RobotFramework_DoIP

v. 0.1.2

Hua Van Thong

08.04.2024

Contents

Chapter 1

Introduction

1.1 Overview

RobotFramework_DoIP is a Robot Framework library specifically designed for interacting with Electronic Control Units (ECUs) using the Diagnostics over Internet Protocol (DoIP).

At its core, DoIP serves as a communication bridge between external diagnostic tools and a vehicle's ECUs. This library, RobotFrameworkDoIP, provides a set of keywords that enable users to perform diagnostic operations and engage with ECUs, facilitating automated testing processes and interaction with vehicles through the DoIP protocol.

The **RobotFramework_DoIP** sources can be found in repository **robotframework-doip**: DoIP

1.2 Abbreviations

Table 1.1: Abbreviation

Abbreviation / Acronym	**Description**
ARP	Address Resolution Protocol
DHCP	Diagnostic Host Configuration Protocol
EID	Entity identifier
GID	Group identifier
ICMP	Internet Control Message Protocol
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
TCP	Transmission Control Protocol
TCP/IP	A family of communication protocols used in computer networks
VIN	Vehicle Identification Number
UDP	User Datagram Protocol

1.3 Terms and definitions

1.3.1 Diagnostic Power Mode

Abstract vehicle internal power supply state, which affects the diagnostic capabilities of all servers on the in-vehicle networks and which identifies the state of all servers of all gateway sub-networks that allow diagnostic communication

1.3.2 Transport protocol

The important thing here is that DoIP does not represent a diagnostic protocol according to ISO 13400 but rather an expanded transport protocol. This means that the transmission of diagnostic packets is defined in DoIP, but the contained diagnostic services continue to be specified and described by diagnostic protocols such as KWP2000 and UDS.

1.3.3 Diagnostics tester

Same as for diagnostics with classic bus systems, a diagnostic tester enables the sending of diagnostic requests. Testers can take the form of external devices, such as in repair shops, or on-board testers in the vehicle. The receiving ECU must, in turn, process the diagnostic requests and return an associated diagnostic response to the tester. However, this requires that DoIP as well as underlying layers be implemented in each directly diagnosable ECU.

1.3.4 Diagnostics gateway

So that a separate implementation is not needed for each ECU, DoIP allows the use of diagnostic gateways. Thus, all ECUs of a vehicle that are connected via a classic bus system or network can be made available in principle. The gateway assumes the role of the intermediary. Requests of the tester are forwarded to internal networks so that a desired ECU can receive and process them. As soon as a response from the requested ECU is available, the gateway routes this back to the tester.

1.3.5 Logical addresses

Address identifying a diagnostic application layer entity.

A diagnostic gateway always requires two pieces of information in order to forward diagnostic requests and responses. First, it needs a logical address that uniquely identifies the ECU to be diagnosed in the vehicle. Second, the gateway must know which messages on the respective bus system or network will be used to send diagnostic requests and to receive diagnostic responses. Both pieces of information must be available for an ECU for it to be accessible via the gateway.

Chapter 2

Description

2.1 Background

Modern cars are becoming computers on wheels. They can use up to 70 or more Electronic Control Units (ECU) for performing various critical functions. These ECUs are embedded systems that maintain and oversee critical functions ranging from fuel injection, cabin room temperature to brakes and suspensions. These devices digest data from local sensors and perform calibrations to keep everything in order.

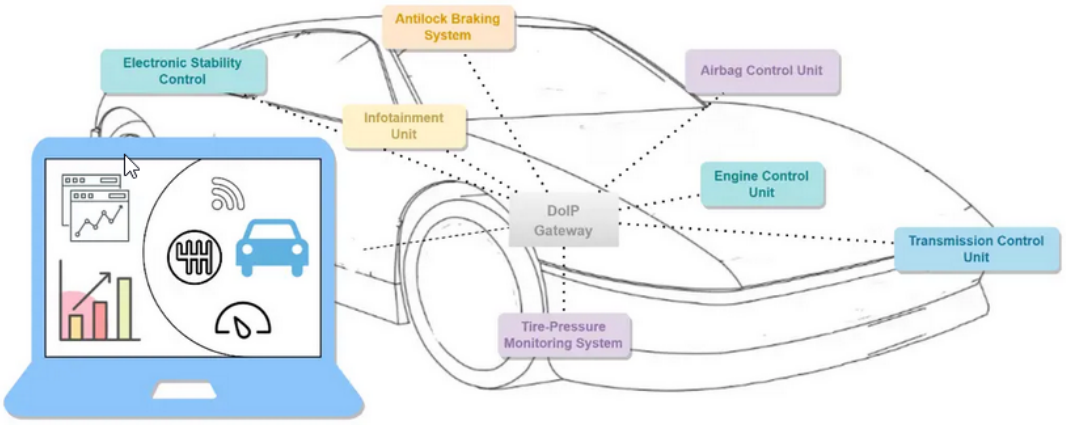

Figure 2.1: ECUs in modern cars

In the automotive industry, "vehicle diagnostics" refers to an inspection of the vehicle to identify any faults and guarantee the smooth operation of all the mechanical, software, and hardware components. These components are usually the ECUs in modern cars.

Typically, a manual vehicle diagnosis is carried out by connecting test equipment to the car's physical ports. These on-site vehicle diagnostic methods are not always a practical choice.

Moving forward, the Original Equipment Manufacturers (OEM) began to give some expensive cars the ability to get over-the-air diagnostics through the network connection. What began with a brand-specific method soon became a standard that we now know as Diagnostics over IP (DoIP).

Nowadays, most of the popular automobile brands such as BMW, Porsche and Ferrari have equipment that can leverage DoIP.

2.2 System Overview

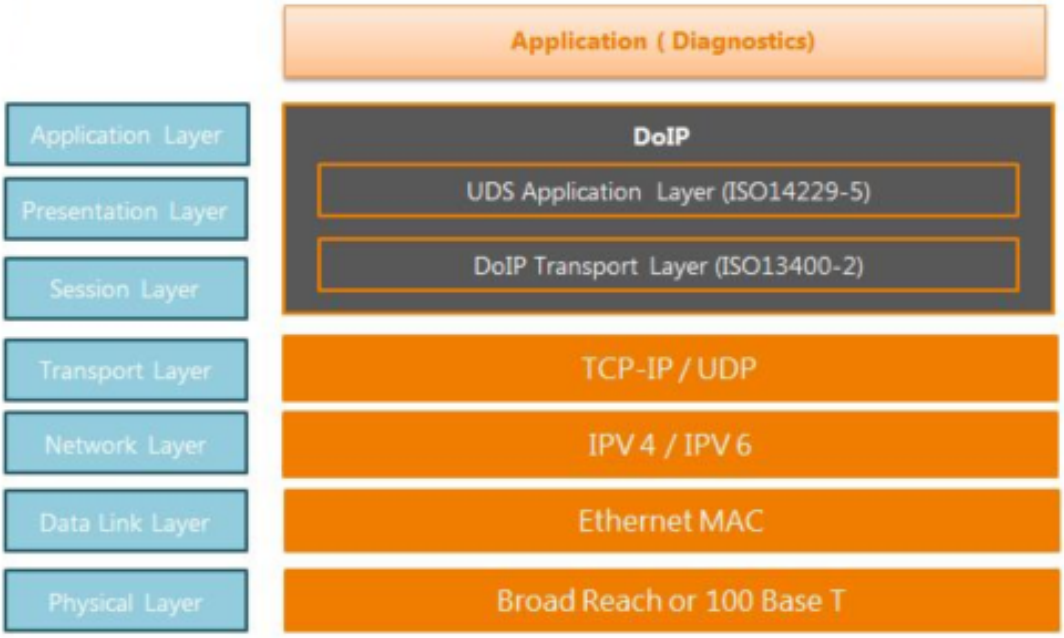

Figure 2.2: System overview

- The DoIP (Diagnostics over Internet Protocol) protocol is a standard for vehicle diagnostics that allows communication between diagnostic devices and electronic control units over Ethernet networks.
- DoIP is a standardized diagnostic transport protocol according to ISO 13400.
  - DoIP Transport Layer (ISO 13400-2) is equipped with features to establish and maintain connection between external tester device and DoIP gateway inside the vehicle.
  - UDS application layer (ISO 14229-5) is the application profile that implements UDS on IP.
- The overall goal of the protocol is to encapsulate diagnostics messages of protocol standards like Unified Diagnostic Services (UDS) and route them to and from the ECU.
- The DoIP gateway or server can be a part of the ECU. A vehicle can have multiple DoIP entities and multiple testing devices and ECUs can route their traffic via a single DoIP entity.
- DoIP uses both User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) for specific phases of the underlying layer. The initial announcement and identification messages are over UDP, after which the communication switches over to TCP.

2.3 DoIP application scenarios

This section will provide some features along with examples:

- Vehicle identification and announcement: Is necessary to detect who is participating in the DoIP communication.
- Request diagnostic message: Request for diagnostic information, which is crucial for diagnosing vehicle issues and ensuring effective communication within the DoIP network.
- Routing Activation: Allows that single Diagnostic Message pathes are activated or not to treat different protocols different (like UDS and OBD) and to also treat single testers different.
- Node information: Provides general information of the single DoIP entity. Usually used by the testers to get the current DoIP protocol relevant information from the single DoIPEntities.
- Alive mechanism: Is used to maintain different tester connections.

2.3.1 Example of Diagnostic Process

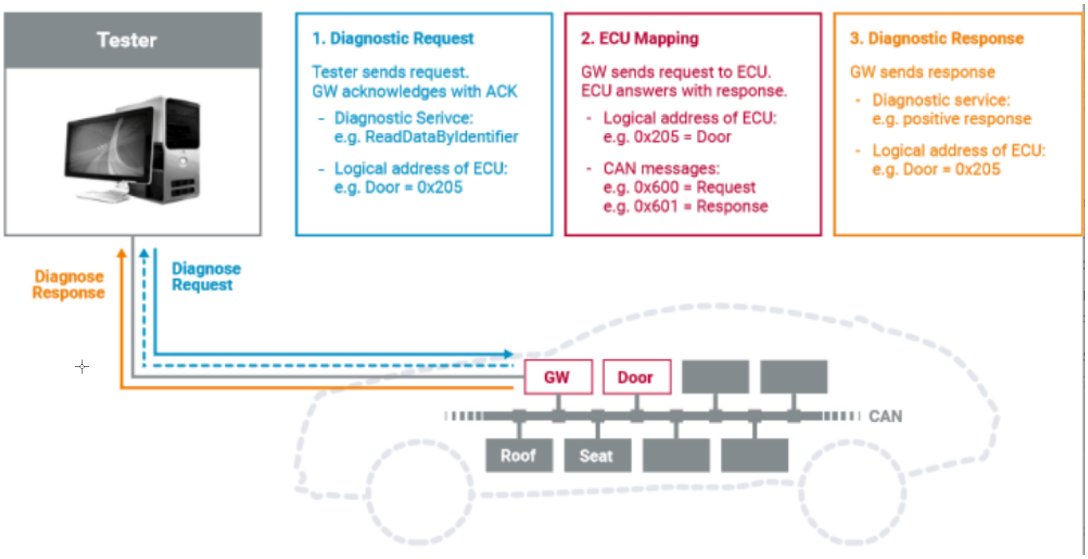

Figure 2.3: System test for diagnostic process

In this example, we will learn how to

```
*** Settings ***
Library    RobotFramework_DoIP

*** Test Cases ***
Test Establish an DoIP connection
    # Establish an connection to ecu ip address and ecu logical address
    Connect To ECU     192.168.108.1      205
    Send Diagnostic Message     600
    ${res}= Receive Diagnostic Message
    Log To Console    ${resp}
    Disconnect

Test Establish an DoIP connection between target tester and ECU
    # Establish an connection to ecu ip address and ecu logical address
    Connect To ECU     192.168.108.1      205       client_ip_address=192.168.108.10    ↩
    ↪ client_logical_address=3584
    Send Diagnostic Message     600
    ${res}= Receive Diagnostic Message
    Log To Console    ${resp}
    Disconnect
```

2.3.2 Example of Vehicle Identification

In a DoIP communication scenario, it is essential to detect and identify the participating vehicles. This allows for effective communication management and ensures that diagnostic messages are appropriately routed to the intended recipients. Vehicle identification helps in monitoring and managing the overall communication network, facilitating seamless interaction between diagnostic tools and vehicle ECUs.

```
*** Settings ***
Library     RobotFramework_DoIP

*** Test Cases ***
Test Request Vehicle Identification
    Connect To ECU      192.168.108.1       205
    Request Vehicle Identification
```

Chapter 3

The Ecu Simulator

This chapter provides a detailed explanation of the utilization of the ECU simulator through DoIP base on doipclient library. It serves for development or testing scenarios where a physical device is not available.
The ECU simulator is designed to receive messages and respond accordingly to the following types of messages:

- Alive Check Request
- Diagnostic Power Mode Request
- Doip Entity Status Request
- Routing Activation Request
- Vehicle Identification Request

3.1 Initialize

This function sets up an instance of an ECU, initializes its attributes with default values, and includes placeholders for various properties that can be customized based on specific requirements.

```
def __init__(self, ecu_type, ip_address, tcp_port, udp_port):
    # Initialize ECU attributes with default values
    self.ecu_type = ecu_type
    self.ip_address = ip_address
    self.tcp_port = tcp_port
    self.udp_port = udp_port
    self.tcp_socket = None
    self.udp_socket = None
    # Set default values for various ECU properties
    # These values might be placeholders and can be updated based on your actual ↩
↪ requirements
    self._ecu_logical_address = 3584
    self._client_logical_address = 3584
    self._logical_address = 55
    self._response_code = doip_message.RoutingActivationResponse.ResponseCode.Success
    self._diagnostic_power_mode = ↩
↪ doip_message.DiagnosticPowerModeResponse.DiagnosticPowerMode.Ready
    self._node_type = 1
    self._max_concurrent_sockets = 16
    self._currently_open_sockets = 1
    self._max_data_size = None
    self._vin = '19676527011956855057'
    self._eid = b'11111'
    self._gid = b'222222'
    self._further_action_required = ↩
↪ doip_message.VehicleIdentificationResponse.FurtherActionCodes.NoFurtherActionRequired
    self._vin_sync_status = ↩
↪ doip_message.VehicleIdentificationResponse.SynchronizationStatusCodes.Synchronized
```

3.2 Start

This method is responsible for initializing and setting up TCP and UDP sockets, binding them to specific IP addresses and ports, and then starting separate threads to handle the communication on these sockets concurrently.

```
def start(self):
    # Create TCP socket
    self.tcp_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    self.tcp_socket.bind((self.ip_address, self.tcp_port))
    self.tcp_socket.listen(5)

    # Create UDP socket
    self.udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    self.udp_socket.bind((self.ip_address, self.udp_port))

    # Start listening on separate threads
    tcp_thread = threading.Thread(target=self.listen_tcp)
    udp_thread = threading.Thread(target=self.listen_udp)

    tcp_thread.start()
    udp_thread.start()
```

Explanation:

1. TCP Socket Setup
   - A TCP socket is created using the socket module with the `socket.AF_INET` family (IPv4) and `socket.SOCK_STREAM` type (TCP).
   - The TCP socket is bound to the specified IP address `self.ip_address` and TCP port `self.tcp_port` .
   - The TCP socket is set to listen for incoming connections with a backlog of 5 connections.
2. UDP Socket Setup
   - A UDP socket is created using the same socket module with the `socket.AF_INET` family (IPv4) and `socket.SOCK_DGRAM type` (UDP).
   - The UDP socket is bound to the specified IP address `self.ip_address` and UDP port `self.udp_port` .
3. Thread Creation
   - Two separate threads `tcp_thread and udp_thread` are created using the threading module.
   - The target parameter of each thread is set to point to specific methods `self.listen_tcp` and `self.listen_udp` , suggesting that these methods likely contain the logic for handling TCP and UDP communication.
4. Thread Start
   - Both threads are started concurrently using the start method, allowing the ECU to handle TCP and UDP communication simultaneously.

3.3 Example

We have provided an example demonstrating the usage of the ECU simulator in the file located at `test_ecu_simulator.py`

```
if __name__ == "__main__":
    # Create and start instances of different ECUs using the factory pattern and abstract class
    factory = ECUFactory()

    positive_ecu = factory.create_ecu(ECUType.POSITIVE_ECU, POSITIVE_ECU_IP, POSITIVE_TCP_PORT, POSITIVE_UDP_PORT)
    negative_ecu = factory.create_ecu(ECUType.NEGATIVE_ECU, NEGATIVE_ECU_IP, NEGATIVE_TCP_PORT, NEGATIVE_UDP_PORT)
    # Start positive and negative ECUs
    positive_ecu.start()
    negative_ecu.start()
```

In the given example, an instance of the ECU is created in `ecu_simulator.py` by specifying the ECU's IP address, TCP port, and UDP port. Subsequently, the start method is invoked to initiate its operation.

Output:

```
TCP Server 172.17.0.5 listening on port 13400
UDP Server 172.17.0.5 listening on port 13400
TCP Server 172.17.0.5 listening on port 12346
UDP Server 172.17.0.5 listening on port 12347
```

Now you can execute the test by running the file located at `test_ecu_simulator.py`

```
def test_positive_ecu_simulator():
    try:
        ip = '172.17.0.5'
        ecu_logical_address = 57344

        # Create a DoIPClient instance for positive ECU simulator
        doip = DoIPClient(ip, ecu_logical_address, activation_type=None)

        # Test various interactions
        print(doip.request_diagnostic_power_mode())
        print(doip.request_entity_status())
        print(doip.request_alive_check())
        print(doip.request_activation(1))
        print(doip.get_entity())
        print(doip.request_vehicle_identification(vin="1" * 17))
        print(doip.request_vehicle_identification(eid=b"1" * 6))

    except Exception as e:
        print(f"Error during positive ECU simulation: {e}")
```

Output:

```
# Diagnostic power mode response
DiagnosticPowerModeResponse (0x4004): { diagnostic_power_mode : DiagnosticPowerMode.Ready }

# Entity status response
EntityStatusResponse (0x4002): { node_type : 1, max_concurrent_sockets : 16, currently_open_sockets : 1, max_data_size : None }

# Alive check response
AliveCheckResponse (0x8): { source_address : 3584 }

# Routing activation response
RoutingActivationResponse (0x6): { client_logical_address : 3584, logical_address : 55, response_code : ResponseCode.Success, reserved : 0, vm_specific : None }

# Get entity response
(('172.17.0.5', 13400), VehicleIdentificationResponse(b'19676527011956855', 3584, b'11111\x00', b'222222', 0, 0))

# Vehicle identification response
VehicleIdentificationResponse (0x4): { vin: "19676527011956855", logical_address : 3584, eid : b'11111\x00', gid : b'222222', further_action_required : FurtherActionCodes.NoFurtherActionRequired, vin_sync_status : SynchronizationStatusCodes.Synchronized }
VehicleIdentificationResponse (0x4): { vin: "19676527011956855", logical_address : 3584, eid : b'11111\x00', gid : b'222222', further_action_required : FurtherActionCodes.NoFurtherActionRequired, vin_sync_status : SynchronizationStatusCodes.Synchronized }
```

Chapter 4

DoipKeywords.py

4.1 Class: DoipKeywords

Imported by:

```
from RobotFramework_DoIP.DoipKeywords import DoipKeywords
```

4.1.1 Method: connect_to_ecu

Description:

Establishing a DoIP connection to an (ECU) within the context of automotive communication.

Parameters:

- **param `ecu_ip_address` (required): The IP address of the ECU to establish a connection. This shoul**
  address like "192.168.1.1" or an IPv6 address like "2001:db8::".
- type `ecu_ip_address`: str
- param `ecu_logical_address` (required): The logical address of the ECU.
- type `ecu_logical_address`: any
- param `tcp_port` (optional): The TCP port used for unsecured data communication (default is **TCP_DATA_UNSECURED**).
- type `tcp_port`: int
- param `udp_port` (optional): The UDP port used for ECU discovery (default is **UDP_DISCOVERY**).
- type `udp_port`: int
- param `activation_type` (optional): The type of activation, which can be the default value (ActivationTypeDefault) or a specific value based on application-specific settings.
- type `activation_type`: RoutingActivationRequest.ActivationType,
- param `protocol_version` (optional): The version of the protocol used for the connection (default is 0x02).
- type `protocol_version`: int
- **param `client_logical_address` (optional): The logical address that this DoIP client will use to ide**
  this should be 0x0E00 to 0x0FFF. Can typically be left as default.
- type `client_logical_address`: int
- **param `client_ip_address` (optional): If specified, attempts to bind to this IP as the source for both**
  Useful if you have multiple network adapters. Can be an IPv4 or IPv6 address just like ecu_ip_address, though the type should match.
- type `client_ip_address`: str
- **param `use_secure` (optional): Enables TLS. If set to True, a default SSL context is used. For more**
  SSL context can be passed directly. Untested. Should be combined with changing tcp_port to 3496.

- type `use_secure`: Union[bool,ssl.SSLContext]
- param `auto_reconnect_tcp` (optional): Attempt to automatically reconnect TCP sockets that were closed by peer
- type `auto_reconnect_tcp`: bool

Return:

None

Exception:

raises ConnectionError: Failed to establish a DoIP connection

Usage:

Explicitly specifies all establishing a connection

- Connect To ECU | 172.17.0.111 | 1863 |
- Connect To ECU | 172.17.0.111 | 1863 | client_ip_address=172.17.0.5 | client_logical_address=1895 |
- Connect To ECU | 172.17.0.111 | 1863 | client_ip_address=172.17.0.5 | client_logical_address=1895 | activation_type=0 |

4.1.2 Method: send_diagnostic_message

Description:

Send a raw diagnostic payload (ie: UDS) to the ECU.

Parameters:

- param `diagnostic_payload`: UDS payload to transmit to the ECU
- type `diagnostic_payload`: string
- param `timeout`: send diagnostic time out (default: A_PROCESSING_TIME)
- type `timeout`: int (s)

Return:

None

Exception:

raises ConnectionRefusedError: DoIP connection attempt failed raises IOError: DoIP negative acknowledgement received

Usage:

Explicitly specifies all diagnostic message properties

- Send Diagnostic Message | 1040 |
- Send Diagnostic Message | 1040 | timeout=10 |

4.1.3 Method: receive_diagnostic_message

Description:

Receive a raw diagnostic payload (ie: UDS) from the ECU.

Parameters:

- param `timeout`: time waiting diagnostic message (default: None)
- type `timeout`: int (s)

Return:

None

Exception:

raises ConnectionRefusedError: DoIP connection attempt failed raises IOError: DoIP negative acknowledgement received

Usage:

Explicitly specifies all diagnostic message properties

- Receive Diagnostic Message |
- Receive Diagnostic Message | timeout=10 |

4.1.4 Method: reconnect_to_ecu

Description:

Attempts to re-establish the connection. Useful after an ECU reset

Parameters:

- param `close_delay`: Time to wait between closing and re-opening socket (default: **A_PROCESSING_TIME**)
- type `close_delay`: int (s)

Return: None

Exception:

raises ConnectionRefusedError: DoIP connection attempt failed

Usage:

Explicitly specifies all diagnostic message properties

- Reconnect To Ecu |
- Reconnect To Ecu | close_delay=10 |

4.1.5 Method: disconnect

Description:

Close the DoIP client

Parameters:

None

Return:

None

Exception:

raises ConnectionRefusedError: DoIP connection attempt failed raises ConnectionAbortedError: close DoIP connection aborted

Usage:

Explicitly specifies all diagnostic message properties

- Disconnect

4.1.6 Method: await_vehicle_announcement

Description:

When an ECU first turns on, it's supposed to broadcast a Vehicle Announcement Message over UDP 3 times to assist DoIP clients in determining ECU IP's and Logical Addresses. Will use an IPv4 socket by default, though this can be overridden with the ipv6 parameter.

Parameters:

- param `udp_port`: The UDP port to listen on. Per the spec this should be 13400, but some VM's use a custom
- one.
- type `udp_port`: int, optional
- param `timeout`: Maximum amount of time to wait for message
- type `timeout`: float, optional
- param `ipv6`: Bool forcing IPV6 socket instead of IPV4 socket
- type `ipv6`: bool, optional
- **param `source_interface`: Interface name (like "eth0") to bind to for use with IPv6. Defaults to No** will use the default interface (which may not be the one connected to the ECU). Does nothing for IPv4, which will bind to all interfaces uses INADDR_ANY.
- type `source_interface`: str, optional

Return:

- return: IP Address of ECU and VehicleAnnouncementMessage object
- rtype: tuple

Exception:

raises TimeoutError: If vehicle announcement not received in time

Usage:

Explicitly specifies all diagnostic message properties

- Await Vehicle Annoucement
- Await Vehicle Annoucement | timeout=10

4.1.7 Method: get_entity

Description:

Sends a VehicleIdentificationRequest and awaits a VehicleIdentificationResponse from the ECU, either with a specified VIN, EIN, or nothing. Equivalent to the request_vehicle_identification() method but can be called without instantiation

Parameters:

- param `udp_port`: The UDP port to listen on. Per the spec this should be 13400, but some VM's use a custom
- one.
- type `udp_port`: int, optional
- param `timeout`: Maximum amount of time to wait for message
- type `timeout`: float, optional
- param `ipv6`: Bool forcing IPV6 socket instead of IPV4 socket
- type `ipv6`: bool, optional
- **param `source_interface`: Interface name (like "eth0") to bind to for use with IPv6. Defaults to No** will use the default interface (which may not be the one connected to the ECU). Does nothing for IPv4, which will bind to all interfaces uses INADDR_ANY.
- type `source_interface`: str, optional

Return:

- return: IP Address of ECU and VehicleAnnouncementMessage object
- rtype: tuple

Exception:

raises TimeoutError: If vehicle announcement not received in time

Usage:

- Get Entity |
- Get Entity | ecu_ip_address=172.17.0.111 |
- Get Entity | ecu_ip_address=172.17.0.111 | protocol_version=0x02

4.1.8 Method: request_entity_status

Description:

Request that the ECU send a DoIP Entity Status Response

Parameters:

None

Return:

None

Exception:

None

Usage:

- Request Entity Status

4.1.9 Method: request_vehicle_identification

Description:

Sends a VehicleIdentificationRequest and awaits a VehicleIdentificationResponse from the ECU, either with a specified VIN, EIN, or nothing

Parameters:

param eid EID of the Vehicle
type eid bytes, optional
param vin VIN of the Vehicle
type vin str, optional

Return:

None

Exception:

None

Usage:

- Request Vehicle Identification
- Request Vehicle Identification | eid=0x123456789abc
- Request Vehicle Identification | vin=0x123456789abc

4.1.10 Method: request_alive_check

Description:

Request that the ECU send an alive check response

Parameters:

None

Return:

None

Exception:

None

Usage:

- Request Vehicle Identification
- Request Vehicle Identification | eid=0x123456789abc
- Request Vehicle Identification | vin=0x123456789abc

4.1.11 Method: request_activation

Description:

Requests a given activation type from the ECU for this connection using payload type 0x0005

Parameters:

- **param `activation_type` (required): The type of activation to request - see Table 47 ("Routing** activation request activation types") of ISO-13400, but should generally be 0 (default) or 1 (regulatory diagnostics)
- type `activation_type`: RoutingActivationRequest.ActivationType
- param `vm_specific` (optional): 4 byte long int
- type `vm_specific`: int, optional
- **param `disable_retry`: Disables retry regardless of auto_reconnect_tcp flag. This is used by activatio** requests during connect/reconnect.
- type `disable_retry`: bool, optional

Return:

None

Exception:

None

Usage:

- Request Routing Activation | ${0x02}
- Request Routing Activation | vm_specific=
- Request Routing Activation | vin=0x123456789abc

4.1.12 Method: request_diagnostic_power_mode

Description:

Request that the ECU send a Diagnostic Power Mode response

Parameters:

None

Return:

None

Exception:

None

Usage:

- Request Diagnostic Power Mode

Chapter 5

RobotFramework_DoIP.py

5.1 Function: get_version

5.2 Function: get_version_date

Chapter 6

__init__.py

6.1 Class: RobotFramework_DoIP

Imported by:

```
from RobotFramework_DoIP.__init__ import RobotFramework_DoIP
```

RobotFrameworkDoIP is a Robot Framework library aimed to provide DoIP protocol for diagnostic message.

Chapter 7

Appendix

About this package:

Table 7.1: Package setup

Setup parameter	Value
Name	RobotFramework_DoIP
Version	0.1.2
Date	08.04.2024
Description	RobotFramework for DoIP Client
Package URL	robotframework-doip
Author	Hua Van Thong
Email	thong.huavan@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 8

History

0.1.0	09/2023
Initial version	
0.1.1	12/2023
Add ecu simulator to use for self test	
0.1.2	4/2024
Update the documentation for DoIP	

RobotFramework_DoIP.pdf
Created at 25.03.2024 - 18:31:19
by GenPackageDoc v. 0.41.1