.0.0`.
   `--port` (int, optional) The port where the agent is listening. Default is 2507.

2. On the host test PC, using the `connect` keyword with the `remote` mode and specify the correct host using the `host` parameter.
3. Use the other keywords in the same way as local testing.

2.5 Example

Kindly be advised that all the examples presented within this session are designed to interact with the DBus service sample server.py residing in the atest/ directory.

2.5.1 Example 1

Scenario: In this example, I will use the 'wait for signal' keyword from the RobotFramework_DBus library to wait for a DBus signal to be emitted using the synchronized mechanism. This means that the keyword following 'wait for signal' will only be executed after the signal is emitted, and the 'wait for signal' keyword will return a value.

```
*** Settings ***
Library    RobotFramework_DBus.DBusManager

*** Test Cases ***
Hello World
   connect    conn_name=test_dbus
   ...        namespace=org.example.HelloWorld
   ...        mode=local

   ${ret}=    Wait For Signal     conn_name=test_dbus
   ...                            signal=YellowMessage
   ...                            timeout=10

   Log To Console    ${ret}

   Disconnect    test_dbus
```

Explanation: In the aforementioned example, we establish a connection to the DBus service **org.example.HelloWorld** using the **'connect'** keyword and name this connection '**test_dbus**'. Subsequently, the **'Wait for signal'** keyword will wait for the DBus service to emit the **YellowMessage** signal within a 10-second timeframe and return the payloads of this signal immediately upon its emission within the specified timeframe (failing if the timeout is exceeded). The content of the payloads will be printed to the console.

2.5.2 Example 2

Scenario: In Example 1, if we include an additional **Wait for signal** keyword to wait for the **GreenMessage** signal to be emitted after waiting for **YellowMessage**, there is a possibility of missing the occurrence of **GreenMessage** if it is emitted before **YellowMessage**. In such a scenario, we can utilize the **Register Signal** keyword to register **GreenMessage** in the watchlist. This ensures continuous monitoring of the signal starting immediately after executing this keyword and prevents the occurrence of missing the emitted **GreenMessage** event.

```
*** Settings ***
Library    RobotFramework_DBus.DBusManager

*** Test Cases ***
Hello World
   connect    conn_name=test_dbus
   ...        namespace=org.example.HelloWorld
   ...        mode=local

   Register Signal    conn_name=test_dbus    signal=GreenMessage

   ${ret}=    Wait For Signal     conn_name=test_dbus
   ...                            signal=YellowMessage
   ...                            timeout=10
   Log To Console    ${ret}

   ${ret}=    Wait For Signal     conn_name=test_dbus
   ...                            signal=GreenMessage
   ...                            timeout=10
   Log To Console    ${ret}

   ${ret}=    Call Dbus Method    test_dbus    Hello    World
   Disconnect    test_dbus
```

Explanation: In the aforementioned example, we establish a connection to the DBus service **org.example.HelloWorld** using the **'connect'** keyword and name this connection '**test_dbus**'. Subsequently, the **'Wait for signal'** keyword will wait for the DBus service to emit the **YellowMessage** signal within a 10-second timeframe and return the payloads of this signal immediately upon its emission within the specified timeframe (failing if the timeout is exceeded). The content of the payloads will be printed to the console.

2.6 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.

2.7 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.

2.8 Sourcecode Documentation

For investigating sourcecode, please refer to QConnectWinapp Documentation

2.9 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. Connect with me at cuong.nguyenhuynhtri@vn.bosch.com.

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

2.10 About

2.10.1 Maintainers

Nguyen Huynh Tri Cuong

2.10.2 Contributors

Nguyen Huynh Tri Cuong

Thomas Pollerspoeck

2.10.3 3rd Party Licenses

You must mention all 3rd party licenses (e.g. OSS) licenses used by your project here. Example:

Name	License	Type
Apache Felix.	Apache 2.0 License.	Dependency

2.10.4 Used Encryption

Declaration of the usage of any encryption (see BIOS Repository Policy §4.a).

2.10.5 License

Chapter 3

__init__.py

3.1 Class: DBusManager

Imported by:

```
from RobotFramework_DBus.__init__ import DBusManager
```

Class to manage all dbus communications.

Chapter 4

priority_queue.py

4.1 Class: PriorityQueue

Imported by:

```
from RobotFramework_DBus.common.priority_queue import PriorityQueue
```

4.1.1 Method: put

Chapter 5

register_keyword.py

5.1 Class: RegisterKeyword

Imported by:

```
from RobotFramework_DBus.common.register_keyword import RegisterKeyword
```

A class that provides a keyword as a callback function for a DBus signal received.

5.1.1 Method: callback_func

Constructor for RegisterKeyword class.
Arguments:

- `observer`
  / *Condition*: optional / *Type*: tuple / *Default*: None /
  Input arguments to be passed to the callback method.

Returns:
(*no returns*)

Chapter 6

scheduled_job.py

6.1 Class: ScheduledJob

Imported by:

```
from RobotFramework_DBus.common.scheduled_job import ScheduledJob
```

A threaded job that executes a function at a specified interval.

6.1.1 Method: stop

Stop the execution of the job.

Returns:

(*no returns*)

6.1.2 Method: run

Start the job execution loop.

Returns:

(*no returns*)

Chapter 7

thread_safe_dict.py

7.1 Class: ThreadSafeDict

Imported by:

```
from RobotFramework_DBus.common.thread_safe_dict import ThreadSafeDict
```

This class provides a dictionary with thread-safe operations by utilizing locks for synchronized access.

7.1.1 Method: clear

Remove all key-value pairs from the dictionary.

Returns:

(*no returns*)

7.1.2 Method: pop

Remove and return the value associated with a given key.

Arguments:

- `key`
  / *Condition*: required / *Type*: Any /
  The key of dictionary to be pop.

Returns:

- ○
  / *Type*: Any /
  Value of the given key.

7.1.3 Method: popitem

Remove and return an arbitrary key-value pair from the dictionary.

Returns:

(*no returns*)

7.1.4 Method: update

Update the dictionary with key-value pairs from another dictionary.

Arguments:

- `dict`
  / *Condition*: optional / *Type*: dict / *Default*: None /
  Another dictionary-like object to update from.

Returns:
(*no returns*)

Chapter 8

utils.py

8.1 Class: Singleton

Imported by:

```
from RobotFramework_DBus.common.utils import Singleton
```

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

8.2 Class: DictToClass

Imported by:

```
from RobotFramework_DBus.common.utils import DictToClass
```

Class for converting dictionary to class object.

8.2.1 Method: validate

8.3 Class: Utils

Imported by:

```
from RobotFramework_DBus.common.utils import Utils
```

Class to implement utilities for supporting development.

8.3.1 Method: make_unique_token

Generates a unique session token of specified length.

The make_unique_token function generates a unique session token of the specified length. The session token can be used to identify and associate a session with a specific client.

The session token is a string value that is guaranteed to be unique for each invocation of this function. It can be used as a secure identifier to track and manage client sessions within the DBusAgent.

Arguments:

- `length`
  / *Condition*: optional / *Type*: int / *Default*: 16 /
  The length of the session token. Defaults to 16.

Returns:

- `token`
  / *Type*: str /
  A unique token.

8.3.2 Method: get_all_descendant_classes

Get all descendant classes of a class

Arguments:

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

Returns:

/ *Type*: list /
Array of descendant classes.

8.3.3 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.

8.3.4 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

8.3.5 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.

8.3.6 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

Chapter 9

dbus_client_agent.py

9.1 Function: run_agent

Run the DBus Agent with the specified configuration.

Description:

> The run_agent function starts the DBus Agent with the provided configuration. It parses the command-line arguments, such as the host and port options, and then starts the agent on the specified host and port.
>
> The agent listens for incoming requests from clients and manages client sessions. It provides a mechanism to execute requests on corresponding DBus services through the assigned Executor instances.
>
> **Command-line Arguments:** --host (str, optional): The host where the agent is running. Default is '0.0.0.0'. --port (int, optional): The port where the agent is listening. Default is 2507.

9.2 Class: DBusClientExecutor

Imported by:

```
from RobotFramework_DBus.dbus_agent.dbus_client_agent import DBusClientExecutor
```

The DBusClientExecutor class represents an executor responsible for handling client requests on specific DBus services. It receives requests from the DBusAgent and executes them on the corresponding DBus service.

9.2.1 Method: connect

Create a proxy object to DBus object.

Returns:

(*no returns*)

9.2.2 Method: disconnect

Disconnect the DBus proxy from the remote object.

Returns:

(*no returns*)

9.2.3 Method: quit

Quit the DBus client.

Returns:

(*no returns*)

9.2.4 Method: get_monitoring_signal_payloads

Get the payloads of a specific signal.

Arguments:

- `signal`
  / *Condition*: required / *Type*: str /
  The name of the DBus signal to get payloads.

Returns:

- `payloads`
  / *Type*: str /
  The signal's payloads.

9.2.5 Method: add_signal_to_captured_dict

Add a signal and its payloads to the captured dictionary when the signal be emited.

Arguments:

- `signal`
  / *Condition*: required / *Type*: str /
  The name of the DBus signal(s) which has been raised.
- `loop`
  / *Condition*: optional / *Type*: EventLoop / *Default*: None /
  The Event loop which is running to wait for the raised signal.
- `payloads`
  / *Condition*: optional / *Type*: Any / *Default*: "" /
  The payloads of the raised signal.

Returns:

(*no returns*)

9.2.6 Method: register_monitored_signal

Register a DBus signal or signals to be monitored for a specific connection.

Arguments:

- `signal`
  / *Condition*: optional / *Type*: str / *Default*: '' /
  The name of the DBus signal(s) to register. It can be a single signal name as a string, or multiple signal names joined by ','. For example: "signal1,signal2,signal3".

Returns:

(*no returns*)

9.2.7 Method: wait_for_signal

Wait for a specific DBus signal to be received within a specified timeout period.

Arguments:

- `wait_signal`
  / *Condition*: optional / *Type*: str / *Default*: '' /
  The name of the DBus signal to wait for.
- `timeout`
  / *Condition*: optional / *Type*: int / *Default*: 0 /
  The maximum time (in seconds) to wait for the signal.

Returns:

- `payloads`
  / *Type*: str /
  The signal payloads.

9.2.8 Method: call_dbus_method

Call a DBus method with the specified method name and input arguments.
Arguments:

- `method_name`
  / *Condition*: optional / *Type*: str / *Default*: '' /
  The name of the DBus method to be called.
- `args`
  / *Condition*: optional / *Type*: tuple / *Default*: None /
  Input arguments to be passed to the method.

Returns:

- `mtd_ret`
  / *Type*: Any /
  Return from called method.

9.3 Class: DBusClientAgent

Imported by:

```
from RobotFramework_DBus.dbus_agent.dbus_client_agent import DBusClientAgent
```

The DBusClientAgent class acts as a mediator between clients and the corresponding DBus services they request. It manages client connections, session tokens, and assigns the appropriate DBusClientExecutor for each client session.

9.3.1 Method: get_session_token

Generates a unique session token for a client and returns it.

Returns:
/ *Type*: str /

The random and unique token.

9.3.2 Method: initialize_dbus_client

Initializes an DBusClientExecutor instance for a specific client.

An DBusClientExecutor is an object responsible for executing requests from clients on corresponding DBus services. By initializing a separate DBusClientExecutor for each client, the DBusClientAgent ensures that requests from different clients are handled independently.

The client session token is a unique identifier that can be used to associate the DBusClientExecutor with the specific client. This allows the DBusClientAgent to route incoming requests to the correct DBusClientExecutor based on the client session token.

Arguments:

- `session`

  / *Condition*: required / *Type*: str /

  The client's session token.

- `namespace`

  / *Condition*: optional / *Type*: str / *Default*: '' /

  The namespace of the DBus service. This identifies the specific service or group of services. It is used to differentiate between different service instances. The namespace should be a string that uniquely identifies the service.

- `object_path`

  / *Condition*: optional / *Type*: str / *Default*: None /

  The object path of the DBus service. This identifies the specific object within the service that the action will be performed on. The object path should be a string that follows the DBus object path naming convention. It typically consists of a hierarchical structure separated by slashes (/).

Returns:

(*no returns*)

9.3.3 Method: connect

Create a proxy object to DBus service.

Returns:

(*no returns*)

9.3.4 Method: disconnect

Disconnect the DBus proxy from the remote object.

Returns:

(*no returns*)

9.3.5 Method: quit

Quit the DBus client.

Returns:

(*no returns*)

9.3.6 Method: get_monitoring_signal_payloads

Get the payloads of a specific signal.

Arguments:

- `session`
  / *Condition*: required / *Type*: str /
  The client's session token.
- `signal`
  / *Condition*: required / *Type*: str /
  The name of the DBus signal to get payloads.

Returns:

- `payloads`
  / *Type*: str /
  The signal's payloads.

9.3.7 Method: register_monitored_signal

Register a DBus signal or signals to be monitored for a specific connection.

Arguments:

- `session`
  / *Condition*: required / *Type*: str /
  The client's session token.
- `signal`
  / *Condition*: required / *Type*: str /
  The name of the DBus signal(s) to register. It can be a single signal name as a string, or multiple signal names joined by ','. For example: "signal1,signal2,signal3".

Returns:

(*no returns*)

9.3.8 Method: wait_for_signal

Wait for a specific DBus signal to be received within a specified timeout period.

Arguments:

- `session`
  / *Condition*: required / *Type*: str /
  The client's session token.
- `wait_signal`
  / *Condition*: optional / *Type*: str / *Default*: '' /
  The name of the DBus signal to wait for.
- `timeout`
  / *Condition*: optional / *Type*: int / *Default*: 0 /
  The maximum time (in seconds) to wait for the signal.

Returns:

- `payloads`
  / *Type*: str /
  The signal payloads.

9.3.9 Method: call_dbus_method

Call a DBus method with the specified method name and input arguments.

Arguments:

- `session`
  / *Condition*: required / *Type*: str /
  The client's session token.
- `method_name`
  / *Condition*: optional / *Type*: str / *Default*: '' /
  The name of the DBus method to be called.
- `args`
  / *Condition*: optional / *Type*: tuple / *Default*: None /
  Input arguments to be passed to the method.

Returns:

- `ret_obj`
  / *Type*: Any /
  Connection object.

Chapter 10

dbus_client.py

10.1 Class: DBusClient

Imported by:

```
from RobotFramework_DBus.dbus_client import DBusClient
```

A client class for interacting with a specific DBus service.

10.1.1 Method: connect

Create a proxy object to DBus object.

Returns:

(*no returns*)

10.1.2 Method: disconnect

Disconnect the DBus proxy from the remote object.

Returns:

(*no returns*)

10.1.3 Method: quit

Quit the DBus client.

Returns:

(*no returns*)

10.1.4 Method: set_signal_received_handler

Set a signal received handler for a specific signal.

Arguments:

- `signal`
  / *Condition*: required / *Type*: str /
  The name of the DBus signal to handle.
- `handler`
  / *Condition*: required / *Type*: str /
  The keyword to handle the received signal. The handler should accept the necessary parameters based on the signal being handled.

Returns:

(*no returns*)

10.1.5 Method: unset_signal_received_handler

Unset a signal received handler for a specific signal.

Arguments:

- `signal`
  / *Condition*: required / *Type*: str /
  The name of the DBus signal to handle.
- `handle_keyword`
  / *Condition*: optional / *Type*: str / *Type*: None /
  The keyword which is handling for signal emitted event.

Returns:

(*no returns*)

10.1.6 Method: add_signal_to_captured_dict

Add a signal and its payloads to the captured dictionary when the signal be emited.

Arguments:

- `signal`
  / *Condition*: required / *Type*: str /
  The name of the DBus signal(s) which has been raised.
- `loop`
  / *Condition*: optional / *Type*: EventLoop / *Default*: None /
  The Event loop which is running to wait for the raised signal.
- `payloads`
  / *Condition*: optional / *Type*: Any / *Default*: "" /
  The payloads of the raised signal.

Returns:

(*no returns*)

10.1.7 Method: register_monitored_signal

Register a DBus signal or signals to be monitored for a specific connection.

Arguments:

- `signal`
  / *Condition*: optional / *Type*: str / *Default*: '' /
  The name of the DBus signal(s) to register. It can be a single signal name as a string, or multiple signal names joined by ','. For example: "signal1,signal2,signal3".

Returns:

(*no returns*)

10.1.8 Method: wait_for_signal

Wait for a specific DBus signal to be received within a specified timeout period.

Arguments:

- `wait_signal`
  / *Condition*: optional / *Type*: str / *Default*: '' /
  The name of the DBus signal to wait for.
- `timeout`
  / *Condition*: optional / *Type*: int / *Default*: 0 /
  The maximum time (in seconds) to wait for the signal.

Returns:

- `payloads`
  / *Type*: str /
  The signal payloads.

10.1.9 Method: call_dbus_method

Call a DBus method with the specified method name and input arguments.

Arguments:

- `method_name`
  / *Condition*: optional / *Type*: str / *Default*: '' /
  The name of the DBus method to be called.
- `args`
  / *Condition*: optional / *Type*: tuple / *Default*: None /
  Input arguments to be passed to the method.

Returns:

/ *Type*: Any /
Return from called method.

10.1.10 Method: call_dbus_method_with_keyword_args

Call a DBus method with the specified method name and input arguments.

Arguments:

- `method_name`
  / *Condition*: optional / *Type*: str / *Default*: '' /
  The name of the DBus method to be called.
- `args`
  / *Condition*: optional / *Type*: tuple / *Default*: None /
  Input arguments to be passed to the method.

Returns:

- `ret_obj`
  / *Type*: Any /
  Connection object.

Chapter 11

dbus_client_remote.py

11.1 Class: DBusClientRemote

Imported by:

```
from RobotFramework_DBus.dbus_client_remote import DBusClientRemote
```

A client class for interacting with a specific DBus service on a remote machine.

11.1.1 Method: connect

Create a proxy object to DBus object.

Returns:

(*no returns*)

11.1.2 Method: disconnect

Disconnect the DBus proxy from the remote object.

Returns:

(*no returns*)

11.1.3 Method: quit

Quit the DBus client.

Returns:

(*no returns*)

11.1.4 Method: do_signal_check

Checking if the signal was emited.

Arguments:

- `signal`
  / *Condition*: required / *Type*: str /
  The name of the DBus signal to check.
- `call_back_func`
  / *Condition*: required / *Type*: callable /
  The function to be callback when receiving the signal.

Returns:

(*no returns*)

11.1.5 Method: set_signal_received_handler

Set a signal received handler for a specific signal.

Arguments:

- `signal`
  / *Condition*: required / *Type*: str /
  The name of the DBus signal to handle.
- `handler`
  / *Condition*: required / *Type*: str /
  The keyword to handle the received signal. The handler should accept the necessary parameters based on the signal being handled.

Returns:

(*no returns*)

11.1.6 Method: unset_signal_received_handler

Unset a signal received handler for a specific signal.

Arguments:

- `signal`
  / *Condition*: required / *Type*: str /
  The name of the DBus signal to handle.

Returns:

(*no returns*)

11.1.7 Method: register_monitored_signal

Register a DBus signal or signals to be monitored for a specific connection.

Arguments:

- `signal`
  / *Condition*: optional / *Type*: str / *Default*: '' /
  The name of the DBus signal(s) to register. It can be a single signal name as a string, or multiple signal names joined by ','. For example: "signal1,signal2,signal3".

Returns:

(*no returns*)

11.1.8 Method: wait_for_signal

Wait for a specific DBus signal to be received within a specified timeout period.

Arguments:

- `wait_signal`
  / *Condition*: optional / *Type*: str / *Default*: '' /
  The name of the DBus signal to wait for.
- `timeout`
  / *Condition*: optional / *Type*: int / *Default*: 0 /
  The maximum time (in seconds) to wait for the signal.

Returns:

- `payloads`
  / *Type*: str /
  The signal payloads.

11.1.9 Method: call_dbus_method

Call a DBus method with the specified method name and input arguments.

Arguments:

- `method_name`
  / *Condition*: optional / *Type*: str / *Default*: '' /
  The name of the DBus method to be called.
- `args`
  / *Condition*: optional / *Type*: tuple / *Default*: None /
  Input arguments to be passed to the method.

Returns:

- `ret_obj`
  / *Type*: Any /
  Connection object.

Chapter 12

dbus_manager.py

12.1 Class: DBusManager

Imported by:

```
from RobotFramework_DBus.dbus_manager import DBusManager
```

Class to manage all DBus connections.

12.1.1 Method: quit

Quit connection manager.

Returns:

(*no returns*)

12.1.2 Method: add_connection

Add a connection to managed dictionary.

Arguments:

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

Returns:

(*no returns*)

12.1.3 Method: remove_connection

Remove a connection by name.

Arguments:

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

Returns:

(*no returns*)

12.1.4 Method: get_connection_by_name

Get an exist connection by name.

Arguments:

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

Returns:

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

12.1.5 Keyword: disconnect

Keyword for disconnecting a connection by name.

Arguments:

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

Returns:

(*no returns*)

12.1.6 Keyword: connect

Keyword used to establish a DBus connection.

Arguments:

- `conn_name`
  / *Condition*: optional / *Type*: str / *Default*: 'default_conn' /
  The name or identifier of the connection instance used to interact with the DBus service. This parameter is optional and can be used to uniquely identify a specific connection when multiple connections are established. If not provided, a default connection will be used.
- `namespace`
  / *Condition*: optional / *Type*: str / *Default*: '' /
  The namespace of the DBus service. This identifies the specific service or group of services. It is used to differentiate between different service instances. The namespace should be a string that uniquely identifies the service.
- `object_path`
  / *Condition*: optional / *Type*: str / *Default*: None /
  The object path of the DBus service. This identifies the specific object within the service that the action will be performed on. The object path should be a string that follows the DBus object path naming convention. It typically consists of a hierarchical structure separated by slashes (/).
- `mode`
  / *Condition*: optional / *Type*: str / *Default*: 'local' /
  The mode of testing the DBus service. Possible values are 'local' or 'remote'. 'local' indicates testing on the current system, while 'remote' indicates testing on a remote system.

- `host`
  / *Condition*: optional / *Type*: str / *Default*: 'localhost' /
  The IP address or hostname of the remote system where the DBus agent is running.
  This parameter is applicable only if mode is set to 'remote'.
- `port`
  / *Condition*: optional / *Type*: int / *Default*: 2507 /
  The port number on which the DBus agent is listening on the remote system.
  This parameter is applicable only if mode is set to 'remote'.

Returns:
(*no returns*)

12.1.7 Keyword: set_signal_received_handler

Keyword used to set a signal received handler for a specific DBus connection and signal.
Arguments:

- `conn_name`
  / *Condition*: optional / *Type*: str / *Default*: 'default_conn' /
  The name of the DBus connection.
- `signal`
  / *Condition*: optional / *Type*: str / *Default*: '' /
  The name of the DBus signal to handle.
- `handler`
  / *Condition*: optional / *Type*: str / *Default*: None /
  The keyword to handle the received signal. The handler should accept the necessary parameters based on the signal being handled.

Returns:
(*no returns*)

12.1.8 Keyword: unset_signal_received_handler

Keyword used to set a signal received handler for a specific DBus connection and signal.
Arguments:

- `conn_name`
  / *Condition*: optional / *Type*: str / *Default*: 'default_conn' /
  The name of the DBus connection.
- `signal`
  / *Condition*: optional / *Type*: str / *Default*: '' /
  The name of the DBus signal to handle.
- `handler`
  / *Condition*: optional / *Type*: str / *Default*: None /
  The robotframework keyword which is handling the signal emitted event.

Returns:
(*no returns*)

12.1.9 Keyword: register_signal

Keyword used to register a DBus signal or signals to be monitored for a specific connection.

Arguments:

- `conn_name`
  / *Condition*: optional / *Type*: str / *Default*: 'default_conn' /
  The name of the DBus connection.
- `signal`
  / *Condition*: optional / *Type*: str / *Default*: '' /
  The name of the DBus signal(s) to register. It can be a single signal name as a string, or multiple signal names joined by ','. For example: "signal1,signal2,signal3".

Returns:

(*no returns*)

12.1.10 Keyword: call_dbus_method

Keyword used to call a DBus method with the specified method name and input arguments.

Arguments:

- `conn_name`
  / *Condition*: optional / *Type*: str / *Default*: 'default_conn' /
  The name of the DBus connection.
- `method_name`
  / *Condition*: optional / *Type*: str / *Default*: '' /
  The name of the DBus method to be called.
- `args`
  / *Condition*: optional / *Type*: tuple / *Default*: None /
  Input arguments to be passed to the method.

Returns:

- `ret_obj`
  / *Type*: Any /
  Return from called method.

12.1.11 Keyword: call_dbus_method_with_keyword_args

Keyword used to call a DBus method with the specified method name and input keyword arguments.

Arguments:

- `conn_name`
  / *Condition*: optional / *Type*: str / *Default*: 'default_conn' /
  The name of the DBus connection.
- `method_name`
  / *Condition*: optional / *Type*: str / *Default*: '' /
  The name of the DBus method to be called.
- `kwargs`
  / *Condition*: optional / *Type*: dict / *Default*: None /
  Input keyword arguments to be passed to the method.

Returns:

/ *Type*: Any /
Return from called method.

12.1.12 Keyword: wait_for_signal

Keyword used to wait for a specific DBus signal to be received within a specified timeout period.

Arguments:

- `conn_name`
  / *Condition*: optional / *Type*: str / *Default*: 'default_conn' /
  The name of the DBus connection.
- `signal`
  / *Condition*: optional / *Type*: str / *Default*: '' /
  The name of the DBus signal to wait for.
- `timeout`
  / *Condition*: optional / *Type*: int / *Default*: 0 /
  The maximum time (in seconds) to wait for the signal.

Returns:

- `payloads`
  / *Type*: str /
  The signal payloads.

Chapter 13

Appendix

About this package:

Table 13.1: Package setup

Setup parameter	Value
Name	RobotFramework_DBus
Version	0.1.0
Date	22.05.2023
Description	TODO
Package URL	robotframework-dbus
Author	TODO
Email	TODO
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 14

History

0.1.0	05/2023
Initial version	

RobotFramework_DBus.pdf
Created at 06.06.2023 - 19:38:31
by GenPackageDoc v. 0.39.0