py-uds

Release 0.3.0

Maciej Dąbrowski

Mar 12, 2024

CONTENTS

CHAPTER

ONE

INSTALLATION

1.1 Released versions

It is **recommended to use released versions** as they were fully tested and are considered as stable. UDS package is distributed via PyPI. All versions and other distribution related details can be found on https://pypi.org/project/py-uds/ page.

- To install the package, run the following command in your command line interface:

```
pip install py-uds
```

- To update your installation to the newest version of the package, run the following command:

```
pip install -U py-uds
```

1.2 Development version

For people who likes risk, but looking for features that are currently under the development, the following command will help with installing the development version of the package that is currently stored on *main* branch of github repository:

```
pip install git+https://github.com/mdabrowski1990/uds.git@main
```

> **Warning:** Use this code on your own responsibility.

CHAPTER

TWO

USER GUIDE

2.1 Diagnostic Messages

Implementation related to diagnostic messages is located in `uds.message` sub-package.

2.1.1 UDS Message Implementation

***Diagnostic messages* implementation is divided into two parts:**

- *UDS Message* - storage for a new diagnostic message definition
- *UDS Message Record* - storage for historic information of a diagnostic message that was either received or transmitted

UDS Message

`UdsMessage` class is meant to provide containers for a new *diagnostic messages* information. Once a diagnostic message object is created, it stores all diagnostic message information that were provided by a user. One can **use these objects to execute complex operations** (provided in other subpackages of `uds`) such as diagnostic messages transmission or segmentation.

Note: All `UdsMessage` **attributes are validated on each value change**, therefore a user will face an exception if one tries to set an invalid (e.g. incompatible with the annotation) value to any of these attributes.

Example code:

```
import uds

# example how to create an object
uds_message = uds.message.UdsMessage(payload=[0x10, 0x03],
                                     addressing_type=uds.transmission_
↪attributes.AddressingType.PHYSICAL)

# raw message attribute reassignment
uds_message.payload = (0x62, 0x01, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEF)

# addressing attribute reassignment
uds_message.addressing_type = uds.transmission_attributes.AddressingType.
↪FUNCTIONAL
```

UDS Message Record

UdsMessageRecord class is meant to provide containers for historic information of *diagnostic messages* that were either transmitted or received.

Note: A **user should not create objects of this class** in normal cases, but one would probably use them quite often as they are returned by other layers of *uds* package.

Warning: All *UdsMessageRecord* **attributes are read only** (they are set only once upon an object creation) as they store historic data and history cannot be changed (*can't it, right?*).

A user will face an exception if one tries to modify any attribute.

2.1.2 UDS Messages Data

Implementation of data parameters that are part of diagnostic messages data.

UDS data parameters:

- *Service Identifiers* - are implemented by:
  - *RequestSID*
  - *ResponseSID*
- *Negative Response Codes*

Service Identifiers

Implementation of *Service Identifier (SID)* values.

RequestSID

Enum *RequestSID* contains definitions of request *Service Identifiers* values.

Warning: *RequestSID* does not contain definition for every `POSSIBLE_REQUEST_SIDS` value as some Request SID values are reserved for further extension by UDS specification and others are ECU specific (defined by ECU's manufacturer).

Note: Use *add_member()* method on *RequestSID* class to add Request SID value.

Example code:

```
import uds

# check if a value (0xBA in the example) is a Request SID value
uds.message.RequestSID.is_request_sid(0xBA)
```

(continues on next page)

(continued from previous page)

```
# check if there is member defined for the value
uds.message.RequestSID.is_member(0xBA)

# example how to add a new Request SID value
new_member = uds.message.RequestSID.add_member("NewRequestSIDMemberName", 0xBA)

# check if the value was added as a new member
uds.message.RequestSID.is_member(new_member)
uds.message.RequestSID.is_member(0xBA)
```

ResponseSID

Enum *ResponseSID* contains definitions of response *Service Identifiers* values.

Warning: *ResponseSID* does not contain definition for every POSSIBLE_RESPONSE_SIDS value as some Response SID values are reserved for further extension by UDS specification and other are ECU specific (defined by ECU's manufacturer).

Note: Use *add_member()* method on *ResponseSID* class to add Response SID.

Example code:

```
import uds

# check if a value (0xFA in the example) is a Response SID value
uds.message.ResponseSID.is_response_sid(0xFA)

# check if there is member defined for the value
uds.message.ResponseSID.is_member(0xFA)

# example how to add a new Response SID value
new_member = uds.message.ResponseSID.add_member("NewResponseSIDMemberName",␣
↪0xFA)

# check if the value was added as a new member
uds.message.ResponseSID.is_member(new_member)
uds.message.ResponseSID.is_member(0xFA)
```

Negative Response Codes

Enum *NRC* contains definitions of all common (defined by ISO 14229) *Negative Response Codes* values.

> **Warning:** *NRC* does not contain definition for every possible NRC value as some of them are reserved for further extension by UDS specification and other are ECU specific (defined by ECU's manufacturer).

Note: Use *add_member()* method on *NRC* class to add NRC value that is specific for the system that you communicate with.

Example code:

```
import uds

# check if a value (0xF0 in the example) is a NRC value
uds.message.NRC.is_member(0xF0)

# example how to add a new NRC value
new_member = uds.message.NRC.add_member("NewNRCMemberName", 0xF0)

# check if the value was added as a new member
uds.message.NRC.is_member(new_member)
uds.message.NRC.is_member(0xF0)
```

2.2 Segmentation

Implementation of *segmentation process* is located in *uds.segmentation* sub-package.

2.2.1 AbstractSegmenter

AbstractSegmenter defines common API and contains common code for all segmenter classes. Each concrete segmenter class handles segmentation process for a specific bus.

> **Warning:** A **user shall not use** *AbstractSegmenter* **directly**, but one is able (and encouraged) to use *AbstractSegmenter* implementation on any of its children classes.

2.2.2 CanSegmenter

CanSegmenter handles segmentation process specific for CAN bus.

Following functionalities are provided by *CanSegmenter*:

- Configuration of the segmenter:

  As a user, you are able to configure *CanSegmenter* parameters which determines the addressing (Addressing Format and Addressing Information of input and output CAN packets) and the content (e.g. Filler Byte value and whether to use CAN Frame Data Optimization) of CAN packets.

Example code:

```
import uds

# define Addressing Information for a CAN Node
can_node_addressing_information = uds.can.CanAddressingInformation(
    addressing_format=uds.can.CanAddressingFormat.NORMAL_11BIT_ADDRESSING,
    tx_physical={"can_id": 0x611},
    rx_physical={"can_id": 0x612},
    tx_functional={"can_id": 0x6FF},
    rx_functional={"can_id": 0x6FE})

# configure CAN Segmenter for this CAN Node
can_segmenter = uds.segmentation.CanSegmenter(addressing_information=can_
↪node_addressing_information,
                                              dlc=8,
                                              use_data_optimization=False,
                                              filler_byte=0xFF)

# change CAN Segmenter configuration
can_segmenter.addressing_information = uds.can.CanAddressingInformation(
    uds.can.CanAddressingFormat.NORMAL_11BIT_ADDRESSING,
    tx_physical={"can_id": 0x612},
    rx_physical={"can_id": 0x611},
    tx_functional={"can_id": 0x6FE},
    rx_functional={"can_id": 0x6FF})
can_segmenter.dlc=0xF
can_segmenter.use_data_optimization = True
can_segmenter.filler_byte = 0xAA
```

- Diagnostic message segmentation:

  As a user, you are able to *segment diagnostic messages* (objects of `UdsMessage` class) into CAN packets (objects for `CanPacket` class).

  Example code:

```
# let's assume that we have `can_segmenter` already configured as presented␣
↪in configuration example above

# define diagnostic message to segment
uds_message_1 = uds.message.UdsMessage(payload=[0x3E, 0x00],
                                       addressing_type=uds.transmission_
↪attributes.AddressingType.FUNCTIONAL)
uds_message_2 = uds.message.UdsMessage(payload=[0x62, 0x10, 0x00] +␣
↪[0x20]*100,
                                       addressing_type=uds.transmission_
↪attributes.AddressingType.PHYSICAL)

# use preconfigured segmenter to segment the diagnostic messages
can_packets_1 = can_segmenter.segmentation(uds_message_1)  # output: Single␣
↪Frame
can_packets_2 = can_segmenter.segmentation(uds_message_2)  # output: First␣
↪Frame with Consecutive Frame(s)
```

Note: It is impossible to segment functionally addressed diagnostic message into First Frame and Consecutive Frame(s) as such result is considered incorrect according to *UDS ISO Standards*.

- CAN packets desegmentation:

  As a user, you are able to *desegment CAN packets* (either objects of `CanPacket` or `CanPacketRecord` class) into diagnostic messages (either objects of `UdsMessage` or `UdsMessageRecord` class).

  Example code:

```
# let's assume that we have `can_segmenter` already configured as presented in configuration example above

# define CAN packets to desegment
can_packets_1 = [
    uds.packet.CanPacket(packet_type=uds.packet.CanPacketType.SINGLE_FRAME,
                         addressing_format=uds.can.CanAddressingFormat.EXTENDED_ADDRESSING,
                         addressing_type=uds.transmission_attributes.AddressingType.FUNCTIONAL,
                         can_id=0x6A5,
                         target_address=0x0C,
                         payload=[0x3E, 0x80])
]
can_packets_2 = [
    uds.packet.CanPacket(packet_type=uds.packet.CanPacketType.FIRST_FRAME,
                         addressing_format=uds.can.CanAddressingFormat.NORMAL_FIXED_ADDRESSING,
                         addressing_type=uds.transmission_attributes.AddressingType.PHYSICAL,
                         target_address=0x12,
                         source_address=0xE0,
                         dlc=8,
                         data_length=15,
                         payload=[0x62, 0x10, 0x00] + 3*[0x20]),
    uds.packet.CanPacket(packet_type=uds.packet.CanPacketType.CONSECUTIVE_FRAME,
                         addressing_format=uds.can.CanAddressingFormat.NORMAL_FIXED_ADDRESSING,
                         addressing_type=uds.transmission_attributes.AddressingType.PHYSICAL,
                         target_address=0x12,
                         source_address=0xE0,
                         dlc=8,
                         sequence_number=1,
                         payload=7*[0x20]),
    uds.packet.CanPacket(packet_type=uds.packet.CanPacketType.CONSECUTIVE_FRAME,
                         addressing_format=uds.can.CanAddressingFormat.NORMAL_FIXED_ADDRESSING,
                         addressing_type=uds.transmission_attributes.AddressingType.PHYSICAL,
                         target_address=0x12,
```

(continues on next page)

(continued from previous page)

```
                          source_address=0xE0,
                          sequence_number=1,
                          payload=2 * [0x20],
                          filler_byte=0x99)
]

# use preconfigured segmenter to desegment the CAN packets
uds_message_1 = can_segmenter.desegmentation(can_packets_1)
uds_message_2 = can_segmenter.desegmentation(can_packets_2)
```

Warning: Desegmentation performs only sanity check of CAN packets content, therefore some inconsistencies with Diagnostic on CAN standard might be silently accepted as long as a message can be unambiguously decoded out of provided CAN packets.

Note: Desegmentation can be performed for any CAN packets (not only those targeting this CAN Node) in any format.

2.3 Transport Interfaces

Transport interfaces are meant to handle Physical (layer 1), Data (layer 2), Network (layer 3) and Transport (layer 4) layers of UDS OSI model which are unique for every communication bus. First two layers (Physical and Data Link) are usually handled by some external packages. Abstract API that is common for all Transport Interfaces (and therefore buses) is defined in *AbstractTransportInterface* class. The implementation is located in *uds.transport_interface* sub-package.

2.3.1 CAN Transport Interfaces

The implementation for Transport Interfaces that can be used with CAN bus is located in `uds.transport_interface.can_transport_interface.py` module.

Common

Common implementation for all all CAN Transport Interfaces is included in *AbstractCanTransportInterface*.

Warning: A **user shall not use** *AbstractCanTransportInterface* **directly**, but one is able (and encouraged) to use *AbstractCanTransportInterface* implementation on any of its children classes.

Configuration

CAN bus specific configuration is set upon calling `uds.transport_interface.can_transport_interface.AbstractCanTransportInterface.__init__()` method. The following configuration parameters are set then:

- Addressing Information of this CAN node - attribute *addressing_information*
- driver for a CAN bus interface - attribute *bus_manager*
- timing parameters (*N_Br*, *N_Cs*) - attributes *n_br* and *n_cs*
- communication timeout parameters (*N_As*, *N_Ar*, *N_Bs*, *N_Cr*) - attributes *n_as_timeout*, *n_ar_timeout*, *n_bs_timeout* and *n_cr_timeout*
- UDS message segmentation parameters (*base DLC of a CAN frame*, flag whether to use *data optimization for CAN frame*, and the value to use for *CAN frame data padding*) - attributes *dlc*, *use_data_optimization*, *filler_byte*,

Most of these attributes (all except *addressing_information*) can be changed after object is created.

Python-CAN

Class *PyCanTransportInterface* contains the implementation of CAN Transport Interface that uses python-can package for receiving and transmitting CAN frames.

Configuration

Configuration is set upon calling `uds.transport_interface.can_transport_interface.PyCanTransportInterface.__init__()` method and from the user perspective it does not provide any additional features to *common* implementation provided by `uds.transport_interface.can_transport_interface.AbstractCanTransportInterface.__init__()`.

Example code:

```
import uds
from can import Bus

# define example python-can bus interface (https://python-can.readthedocs.io/en/stable/
↪bus.html#bus-api)
python_can_interface = Bus(interface="kvaser", channel=0, fd=True, receive_own_
↪messages=True)

# define Addressing Information for a CAN Node
can_node_addressing_information = uds.can.CanAddressingInformation(
    addressing_format=uds.can.CanAddressingFormat.NORMAL_11BIT_ADDRESSING,
    tx_physical={"can_id": 0x611},
    rx_physical={"can_id": 0x612},
    tx_functional={"can_id": 0x6FF},
    rx_functional={"can_id": 0x6FE})

# configure CAN Transport Interface for this CAN Node
can_transport_interface = uds.transport_interface.PyCanTransportInterface(
    can_bus_manager=python_can_interface,
    addressing_information=can_node_addressing_information,
    n_as_timeout=50,
```

(continues on next page)

(continued from previous page)

```
    n_ar_timeout=900,
    n_bs_timeout=50,
    n_br=10,
    n_cs=0,
    n_cr_timeout = 900,
    dlc=0xF,
    use_data_optimization=True,
    filler_byte=0x55)

# change CAN Transport Interface configuration
can_transport_interface.n_as_timeout = uds.transport_interface.PyCanTransportInterface.N_
↪AS_TIMEOUT
can_transport_interface.n_ar_timeout = uds.transport_interface.PyCanTransportInterface.N_
↪AR_TIMEOUT
can_transport_interface.n_bs_timeout = uds.transport_interface.PyCanTransportInterface.N_
↪BS_TIMEOUT
can_transport_interface.n_br = uds.transport_interface.PyCanTransportInterface.DEFAULT_N_
↪BR
can_transport_interface.n_cs = uds.transport_interface.PyCanTransportInterface.DEFAULT_N_
↪CS
can_transport_interface.n_cr_timeout = uds.transport_interface.PyCanTransportInterface.N_
↪CR_TIMEOUT
can_transport_interface.dlc = 8
can_transport_interface.use_data_optimization = False
can_transport_interface.filler_byte = 0xAA
```

Send Packet

Once an object of *PyCanTransportInterface* class is created, there are two methods which can be used to transmit CAN packets:

- *send_packet()* - for synchronous implementation
- *async_send_packet()* - for asynchronous implementation

Example synchronous code:

```
# let's assume that we have `can_transport_interface` already configured as presented in␣
↪configuration example above

# define some UDS message to send
message = uds.message.UdsMessage(addressing_type=uds.transmission_attributes.
↪AddressingType.PHYSICAL,
                                 payload=[0x10, 0x03])

# segment the message to create a CAN packet
can_packet = can_transport_interface.segmenter.segmentation(message)[0]

# send CAN packet and receive CAN packet record with historic information about the␣
↪transmission and the transmitted CAN packet
can_packet_record = can_transport_interface.send_packet(can_packet)
```

Example asynchronous code:

```
# let's assume that we have `can_transport_interface` already configured as presented in␣
↪configuration example above

# define some UDS message to send
message = uds.message.UdsMessage(addressing_type=uds.transmission_attributes.
↪AddressingType.PHYSICAL,
                                 payload=[0x10, 0x03])

# segment the message to create a CAN packet
can_packet = can_transport_interface.segmenter.segmentation(message)[0]

# send CAN packet and receive CAN packet record with historic information about the␣
↪transmission and the transmitted CAN packet
can_packet_record = await can_transport_interface.async_send_packet(can_packet)
```

Note: In the example above, only a coroutine code was presented. If you need a manual how to run an asynchronous program, visit https://docs.python.org/3/library/asyncio-runner.html#running-an-asyncio-program.

Warning: Synchronous and asynchronous implementation shall not be mixed, so use either *send_packet()* and *receive_packet()* (synchronous) or *async_send_packet()* and *async_receive_packet()* (asynchronous) methods for transmitting and receiving CAN Packets.

See also:

Examples for python-can Transport Interface

Receive Packet

Once an object of *PyCanTransportInterface* class is created, there are two methods which can be used to receive CAN packets:

- *receive_packet()* - for synchronous implementation
- *async_receive_packet()* - for asynchronous implementation

Example synchronous code:

```
# let's assume that we have `can_transport_interface` already configured as presented in␣
↪configuration example above

# receive a CAN packet with timeout set to 1000 ms
can_packet_record = can_transport_interface.receive_packet(timeout=1000)
```

Example asynchronous code:

```
# let's assume that we have `can_transport_interface` already configured as presented in␣
↪configuration example above

# receive a CAN packet with timeout set to 1000 ms
can_packet_record = await can_transport_interface.async_receive_packet(timeout=1000)
```

Note: In the example above, only a coroutine code was presented. If you need a manual how to run an asynchronous program, visit https://docs.python.org/3/library/asyncio-runner.html#running-an-asyncio-program.

Warning: Synchronous and asynchronous implementation shall not be mixed, so use either *send_packet()* and *receive_packet()* (synchronous) or *async_send_packet()* and *async_receive_packet()* (asynchronous) methods for transmitting and receiving CAN Packets.

See also:

Examples for python-can Transport Interface

2.3.2 FlexRay Transport Interfaces

FlexRay FEATURE IS PLANNED BUT NOT IMPLEMENTED YET, THEREFORE THERE ARE NO MORE INFORMATION TO DISPLAY.

2.3.3 Ethernet Transport Interfaces

Ethernet FEATURE IS PLANNED BUT NOT IMPLEMENTED YET, THEREFORE THERE ARE NO MORE INFORMATION TO DISPLAY.

2.3.4 K-Line Transport Interfaces

K-Line FEATURE IS PLANNED BUT NOT IMPLEMENTED YET, THEREFORE THERE ARE NO MORE INFORMATION TO DISPLAY.

2.3.5 LIN Transport Interfaces

LIN FEATURE IS PLANNED BUT NOT IMPLEMENTED YET, THEREFORE THERE ARE NO MORE INFORMATION TO DISPLAY.

2.3.6 Custom Transport Interfaces

THIS FEATURE IS PLANNED BUT NOT IMPLEMENTED YET, THEREFORE THERE ARE NO MORE INFORMATION TO DISPLAY.

2.4 Client Simulation

This chapter describes how to simulate client (diagnostic tester or any other node which sends its request to other ECUs) in UDS communication. Client simulation enables sending diagnostic requests and receiving diagnostic responses from connected nodes.

THIS FEATURE IS PLANNED BUT NOT IMPLEMENTED YET, THEREFORE THERE ARE NO MORE INFORMATION TO DISPLAY.

2.5 Server Simulation

This chapter describes how to simulate server (any ECU that is recipient of diagnostic requests) in UDS communication. Server simulation supports defining diagnostic responses to incoming requests and then python program send them automatically according to the configuration provided by the user.

THIS FEATURE IS PLANNED BUT NOT IMPLEMENTED YET, THEREFORE THERE ARE NO MORE INFORMATION TO DISPLAY.

CHAPTER

THREE

EXAMPLES

Location of all example files: https://github.com/mdabrowski1990/uds/tree/main/examples

3.1 CAN

Code examples of UDS protocol communication over CAN bus.

See also:

https://github.com/mdabrowski1990/uds/tree/main/examples/can

3.1.1 Python-CAN

Examples with python-can package being used for controlling CAN bus (handling CAN frames transmission and reception).

See also:

https://github.com/mdabrowski1990/uds/tree/main/examples/can/python-can

Kvaser interface

- Send CAN packets (synchronous implementation):

```
from pprint import pprint
from time import sleep

from can import Bus
from uds.transport_interface import PyCanTransportInterface
from uds.can import CanAddressingInformation, CanAddressingFormat
from uds.message import UdsMessage
from uds.transmission_attributes import AddressingType


def main():
    # configure CAN interfaces
    kvaser_interface_1 = Bus(interface="kvaser", channel=0, fd=True, receive_own_
↪messages=True)
    kvaser_interface_2 = Bus(interface="kvaser", channel=1, fd=True, receive_own_
↪messages=True)
```

(continues on next page)

(continued from previous page)

```
    # configure Addressing Information of a CAN Node
    addressing_information = CanAddressingInformation(
        addressing_format=CanAddressingFormat.NORMAL_11BIT_ADDRESSING,
        tx_physical={"can_id": 0x611},
        rx_physical={"can_id": 0x612},
        tx_functional={"can_id": 0x6FF},
        rx_functional={"can_id": 0x6FE})

    # create Transport Interface object for UDS communication
    can_ti = PyCanTransportInterface(can_bus_manager=kvaser_interface_1,
                                     addressing_information=addressing_information)

    # define UDS Messages to send
    message_1 = UdsMessage(addressing_type=AddressingType.PHYSICAL, payload=[0x10, 0x03])
    message_2 = UdsMessage(addressing_type=AddressingType.FUNCTIONAL, payload=[0x3E])

    # create CAN packets that carries those UDS Messages
    packet_1 = can_ti.segmenter.segmentation(message_1)[0]
    packet_2 = can_ti.segmenter.segmentation(message_2)[0]

    # send CAN Packet 1
    record_1 = can_ti.send_packet(packet_1)
    pprint(record_1.__dict__)

    # send CAN Packet 2
    record_2 = can_ti.send_packet(packet_2)
    pprint(record_2.__dict__)

    # close connections with CAN interfaces
    del can_ti
    sleep(0.1)  # wait to make sure all tasks are closed
    kvaser_interface_1.shutdown()
    kvaser_interface_2.shutdown()


if __name__ == "__main__":
    main()
```

- Send CAN packets (asynchronous implementation):

```
import asyncio
from pprint import pprint

from can import Bus
from uds.transport_interface import PyCanTransportInterface
from uds.can import CanAddressingInformation, CanAddressingFormat
from uds.message import UdsMessage
from uds.transmission_attributes import AddressingType


async def main():
```

(continues on next page)

(continued from previous page)

```
    # configure CAN interfaces
    kvaser_interface_1 = Bus(interface="kvaser", channel=0, fd=True, receive_own_
↪messages=True)
    kvaser_interface_2 = Bus(interface="kvaser", channel=1, fd=True, receive_own_
↪messages=True)

    # configure Addressing Information of a CAN Node
    addressing_information = CanAddressingInformation(
        addressing_format=CanAddressingFormat.NORMAL_11BIT_ADDRESSING,
        tx_physical={"can_id": 0x611},
        rx_physical={"can_id": 0x612},
        tx_functional={"can_id": 0x6FF},
        rx_functional={"can_id": 0x6FE})

    # create Transport Interface object for UDS communication
    can_ti = PyCanTransportInterface(can_bus_manager=kvaser_interface_1,
                                     addressing_information=addressing_information)

    # define UDS Messages to send
    message_1 = UdsMessage(addressing_type=AddressingType.PHYSICAL, payload=[0x10, 0x03])
    message_2 = UdsMessage(addressing_type=AddressingType.FUNCTIONAL, payload=[0x3E])

    # create CAN packets that carries those UDS Messages
    packet_1 = can_ti.segmenter.segmentation(message_1)[0]
    packet_2 = can_ti.segmenter.segmentation(message_2)[0]

    # send CAN Packet 1
    record_1 = await can_ti.async_send_packet(packet_1)
    pprint(record_1.__dict__)

    # send CAN Packet 2
    record_2 = await can_ti.async_send_packet(packet_2)
    pprint(record_2.__dict__)

    # close connections with CAN interfaces
    del can_ti
    await asyncio.sleep(0.1)  # wait to make sure all tasks are closed
    kvaser_interface_1.shutdown()
    kvaser_interface_2.shutdown()


if __name__ == "__main__":
    asyncio.run(main())
```

- Receive CAN packets (synchronous implementation):

```
from pprint import pprint
from threading import Timer
from time import sleep

from can import Bus, Message
from uds.transport_interface import PyCanTransportInterface
```

(continues on next page)

(continued from previous page)

```
from uds.can import CanAddressingInformation, CanAddressingFormat


def main():
    # configure CAN interfaces
    kvaser_interface_1 = Bus(interface="kvaser", channel=0, fd=True, receive_own_
↪messages=True)
    kvaser_interface_2 = Bus(interface="kvaser", channel=1, fd=True, receive_own_
↪messages=True)

    # configure Addressing Information of a CAN Node
    addressing_information = CanAddressingInformation(
        addressing_format=CanAddressingFormat.NORMAL_11BIT_ADDRESSING,
        tx_physical={"can_id": 0x611},
        rx_physical={"can_id": 0x612},
        tx_functional={"can_id": 0x6FF},
        rx_functional={"can_id": 0x6FE})

    # create Transport Interface object for UDS communication
    can_ti = PyCanTransportInterface(can_bus_manager=kvaser_interface_1,
                                     addressing_information=addressing_information)

    # some frames to be received later on
    frame_1 = Message(arbitration_id=0x6FE, data=[0x10, 0x03])
    frame_2 = Message(arbitration_id=0x611, data=[0x10, 0x03])  # shall be ignored, as␣
↪it is not observed CAN ID
    frame_3 = Message(arbitration_id=0x612, data=[0x3E, 0x00, 0xAA, 0xAA, 0xAA, 0xAA,␣
↪0xAA, 0xAA])

    # receive CAN packet 1
    Timer(interval=0.1, function=kvaser_interface_1.send, args=(frame_1,)).start()  #␣
↪schedule transmission of frame 1
    record_1 = can_ti.receive_packet(timeout=1000)  # receive CAN packet 1 carried by␣
↪frame 1
    pprint(record_1.__dict__)  # show attributes of CAN packet record 1

    # receive CAN packet 2
    Timer(interval=0.3, function=kvaser_interface_1.send, args=(frame_2,)).start()  #␣
↪schedule transmission of frame 2
    Timer(interval=0.8, function=kvaser_interface_1.send, args=(frame_3,)).start()  #␣
↪schedule transmission of frame 3
    record_2 = can_ti.receive_packet(timeout=1000)  # receive CAN packet 2 carried by␣
↪frame 3
    pprint(record_2.__dict__)  # show attributes of CAN packet record 2

    # close connections with CAN interfaces
    del can_ti
    sleep(0.1)  # wait to make sure all tasks are closed
    kvaser_interface_1.shutdown()
    kvaser_interface_2.shutdown()
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    main()
```

- Receive CAN packets (asynchronous implementation):

```
import asyncio
from pprint import pprint

from can import Bus, Message
from uds.transport_interface import PyCanTransportInterface
from uds.can import CanAddressingInformation, CanAddressingFormat


async def main():
    # configure CAN interfaces
    kvaser_interface_1 = Bus(interface="kvaser", channel=0, fd=True, receive_own_
↪messages=True)
    kvaser_interface_2 = Bus(interface="kvaser", channel=1, fd=True, receive_own_
↪messages=True)

    # configure Addressing Information of a CAN Node
    addressing_information = CanAddressingInformation(
        addressing_format=CanAddressingFormat.NORMAL_11BIT_ADDRESSING,
        tx_physical={"can_id": 0x611},
        rx_physical={"can_id": 0x612},
        tx_functional={"can_id": 0x6FF},
        rx_functional={"can_id": 0x6FE})

    # create Transport Interface object for UDS communication
    can_ti = PyCanTransportInterface(can_bus_manager=kvaser_interface_1,
                                     addressing_information=addressing_information)

    # some frames to be received later on
    frame_1 = Message(arbitration_id=0x6FE, data=[0x10, 0x03])
    frame_2 = Message(arbitration_id=0x611, data=[0x10, 0x03])  # shall be ignored, as␣
↪it is not observed CAN ID
    frame_3 = Message(arbitration_id=0x612, data=[0x3E, 0x00, 0xAA, 0xAA, 0xAA, 0xAA,␣
↪0xAA, 0xAA])

    # receive CAN packet 1
    kvaser_interface_2.send(frame_1)  # transmit CAN Frame 1
    record_1 = await can_ti.async_receive_packet(timeout=1000)   # receive CAN packet 1␣
↪carried by frame 1
    pprint(record_1.__dict__)  # show attributes of CAN packet record 1

    # receive CAN packet 2
    kvaser_interface_2.send(frame_2)  # transmit CAN Frame 2
    kvaser_interface_2.send(frame_3)  # transmit CAN Frame 3
    record_2 = await can_ti.async_receive_packet(timeout=1000)
    pprint(record_2.__dict__)  # show attributes of CAN packet record 2

    # close connections with CAN interfaces
```

(continues on next page)

(continued from previous page)

```
    del can_ti
    await asyncio.sleep(0.1)  # wait to make sure all tasks are closed
    kvaser_interface_1.shutdown()
    kvaser_interface_2.shutdown()


if __name__ == "__main__":
    asyncio.run(main())
```

See also:

https://github.com/mdabrowski1990/uds/tree/main/examples/can/python-can/kvaser

CHAPTER

FOUR

API REFERENCE

This page contains auto-generated API reference documentation[1].

4.1 uds

Package for handling Unified Diagnostic Services (UDS) protocol defined by ISO-14229.

The package is meant to provide tools that enables:

- monitoring UDS communication
- simulation of any UDS node (either a client or a server)
- testing of a device that supports UDS
- injection of communication faults on any layers 3-7 of UDS OSI Model

The package is created with an idea to support any communication bus:

- CAN
- LIN
- Ethernet
- FlexRay
- K-Line

4.1.1 Subpackages

uds.can

A subpackage with CAN bus specific implementation.

It provides tools for:

- definition of CAN specific attributes: - CAN Addressing Formats - Flow Status
- handlers for CAN frame fields: - DLC - CAN ID
- handler for CAN specific packets: - Single Frame - First Frame - Consecutive Frame - Flow Status

[1] Created with sphinx-autoapi

Submodules

uds.can.abstract_addressing_information

Abstract definition of Addressing Information handler.

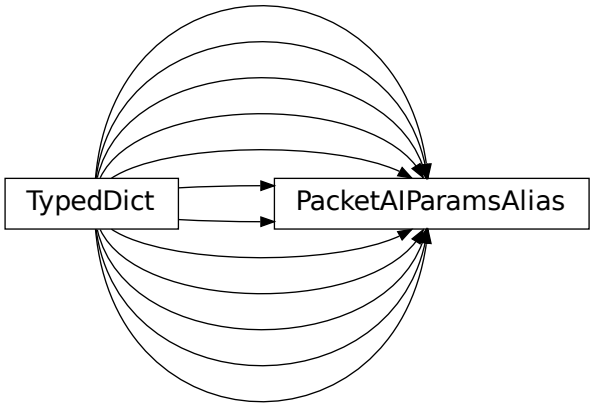
Module Contents

Classes

PacketAIParamsAlias	Alias of *Addressing Information* parameters of CAN packets stream.
AbstractCanAddressingInformation	Abstract definition of CAN Entity (either server or client) Addressing Information.

class uds.can.abstract_addressing_information.**PacketAIParamsAlias**

Bases: TypedDict



Alias of *Addressing Information* parameters of CAN packets stream.

Initialize self. See help(type(self)) for accurate signature.

addressing_format: *uds.can.addressing_format.CanAddressingFormat*

addressing_type: **uds.transmission_attributes.AddressingType**

can_id: **int**

target_address: **int | None**

source_address: **int | None**

address_extension: int | None

class uds.can.abstract_addressing_information.**AbstractCanAddressingInformation**(*rx_physical*, *tx_physical*, *rx_functional*, *tx_functional*)

Bases: abc.ABC

Abstract definition of CAN Entity (either server or client) Addressing Information.

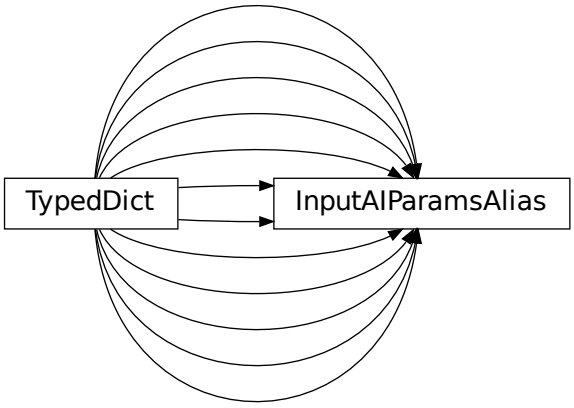
Configure Addressing Information of a CAN Entity.

Parameters

- **rx_physical** (InputAIParamsAlias) – Addressing Information parameters used for incoming physically addressed communication.
- **tx_physical** (InputAIParamsAlias) – Addressing Information parameters used for outgoing physically addressed communication.
- **rx_functional** (InputAIParamsAlias) – Addressing Information parameters used for incoming functionally addressed communication.
- **tx_functional** (InputAIParamsAlias) – Addressing Information parameters used for outgoing functionally addressed communication.

class InputAIParamsAlias

Bases: TypedDict



Alias of *Addressing Information* configuration parameters.

Initialize self. See help(type(self)) for accurate signature.

can_id: int

target_address: int

source_address: int

address_extension: int

abstract property addressing_format: *uds.can.addressing_format.CanAddressingFormat*

CAN Addressing format used.

Return type
uds.can.addressing_format.CanAddressingFormat

property rx_packets_physical_ai: *PacketAIParamsAlias*

Addressing Information parameters of incoming physically addressed CAN packets.

Return type
PacketAIParamsAlias

property tx_packets_physical_ai: *PacketAIParamsAlias*

Addressing Information parameters of outgoing physically addressed CAN packets.

Return type
PacketAIParamsAlias

property rx_packets_functional_ai: *PacketAIParamsAlias*

Addressing Information parameters of incoming functionally addressed CAN packets.

Return type
PacketAIParamsAlias

property tx_packets_functional_ai: *PacketAIParamsAlias*

Addressing Information parameters of outgoing functionally addressed CAN packets.

Return type
PacketAIParamsAlias

ADDRESSING_FORMAT_NAME: str = 'addressing_format'

Name of *CAN Addressing Format* parameter in Addressing Information.

ADDRESSING_TYPE_NAME: str

Name of *Addressing Type* parameter in Addressing Information.

CAN_ID_NAME: str = 'can_id'

Name of CAN Identifier parameter in Addressing Information.

TARGET_ADDRESS_NAME: str

Name of Target Address parameter in Addressing Information.

SOURCE_ADDRESS_NAME: str

Name of Source Address parameter in Addressing Information.

ADDRESS_EXTENSION_NAME: str = 'address_extension'

Name of Address Extension parameter in Addressing Information.

AI_DATA_BYTES_NUMBER: int

Number of CAN Frame data bytes that are used to carry Addressing Information.

abstract classmethod validate_packet_ai(*addressing_type*, *can_id=None*, *target_address=None*, *source_address=None*, *address_extension=None*)

Validate Addressing Information parameters of a CAN packet.

Parameters

- **addressing_type** (*uds.transmission_attributes.AddressingType*) – Addressing type to validate.
- **can_id** (*Optional[int]*) – CAN Identifier value to validate.
- **target_address** (*Optional[int]*) – Target Address value to validate.
- **source_address** (*Optional[int]*) – Source Address value to validate.
- **address_extension** (*Optional[int]*) – Address Extension value to validate.

Raises

- ***InconsistentArgumentsError*** – Provided values are not consistent with each other (cannot be used together) or with the Addressing format used.
- ***UnusedArgumentError*** – Provided parameter is not supported by Addressing format used.

Returns

Normalized dictionary with the provided information.

Return type

PacketAIParamsAlias

uds.can.addressing_format

Implementation of CAN Addressing Formats.

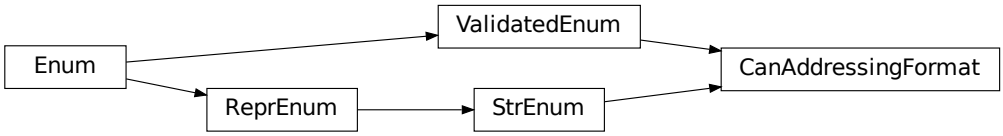
Module Contents

Classes

CanAddressingFormat	Definition of CAN addressing formats.

class uds.can.addressing_format.**CanAddressingFormat**

Bases: aenum.StrEnum, uds.utilities.ValidatedEnum



Definition of CAN addressing formats.

CAN addressing formats determines how (in which fields of a CAN Packet) *Network Address Information (N_AI)* is provided.

Initialize self. See help(type(self)) for accurate signature.

NORMAL_11BIT_ADDRESSING: *CanAddressingFormat* = 'Normal 11-bit Addressing'

Normal addressing format that uses 11-bit CAN Identifiers.

NORMAL_FIXED_ADDRESSING: *CanAddressingFormat* = 'Normal Fixed Addressing'

Normal fixed addressing format. It is a subformat of *Normal addressing* which uses 29-bit CAN Identifiers only.

EXTENDED_ADDRESSING: *CanAddressingFormat* = 'Extended Addressing'

Extended addressing format.

MIXED_11BIT_ADDRESSING: *CanAddressingFormat* = 'Mixed 11-bit Addressing'

Mixed addressing with 11-bit CAN ID format. It is a subformat of *mixed addressing*.

MIXED_29BIT_ADDRESSING: *CanAddressingFormat* = 'Mixed 29-bit Addressing'

Mixed addressing with 29-bit CAN ID format. It is a subformat of *mixed addressing*.

uds.can.addressing_information

Implementation of CAN Addressing Information.

This module contains helper class for managing *Addressing Information* on CAN bus.

Module Contents

Classes

CanAddressingInformation	CAN Entity (either server or client) Addressing Information.

class uds.can.addressing_information.**CanAddressingInformation**

CAN Entity (either server or client) Addressing Information.

Create object of CAN Entity (either server or client) Addressing Information.

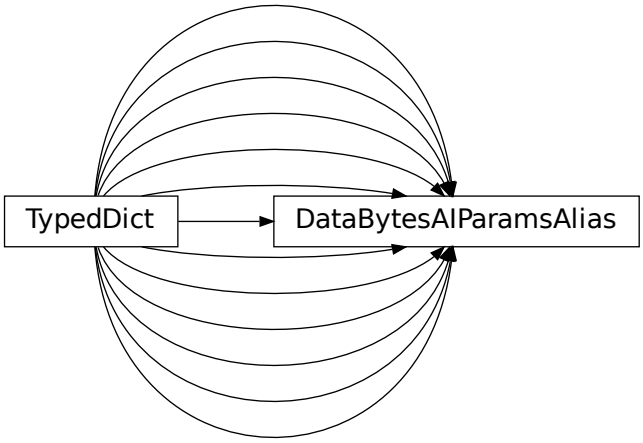
Parameters

- **addressing_format** – CAN Addressing format used by CAN Entity.

- **rx_physical** – Addressing Information parameters used for incoming physically addressed communication.
- **tx_physical** – Addressing Information parameters used for outgoing physically addressed communication.
- **rx_functional** – Addressing Information parameters used for incoming functionally addressed communication.
- **tx_functional** – Addressing Information parameters used for outgoing functionally addressed communication.

class DataBytesAIParamsAlias

Bases: TypedDict



Alias of *Addressing Information* parameters encoded in data field.

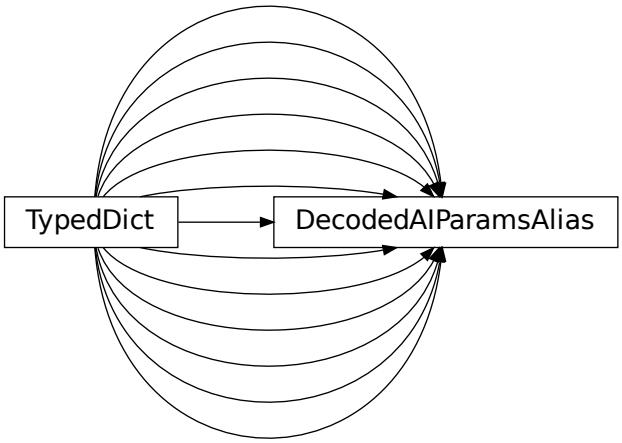
Initialize self. See help(type(self)) for accurate signature.

target_address: int

address_extension: int

class DecodedAIParamsAlias

Bases: TypedDict



Alias of *Addressing Information* parameters encoded in CAN ID and data field.

Initialize self. See help(type(self)) for accurate signature.

addressing_type: uds.transmission_attributes.AddressingType | None

target_address: int | None

source_address: int | None

address_extension: int | None

ADDRESSING_INFORMATION_MAPPING: Dict[*uds.can.addressing_format.CanAddressingFormat*, Type[*uds.can.abstract_addressing_information.AbstractCanAddressingInformation*]]

Dictionary with CAN Addressing format mapping to Addressing Information handler classes.

classmethod validate_packet_ai(*addressing_format*, *addressing_type*, *can_id=None*, *target_address=None*, *source_address=None*, *address_extension=None*)

Validate Addressing Information parameters of a CAN packet.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN addressing format value to validate.
- **addressing_type** (*uds.transmission_attributes.AddressingType*) – Addressing type to validate.
- **can_id** (*Optional[int]*) – CAN Identifier value to validate.
- **target_address** (*Optional[int]*) – Target Address value to validate.
- **source_address** (*Optional[int]*) – Source Address value to validate.
- **address_extension** (*Optional[int]*) – Address Extension value to validate.

Returns

Normalized dictionary with the provided Addressing Information.

Return type
uds.can.abstract_addressing_information.PacketAIParamsAlias

classmethod validate_ai_data_bytes(*addressing_format*, *ai_data_bytes*)

Validate Addressing Information stored in CAN data bytes.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN Addressing Format used.
- **ai_data_bytes** (*uds.utilities.RawBytesAlias*) – Data bytes to validate.

Raises
InconsistentArgumentsError – Provided number of Addressing Information data bytes does not match Addressing Format used.

Return type
None

classmethod decode_packet_ai(*addressing_format*, *can_id*, *ai_data_bytes*)

Decode Addressing Information parameters from CAN ID and data bytes.

> **Warning:** This methods might not extract full Addressing Information from the provided data as some of them are system specific.
>
> For example, Addressing Type will not be decoded when either Normal 11bit, Extended or Mixed 11bit addressing format is used as the Addressing Type (in such case) depends on system specific behaviour.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN Addressing Format used.
- **can_id** (*int*) – Value of CAN Identifier.
- **ai_data_bytes** (*uds.utilities.RawBytesAlias*) – Data bytes containing Addressing Information. This parameter shall contain either 0 or 1 byte that is located at the beginning of a CAN frame data field. Number of these bytes depends on *CAN Addressing Format* used.

Returns
Dictionary with Addressing Information decoded out of the provided CAN ID and data bytes.

Return type
DecodedAIParamsAlias

classmethod decode_ai_data_bytes(*addressing_format*, *ai_data_bytes*)

Decode Addressing Information from CAN data bytes.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN Addressing Format used.
- **ai_data_bytes** (*uds.utilities.RawBytesAlias*) – Data bytes containing Addressing Information. This parameter shall contain either 0 or 1 byte that is located at the beginning of a CAN frame data field. Number of these bytes depends on *CAN Addressing Format* used.

Raises
NotImplementedError – There is missing implementation for the provided Addressing Format. Please create an issue in our Issues Tracking System with detailed description if you face this error.

Returns
Dictionary with Addressing Information decoded out of the provided data bytes.

Return type
DataBytesAIParamsAlias

classmethod encode_ai_data_bytes(*addressing_format*, *target_address=None*, *address_extension=None*)

Generate a list of data bytes that carry Addressing Information.

Parameters
- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN Addressing Format used.
- **target_address** (*Optional[int]*) – Target Address value used.
- **address_extension** (*Optional[int]*) – Source Address value used.

Raises
NotImplementedError – There is missing implementation for the provided Addressing Format. Please create an issue in our Issues Tracking System with detailed description if you face this error.

Returns
List of data bytes that carry Addressing Information in CAN frame Data field.

Return type
uds.utilities.RawBytesListAlias

classmethod get_ai_data_bytes_number(*addressing_format*)

Get number of data bytes that are used to carry Addressing Information.

Parameters
addressing_format (uds.can.addressing_format.CanAddressingFormat) – CAN Addressing Format used.

Returns
Number of data bytes in a CAN Packet that are used to carry Addressing Information for provided CAN Addressing Format.

Return type
int

uds.can.consecutive_frame

Implementation specific for Consecutive Frame CAN packets.

This module contains implementation specific for *Consecutive Frame* packets - that includes *Sequence Number (SN)* parameter.

Module Contents

Classes

CanConsecutiveFrameHandler	Helper class that provides utilities for Consecutive Frame CAN Packets.

class uds.can.consecutive_frame.**CanConsecutiveFrameHandler**

Helper class that provides utilities for Consecutive Frame CAN Packets.

CONSECUTIVE_FRAME_N_PCI: int = 2

Consecutive Frame N_PCI value.

SN_BYTES_USED: int = 1

Number of CAN Frame data bytes used to carry CAN Packet Type and Sequence Number in Consecutive Frame.

classmethod create_valid_frame_data(*, *addressing_format*, *payload*, *sequence_number*, *dlc=None*, *filler_byte=DEFAULT_FILLER_BYTE*, *target_address=None*, *address_extension=None*)

Create a data field of a CAN frame that carries a valid Consecutive Frame packet.

Note: This method can only be used to create a valid (compatible with ISO 15765 - Diagnostic on CAN) output. Use *create_any_frame_data()* to create data bytes for a Consecutive Frame with any (also incompatible with ISO 15765) parameters values.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN addressing format used by a considered Consecutive Frame.
- **payload** (*uds.utilities.RawBytesAlias*) – Payload of a diagnostic message that is carried by a considered CAN packet.
- **sequence_number** (*int*) – Value of Sequence Number parameter.
- **dlc** (*Optional[int]*) – DLC value of a CAN frame that carries a considered CAN Packet.
  - None - use CAN Data Frame Optimization (CAN ID value will be automatically determined)
  - int type value - DLC value to set. CAN Data Padding will be used to fill the unused data bytes.
- **filler_byte** (*int*) – Filler Byte value to use for CAN Frame Data Padding.
- **target_address** (*Optional[int]*) – Target Address value carried by this CAN Packet. The value must only be provided if *addressing_format* uses Target Address parameter.
- **address_extension** (*Optional[int]*) – Address Extension value carried by this CAN packet. The value must only be provided if *addressing_format* uses Address Extension parameter.

Raises

InconsistentArgumentsError – Provided *payload* contains invalid number of bytes to fit it into a properly defined Consecutive Frame data field.

Returns
Raw bytes of CAN frame data for the provided Consecutive Frame packet information.

Return type
uds.utilities.RawBytesListAlias

classmethod create_any_frame_data(*, *addressing_format*, *payload*, *sequence_number*, *dlc*, *filler_byte=DEFAULT_FILLER_BYTE*, *target_address=None*, *address_extension=None*)

Create a data field of a CAN frame that carries a Consecutive Frame packet.

Note: You can use this method to create Consecutive Frame data bytes with any (also inconsistent with ISO 15765) parameters values. It is recommended to use *create_valid_frame_data()* to create data bytes for a Consecutive Frame with valid (compatible with ISO 15765) parameters values.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN addressing format used by a considered Consecutive Frame.
- **payload** (*uds.utilities.RawBytesAlias*) – Payload of a diagnostic message that is carried by a considered CAN packet.
- **sequence_number** (*int*) – Value of Sequence Number parameter.
- **dlc** (*int*) – DLC value of a CAN frame that carries a considered CAN Packet.
- **filler_byte** (*int*) – Filler Byte value to use for CAN Frame Data Padding.
- **target_address** (*Optional[int]*) – Target Address value carried by this CAN Packet. The value must only be provided if *addressing_format* uses Target Address parameter.
- **address_extension** (*Optional[int]*) – Address Extension value carried by this CAN packet. The value must only be provided if *addressing_format* uses Address Extension parameter.

Raises
InconsistentArgumentsError – Provided *payload* contains too many bytes to fit it into a Consecutive Frame data field.

Returns
Raw bytes of CAN frame data for the provided Consecutive Frame packet information.

Return type
uds.utilities.RawBytesListAlias

classmethod is_consecutive_frame(*addressing_format*, *raw_frame_data*)

Check if provided data bytes encodes a Consecutive Frame packet.

Warning: The method does not validate the content of the provided frame data bytes. Only, *CAN Packet Type (N_PCI)* parameter is checked whether contain Consecutive Frame N_PCI value.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN Addressing Format used.

- **raw_frame_data** (*uds.utilities.RawBytesAlias*) – Raw data bytes of a CAN frame to check.

Returns
True if provided data bytes carries Consecutive Frame, False otherwise.

Return type
bool

classmethod decode_payload(*addressing_format*, *raw_frame_data*)

Extract diagnostic message payload from Consecutive Frame data bytes.

> **Warning:** The output might contain additional filler bytes (they are not part of diagnostic message payload) that were added during *CAN Frame Data Padding*. The presence of filler bytes in *Consecutive Frame* cannot be determined basing solely on the information contained in a Consecutive Frame data bytes.

> **Warning:** The method does not validate the content of the provided frame data bytes. There is no guarantee of the proper output when frame data in invalid format (incompatible with ISO 15765) is provided.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN Addressing Format used.
- **raw_frame_data** (*uds.utilities.RawBytesAlias*) – Raw data bytes of a considered CAN frame.

Raises
ValueError – Provided frame data of a CAN frames does not carry a Consecutive Frame CAN packet.

Returns
Payload bytes (with potential Filler Bytes) of a diagnostic message carried by a considered Consecutive Frame.

Return type
uds.utilities.RawBytesListAlias

classmethod decode_sequence_number(*addressing_format*, *raw_frame_data*)

Extract a value of Sequence Number from Consecutive Frame data bytes.

> **Warning:** The method does not validate the content of the provided frame data bytes. There is no guarantee of the proper output when frame data in invalid format (incompatible with ISO 15765) is provided.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN Addressing Format used.

- **raw_frame_data** (*uds.utilities.RawBytesAlias*) – Raw data bytes of a considered CAN frame.

Raises
ValueError – Provided frame data of a CAN frames does not carry a Consecutive Frame CAN packet.

Returns
Extracted value of Sequence Number.

Return type
int

classmethod get_min_dlc(*addressing_format*, *payload_length=1*)

Get the minimum value of a CAN frame DLC to carry a Consecutive Frame packet.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN addressing format that considered CAN packet uses.
- **payload_length** (*int*) – Number of payload bytes that considered CAN packet carries.

Raises

- **TypeError** – Provided value of Payload Length is not integer value.
- **ValueError** – Provided value of Payload Length is out of range.
- ***InconsistentArgumentsError*** – Provided Addressing Format and Payload Length values cannot be used together.

Returns
The lowest value of DLC that enables to fit in provided Consecutive Frame packet data.

Return type
int

classmethod get_max_payload_size(*addressing_format=None*, *dlc=None*)

Get the maximum size of a payload that can fit into Consecutive Frame data bytes.

Parameters

- **addressing_format** (*Optional[*uds.can.addressing_format.CanAddressingFormat*]*) – CAN addressing format that considered CAN packet uses. Leave None to get the result for CAN addressing format that does not use data bytes for carrying addressing information.
- **dlc** (*Optional[int]*) – DLC value of a CAN frame that carries a considered CAN Packet. Leave None to get the result for the greatest possible DLC value.

Raises
InconsistentArgumentsError – Consecutive Frame packet cannot use provided attributes according to ISO 15765.

Returns
The maximum number of payload bytes that could fit into a considered Consecutive Frame.

Return type
int

classmethod validate_frame_data(*addressing_format*, *raw_frame_data*)

Validate whether data field of a CAN Packet carries a properly encoded Consecutive Frame.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN Addressing Format used.
- **raw_frame_data** (*uds.utilities.RawBytesAlias*) – Raw data bytes of a CAN frame to validate.

Raises

- **ValueError** – Provided frame data of a CAN frames does not carry a Consecutive Frame CAN packet.
- ***InconsistentArgumentsError*** – Provided frame data of a CAN frames does not carry a properly encoded Consecutive Frame CAN packet.

Return type
None

classmethod __encode_sn(*sequence_number*)

Create Consecutive Frame data bytes with CAN Packet Type and Sequence Number parameters.

Parameters
sequence_number (*int*) – Value of the sequence number parameter.

Returns
Consecutive Frame data bytes containing CAN Packet Type and Sequence Number parameters.

Return type
uds.utilities.RawBytesListAlias

uds.can.extended_addressing_information

Implementation of Extended Addressing Information handler.

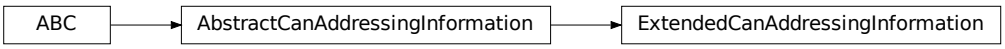
Module Contents

Classes

ExtendedCanAddressingInformation	Addressing Information of CAN Entity (either server or client) that uses Extended Addressing format.

class uds.can.extended_addressing_information.**ExtendedCanAddressingInformation**(*rx_physical*, *tx_physical*, *rx_functional*, *tx_functional*)

Bases: *uds.can.abstract_addressing_information.AbstractCanAddressingInformation*



Addressing Information of CAN Entity (either server or client) that uses Extended Addressing format.

Configure Addressing Information of a CAN Entity.

Parameters

- **rx_physical** (InputAIParamsAlias) – Addressing Information parameters used for incoming physically addressed communication.
- **tx_physical** (InputAIParamsAlias) – Addressing Information parameters used for outgoing physically addressed communication.
- **rx_functional** (InputAIParamsAlias) – Addressing Information parameters used for incoming functionally addressed communication.
- **tx_functional** (InputAIParamsAlias) – Addressing Information parameters used for outgoing functionally addressed communication.

property addressing_format: *uds.can.addressing_format.CanAddressingFormat*

CAN Addressing format used.

Return type
uds.can.addressing_format.CanAddressingFormat

AI_DATA_BYTES_NUMBER: int = 1

Number of CAN Frame data bytes that are used to carry Addressing Information.

classmethod validate_packet_ai(*addressing_type*, *can_id=None*, *target_address=None*, *source_address=None*, *address_extension=None*)

Validate Addressing Information parameters of a CAN packet that uses Extended Addressing format.

Parameters

- **addressing_type** (*uds.transmission_attributes.AddressingType*) – Addressing type to validate.
- **can_id** (*Optional[int]*) – CAN Identifier value to validate.
- **target_address** (*Optional[int]*) – Target Address value to validate.
- **source_address** (*Optional[int]*) – Source Address value to validate.
- **address_extension** (*Optional[int]*) – Address Extension value to validate.

Raises

- *InconsistentArgumentsError* – Provided CAN ID value is incompatible with Extended Addressing format.
- *UnusedArgumentError* – Provided parameter is not supported by this Addressing format.

Returns
Normalized dictionary with the provided Addressing Information.

Return type
uds.can.abstract_addressing_information.PacketAIParamsAlias

uds.can.first_frame

Implementation specific for First Frame CAN packets.

This module contains implementation specific for *First Frame* packets - that includes *First Frame Data Length (FF_DL)* parameter.

Module Contents

Classes

CanFirstFrameHandler	Helper class that provides utilities for First Frame CAN Packets.

class uds.can.first_frame.**CanFirstFrameHandler**

Helper class that provides utilities for First Frame CAN Packets.

FIRST_FRAME_N_PCI: int = 1

First Frame N_PCI value.

MAX_SHORT_FF_DL_VALUE: int = 4095

Maximum value of *First Frame Data Length (FF_DL)* for which short format of FF_DL is used.

MAX_LONG_FF_DL_VALUE: int = 4294967295

Maximum value of *First Frame Data Length (FF_DL)*.

SHORT_FF_DL_BYTES_USED: int = 2

Number of CAN Frame data bytes used to carry CAN Packet Type and First Frame Data Length (FF_DL). This value is valid only for the short format using FF_DL <= 4095.

LONG_FF_DL_BYTES_USED: int = 6

Number of CAN Frame data bytes used to carry CAN Packet Type and First Frame Data Length (FF_DL). This value is valid only for the long format using FF_DL > 4095.

classmethod create_valid_frame_data(*, *addressing_format*, *payload*, *dlc*, *ff_dl*, *target_address=None*, *address_extension=None*)

Create a data field of a CAN frame that carries a valid First Frame packet.

> **Note:** This method can only be used to create a valid (compatible with ISO 15765 - Diagnostic on CAN) output. Use *create_any_frame_data()* to create data bytes for a First Frame with any (also incompatible with ISO 15765) parameters values.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN addressing format used by a considered First Frame.
- **payload** (*uds.utilities.RawBytesAlias*) – Payload of a diagnostic message that is carried by a considered CAN packet.
- **dlc** (*int*) – DLC value of a CAN frame that carries a considered CAN Packet.

- **ff_dl** (*int*) – Value of First Frame Data Length parameter that is carried by a considered CAN packet.
- **target_address** (*Optional[int]*) – Target Address value carried by this CAN Packet. The value must only be provided if *addressing_format* uses Target Address parameter.
- **address_extension** (*Optional[int]*) – Address Extension value carried by this CAN packet. The value must only be provided if *addressing_format* uses Address Extension parameter.

Raises
InconsistentArgumentsError – Provided *payload* contains incorrect number of bytes to fit them into a First Frame data field using provided parameters.

Returns
Raw bytes of CAN frame data for the provided First Frame packet information.

Return type
uds.utilities.RawBytesListAlias

classmethod create_any_frame_data(*, *addressing_format*, *payload*, *dlc*, *ff_dl*, *long_ff_dl_format=False*, *target_address=None*, *address_extension=None*)

Create a data field of a CAN frame that carries a First Frame packet.

Note: You can use this method to create First Frame data bytes with any (also inconsistent with ISO 15765) parameters values. It is recommended to use *create_valid_frame_data()* to create data bytes for a First Frame with valid (compatible with ISO 15765) parameters values.

Parameters

- **addressing_format** (*uds.can.addressing_format.CanAddressingFormat*) – CAN addressing format used by a considered First Frame.
- **payload** (*uds.utilities.RawBytesAlias*) – Payload of a diagnostic message that is carried by a considered CAN packet.
- **dlc** (*int*) – DLC value of a CAN frame that carries a considered CAN Packet.
- **ff_dl** (*int*) – Value of First Frame Data Length parameter that is carried by a considered CAN packet.
- **long_ff_dl_format** (*bool*) – Information whether to use long or short format of First Frame Data Length.
- **target_address** (*Optional[int]*) – Target Address value carried by this CAN Packet. The value must only be provided if *addressing_format* uses Target Address parameter.
- **address_extension** (*Optional[int]*) – Address Extension value carried by this CAN packet. The value must only be provided if *addressing_format* uses Address Extension parameter.

Raises
InconsistentArgumentsError – Provided *payload* contains incorrect number of bytes to fit them into a First Frame data field using provided parameters.

Returns
Raw bytes of CAN frame data for the provided First Frame packet information.

Return type
uds.utilities.RawBytesListAlias

classmethod is_first_frame(*addressing_format*, *raw_frame_data*)

Check if provided data bytes encodes a First Frame packet.

> **Warning:** The method does not validate the content (e.g. FF_DL parameter) of the packet.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN Addressing Format used.
- **raw_frame_data** (*uds.utilities.RawBytesAlias*) – Raw data bytes of a CAN frame to check.

Returns
True if provided data bytes carries First Frame, False otherwise.

Return type
bool

classmethod decode_payload(*addressing_format*, *raw_frame_data*)

Extract a value of payload from First Frame data bytes.

> **Warning:** The method does not validate the content of the provided frame data bytes. There is no guarantee of the proper output when frame data in invalid format (incompatible with ISO 15765) is provided.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN Addressing Format used.
- **raw_frame_data** (*uds.utilities.RawBytesAlias*) – Raw data bytes of a considered CAN frame.

Returns
Payload bytes of a diagnostic message carried by a considered First Frame.

Return type
uds.utilities.RawBytesListAlias

classmethod decode_ff_dl(*addressing_format*, *raw_frame_data*)

Extract a value of First Frame Data Length from First Frame data bytes.

> **Warning:** The method does not validate the content of the provided frame data bytes. There is no guarantee of the proper output when frame data in invalid format (incompatible with ISO 15765) is provided.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN Addressing Format used.
- **raw_frame_data** (*uds.utilities.RawBytesAlias*) – Raw data bytes of a considered CAN frame.

Raises
NotImplementedError – There is missing implementation for the provided First Frame Data Length format. Please create an issue in our Issues Tracking System with detailed description if you face this error.

Returns
Extracted value of First Frame Data Length.

Return type
int

classmethod get_payload_size(*addressing_format*, *dlc*, *long_ff_dl_format=False*)

Get the size of a payload that can fit into First Frame data bytes.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN addressing format that considered CAN packet uses.
- **dlc** (*int*) – DLC value of a CAN frame that carries a considered CAN Packet.
- **long_ff_dl_format** (*bool*) – Information whether to use long or short format of First Frame Data Length.

Raises
ValueError – First Frame packet cannot use provided attributes according to ISO 15765.

Returns
The maximum number of payload bytes that could fit into a considered First Frame.

Return type
int

classmethod validate_frame_data(*addressing_format*, *raw_frame_data*)

Validate whether data field of a CAN Packet carries a properly encoded First Frame.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN Addressing Format used.
- **raw_frame_data** (*uds.utilities.RawBytesAlias*) – Raw data bytes of a CAN frame to validate.

Raises
ValueError – Provided frame data of a CAN frames does not carry a First Frame CAN packet.

Return type
None

classmethod validate_ff_dl(*ff_dl*, *long_ff_dl_format=None*, *dlc=None*, *addressing_format=None*)

Validate a value of First Frame Data Length.

Parameters

- **ff_dl** (*int*) – First Frame Data Length value to validate.

- **long_ff_dl_format** (*Optional[bool]*) – Information whether long or short format of First Frame Data Length is used.
  - None - do not perform compatibility check with the FF_DL format
  - True - perform compatibility check with long FF_DL format
  - False - perform compatibility check with short FF_DL format
- **dlc** (*Optional[int]*) – Value of DLC to use for First Frame Data Length value validation. Leave None if you do not want to validate whether First Frame shall be used in this case.
- **addressing_format** (*Optional[uds.can.addressing_format.CanAddressingFormat]*) – Value of CAN Addressing Format to use for First Frame Data Length value validation. Leave None if you do not want to validate whether First Frame shall be used in this case.

Raises

- **TypeError** – Provided value of First Frame Data Length is not integer.
- **ValueError** – Provided value of First Frame Data Length is out of range.
- ***InconsistentArgumentsError*** – Single Frame shall be used instead of First Frame to transmit provided number of payload bytes represented by FF_DL value.

Return type
None

classmethod __extract_ff_dl_data_bytes(*addressing_format*, *raw_frame_data*)

Extract data bytes that carries CAN Packet Type and First Frame Data Length parameters.

> **Warning:** This method does not check whether provided *raw_frame_data* actually contains First Frame.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN Addressing Format used.
- **raw_frame_data** (*uds.utilities.RawBytesAlias*) – Raw data bytes of a considered CAN frame.

Returns
Extracted data bytes with CAN Packet Type and First Frame Data Length parameters.

Return type
uds.utilities.RawBytesListAlias

classmethod __encode_valid_ff_dl(*ff_dl*, *dlc*, *addressing_format*)

Create First Frame data bytes with CAN Packet Type and First Frame Data Length parameters.

Note: This method can only be used to create a valid (compatible with ISO 15765 - Diagnostic on CAN) output. First Frame Data Length value validation (whether it is too low according to ISO 15765) is not performed though.

Parameters

- **ff_dl** (*int*) – Value to put into a slot of First Frame Data Length.

- **dlc** (*int*) – Value of DLC to use for First Frame Data Length value validation.
- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – Value of CAN Addressing Format to use for First Frame Data Length value validation.

Returns
First Frame data bytes containing CAN Packet Type and First Frame Data Length parameters.

Return type
uds.utilities.RawBytesListAlias

classmethod __encode_any_ff_dl(*ff_dl*, *long_ff_dl_format=False*)

Create First Frame data bytes with CAN Packet Type and First Frame Data Length parameters.

Note: This method can be used to create any (also incompatible with ISO 15765 - Diagnostic on CAN) output.

Parameters

- **ff_dl** (*int*) – Value to put into a slot of First Frame Data Length.
- **long_ff_dl_format** (*bool*) – Information whether to use long or short format of First Frame Data Length.

Raises

- **ValueError** – Provided First Frame Data Length value is out of the parameter values range.
- ***InconsistentArgumentsError*** – Provided First Frame Data Length value cannot fit into the short format.

Returns
First Frame data bytes containing CAN Packet Type and First Frame Data Length parameters.

Return type
uds.utilities.RawBytesListAlias

uds.can.flow_control

Implementation specific for Flow Control CAN packets.

This module contains implementation of *Flow Control* packet attributes:

- *Flow Status*
- *Block Size*
- *Separation Time minimum (STmin)*

Module Contents

Classes

CanFlowStatus	Definition of Flow Status values.
CanSTminTranslator	Helper class that provides STmin values mapping.
CanFlowControlHandler	Helper class that provides utilities for Flow Control CAN Packets.

exception uds.can.flow_control.**UnrecognizedSTminWarning**

Bases: Warning

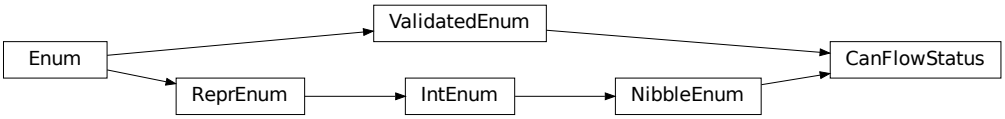
Warning about STmin value that is reserved and therefore not implemented.

Note: If you have a documentation that defines a meaning of *STmin* value for which this warning was raised, please create a request in issues management system and provide this documentation for us.

Initialize self. See help(type(self)) for accurate signature.

class uds.can.flow_control.**CanFlowStatus**

Bases: uds.utilities.NibbleEnum, uds.utilities.ValidatedEnum



Definition of Flow Status values.

Flow Status (FS) is a 4-bit value that enables controlling Consecutive Frames transmission.

Initialize self. See help(type(self)) for accurate signature.

ContinueToSend: *CanFlowStatus* = 0

Asks to resume Consecutive Frames transmission.

Wait: *CanFlowStatus* = 1

Asks to pause Consecutive Frames transmission.

Overflow: *CanFlowStatus* = 2

Asks to abort transmission of a diagnostic message.

class uds.can.flow_control.**CanSTminTranslator**

Helper class that provides STmin values mapping.

Separation Time minimum (STmin) informs about minimum time gap between a transmission of two following Consecutive Frames.

MAX_STMIN_TIME: uds.utilities.TimeMillisecondsAlias = 127

Maximal time value (in milliseconds) of STmin.

MIN_VALUE_MS_RANGE: int = 0

Minimal value of STmin in milliseconds range (raw value and time value in milliseconds are equal).

MAX_VALUE_MS_RANGE: int = 127

Maximal value of STmin in milliseconds range (raw value and time value in milliseconds are equal).

MIN_RAW_VALUE_100US_RANGE: int = 241

Minimal raw value of STmin in 100 microseconds range.

MAX_RAW_VALUE_100US_RANGE: int = 249

Maximal raw value of STmin in 100 microseconds range.

MIN_TIME_VALUE_100US_RANGE: uds.utilities.TimeMillisecondsAlias = 0.1

Minimal time value (in milliseconds) of STmin in 100 microseconds range.

MAX_TIME_VALUE_100US_RANGE: uds.utilities.TimeMillisecondsAlias = 0.9

Maximal time value (in milliseconds) of STmin in 100 microseconds range.

__FLOATING_POINT_ACCURACY: int = 10

Accuracy used for floating point values (rounding is necessary due to float operation in python).

classmethod decode(*raw_value*)

Map raw value of STmin into time value.

Note: According to ISO 15765-2, if a raw value of STmin that is not recognized by its recipient, then the longest STmin time value (0x7F = 127 ms) shall be used instead.

Parameters
raw_value (*int*) – Raw value of STmin.

Returns
STmin time in milliseconds.

Return type
uds.utilities.TimeMillisecondsAlias

classmethod encode(*time_value*)

Map time value of STmin into raw value.

Parameters
time_value (*uds.utilities.TimeMillisecondsAlias*) – STmin time in milliseconds.

Raises

- **TypeError** – Provided value is not time in milliseconds.

- **ValueError** – Value out of supported range.

Returns
Raw value of STmin.

Return type
int

classmethod is_time_value(*value*)

Check if provided value is a valid time value of STmin.

Parameters
value (*uds.utilities.TimeMillisecondsAlias*) – Value to check.

Returns
True if provided value is a valid time value of STmin, False otherwise.

Return type
bool

classmethod _is_ms_value(*value*)

Check if provided argument is STmin time value in milliseconds.

Parameters
value (*uds.utilities.TimeMillisecondsAlias*) – Value to check.

Returns
True if provided valid value of STmin time in milliseconds, False otherwise.

Return type
bool

classmethod _is_100us_value(*value*)

Check if provided argument is STmin time value in 100 microseconds.

Parameters
value (*uds.utilities.TimeMillisecondsAlias*) – Value to check.

Returns
True if provided valid value of STmin time in 100 microseconds, False otherwise.

Return type
bool

class uds.can.flow_control.**CanFlowControlHandler**

Helper class that provides utilities for Flow Control CAN Packets.

FLOW_CONTROL_N_PCI: int = 3

N_PCI value of Flow Control.

FS_BYTES_USED: int = 3

Number of CAN Frame data bytes used to carry CAN Packet Type, Flow Status, Block Size and STmin.

BS_BYTE_POSITION: int = 1

Position of a data byte with *Block Size* parameter.

STMIN_BYTE_POSITION: int = 2

Position of a data byte with *STmin* parameter.

classmethod create_valid_frame_data(*, *addressing_format*, *flow_status*, *block_size=None*, *st_min=None*, *dlc=None*, *filler_byte=DEFAULT_FILLER_BYTE*, *target_address=None*, *address_extension=None*)

Create a data field of a CAN frame that carries a valid Flow Control packet.

Note: This method can only be used to create a valid (compatible with ISO 15765 - Diagnostic on CAN) output. Use `create_any_frame_data()` to create data bytes for a Flow Control with any (also incompatible with ISO 15765) parameters values.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN addressing format used by a considered Flow Control.
- **flow_status** (CanFlowStatus) – Value of Flow Status parameter.
- **block_size** (*Optional[int]*) – Value of Block Size parameter. This parameter is only required with ContinueToSend Flow Status, leave None otherwise.
- **st_min** (*Optional[int]*) – Value of Separation Time minimum (STmin) parameter. This parameter is only required with ContinueToSend Flow Status, leave None otherwise.
- **dlc** (*Optional[int]*) – DLC value of a CAN frame that carries a considered CAN Packet.
  - None - use CAN Data Frame Optimization (CAN ID value will be automatically determined)
  - int type value - DLC value to set. CAN Data Padding will be used to fill the unused data bytes.
- **filler_byte** (*int*) – Filler Byte value to use for CAN Frame Data Padding.
- **target_address** (*Optional[int]*) – Target Address value carried by this CAN Packet. The value must only be provided if *addressing_format* uses Target Address parameter.
- **address_extension** (*Optional[int]*) – Address Extension value carried by this CAN packet. The value must only be provided if *addressing_format* uses Address Extension parameter.

Raises

InconsistentArgumentsError – Invalid DLC value was provided.

Returns

Raw bytes of CAN frame data for the provided Flow Control packet information.

Return type

uds.utilities.RawBytesListAlias

classmethod create_any_frame_data(*, *addressing_format*, *flow_status*, *dlc*, *block_size=None*, *st_min=None*, *filler_byte=DEFAULT_FILLER_BYTE*, *target_address=None*, *address_extension=None*)

Create a data field of a CAN frame that carries a Flow Control packet.

Note: You can use this method to create Flow Control data bytes with any (also inconsistent with ISO 15765) parameters values. It is recommended to use `create_valid_frame_data()` to create data bytes for a Flow Control with valid (compatible with ISO 15765) parameters values.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN addressing format used by a considered Flow Control.
- **flow_status** (CanFlowStatus) – Value of Flow Status parameter.
- **st_min** (*Optional[int]*) – Value of Separation Time minimum (STmin) parameter. Leave None to not insert this parameter in a Flow Control data bytes.
- **block_size** (*Optional[int]*) – Value of Block Size parameter. Leave None to not insert this parameter in a Flow Control data bytes.
- **dlc** (*int*) – DLC value of a CAN frame that carries a considered CAN Packet.
- **filler_byte** (*int*) – Filler Byte value to use for CAN Frame Data Padding.
- **target_address** (*Optional[int]*) – Target Address value carried by this CAN Packet. The value must only be provided if *addressing_format* uses Target Address parameter.
- **address_extension** (*Optional[int]*) – Address Extension value carried by this CAN packet. The value must only be provided if *addressing_format* uses Address Extension parameter.

Raises
InconsistentArgumentsError – DLC value is too small.

Returns
Raw bytes of CAN frame data for the provided Flow Control packet information.

Return type
uds.utilities.RawBytesListAlias

classmethod is_flow_control(*addressing_format*, *raw_frame_data*)

Check if provided data bytes encodes a Flow Control packet.

> **Warning:** The method does not validate the content of the provided frame data bytes. Only, *CAN Packet Type (N_PCI)* parameter is checked whether contain Flow Control N_PCI value.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN Addressing Format used.
- **raw_frame_data** (*uds.utilities.RawBytesAlias*) – Raw data bytes of a CAN frame to check.

Returns
True if provided data bytes carries Flow Control, False otherwise.

Return type
bool

classmethod decode_flow_status(*addressing_format*, *raw_frame_data*)

Extract Flow Status value from Flow Control data bytes.

> **Warning:** The method does not validate the content of the provided frame data bytes. There is no guarantee of the proper output when frame data in invalid format (incompatible with ISO 15765) is provided.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN Addressing Format used.
- **raw_frame_data** (*uds.utilities.RawBytesAlias*) – Raw data bytes of a considered CAN frame.

Raises

ValueError – Provided frame data of a CAN frames does not carry a Flow Control CAN packet.

Returns

Flow Status value carried by a considered Flow Control.

Return type

CanFlowStatus

classmethod decode_block_size(*addressing_format*, *raw_frame_data*)

Extract Block Size value from Flow Control data bytes.

> **Warning:** The method does not validate the content of the provided frame data bytes. There is no guarantee of the proper output when frame data in invalid format (incompatible with ISO 15765) is provided.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN Addressing Format used.
- **raw_frame_data** (*uds.utilities.RawBytesAlias*) – Raw data bytes of a considered CAN frame.

Raises

ValueError – Provided frame data of a CAN frames does not carry a Flow Control CAN packet with Continue To Send Flow Status.

Returns

Block Size value carried by a considered Flow Control.

Return type

int

classmethod decode_st_min(*addressing_format*, *raw_frame_data*)

Extract STmin value from Flow Control data bytes.

> **Warning:** The method does not validate the content of the provided frame data bytes. There is no guarantee of the proper output when frame data in invalid format (incompatible with ISO 15765) is provided.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN Addressing Format used.
- **raw_frame_data** (*uds.utilities.RawBytesAlias*) – Raw data bytes of a considered CAN frame.

Raises
ValueError – Provided frame data of a CAN frames does not carry a Flow Control CAN packet with Continue To Send Flow Status.

Returns
Separation Time minimum (STmin) value carried by a considered Flow Control.

Return type
int

classmethod get_min_dlc(*addressing_format*)

Get the minimum value of a CAN frame DLC to carry a Flow Control packet.

Parameters
addressing_format (uds.can.addressing_format.CanAddressingFormat) – CAN addressing format that considered CAN packet uses.

Returns
The lowest value of DLC that enables to fit in provided Flow Control packet data.

Return type
int

classmethod validate_frame_data(*addressing_format*, *raw_frame_data*)

Validate whether data field of a CAN Packet carries a properly encoded Flow Control.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN Addressing Format used.
- **raw_frame_data** (*uds.utilities.RawBytesAlias*) – Raw data bytes of a CAN frame to validate.

Raises
ValueError – Provided frame data of a CAN frames does not carry a properly encoded Flow Control CAN packet.

Return type
None

classmethod __encode_valid_flow_status(*flow_status*, *block_size=None*, *st_min=None*, *filler_byte=DEFAULT_FILLER_BYTE*)

Create Flow Control data bytes with CAN Packet Type and Flow Status, Block Size and STmin parameters.

Note: This method can only be used to create a valid (compatible with ISO 15765 - Diagnostic on CAN) output.

Parameters

- **flow_status** (CanFlowStatus) – Value of Flow Status parameter.
- **block_size** (*Optional[int]*) – Value of Block Size parameter. This parameter is only required with ContinueToSend Flow Status, leave None otherwise.
- **st_min** (*Optional[int]*) – Value of Separation Time minimum (STmin) parameter. This parameter is only required with ContinueToSend Flow Status, leave None otherwise.
- **filler_byte** (*int*) – Filler Byte value to use for CAN Frame Data Padding.

Returns
Flow Control data bytes with CAN Packet Type and Flow Status, Block Size and STmin parameters.

Return type
uds.utilities.RawBytesListAlias

classmethod __encode_any_flow_status(*flow_status*, *block_size=None*, *st_min=None*)

Create Flow Control data bytes with CAN Packet Type and Flow Status, Block Size and STmin parameters.

Note: This method can be used to create any (also incompatible with ISO 15765 - Diagnostic on CAN) output.

Parameters

- **flow_status** (*int*) – Value of Flow Status parameter.
- **block_size** (*Optional[int]*) – Value of Block Size parameter. Leave None to skip the Block Size byte in the output.
- **st_min** (*Optional[int]*) – Value of Separation Time minimum (STmin) parameter. Leave None to skip the STmin byte in the output.

Returns
Flow Control data bytes with CAN Packet Type and Flow Status, Block Size and STmin parameters. Some of the parameters might be missing if certain arguments were provided.

Return type
uds.utilities.RawBytesListAlias

uds.can.frame_fields

Implementation for CAN frame fields that are influenced by UDS.

Handlers for *CAN Frame* fields:

- CAN Identifier
- DLC
- Data

Module Contents

Classes

CanIdHandler	Helper class that provides utilities for CAN Identifier field.
CanDlcHandler	Helper class that provides utilities for CAN Data Length Code field.

Attributes

DEFAULT_FILLER_BYTE	Default value of Filler Byte.

uds.can.frame_fields.**DEFAULT_FILLER_BYTE: int = 204**

Default value of Filler Byte. Filler Bytes are used for *CAN Frame Data Padding*. .. note:: The value is specified by ISO 15765-2:2016 (chapter 10.4.2.1).

class uds.can.frame_fields.**CanIdHandler**

Helper class that provides utilities for CAN Identifier field.

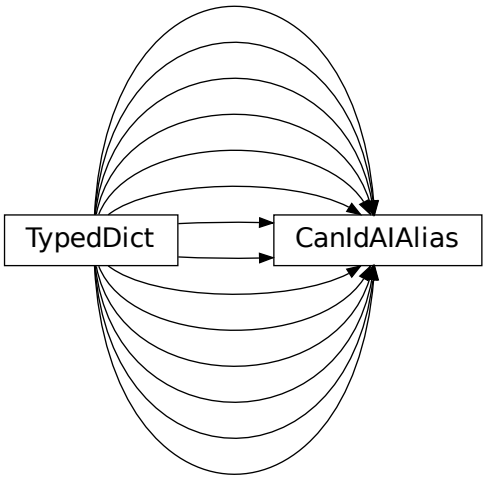
CAN Identifier (CAN ID) is a CAN frame field that informs about a sender and a content of CAN frames.

CAN bus supports two formats of CAN ID:

- Standard (11-bit Identifier)
- Extended (29-bit Identifier)

class CanIdAIAlias

Bases: TypedDict



Alias of *Addressing Information* that is carried by CAN Identifier.

Initialize self. See help(type(self)) for accurate signature.

addressing_type: uds.transmission_attributes.AddressingType | None

target_address: int | None

source_address: int | None

MIN_STANDARD_VALUE: int = 0

Minimum value of Standard (11-bit) CAN ID.

MAX_STANDARD_VALUE: int

Maximum value of Standard (11-bit) CAN ID.

MIN_EXTENDED_VALUE: int

Minimum value of Extended (29-bit) CAN ID.

MAX_EXTENDED_VALUE: int

Maximum value of Extended (29-bit) CAN ID.

NORMAL_FIXED_PHYSICAL_ADDRESSING_OFFSET: int = 416940032

Minimum value of physically addressed CAN ID in Normal Fixed Addressing format.

NORMAL_FIXED_FUNCTIONAL_ADDRESSING_OFFSET: int = 417005568

Minimum value of functionally addressed CAN ID in Normal Fixed Addressing format.

MIXED_29BIT_PHYSICAL_ADDRESSING_OFFSET: int = 416153600

Minimum value of physically addressed CAN ID in Mixed 29-bit Addressing format.

MIXED_29BIT_FUNCTIONAL_ADDRESSING_OFFSET: int = 416088064

Minimum value of functionally addressed CAN ID in Mixed 29-bit Addressing format.

ADDRESSING_TYPE_NAME: str = 'addressing_type'

Name of *Addressing Type* parameter in Addressing Information.

TARGET_ADDRESS_NAME: str = 'target_address'

Name of Target Address parameter in Addressing Information.

SOURCE_ADDRESS_NAME: str = 'source_address'

Name of Source Address parameter in Addressing Information.

classmethod decode_can_id(*addressing_format*, *can_id*)

Extract Addressing Information out of CAN ID.

> **Warning:** This methods might not extract any Addressing Information from the provided CAN ID as some of these information are system specific.
>
> For example, Addressing Type (even though it always depends on CAN ID value) will not be decoded when either Normal 11bit, Extended or Mixed 11bit addressing format is used as the Addressing Type (in such case) depends on system specific behaviour.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – Addressing format used.
- **can_id** (*int*) – CAN ID from which Addressing Information to be extracted.

Raises

NotImplementedError – There is missing implementation for the provided Addressing Format. Please create an issue in our Issues Tracking System with detailed description if you face this error.

Returns

Dictionary with Addressing Information decoded out of the provided CAN ID.

Return type

CanIdAIAlias

classmethod decode_normal_fixed_addressed_can_id(*can_id*)

Extract Addressing Information out of CAN ID for Normal Fixed CAN Addressing format.

Parameters
can_id (*int*) – CAN ID from which Addressing Information to be extracted.

Raises

- **ValueError** – Provided CAN ID is not compatible with Normal Fixed Addressing format.
- **NotImplementedError** – There is missing implementation for the provided CAN ID. Please create an issue in our Issues Tracking System with detailed description if you face this error.

Returns
Dictionary with Addressing Information decoded out of the provided CAN ID.

Return type
CanIdAIAlias

classmethod decode_mixed_addressed_29bit_can_id(*can_id*)

Extract Addressing Information out of CAN ID for Mixed 29-bit CAN Addressing format.

Parameters
can_id (*int*) – CAN ID from which Addressing Information to be extracted.

Raises

- **ValueError** – Provided CAN ID is not compatible with Mixed 29-bit Addressing format.
- **NotImplementedError** – There is missing implementation for the provided CAN ID. Please create an issue in our Issues Tracking System with detailed description if you face this error.

Returns
Dictionary with Addressing Information decoded out of the provided CAN ID.

Return type
CanIdAIAlias

classmethod encode_normal_fixed_addressed_can_id(*addressing_type*, *target_address*, *source_address*)

Generate CAN ID value for Normal Fixed CAN Addressing format.

Parameters

- **addressing_type** (*uds.transmission_attributes.AddressingType*) – Addressing type used.
- **target_address** (*int*) – Target address value to use.
- **source_address** (*int*) – Source address value to use.

Raises
NotImplementedError – There is missing implementation for the provided Addressing Type. Please create an issue in our Issues Tracking System with detailed description if you face this error.

Returns
Value of CAN ID (compatible with Normal Fixed Addressing Format) that was generated from the provided values.

Return type
int

classmethod encode_mixed_addressed_29bit_can_id(*addressing_type*, *target_address*, *source_address*)

Generate CAN ID value for Mixed 29-bit CAN Addressing format.

Parameters

- **addressing_type** (*uds.transmission_attributes.AddressingType*) – Addressing type used.
- **target_address** (*int*) – Target address value to use.
- **source_address** (*int*) – Source address value to use.

Raises

NotImplementedError – There is missing implementation for the provided Addressing Type. Please create an issue in our Issues Tracking System with detailed description if you face this error.

Returns

Value of CAN ID (compatible with Mixed 29-bit Addressing Format) that was generated from the provided values.

Return type

int

classmethod is_compatible_can_id(*can_id*, *addressing_format*, *addressing_type=None*)

Check if the provided value of CAN ID is compatible with addressing format used.

Parameters

- **can_id** (*int*) – Value to check.
- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – Addressing format used.
- **addressing_type** (*Optional[uds.transmission_attributes.AddressingType]*) – Addressing type for which consistency check to be performed. Leave None to not perform consistency check with Addressing Type.

Raises

NotImplementedError – There is missing implementation for the provided Addressing Format. Please create an issue in our Issues Tracking System with detailed description if you face this error.

Returns

True if CAN ID value is compatible with provided addressing values, False otherwise.

Return type

bool

classmethod is_normal_11bit_addressed_can_id(*can_id*)

Check if the provided value of CAN ID is compatible with Normal 11-bit Addressing format.

Parameters

can_id (*int*) – Value to check.

Returns

True if value is a valid CAN ID for Normal 11-bit Addressing format, False otherwise.

Return type

bool

classmethod is_normal_fixed_addressed_can_id(*can_id*, *addressing_type=None*)

Check if the provided value of CAN ID is compatible with Normal Fixed Addressing format.

Parameters

- **can_id** (*int*) – Value to check.
- **addressing_type** (*Optional[uds.transmission_attributes.AddressingType]*) – Addressing type for which consistency check to be performed. Leave None to not perform consistency check with Addressing Type.

Returns

True if value is a valid CAN ID for Normal Fixed Addressing format and consistent with the provided Addressing Type, False otherwise.

Return type

bool

classmethod is_extended_addressed_can_id(*can_id*)

Check if the provided value of CAN ID is compatible with Extended Addressing format.

Parameters

can_id (*int*) – Value to check.

Returns

True if value is a valid CAN ID for Extended Addressing format, False otherwise.

Return type

bool

classmethod is_mixed_11bit_addressed_can_id(*can_id*)

Check if the provided value of CAN ID is compatible with Mixed 11-bit Addressing format.

Parameters

can_id (*int*) – Value to check.

Returns

True if value is a valid CAN ID for Mixed 11-bit Addressing format, False otherwise.

Return type

bool

classmethod is_mixed_29bit_addressed_can_id(*can_id*, *addressing_type=None*)

Check if the provided value of CAN ID is compatible with Mixed 29-bit Addressing format.

Parameters

- **can_id** (*int*) – Value to check.
- **addressing_type** (*Optional[uds.transmission_attributes.AddressingType]*) – Addressing type for which consistency check to be performed. Leave None to not perform consistency check with Addressing Type.

Returns

True if value is a valid CAN ID for Mixed 29-bit Addressing format and consistent with the provided Addressing Type, False otherwise.

Return type

bool

classmethod is_can_id(*value*)

Check if the provided value is either Standard (11-bit) or Extended (29-bit) CAN ID.

Parameters
value (*int*) – Value to check.

Returns
True if value is a valid CAN ID, False otherwise.

Return type
bool

classmethod is_standard_can_id(*can_id*)

Check if the provided value is Standard (11-bit) CAN ID.

Parameters
can_id (*int*) – Value to check.

Returns
True if value is a valid 11-bit CAN ID, False otherwise.

Return type
bool

classmethod is_extended_can_id(*can_id*)

Check if the provided value is Extended (29-bit) CAN ID.

Parameters
can_id (*int*) – Value to check.

Returns
True if value is a valid 29-bit CAN ID, False otherwise.

Return type
bool

classmethod validate_can_id(*value*, *extended_can_id=None*)

Validate whether provided value is either Standard or Extended CAN ID.

Parameters

- **value** (*int*) – Value to validate.
- **extended_can_id** (*Optional[bool]*) – Flag whether to perform consistency check with CAN ID format.
  - None - does not check the format of the value
  - True - verify that the value uses Extended (29-bit) CAN ID format
  - False - verify that the value uses Standard (11-bit) CAN ID format

Raises

- **TypeError** – Provided value is not int type.
- **ValueError** – Provided value is out of CAN Identifier values range.

Return type
None

class uds.can.frame_fields.**CanDlcHandler**

Helper class that provides utilities for CAN Data Length Code field.

CAN Data Length Code (DLC) is a CAN frame field that informs about number of data bytes carried by CAN frames.

CAN DLC supports two values ranges:

- 0x0-0x8 - linear range which is supported by CLASSICAL CAN and CAN FD
- 0x9-0xF - discrete range which is supported by CAN FD only

__DLC_VALUES: Tuple[int, Ellipsis]

__DATA_BYTES_NUMBERS: Tuple[int, Ellipsis] = (0, 1, 2, 3, 4, 5, 6, 7, 8, 12, 16, 20, 24, 32, 48, 64)

__DLC_MAPPING: Dict[int, int]

__DATA_BYTES_NUMBER_MAPPING: Dict[int, int]

__DLC_SPECIFIC_FOR_CAN_FD: Set[int]

MIN_DATA_BYTES_NUMBER: int

Minimum number of data bytes in a CAN frame.

MAX_DATA_BYTES_NUMBER: int

Maximum number of data bytes in a CAN frame.

MIN_DLC_VALUE: int

Minimum value of DLC parameter.

MAX_DLC_VALUE: int

Maximum value of DLC parameter.

MIN_BASE_UDS_DLC: int = 8

Minimum CAN DLC value that can be used for UDS communication. Lower values of DLC are only allowed when *CAN Frame Data Optimization* is used.

classmethod decode_dlc(*dlc*)

Map a value of CAN DLC into a number of data bytes.

Parameters
dlc (*int*) – Value of CAN DLC.

Returns
Number of data bytes in a CAN frame that is represented by provided DLC value.

Return type
int

classmethod encode_dlc(*data_bytes_number*)

Map a number of data bytes in a CAN frame into DLC value.

Parameters
data_bytes_number (*int*) – Number of data bytes in a CAN frame.

Returns
DLC value of a CAN frame that represents provided number of data bytes.

Return type
int

classmethod get_min_dlc(*data_bytes_number*)

Get a minimum value of CAN DLC that is required to carry the provided number of data bytes in a CAN frame.

Parameters
data_bytes_number (*int*) – Number of data bytes in a CAN frame.

Returns
Minimum CAN DLC value that is required to carry provided number of data bytes in a CAN frame.

Return type
int

`classmethod is_can_fd_specific_dlc(dlc)`

Check whether the provided DLC value is CAN FD specific.

Parameters
dlc (*int*) – Value of DLC to check.

Returns
True if provided DLC value is CAN FD specific, False otherwise.

Return type
bool

`classmethod validate_dlc(value)`

Validate whether the provided value is a valid value of CAN DLC.

Parameters
value (*int*) – Value to validate.

Raises

- **TypeError** – Provided values is not int type.
- **ValueError** – Provided value is not a valid DLC value.

Return type
None

`classmethod validate_data_bytes_number(value, exact_value=True)`

Validate whether provided value is a valid number of data bytes that might be carried a CAN frame.

Parameters

- **value** (*int*) – Value to validate.
- **exact_value** (*bool*) – Informs whether the value must be the exact number of data bytes in a CAN frame.
  - True - provided value must be the exact number of data bytes to be carried by a CAN frame.
  - False - provided value must be a number of data bytes that could be carried by a CAN frame (*CAN Frame Data Padding* is allowed).

Raises

- **TypeError** – Provided values is not int type.
- **ValueError** – Provided value is not number of data bytes that matches the criteria.

Return type
None

uds.can.mixed_addressing_information

Implementation of Mixed Addressing Information handlers.

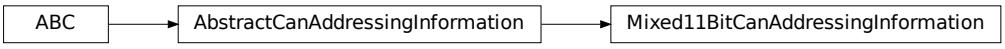
Module Contents

Classes

Mixed11BitCanAddressingInformation	Addressing Information of CAN Entity (either server or client) that uses Mixed 11-bit Addressing format.
Mixed29BitCanAddressingInformation	Addressing Information of CAN Entity (either server or client) that uses Mixed 29-bit Addressing format.

class uds.can.mixed_addressing_information.**Mixed11BitCanAddressingInformation**(*rx_physical*, *tx_physical*, *rx_functional*, *tx_functional*)

Bases: *uds.can.abstract_addressing_information.AbstractCanAddressingInformation*



Addressing Information of CAN Entity (either server or client) that uses Mixed 11-bit Addressing format.

Configure Addressing Information of a CAN Entity.

Parameters

- **rx_physical** (InputAIParamsAlias) – Addressing Information parameters used for incoming physically addressed communication.
- **tx_physical** (InputAIParamsAlias) – Addressing Information parameters used for outgoing physically addressed communication.
- **rx_functional** (InputAIParamsAlias) – Addressing Information parameters used for incoming functionally addressed communication.
- **tx_functional** (InputAIParamsAlias) – Addressing Information parameters used for outgoing functionally addressed communication.

property addressing_format: *uds.can.addressing_format.CanAddressingFormat*

CAN Addressing format used.

Return type
uds.can.addressing_format.CanAddressingFormat

AI_DATA_BYTES_NUMBER: int = 1

Number of CAN Frame data bytes that are used to carry Addressing Information.

classmethod validate_packet_ai(*addressing_type*, *can_id=None*, *target_address=None*, *source_address=None*, *address_extension=None*)

Validate Addressing Information parameters of a CAN packet that uses Mixed 11-bit Addressing format.

Parameters

- **addressing_type** (*uds.transmission_attributes.AddressingType*) – Addressing type to validate.
- **can_id** (*Optional[int]*) – CAN Identifier value to validate.
- **target_address** (*Optional[int]*) – Target Address value to validate.
- **source_address** (*Optional[int]*) – Source Address value to validate.
- **address_extension** (*Optional[int]*) – Address Extension value to validate.

Raises

- ***InconsistentArgumentsError*** – Provided CAN ID value is incompatible with Mixed 11-bit Addressing format.
- ***UnusedArgumentError*** – Provided parameter is not supported by this Addressing format.

Returns

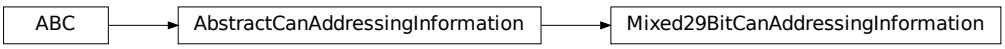
Normalized dictionary with the provided Addressing Information.

Return type

uds.can.abstract_addressing_information.PacketAIParamsAlias

class uds.can.mixed_addressing_information.**Mixed29BitCanAddressingInformation**(*rx_physical*, *tx_physical*, *rx_functional*, *tx_functional*)

Bases: *uds.can.abstract_addressing_information.AbstractCanAddressingInformation*



Addressing Information of CAN Entity (either server or client) that uses Mixed 29-bit Addressing format.

Configure Addressing Information of a CAN Entity.

Parameters

- **rx_physical** (InputAIParamsAlias) – Addressing Information parameters used for incoming physically addressed communication.
- **tx_physical** (InputAIParamsAlias) – Addressing Information parameters used for outgoing physically addressed communication.
- **rx_functional** (InputAIParamsAlias) – Addressing Information parameters used for incoming functionally addressed communication.
- **tx_functional** (InputAIParamsAlias) – Addressing Information parameters used for outgoing functionally addressed communication.

property addressing_format: *uds.can.addressing_format.CanAddressingFormat*

CAN Addressing format used.

Return type
uds.can.addressing_format.CanAddressingFormat

AI_DATA_BYTES_NUMBER: int = 1

Number of CAN Frame data bytes that are used to carry Addressing Information.

classmethod validate_packet_ai(*addressing_type*, *can_id=None*, *target_address=None*, *source_address=None*, *address_extension=None*)

Validate Addressing Information parameters of a CAN packet that uses Mixed 29-bit Addressing format.

Parameters

- **addressing_type** (*uds.transmission_attributes.AddressingType*) – Addressing type to validate.
- **can_id** (*Optional[int]*) – CAN Identifier value to validate.
- **target_address** (*Optional[int]*) – Target Address value to validate.
- **source_address** (*Optional[int]*) – Source Address value to validate.
- **address_extension** (*Optional[int]*) – Address Extension value to validate.

Raises
InconsistentArgumentsError – Provided Target Address, Source Address or CAN ID values are incompatible with each other or Mixed 29-bit Addressing format.

Returns
Normalized dictionary with the provided Addressing Information.

Return type
uds.can.abstract_addressing_information.PacketAIParamsAlias

uds.can.normal_addressing_information

Implementation of Normal Addressing Information handlers.

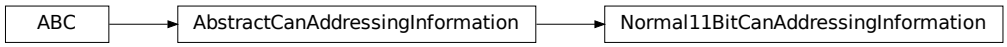
Module Contents

Classes

Normal11BitCanAddressingInformation	Addressing Information of CAN Entity (either server or client) that uses Normal 11-bit Addressing format.
NormalFixedCanAddressingInformation	Addressing Information of CAN Entity (either server or client) that uses Normal Fixed Addressing format.

class uds.can.normal_addressing_information.**Normal11BitCanAddressingInformation**(*rx_physical*, *tx_physical*, *rx_functional*, *tx_functional*)

Bases: *uds.can.abstract_addressing_information.AbstractCanAddressingInformation*



Addressing Information of CAN Entity (either server or client) that uses Normal 11-bit Addressing format.

Configure Addressing Information of a CAN Entity.

Parameters

- **rx_physical** (InputAIParamsAlias) – Addressing Information parameters used for incoming physically addressed communication.
- **tx_physical** (InputAIParamsAlias) – Addressing Information parameters used for outgoing physically addressed communication.
- **rx_functional** (InputAIParamsAlias) – Addressing Information parameters used for incoming functionally addressed communication.
- **tx_functional** (InputAIParamsAlias) – Addressing Information parameters used for outgoing functionally addressed communication.

property addressing_format: *uds.can.addressing_format.CanAddressingFormat*

CAN Addressing format used.

Return type
uds.can.addressing_format.CanAddressingFormat

AI_DATA_BYTES_NUMBER: int = 0

Number of CAN Frame data bytes that are used to carry Addressing Information.

classmethod validate_packet_ai(*addressing_type*, *can_id=None*, *target_address=None*, *source_address=None*, *address_extension=None*)

Validate Addressing Information parameters of a CAN packet that uses Normal 11-bit Addressing format.

Parameters

- **addressing_type** (*uds.transmission_attributes.AddressingType*) – Addressing type to validate.
- **can_id** (*Optional[int]*) – CAN Identifier value to validate.
- **target_address** (*Optional[int]*) – Target Address value to validate.
- **source_address** (*Optional[int]*) – Source Address value to validate.
- **address_extension** (*Optional[int]*) – Address Extension value to validate.

Raises

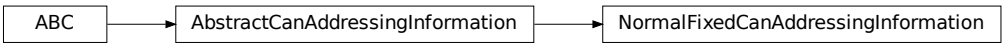
- *InconsistentArgumentsError* – Provided CAN ID value is incompatible with Normal 11-bit Addressing format.
- *UnusedArgumentError* – Provided parameter is not supported by this Addressing format.

Returns
Normalized dictionary with the provided Addressing Information.

Return type
uds.can.abstract_addressing_information.PacketAIParamsAlias

class uds.can.normal_addressing_information.**NormalFixedCanAddressingInformation**(*rx_physical*, *tx_physical*, *rx_functional*, *tx_functional*)

Bases: *uds.can.abstract_addressing_information.AbstractCanAddressingInformation*



Addressing Information of CAN Entity (either server or client) that uses Normal Fixed Addressing format.

Configure Addressing Information of a CAN Entity.

Parameters

- **rx_physical** (InputAIParamsAlias) – Addressing Information parameters used for incoming physically addressed communication.
- **tx_physical** (InputAIParamsAlias) – Addressing Information parameters used for outgoing physically addressed communication.
- **rx_functional** (InputAIParamsAlias) – Addressing Information parameters used for incoming functionally addressed communication.
- **tx_functional** (InputAIParamsAlias) – Addressing Information parameters used for outgoing functionally addressed communication.

property **addressing_format**: *uds.can.addressing_format.CanAddressingFormat*

CAN Addressing format used.

Return type
uds.can.addressing_format.CanAddressingFormat

AI_DATA_BYTES_NUMBER: int = 0

Number of CAN Frame data bytes that are used to carry Addressing Information.

classmethod **validate_packet_ai**(*addressing_type*, *can_id=None*, *target_address=None*, *source_address=None*, *address_extension=None*)

Validate Addressing Information parameters of a CAN packet that uses Normal Fixed Addressing format.

Parameters

- **addressing_type** (*uds.transmission_attributes.AddressingType*) – Addressing type to validate.
- **can_id** (*Optional[int]*) – CAN Identifier value to validate.
- **target_address** (*Optional[int]*) – Target Address value to validate.
- **source_address** (*Optional[int]*) – Source Address value to validate.
- **address_extension** (*Optional[int]*) – Address Extension value to validate.

Raises

- *InconsistentArgumentsError* – Provided Target Address, Source Address or CAN ID values are incompatible with each other or Normal Fixed Addressing format.

- *UnusedArgumentError* – Provided parameter is not supported by this Addressing format.

Returns
Normalized dictionary with the provided Addressing Information.

Return type
uds.can.abstract_addressing_information.PacketAIParamsAlias

uds.can.single_frame

Implementation specific for Single Frame CAN packets.

This module contains implementation specific for *Single Frame* packets - that includes *Single Frame Data Length (SF_DL)* parameter.

Module Contents

Classes

CanSingleFrameHandler	Helper class that provides utilities for Single Frame CAN Packets.

class uds.can.single_frame.**CanSingleFrameHandler**

Helper class that provides utilities for Single Frame CAN Packets.

SINGLE_FRAME_N_PCI: int = 0

N_PCI value of Single Frame.

MAX_DLC_VALUE_SHORT_SF_DL: int = 8

Maximum value of DLC for which short *Single Frame Data Length* format shall be used.

SHORT_SF_DL_BYTES_USED: int = 1

Number of CAN Frame data bytes used to carry CAN Packet Type and Single Frame Data Length (SF_DL). This value is valid only for the short format using DLC <= 8.

LONG_SF_DL_BYTES_USED: int = 2

Number of CAN Frame data bytes used to carry CAN Packet Type and Single Frame Data Length (SF_DL). This value is valid only for the long format using DLC > 8.

classmethod create_valid_frame_data(*, *addressing_format*, *payload*, *dlc=None*, *filler_byte=DEFAULT_FILLER_BYTE*, *target_address=None*, *address_extension=None*)

Create a data field of a CAN frame that carries a valid Single Frame packet.

Note: This method can only be used to create a valid (compatible with ISO 15765 - Diagnostic on CAN) output. Use *create_any_frame_data()* to create data bytes for a Single Frame with any (also incompatible with ISO 15765) parameters values.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN addressing format used by a considered Single Frame.

- **payload** (*uds.utilities.RawBytesAlias*) – Payload of a diagnostic message that is carried by a considered CAN packet.
- **dlc** (*Optional[int]*) – DLC value of a CAN frame that carries a considered CAN Packet.
  - None - use CAN Data Frame Optimization (CAN ID value will be automatically determined)
  - int type value - DLC value to set. CAN Data Padding will be used to fill the unused data bytes.
- **filler_byte** (*int*) – Filler Byte value to use for CAN Frame Data Padding.
- **target_address** (*Optional[int]*) – Target Address value carried by this CAN Packet. The value must only be provided if *addressing_format* uses Target Address parameter.
- **address_extension** (*Optional[int]*) – Address Extension value carried by this CAN packet. The value must only be provided if *addressing_format* uses Address Extension parameter.

Raises

InconsistentArgumentsError – Provided *payload* contains invalid number of bytes to fit it into a properly defined Single Frame data field.

Returns

Raw bytes of CAN frame data for the provided Single Frame packet information.

Return type

uds.utilities.RawBytesListAlias

classmethod create_any_frame_data(*, *addressing_format*, *payload*, *dlc*, *sf_dl_short*, *sf_dl_long=None*, *filler_byte=DEFAULT_FILLER_BYTE*, *target_address=None*, *address_extension=None*)

Create a data field of a CAN frame that carries a Single Frame packet.

Note: You can use this method to create Single Frame data bytes with any (also inconsistent with ISO 15765) parameters values. It is recommended to use *create_valid_frame_data()* to create data bytes for a Single Frame with valid (compatible with ISO 15765) parameters values.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN addressing format used by a considered Single Frame.
- **payload** (*uds.utilities.RawBytesAlias*) – Payload of a diagnostic message that is carried by a considered CAN packet.
- **dlc** (*int*) – DLC value of a CAN frame that carries a considered CAN Packet.
- **sf_dl_short** (*int*) – Value to put into a slot of Single Frame Data Length in short format.
- **sf_dl_long** (*Optional[int]*) – Value to put into a slot of Single Frame Data Length in long format. Leave None to use short (1-byte-long) format of Single Frame Data Length.
- **filler_byte** (*int*) – Filler Byte value to use for CAN Frame Data Padding.
- **target_address** (*Optional[int]*) – Target Address value carried by this CAN Packet. The value must only be provided if *addressing_format* uses Target Address parameter.

- **address_extension** (*Optional[int]*) – Address Extension value carried by this CAN packet. The value must only be provided if *addressing_format* uses Address Extension parameter.

Raises
InconsistentArgumentsError – Provided *payload* contains too many bytes to fit it into a Single Frame data field.

Returns
Raw bytes of CAN frame data for the provided Single Frame packet information.

Return type
uds.utilities.RawBytesListAlias

classmethod is_single_frame(*addressing_format*, *raw_frame_data*)

Check if provided data bytes encodes a Single Frame packet.

> **Warning:** The method does not validate the content (e.g. SF_DL parameter) of the provided frame data bytes. Only, *CAN Packet Type (N_PCI)* parameter is checked whether contain Single Frame N_PCI value.

Parameters

- **addressing_format** (*uds.can.addressing_format.CanAddressingFormat*) – CAN Addressing Format used.
- **raw_frame_data** (*uds.utilities.RawBytesAlias*) – Raw data bytes of a CAN frame to check.

Returns
True if provided data bytes carries Single Frame, False otherwise.

Return type
bool

classmethod decode_payload(*addressing_format*, *raw_frame_data*)

Extract diagnostic message payload from Single Frame data bytes.

> **Warning:** The method does not validate the content of the provided frame data bytes. There is no guarantee of the proper output when frame data in invalid format (incompatible with ISO 15765) is provided.

Parameters

- **addressing_format** (*uds.can.addressing_format.CanAddressingFormat*) – CAN Addressing Format used.
- **raw_frame_data** (*uds.utilities.RawBytesAlias*) – Raw data bytes of a considered CAN frame.

Returns
Payload bytes of a diagnostic message carried by a considered Single Frame.

Return type
uds.utilities.RawBytesListAlias

classmethod decode_sf_dl(*addressing_format*, *raw_frame_data*)

Extract a value of Single Frame Data Length from Single Frame data bytes.

> **Warning:** The method does not validate the content of the provided frame data bytes. There is no guarantee of the proper output when frame data in invalid format (incompatible with ISO 15765) is provided.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN Addressing Format used.
- **raw_frame_data** (*uds.utilities.RawBytesAlias*) – Raw data bytes of a considered CAN frame.

Raises

NotImplementedError – There is missing implementation for the provided Single Frame Data Length format. Please create an issue in our Issues Tracking System with detailed description if you face this error.

Returns

Extracted value of Single Frame Data Length.

Return type

int

classmethod get_min_dlc(*addressing_format*, *payload_length*)

Get the minimum value of a CAN frame DLC to carry a Single Frame packet.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN addressing format that considered CAN packet uses.
- **payload_length** (*int*) – Number of payload bytes that considered CAN packet carries.

Returns

The lowest value of DLC that enables to fit in provided Single Frame packet data.

Return type

int

classmethod get_max_payload_size(*addressing_format=None*, *dlc=None*)

Get the maximum size of a payload that can fit into Single Frame data bytes.

Parameters

- **addressing_format** (*Optional[*uds.can.addressing_format.CanAddressingFormat*]*) – CAN addressing format that considered CAN packet uses. Leave None to get the result for CAN addressing format that does not use data bytes for carrying addressing information.
- **dlc** (*Optional[int]*) – DLC value of a CAN frame that carries a considered CAN Packet. Leave None to get the result for the greatest possible DLC value.

Raises

InconsistentArgumentsError – Single Frame packet cannot use provided attributes according to ISO 15765.

Returns
The maximum number of payload bytes that could fit into a considered Single Frame.

Return type
int

classmethod get_sf_dl_bytes_number(*dlc*)

Get number of data bytes used for carrying CAN Packet Type and Single Frame Data Length parameters.

Parameters
dlc (*int*) – DLC value of a considered CAN frame.

Returns
The number of bytes used for carrying CAN Packet Type and Single Frame Data Length parameters.

Return type
int

classmethod validate_frame_data(*addressing_format*, *raw_frame_data*)

Validate whether data field of a CAN Packet carries a properly encoded Single Frame.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN Addressing Format used.
- **raw_frame_data** (*uds.utilities.RawBytesAlias*) – Raw data bytes of a CAN frame to validate.

Raises

- **ValueError** – Provided frame data of a CAN frames does not carry a Single Frame CAN packet.
- ***InconsistentArgumentsError*** – Provided frame data of a CAN frames does not carry a properly encoded Single Frame CAN packet.

Return type
None

classmethod validate_sf_dl(*sf_dl*, *dlc*, *addressing_format=None*)

Validate a value of Single Frame Data Length.

Parameters

- **sf_dl** (*int*) – Single Frame Data Length value to validate.
- **dlc** (*int*) – DLC value to validate.
- **addressing_format** (*Optional[*uds.can.addressing_format.CanAddressingFormat*]*) – Value of CAN Addressing Format to use for Single Frame Data Length value validation. Leave None if you do not want to validate whether payload can fit into a CAN Frame with considered DLC.

Raises

- **TypeError** – Provided value of Single Frame Data Length is not integer.
- **ValueError** – Provided value of Single Frame Data Length is too small.
- ***InconsistentArgumentsError*** – It is impossible for a Single Frame with provided DLC to contain as many payload bytes as the provided value of Single Frame Data Length.

Return type
None

classmethod __validate_payload_length(*payload_length*, *ai_data_bytes_number*)

Validate value of payload length.

Parameters

- **payload_length** (*int*) – Value to validate.
- **ai_data_bytes_number** (*int*) – Number of data byte that carry Addressing Information.

Raises

- **TypeError** – Provided value of payload length is not integer.
- **ValueError** – Provided value of payload length is less or equal to 0.
- ***InconsistentArgumentsError*** – Provided value of payload length is greater than maximum value.

Return type
None

classmethod __extract_sf_dl_data_bytes(*addressing_format*, *raw_frame_data*)

Extract data bytes that carries CAN Packet Type and Single Frame Data Length parameters.

> **Warning:** This method does not check whether provided *raw_frame_data* actually contains Single Frame.

Parameters

- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – CAN Addressing Format used.
- **raw_frame_data** (*uds.utilities.RawBytesAlias*) – Raw data bytes of a considered CAN frame.

Returns
Extracted data bytes with CAN Packet Type and Single Frame Data Length parameters.

Return type
uds.utilities.RawBytesListAlias

classmethod __encode_valid_sf_dl(*sf_dl*, *dlc*, *addressing_format*)

Create Single Frame data bytes with CAN Packet Type and Single Frame Data Length parameters.

Note: This method can only be used to create a valid (compatible with ISO 15765 - Diagnostic on CAN) output.

Parameters

- **sf_dl** (*int*) – Number of payload bytes carried by a considered Single Frame.
- **dlc** (*int*) – DLC value of a CAN Frame to carry this information.
- **addressing_format** (uds.can.addressing_format.CanAddressingFormat) – Value of CAN Addressing Format to use for Single Frame Data Length value validation.

Returns
Single Frame data bytes containing CAN Packet Type and Single Frame Data Length parameters.

Return type
uds.utilities.RawBytesListAlias

classmethod __encode_any_sf_dl(*sf_dl_short=0*, *sf_dl_long=None*)

Create Single Frame data bytes with CAN Packet Type and Single Frame Data Length parameters.

Note: This method can be used to create any (also incompatible with ISO 15765 - Diagnostic on CAN) output.

Parameters

- **sf_dl_short** (*int*) – Value to put into a slot of Single Frame Data Length in short format.
- **sf_dl_long** (*Optional[int]*) – Value to put into a slot of Single Frame Data Length in long format. Leave None to use short (1-byte-long) format of Single Frame Data Length.

Returns
Single Frame data bytes containing CAN Packet Type and Single Frame Data Length parameters.

Return type
uds.utilities.RawBytesListAlias

uds.message

A subpackage with tools for handling diagnostic message.

It provides tools for:

- creating new diagnostic message
- storing historic information about diagnostic message that were either received or transmitted
- Service Identifiers (SID) definition
- Negative Response Codes (NRC) definition

Submodules

uds.message.nrc

Module with an entire Negative Response Code (NRC) data parameters implementation.

Note: Explanation of *NRC* values meaning is located in appendix A1 of ISO 14229-1 standard.

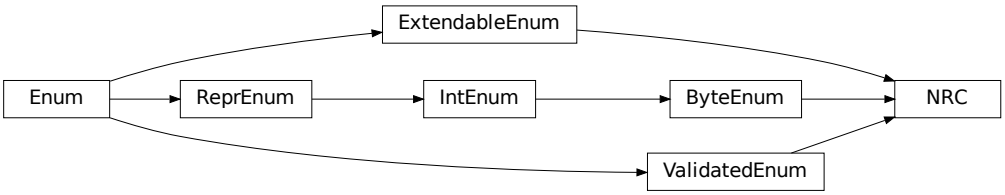
Module Contents

Classes

NRC	Negative Response Codes (NRC) values.

class uds.message.nrc.**NRC**

Bases: uds.utilities.ByteEnum, uds.utilities.ValidatedEnum, uds.utilities.ExtendableEnum



Negative Response Codes (NRC) values.

Negative Response Code <knowledge-base-nrc> is a data parameter located in the last byte of a negative response message. NRC informs why a server is not sending a positive response message.

Initialize self. See help(type(self)) for accurate signature.

GeneralReject: *NRC* = 16

GeneralReject (0x10) NRC indicates that the requested action has been rejected by the server.

ServiceNotSupported: *NRC* = 17

ServiceNotSupported (0x11) NRC indicates that the requested action will not be taken because the server does not support the requested service.

SubFunctionNotSupported: *NRC* = 18

SubFunctionNotSupported (0x12) NRC indicates that the requested action will not be taken because the server does not support the service specific parameters of the request message.

IncorrectMessageLengthOrInvalidFormat: *NRC* = 19

IncorrectMessageLengthOrInvalidFormat (0x13) NRC indicates that the requested action will not be taken because the length of the received request message does not match the prescribed length for the specified service or the format of the parameters do not match the prescribed format for the specified service.

ResponseTooLong: *NRC* = 20

ResponseTooLong (0x14) NRC shall be reported by the server if the response to be generated exceeds the maximum number of bytes available by the underlying network layer. This could occur if the response message exceeds the maximum size allowed by the underlying transport protocol or if the response message exceeds the server buffer size allocated for that purpose.

BusyRepeatRequest: *NRC* = 33

BusyRepeatRequest (0x21) NRC indicates that the server is temporarily too busy to perform the requested operation. In this circumstance the client shall perform repetition of the "identical request message" or "another request message". The repetition of the request shall be delayed by a time specified in the respective implementation documents.

ConditionsNotCorrect: *NRC* = 34

ConditionsNotCorrect (0x22) NRC indicates that the requested action will not be taken because the server prerequisite conditions are not met.

RequestSequenceError: *NRC* = 36

RequestSequenceError (0x24) NRC indicates that the requested action will not be taken because the server expects a different sequence of request messages or message as sent by the client. This may occur when sequence sensitive requests are issued in the wrong order.

NoResponseFromSubnetComponent: *NRC* = 37

NoResponseFromSubnetComponent (0x25) NRC indicates that the server has received the request but the requested action could not be performed by the server as a subnet component which is necessary to supply the requested information did not respond within the specified time.

FailurePreventsExecutionOfRequestedAction: *NRC* = 38

FailurePreventsExecutionOfRequestedAction (0x26) NRC indicates that the requested action will not be taken because a failure condition, identified by a DTC (with at least one DTC status bit for TestFailed, Pending, Confirmed or TestFailedSinceLastClear set to 1), has occurred and that this failure condition prevents the server from performing the requested action.

RequestOutOfRange: *NRC* = 49

RequestOutOfRange (0x31) NRC indicates that the requested action will not be taken because the server has detected that the request message contains a parameter which attempts to substitute a value beyond its range of authority (e.g. attempting to substitute a data byte of 111 when the data is only defined to 100), or which attempts to access a DataIdentifier/RoutineIdentifer that is not supported or not supported in active session.

SecurityAccessDenied: *NRC* = 51

SecurityAccessDenied (0x33) NRC indicates that the requested action will not be taken because the server's security strategy has not been satisfied by the client.

AuthenticationRequired: *NRC* = 52

AuthenticationRequired (0x34) NRC indicates that the requested service will not be taken because the client has insufficient rights based on its Authentication state.

InvalidKey: *NRC* = 53

InvalidKey (0x35) NRC indicates that the server has not given security access because the key sent by the client did not match with the key in the server's memory. This counts as an attempt to gain security.

ExceedNumberOfAttempts: *NRC* = 54

ExceedNumberOfAttempts (0x36) NRC indicates that the requested action will not be taken because the client has unsuccessfully attempted to gain security access more times than the server's security strategy will allow.

RequiredTimeDelayNotExpired: *NRC* = 55

RequiredTimeDelayNotExpired (0x37) NRC indicates that the requested action will not be taken because the client's latest attempt to gain security access was initiated before the server's required timeout period had elapsed.

SecureDataTransmissionRequired: *NRC* = 56

SecureDataTransmissionRequired (0x38) NRC indicates that the requested service will not be taken because the requested action is required to be sent using a secured communication channel.

SecureDataTransmissionNotAllowed: *NRC* = 57

SecureDataTransmissionNotAllowed (0x39) NRC indicates that this message was received using the SecuredDataTransmission (SID 0x84) service. However, the requested action is not allowed to be sent using the SecuredDataTransmission (0x84) service.

SecureDataVerificationFailed: *NRC* = 58

SecureDataVerificationFailed (0x3A) NRC indicates that the message failed in the security sub-layer.

CertificateVerificationFailed_InvalidTimePeriod: *NRC* = 80

CertificateVerificationFailed_InvalidTimePeriod (0x50) NRC indicates that date and time of the server does not match the validity period of the Certificate.

CertificateVerificationFailed_InvalidSignature: *NRC* = 81

CertificateVerificationFailed_InvalidSignature (0x51) NRC indicates that signature of the Certificate could not be verified.

CertificateVerificationFailed_InvalidChainOfTrust: *NRC* = 82

CertificateVerificationFailed_InvalidChainOfTrust (0x52) NRC indicates that The Certificate could not be verified against stored information about the issuing authority.

CertificateVerificationFailed_InvalidType: *NRC* = 83

CertificateVerificationFailed_InvalidType (0x53) NRC indicates that the Certificate does not match the current requested use case.

CertificateVerificationFailed_InvalidFormat: *NRC* = 84

CertificateVerificationFailed_InvalidFormat (0x54) NRC indicates that the Certificate could not be evaluated because the format requirement has not been met.

CertificateVerificationFailed_InvalidContent: *NRC* = 85

CertificateVerificationFailed_InvalidContent (0x55) NRC indicates that the Certificate could not be verified because the content does not match.

CertificateVerificationFailed_InvalidScope: *NRC* = 86

CertificateVerificationFailed_InvalidScope (0x56) NRC indicates that the scope of the Certificate does not match the contents of the server.

CertificateVerificationFailed_InvalidCertificate: *NRC* = 87

CertificateVerificationFailed_InvalidCertificate (0x57) NRC indicates that the Certificate received from client is invalid, because the server has revoked access for some reason.

OwnershipVerificationFailed: *NRC* = 88

OwnershipVerificationFailed (0x58) NRC indicates that delivered Ownership does not match the provided challenge or could not verified with the own private key.

ChallengeCalculationFailed: *NRC* = 89

ChallengeCalculationFailed (0x59) NRC indicates that the challenge could not be calculated on the server side.

SettingAccessRightsFailed: *NRC* = 90

SettingAccessRightsFailed (0x5A) NRC indicates that the server could not set the access rights.

SessionKeyCreationOrDerivationFailed: *NRC* = 91

SessionKeyCreationOrDerivationFailed (0x5B) NRC indicates that the server could not create or derive a session key.

ConfigurationDataUsageFailed: *NRC* = 92

ConfigurationDataUsageFailed (0x5C) NRC indicates that the server could not work with the provided configuration data.

DeAuthenticationFailed: *NRC* = 93

DeAuthenticationFailed (0x5D) NRC indicates that DeAuthentication was not successful, server could still be unprotected.

UploadDownloadNotAccepted: *NRC* = 112

UploadDownloadNotAccepted (0x70) NRC indicates that an attempt to upload/download to a server's memory cannot be accomplished due to some fault conditions.

TransferDataSuspended: *NRC* = 113

TransferDataSuspended (0x71) NRC indicates that a data transfer operation was halted due to some fault. The active transferData sequence shall be aborted.

GeneralProgrammingFailure: *NRC* = 114

GeneralProgrammingFailure (0x72) NRC indicates that the server detected an error when erasing or programming a memory location in the permanent memory device (e.g. Flash Memory).

WrongBlockSequenceCounter: *NRC* = 115

WrongBlockSequenceCounter (0x73) NRC indicates that the server detected an error in the sequence of blockSequenceCounter values. Note that the repetition of a TransferData request message with a blockSequenceCounter equal to the one included in the previous TransferData request message shall be accepted by the server.

RequestCorrectlyReceived_ResponsePending: *NRC* = 120

RequestCorrectlyReceived_ResponsePending (0x78) NRC ndicates that the request message was received correctly, and that all parameters in the request message were valid (these checks can be delayed until after sending this NRC if executing the boot software), but the action to be performed is not yet completed and the server is not yet ready to receive another request. As soon as the requested service has been completed, the server shall send a positive response message or negative response message with a response code different from this.

SubFunctionNotSupportedInActiveSession: *NRC* = 126

SubFunctionNotSupportedInActiveSession (0x7E) NRC indicates that the requested action will not be taken because the server does not support the requested SubFunction in the session currently active. This NRC shall only be used when the requested SubFunction is known to be supported in another session, otherwise response code SubFunctionNotSupported shall be used.

ServiceNotSupportedInActiveSession: *NRC* = 127

ServiceNotSupportedInActiveSession (0x7F) NRC indicates that the requested action will not be taken because the server does not support the requested service in the session currently active. This NRC shall only be used when the requested service is known to be supported in another session, otherwise response code serviceNotSupported shall be used.

RpmTooHigh: *NRC* = 129

RpmTooHigh (0x81) NRC indicates that the requested action will not be taken because the server prerequisite condition for RPM is not met (current RPM is above a preprogrammed maximum threshold).

RpmTooLow: *NRC* = 130

RpmTooLow (0x82) NRC indicates that the requested action will not be taken because the server prerequisite condition for RPM is not met (current RPM is below a preprogrammed minimum threshold).

EngineIsRunning: *NRC* = 131

EngineIsRunning (0x83) NRC is required for those actuator tests which cannot be actuated while the Engine is running. This is different from RPM too high negative response, and shall be allowed.

EngineIsNotRunning: *NRC* = 132

EngineIsNotRunning (0x84) NRC is required for those actuator tests which cannot be actuated unless the Engine is running. This is different from RPM too low negative response, and shall be allowed.

EngineRunTimeTooLow: *NRC* = 133

EngineRunTimeTooLow (0x85) NRC indicates that the requested action will not be taken because the server

prerequisite condition for engine run time is not met (current engine run time is below a preprogrammed limit).

TemperatureTooHigh: *NRC* = 134

TemperatureTooHigh (0x86) NRC indicates that the requested action will not be taken because the serve prerequisite condition for temperature is not met (current temperature is above a preprogrammed maximum threshold).

TemperatureTooLow: *NRC* = 135

TemperatureTooLow (0x87) NRC indicates that the requested action will not be taken because the server prerequisite condition for temperature is not met (current temperature is below a preprogrammed minimum threshold).

VehicleSpeedTooHigh: *NRC* = 136

VehicleSpeedTooHigh (0x88) NRC indicates that the requested action will not be taken because the server prerequisite condition for vehicle speed is not met (current VS is above a preprogrammed maximum threshold).

VehicleSpeedTooLow: *NRC* = 137

VehicleSpeedTooLow (0x89) NRC indicates that the requested action will not be taken because the server prerequisite condition for vehicle speed is not met (current VS is below a preprogrammed minimum threshold).

ThrottleOrPedalTooHigh: *NRC* = 138

ThrottleOrPedalTooHigh (0x8A) NRC indicates that the requested action will not be taken because the server prerequisite condition for throttle/pedal position is not met (current throttle/pedal position is above a preprogrammed maximum threshold).

ThrottleOrPedalTooLow: *NRC* = 139

ThrottleOrPedalTooLow (0x8B) NRC indicates that the requested action will not be taken because the server prerequisite condition for throttle/pedal position is not met (current throttle/pedal position is below a preprogrammed minimum threshold).

TransmissionRangeNotInNeutral: *NRC* = 140

TransmissionRangeNotInNeutral (0x8C) NRC indicates that the requested action will not be taken because the server prerequisite condition for being in neutral is not met (current transmission range is not in neutral).

TransmissionRangeNotInGear: *NRC* = 141

TransmissionRangeNotInGear (0x8D) NRC indicates that the requested action will not be taken because the server prerequisite condition for being in gear is not met (current transmission range is not in gear).

BrakeSwitchOrSwitchesNotClosed: *NRC* = 143

BrakeSwitchOrSwitchesNotClosed (0x8F) NRC indicates that for safety reasons, this is required for certain tests before it begins, and shall be maintained for the entire duration of the test.

ShifterLeverNotInPark: *NRC* = 144

ShifterLeverNotInPark (0x90) NRC indicates that for safety reasons, this is required for certain tests before it begins, and shall be maintained for the entire duration of the test.

TorqueConvertClutchLocked: *NRC* = 145

TorqueConvertClutchLocked (0x91) RC indicates that the requested action will not be taken because the server prerequisite condition for torque converter clutch is not met (current torque converter clutch status above a preprogrammed limit or locked).

VoltageTooHigh: *NRC* = 146

VoltageTooHigh (0x92) NRC indicates that the requested action will not be taken because the server prerequisite condition for voltage at the primary pin of the server (ECU) is not met (current voltage is above a preprogrammed maximum threshold).

VoltageTooLow: *NRC* = 147

VoltageTooLow (0x93) NRC indicates that the requested action will not be taken because the server prerequisite condition for voltage at the primary pin of the server (ECU) is not met (current voltage is below a preprogrammed minimum threshold).

ResourceTemporarilyNotAvailable: *NRC* = 148

ResourceTemporarilyNotAvailable (0x94) NRC indicates that the server has received the request but the requested action could not be performed by the server because an application which is necessary to supply the requested information is temporality not available. This NRC is in general supported by each diagnostic service, as not otherwise stated in the data link specific implementation document, therefore it is not listed in the list of applicable response codes of the diagnostic services.

uds.message.service_identifiers

Service Identifier (SID) data parameter implementation.

Note: *Service Identifiers* values and their meanings are defined by ISO 14229-1 and SAE J1979 standards.

Module Contents

Classes

RequestSID	Request Service Identifier values.
ResponseSID	Response Service Identifier values.

Attributes

ALL_REQUEST_SIDS	Set with all possible values of Request SID data parameter according to SAE J1979 and ISO 14229 standards.
ALL_RESPONSE_SIDS	Set with all possible values of Response SID data parameter according to SAE J1979 and ISO 14229 standards.

uds.message.service_identifiers.**ALL_REQUEST_SIDS: uds.utilities.RawBytesSetAlias**

Set with all possible values of Request SID data parameter according to SAE J1979 and ISO 14229 standards.

uds.message.service_identifiers.**ALL_RESPONSE_SIDS: uds.utilities.RawBytesSetAlias**

Set with all possible values of Response SID data parameter according to SAE J1979 and ISO 14229 standards.

exception uds.message.service_identifiers.**UnrecognizedSIDWarning**

Bases: Warning

Warning about SID value that is legit but not recognized by the package.

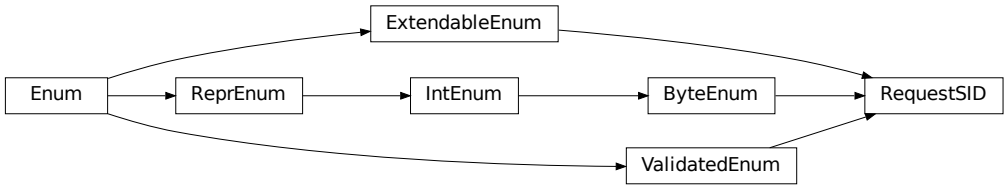
Note: If you want to register a SID value, you need to define members (for this SID) manually using *add_member()* method (on *RequestSID* and *ResponseSID* classes).

You can also create feature request in the UDS project issues management system to register a SID value (for which this warning was raised).

Initialize self. See help(type(self)) for accurate signature.

class uds.message.service_identifiers.**RequestSID**

Bases: uds.utilities.ByteEnum, uds.utilities.ValidatedEnum, uds.utilities.ExtendableEnum



Request Service Identifier values.

Note: Request *SID* is always the first payload byte of all request message.

Initialize self. See help(type(self)) for accurate signature.

DiagnosticSessionControl: *RequestSID* = 16

ECUReset: *RequestSID* = 17

SecurityAccess: *RequestSID* = 39

CommunicationControl: *RequestSID* = 40

Authentication: *RequestSID* = 41

TesterPresent: *RequestSID* = 62

ControlDTCSetting: *RequestSID* = 133

ResponseOnEvent: *RequestSID* = 134

LinkControl: *RequestSID* = 135

ReadDataByIdentifier: *RequestSID* = 34

ReadMemoryByAddress: *RequestSID* = 35

ReadScalingDataByIdentifier: *RequestSID* = 36

ReadDataByPeriodicIdentifier: *RequestSID* = 42

DynamicallyDefineDataIdentifier: *RequestSID* = 44

WriteDataByIdentifier: *RequestSID* = 46

WriteMemoryByAddress: *RequestSID* = 61

ClearDiagnosticInformation: *RequestSID* = 20

ReadDTCInformation: *RequestSID* = 25

InputOutputControlByIdentifier: *RequestSID* = 47

RoutineControl: *RequestSID* = 49

RequestDownload: *RequestSID* = 52

RequestUpload: *RequestSID* = 53

TransferData: *RequestSID* = 54

RequestTransferExit: *RequestSID* = 55

RequestFileTransfer: *RequestSID* = 56

SecuredDataTransmission: *RequestSID* = 132

classmethod is_request_sid(*value*)

Check whether given value is Service Identifier (SID).

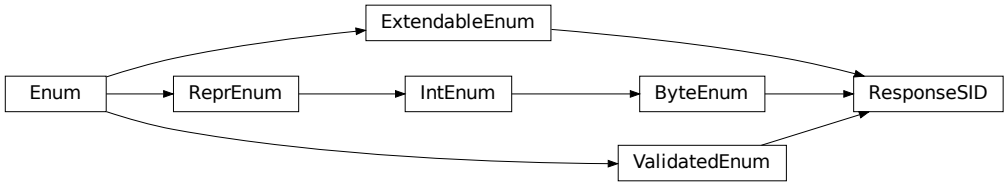
Parameters
value (*int*) – Value to check.

Returns
True if value is valid SID, else False.

Return type
bool

class uds.message.service_identifiers.**ResponseSID**

Bases: uds.utilities.ByteEnum, uds.utilities.ValidatedEnum, uds.utilities.ExtendableEnum



Response Service Identifier values.

Note: Response *SID* is always the first payload byte of all request message.

Warning: This enum contains multiple members (for all the services as *RequestSID*), but most of them are dynamically (implicitly) added and invisible in the documentation.

Initialize self. See help(type(self)) for accurate signature.

NegativeResponse: *ResponseSID* **= 127**

classmethod is_response_sid(*value*)

Check whether given value is Response Service Identifier (RSID).

Parameters
value (*int*) – Value to check.

Returns
True if value is valid RSID, else False.

Return type
bool

uds.message.uds_message

Module with common implementation of all diagnostic messages (requests and responses).

Diagnostic message are defined on upper layers of UDS OSI Model.

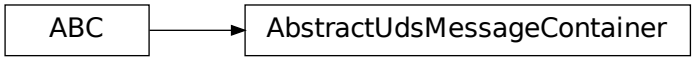
Module Contents

Classes

AbstractUdsMessageContainer	Abstract definition of a container with a diagnostic message information.
UdsMessage	Definition of a diagnostic message.
UdsMessageRecord	Storage for historic information about a diagnostic message that was either received or transmitted.

class uds.message.uds_message.**AbstractUdsMessageContainer**

Bases: abc.ABC



Abstract definition of a container with a diagnostic message information.

abstract property payload: uds.utilities.RawBytesTupleAlias

Raw payload bytes carried by this diagnostic message.

Return type
uds.utilities.RawBytesTupleAlias

abstract property addressing_type: uds.transmission_attributes.AddressingType

Addressing for which this diagnostic message is relevant.

Return type
uds.transmission_attributes.AddressingType

abstract __eq__(*other*)

Compare with other object.

Parameters
other (*object*) – Object to compare.

Returns
True if other object has the same type and carries the same diagnostic message, otherwise False.

Return type
bool

class uds.message.uds_message.**UdsMessage**(*payload*, *addressing_type*)

Bases: *AbstractUdsMessageContainer*

Definition of a diagnostic message.

Objects of this class act as a storage for all relevant attributes of a *diagnostic message*. Later on, such object might be used in a segmentation process or to transmit the message. Once a message is transmitted, its historic data would be stored in *UdsMessageRecord*.

Create a storage for a single diagnostic message.

Parameters

- **payload** (*uds.utilities.RawBytesAlias*) – Raw payload bytes carried by this diagnostic message.
- **addressing_type** (*uds.transmission_attributes.AddressingType*) – Addressing for which this diagnostic message is relevant.

property payload: uds.utilities.RawBytesTupleAlias

Raw payload bytes carried by this diagnostic message.

Return type
uds.utilities.RawBytesTupleAlias

property addressing_type: uds.transmission_attributes.AddressingType

Addressing for which this diagnostic message is relevant.

Return type
uds.transmission_attributes.AddressingType

__eq__(*other*)

Compare with other object.

Parameters
other (*object*) – Object to compare.

Returns
True if other object has the same type and carries the same diagnostic message, otherwise False.

Return type
bool

class uds.message.uds_message.**UdsMessageRecord**(*packets_records*)

Bases: *AbstractUdsMessageContainer*

Storage for historic information about a diagnostic message that was either received or transmitted.

Create a record of historic information about a diagnostic message that was either received or transmitted.

Parameters
packets_records (*uds.packet.PacketsRecordsSequence*) – Sequence (in transmission order) of UDS packets records that carried this diagnostic message.

property packets_records: uds.packet.PacketsRecordsTuple

Sequence (in transmission order) of UDS packets records that carried this diagnostic message.

UDS packets sequence is a complete sequence of packets that was exchanged during this diagnostic message transmission.

Return type
uds.packet.PacketsRecordsTuple

property payload: uds.utilities.RawBytesTupleAlias

Raw payload bytes carried by this diagnostic message.

Return type
uds.utilities.RawBytesTupleAlias

property addressing_type: uds.transmission_attributes.AddressingType

Addressing which was used to transmit this diagnostic message.

Return type
uds.transmission_attributes.AddressingType

property direction: uds.transmission_attributes.TransmissionDirection

Information whether this message was received or sent.

Return type
uds.transmission_attributes.TransmissionDirection

property transmission_start: datetime.datetime

Time stamp when transmission of this message was initiated.

It is determined by a moment of time when the first packet (that carried this message) was published to a bus (either received or transmitted).

Returns
Time stamp when transmission of this message was initiated.

Return type
datetime.datetime

property transmission_end: datetime.datetime

Time stamp when transmission of this message was completed.

It is determined by a moment of time when the last packet (that carried this message) was published to a bus (either received or transmitted).

Returns
Time stamp when transmission of this message was completed.

Return type
datetime.datetime

__eq__(*other*)

Compare with other object.

Parameters
other (*object*) – Object to compare.

Returns
True if other object has the same type and carries the same diagnostic message, otherwise False.

Return type
bool

static __validate_packets_records(*value*)

Validate whether the argument contains UDS Packets records.

Parameters
value (*uds.packet.PacketsRecordsSequence*) – Value to validate.

Raises

- **TypeError** – UDS Packet Records sequence is not list or tuple type.
- **ValueError** – At least one of UDS Packet Records sequence elements is not an object of AbstractUdsPacketRecord class.

Return type
None

uds.packet

A subpackage with tools for handling UDS packets.

It provides tools for:

- data definitions for various UDS packet fields (*N_AI*, *N_PCI*, *N_Data*)
- creating new packets
- storing historic information about packets that were either received or transmitted

Submodules

uds.packet.abstract_can_packet_container

Abstract definition of CAN packets container.

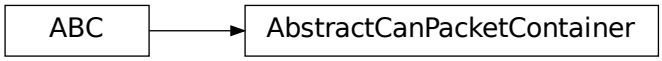
Module Contents

Classes

AbstractCanPacketContainer	Abstract definition of CAN Packets containers.

class uds.packet.abstract_can_packet_container.**AbstractCanPacketContainer**

Bases: abc.ABC



Abstract definition of CAN Packets containers.

abstract property raw_frame_data: uds.utilities.RawBytesTupleAlias

Raw data bytes of a CAN frame that carries this CAN packet.

Return type
uds.utilities.RawBytesTupleAlias

abstract property can_id: int

CAN Identifier (CAN ID) of a CAN Frame that carries this CAN packet.

Return type
int

abstract property addressing_format: uds.can.CanAddressingFormat

CAN addressing format used by this CAN packet.

Return type
uds.can.CanAddressingFormat

abstract property addressing_type: uds.transmission_attributes.AddressingType

Addressing type for which this CAN packet is relevant.

Return type
uds.transmission_attributes.AddressingType

property dlc: int

Value of Data Length Code (DLC) of a CAN Frame that carries this CAN packet.

Return type
int

property packet_type: *uds.packet.can_packet_type.CanPacketType*

Type (N_PCI value) of this CAN packet.

Return type
uds.packet.can_packet_type.CanPacketType

property target_address: int | None

Target Address (TA) value of this CAN Packet.

Target Address value is used with following *addressing formats*:

- *Normal Fixed Addressing*
- *Extended Addressing*
- *Mixed 29-bit Addressing*

None in other cases.

Return type
Optional[int]

property source_address: int | None

Source Address (SA) value of this CAN Packet.

Source Address value is used with following *addressing formats*:

- *Normal Fixed Addressing*
- *Mixed 29-bit Addressing*

None in other cases.

Return type
Optional[int]

property address_extension: int | None

Address Extension (AE) value of this CAN Packet.

Address Extension is used with following *addressing formats*:

- *Mixed Addressing* - either: - *Mixed 11-bit Addressing* - *Mixed 29-bit Addressing*

None in other cases.

Return type
Optional[int]

property data_length: int | None

Payload bytes number of a diagnostic message that is carried by this CAN packet.

Data length is only provided by packets of following types:

- *Single Frame* - *Single Frame Data Length*
- *First Frame* - *First Frame Data Length*

None in other cases.

Return type
Optional[int]

property sequence_number: int | None

Sequence Number carried by this CAN packet.

***Sequence Number* is only provided by packets of following types:**

- *Consecutive Frame*

None in other cases.

Return type
Optional[int]

property payload: uds.utilities.RawBytesTupleAlias | None

Diagnostic message payload carried by this CAN packet.

Payload is only provided by packets of following types:

- *Single Frame*
- *First Frame*
- *Consecutive Frame*

None in other cases.

> **Warning:** For *Consecutive Frames* this value might contain additional filler bytes (they are not part of diagnostic message payload) that were added during *CAN Frame Data Padding*. The presence of filler bytes in *Consecutive Frame* cannot be determined basing solely on the information contained in this packet object.

Return type
Optional[uds.utilities.RawBytesTupleAlias]

property flow_status: uds.can.CanFlowStatus | None

Flow Status carried by this CAN packet.

***Flow Status* is only provided by packets of following types:**

- *Flow Control*

None in other cases.

Return type
Optional[uds.can.CanFlowStatus]

property block_size: int | None

Block Size value carried by this CAN packet.

***Block Size* is only provided by packets of following types:**

- *Flow Control*

None in other cases.

Return type
Optional[int]

property st_min: int | None

Separation Time minimum (STmin) value carried by this CAN packet.

***STmin* is only provided by packets of following types:**

- *Flow Control*

None in other cases.

Return type
Optional[int]

get_addressing_information()

Get Addressing Information carried by this packet.

Returns
Addressing Information decoded from CAN ID and CAN Frame data of this packet.

Return type
uds.can.CanAddressingInformation.DecodedAIParamsAlias

uds.packet.abstract_packet

Abstract definition of UDS packets that is common for all bus types.

Module Contents

Classes

AbstractUdsPacketContainer	Abstract definition of a container with UDS Packet information.
AbstractUdsPacket	Abstract definition of UDS Packet (Network Protocol Data Unit - N_PDU).
AbstractUdsPacketRecord	Abstract definition of a storage for historic information about transmitted or received UDS Packet.

Attributes

PacketsContainersSequence	Alias for a sequence filled with packet or packet record object.
PacketsTuple	Alias for a packet objects tuple.
PacketsRecordsTuple	Alias for a packet record objects tuple.
PacketsRecordsSequence	Alias for a packet record objects sequence.

class uds.packet.abstract_packet.**AbstractUdsPacketContainer**

Bases: abc.ABC

Abstract definition of a container with UDS Packet information.

abstract property raw_frame_data: uds.utilities.RawBytesTupleAlias

Raw data bytes of a frame that carries this packet.

Return type
uds.utilities.RawBytesTupleAlias

abstract property addressing_type: *uds.transmission_attributes.addressing.AddressingType*

Addressing for which this packet is relevant.

Return type
uds.transmission_attributes.addressing.AddressingType

abstract property packet_type: *uds.packet.abstract_packet_type.AbstractUdsPacketType*

Type (N_PCI value) of this UDS packet.

Return type
uds.packet.abstract_packet_type.AbstractUdsPacketType

abstract property data_length: int | None

Payload bytes number of a diagnostic message which was carried by this packet.

Return type
Optional[int]

abstract property payload: uds.utilities.RawBytesTupleAlias | None

Raw payload bytes of a diagnostic message that are carried by this packet.

Return type
Optional[uds.utilities.RawBytesTupleAlias]

class uds.packet.abstract_packet.**AbstractUdsPacket**

Bases: *AbstractUdsPacketContainer*

Abstract definition of UDS Packet (Network Protocol Data Unit - N_PDU).

abstract property raw_frame_data: uds.utilities.RawBytesTupleAlias

Raw data bytes of a frame that carries this packet.

Return type
uds.utilities.RawBytesTupleAlias

abstract property addressing_type:
uds.transmission_attributes.addressing.AddressingType

Addressing for which this packet is relevant.

Return type
uds.transmission_attributes.addressing.AddressingType

abstract property packet_type:
uds.packet.abstract_packet_type.AbstractUdsPacketType

Type (N_PCI value) of this UDS packet.

Return type
uds.packet.abstract_packet_type.AbstractUdsPacketType

abstract property data_length: int | None

Payload bytes number of a diagnostic message which was carried by this packet.

Return type
Optional[int]

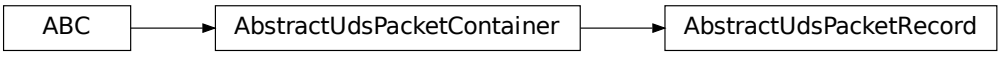
abstract property payload: uds.utilities.RawBytesTupleAlias | None

Raw payload bytes of a diagnostic message that are carried by this packet.

Return type
Optional[uds.utilities.RawBytesTupleAlias]

class uds.packet.abstract_packet.**AbstractUdsPacketRecord**(*frame*, *direction*, *transmission_time*)

Bases: *AbstractUdsPacketContainer*



Abstract definition of a storage for historic information about transmitted or received UDS Packet.

Create a record of historic information about a packet that was either received or transmitted.

Parameters

- **frame** (*Any*) – Frame that carried this UDS packet.
- **direction** (uds.transmission_attributes.transmission_direction.TransmissionDirection) – Information whether this packet was transmitted or received.
- **transmission_time** (*datetime.datetime*) – Time stamp when this packet was fully transmitted on a bus.

property frame: Any

Frame that carried this packet.

Return type
Any

property direction:
uds.transmission_attributes.transmission_direction.TransmissionDirection

Information whether this packet was transmitted or received.

Return type
uds.transmission_attributes.transmission_direction.TransmissionDirection

property transmission_time: datetime.datetime

Time when this packet was fully transmitted on a bus.

Return type
datetime.datetime

abstract property raw_frame_data: uds.utilities.RawBytesTupleAlias

Raw data bytes of a frame that carries this packet.

Return type
uds.utilities.RawBytesTupleAlias

abstract property addressing_type:
uds.transmission_attributes.addressing.AddressingType

Addressing for which this packet is relevant.

Return type
uds.transmission_attributes.addressing.AddressingType

abstract property packet_type:
uds.packet.abstract_packet_type.AbstractUdsPacketType

Type (N_PCI value) of this UDS packet.

Return type
uds.packet.abstract_packet_type.AbstractUdsPacketType

abstract property data_length: int | None

Payload bytes number of a diagnostic message which was carried by this packet.

Return type
Optional[int]

abstract property payload: uds.utilities.RawBytesTupleAlias | None

Raw payload bytes of a diagnostic message that are carried by this packet.

Return type
Optional[uds.utilities.RawBytesTupleAlias]

abstract static _validate_frame(*value*)

Validate a frame argument.

Parameters
value (*Any*) – Value to validate.

Raises

- **TypeError** – The frame argument has unsupported.
- **ValueError** – Some attribute of the frame argument is missing or its value is unexpected.

Return type
None

uds.packet.abstract_packet.**PacketsContainersSequence**
Alias for a sequence filled with packet or packet record object.

uds.packet.abstract_packet.**PacketsTuple**
Alias for a packet objects tuple.

uds.packet.abstract_packet.**PacketsRecordsTuple**
Alias for a packet record objects tuple.

uds.packet.abstract_packet.**PacketsRecordsSequence**
Alias for a packet record objects sequence.

uds.packet.abstract_packet_type

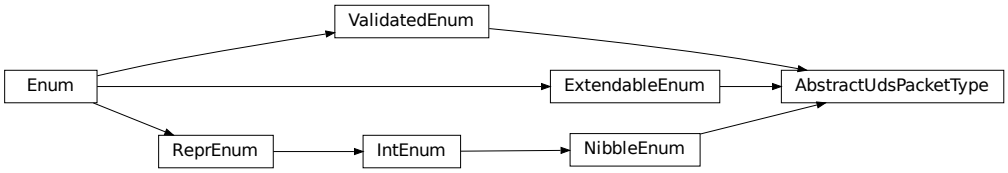
Common (abstract) implementation of UDS Packet Types.

Module Contents

Classes

AbstractUdsPacketType	Abstract definition of UDS packet type.

class uds.packet.abstract_packet_type.**AbstractUdsPacketType**
Bases: uds.utilities.NibbleEnum, uds.utilities.ValidatedEnum, uds.utilities.ExtendableEnum



Abstract definition of UDS packet type.

Packet type information is carried by *Network Protocol Control Information (N_PCI)*. Enums with packet types (N_PCI) values for certain buses (e.g. CAN, LIN, FlexRay) must inherit after this class.

Note: There are differences in values for each bus (e.g. LIN does not use Flow Control).

Initialize self. See help(type(self)) for accurate signature.

abstract classmethod is_initial_packet_type(*value*)
Check whether given argument is a member or a value of packet type that initiates a diagnostic message.

Parameters
value (AbstractUdsPacketType) – Value to check.

Returns
True if given argument is a packet type that initiates a diagnostic message, else False.

Return type
bool

uds.packet.can_packet

CAN bus specific implementation of UDS packets.

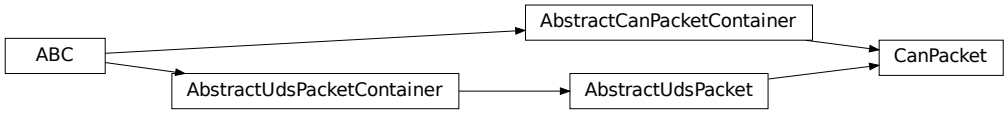
Module Contents

Classes

CanPacket	Definition of a CAN packet.

class uds.packet.can_packet.**CanPacket**(*, *packet_type*, *addressing_format*, *addressing_type*, *can_id=None*, *target_address=None*, *source_address=None*, *address_extension=None*, *dlc=None*, ***packet_type_specific_kwargs*)

Bases: *uds.packet.abstract_can_packet_container.AbstractCanPacketContainer*, *uds.packet.abstract_packet.AbstractUdsPacket*



Definition of a CAN packet.

Objects of this class act as a storage for all relevant attributes of a *CAN packet*.

Create a storage for a single CAN packet.

Parameters

- **packet_type** (uds.packet.can_packet_type.CanPacketType) – Type of this CAN packet.
- **addressing_format** (*uds.can.CanAddressingFormat*) – CAN addressing format that this CAN packet uses.
- **addressing_type** (*uds.transmission_attributes.AddressingType*) – Addressing type for which this CAN packet is relevant.
- **can_id** (*Optional[int]*) – CAN Identifier value that is used by this packet. Leave None if other arguments unambiguously determine CAN ID value.

- **target_address** (*Optional[int]*) – Target Address value carried by this CAN Packet. Leave None if provided *addressing_format* does not use Target Address parameter or the value of Target Address was provided in *can_id* parameter.
- **source_address** (*Optional[int]*) – Source Address value carried by this CAN packet. Leave None if provided *addressing_format* does not use Source Address parameter or the value of Source Address was provided in *can_id* parameter.
- **address_extension** (*Optional[int]*) – Address Extension value carried by this CAN packet. Leave None if provided *addressing_format* does not use Address Extension parameter.
- **dlc** (*Optional[int]*) – DLC value of a CAN frame that carries this CAN Packet.
  - None - use CAN Data Frame Optimization (CAN ID value will be automatically determined)
  - int type value - DLC value to set. CAN Data Padding will be used to fill unused data bytes.

  Warning: You have to provide DLC value for packets of First Frame type.

- **packet_type_specific_kwargs** (*Any*) – Arguments that are specific for provided CAN Packet Type.
  - **parameter filler_byte**
    (optional for: SF, CF and FC) Filler Byte value to use for CAN Frame Data Padding.
  - **parameter payload**
    (required for: SF, FF and CF) Payload of a diagnostic message that is carried by this CAN packet.
  - **parameter data_length**
    (required for: FF) Number of payload bytes of a diagnostic message initiated by this First Frame packet.
  - **parameter sequence_number**
    (required for: CF) Sequence number value of this Consecutive Frame.
  - **parameter flow_status**
    (required for: FC) Flow status information carried by this Flow Control frame.
  - **parameter block_size**
    (required for: FC with ContinueToSend Flow Status) Block size information carried by this Flow Control frame.
  - **parameter st_min**
    (required for: FC with ContinueToSend Flow Status) Separation Time minimum information carried by this Flow Control frame.

property raw_frame_data: uds.utilities.RawBytesTupleAlias

Raw data bytes of a CAN frame that carries this CAN packet.

Return type
uds.utilities.RawBytesTupleAlias

property can_id: int

CAN Identifier (CAN ID) of a CAN Frame that carries this CAN packet.

Return type
int

property addressing_format: uds.can.CanAddressingFormat

CAN addressing format used by this CAN packet.

Return type
uds.can.CanAddressingFormat

property addressing_type: uds.transmission_attributes.AddressingType

Addressing type for which this CAN packet is relevant.

Return type
uds.transmission_attributes.AddressingType

property dlc: int

Value of Data Length Code (DLC) of a CAN Frame that carries this CAN packet.

Return type
int

property packet_type: *uds.packet.can_packet_type.CanPacketType*

Type (N_PCI value) of this CAN packet.

Return type
uds.packet.can_packet_type.CanPacketType

property target_address: int | None

Target Address (TA) value of this CAN Packet.

Target Address value is used with following *addressing formats*:

- *Normal Fixed Addressing*
- *Extended Addressing*
- *Mixed 29-bit Addressing*

None in other cases.

Return type
Optional[int]

property source_address: int | None

Source Address (SA) value of this CAN Packet.

Source Address value is used with following *addressing formats*:

- *Normal Fixed Addressing*
- *Mixed 29-bit Addressing*

None in other cases.

Return type
Optional[int]

property address_extension: int | None

Address Extension (AE) value of this CAN Packet.

Address Extension is used with following *addressing formats*:

- *Mixed Addressing* - either: - *Mixed 11-bit Addressing* - *Mixed 29-bit Addressing*

None in other cases.

Return type
Optional[int]

set_address_information(*, *addressing_format*, *addressing_type*, *can_id=None*, *target_address=None*, *source_address=None*, *address_extension=None*)

Change addressing information for this CAN packet.

This function enables to change an entire *Network Address Information* for a *CAN packet*.

Parameters

- **addressing_format** (*uds.can.CanAddressingFormat*) – CAN addressing format that this CAN packet uses.
- **addressing_type** (*uds.transmission_attributes.AddressingType*) – Addressing type for which this CAN packet is relevant.
- **can_id** (*Optional[int]*) – CAN Identifier value that is used by this packet. Leave None if other arguments unambiguously determine CAN ID value.
- **target_address** (*Optional[int]*) – Target Address value carried by this CAN Packet. Leave None if provided *addressing_format* does not use Target Address parameter or the value of Target Address was provided in *can_id* parameter.
- **source_address** (*Optional[int]*) – Source Address value carried by this CAN packet. Leave None if provided *addressing_format* does not use Source Address parameter or the value of Source Address was provided in *can_id* parameter.
- **address_extension** (*Optional[int]*) – Address Extension value carried by this CAN packet. Leave None if provided *addressing_format* does not use Address Extension parameter.

Raises
NotImplementedError – There is missing implementation for the provided Addressing Format. Please create an issue in our Issues Tracking System with detailed description if you face this error.

Return type
None

set_address_information_normal_11bit(*addressing_type*, *can_id*)

Change addressing information for this CAN packet to use Normal 11-bit Addressing format.

Parameters

- **addressing_type** (*uds.transmission_attributes.AddressingType*) – Addressing type for which this CAN packet is relevant.
- **can_id** (*int*) – CAN Identifier value that is used by this packet.

Return type
None

set_address_information_normal_fixed(*addressing_type*, *can_id=None*, *target_address=None*, *source_address=None*)

Change addressing information for this CAN packet to use Normal Fixed Addressing format.

Parameters

- **addressing_type** (*uds.transmission_attributes.AddressingType*) – Addressing type for which this CAN packet is relevant.

- **can_id** (*Optional[int]*) – CAN Identifier value that is used by this packet. Leave None if the values of *target_address* and *source_address* parameters are provided.
- **target_address** (*Optional[int]*) – Target Address value carried by this CAN Packet. Leave None if the value of *can_id* parameter is provided.
- **source_address** (*Optional[int]*) – Source Address value carried by this CAN packet. Leave None if the value of *can_id* parameter is provided.

Return type
None

set_address_information_extended(*addressing_type*, *can_id*, *target_address*)

Change addressing information for this CAN packet to use Extended Addressing format.

Parameters

- **addressing_type** (*uds.transmission_attributes.AddressingType*) – Addressing type for which this CAN packet is relevant.
- **can_id** (*int*) – CAN Identifier value that is used by this packet.
- **target_address** (*int*) – Target Address value carried by this CAN Packet.

Return type
None

set_address_information_mixed_11bit(*addressing_type*, *can_id*, *address_extension*)

Change addressing information for this CAN packet to use Mixed 11-bit Addressing format.

Parameters

- **addressing_type** (*uds.transmission_attributes.AddressingType*) – Addressing type for which this CAN packet is relevant.
- **can_id** (*int*) – CAN Identifier value that is used by this packet.
- **address_extension** (*int*) – Address Extension value carried by this CAN packet.

Return type
None

set_address_information_mixed_29bit(*addressing_type*, *address_extension*, *can_id=None*, *target_address=None*, *source_address=None*)

Change addressing information for this CAN packet to use Mixed 29-bit Addressing format.

Parameters

- **addressing_type** (*uds.transmission_attributes.AddressingType*) – Addressing type for which this CAN packet is relevant.
- **can_id** (*Optional[int]*) – CAN Identifier value that is used by this packet. Leave None if the values of *target_address* and *source_address* parameters are provided.
- **target_address** (*Optional[int]*) – Target Address value carried by this CAN Packet. Leave None if the value of *can_id* parameter is provided.
- **source_address** (*Optional[int]*) – Source Address value carried by this CAN packet. Leave None if the value of *can_id* parameter is provided.
- **address_extension** (*int*) – Address Extension value carried by this CAN packet.

Return type
None

set_packet_data(*, *packet_type*, *dlc=None*, ***packet_type_specific_kwargs*)

Change packet type and data field of this CAN packet.

This function enables to change an entire *Network Data Field* and *Network Protocol Control Information* for a *CAN packet*.

Parameters

- **packet_type** (uds.packet.can_packet_type.CanPacketType) – Type of this CAN packet.
- **dlc** (*Optional[int]*) – DLC value of a CAN frame that carries this CAN Packet.
  - None - use CAN Data Frame Optimization (CAN ID value will be automatically determined)
  - int type value - DLC value to set. CAN Data Padding will be used to fill unused data bytes.

  Warning: You have to provide DLC value for packets of First Frame type.

- **packet_type_specific_kwargs** (*Any*) – Arguments that are specific for provided CAN Packet Type.
  - **parameter filler_byte**
    (optional for: SF, CF and FC) Filler Byte value to use for CAN Frame Data Padding.
  - **parameter payload**
    (required for: SF, FF and CF) Payload of a diagnostic message that is carried by this CAN packet.
  - **parameter data_length**
    (required for: FF) Number of payload bytes of a diagnostic message initiated by this First Frame packet.
  - **parameter sequence_number**
    (required for: CF) Sequence number value of this Consecutive Frame.
  - **parameter flow_status**
    (required for: FC) Flow status information carried by this Flow Control frame.
  - **parameter block_size**
    (required for: FC with ContinueToSend Flow Status) Block size information carried by this Flow Control frame.
  - **parameter st_min**
    (required for: FC with ContinueToSend Flow Status) Separation Time minimum information carried by this Flow Control frame.

Raises

NotImplementedError – There is missing implementation for the provided CAN Packet Type. Please create an issue in our Issues Tracking System with detailed description if you face this error.

Return type

None

set_single_frame_data(*payload*, *dlc=None*, *filler_byte=DEFAULT_FILLER_BYTE*)

Change packet type (to Single Frame) and data field of this CAN packet.

This function enables to change an entire *Network Data Field* and *Network Protocol Control Information* for a *Single Frame*.

Parameters

- **payload** (*uds.utilities.RawBytesAlias*) – Payload of a diagnostic message that is carried by this CAN packet.
- **dlc** (*Optional[int]*) – DLC value of a CAN frame that carries this CAN Packet.
  - None - use CAN Data Frame Optimization (CAN ID value will be automatically determined)
  - int type value - DLC value to set. CAN Data Padding will be used to fill unused data bytes.
- **filler_byte** (*int*) – Filler Byte value to use for CAN Frame Data Padding.

Return type
None

set_first_frame_data(*dlc*, *payload*, *data_length*)

Change packet type (to First Frame) and data field of this CAN packet.

This function enables to change an entire *Network Data Field* and *Network Protocol Control Information* for a *First Frame*.

Parameters

- **dlc** (*int*) – DLC value of a CAN frame that carries this CAN Packet.
- **payload** (*uds.utilities.RawBytesAlias*) – Payload of a diagnostic message that is carried by this CAN packet.
- **data_length** (*int*) – Number of payload bytes of a diagnostic message initiated by this First Frame packet.

Return type
None

set_consecutive_frame_data(*payload*, *sequence_number*, *dlc=None*, *filler_byte=DEFAULT_FILLER_BYTE*)

Change packet type (to Consecutive Frame) and data field of this CAN packet.

This function enables to change an entire *Network Data Field* and *Network Protocol Control Information* for a *Consecutive Frame*.

Parameters

- **payload** (*uds.utilities.RawBytesAlias*) – Payload of a diagnostic message that is carried by this CAN packet.
- **sequence_number** (*int*) – Sequence number value of this Consecutive Frame.
- **dlc** (*Optional[int]*) – DLC value of a CAN frame that carries this CAN Packet.
  - None - use CAN Data Frame Optimization (CAN ID value will be automatically determined)
  - int type value - DLC value to set. CAN Data Padding will be used to fill unused data bytes.

- **filler_byte** (*int*) – Filler Byte value to use for CAN Frame Data Padding.

Return type
None

set_flow_control_data(*flow_status*, *block_size=None*, *st_min=None*, *dlc=None*, *filler_byte=DEFAULT_FILLER_BYTE*)

Change packet type (to Flow Control) and data field of this CAN packet.

This function enables to change an entire *Network Data Field* and *Network Protocol Control Information* for a *Flow Control*.

Parameters

- **flow_status** (*uds.can.CanFlowStatus*) – Flow status information carried by this Flow Control frame.
- **block_size** (*Optional[int]*) – Block size information carried by this Flow Control frame.
- **st_min** (*Optional[int]*) – Separation Time minimum information carried by this Flow Control frame.
- **dlc** (*Optional[int]*) – DLC value of a CAN frame that carries this CAN Packet.
  - None - use CAN Data Frame Optimization (CAN ID value will be automatically determined)
  - int type value - DLC value to set. CAN Data Padding will be used to fill unused data bytes.
- **filler_byte** (*int*) – Filler Byte value to use for CAN Frame Data Padding.

Return type
None

__validate_unambiguous_ai_change(*addressing_format*)

Validate whether CAN Addressing Format change to provided value is ambiguous.

Parameters
addressing_format (*uds.can.CanAddressingFormat*) – Desired value of CAN Addressing Format.

Raises
AmbiguityError – Cannot change value because the operation is ambiguous.

Return type
None

__update_ai_data_byte()

Update the value of *raw_frame_data* attribute after Addressing Information change.

Return type
None

uds.packet.can_packet_record

CAN bus specific implementation of UDS packets records.

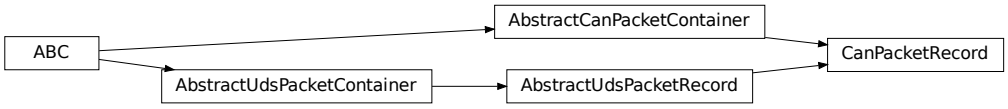
Module Contents

Classes

CanPacketRecord	Definition of a CAN packet record.

class uds.packet.can_packet_record.**CanPacketRecord**(*frame*, *direction*, *addressing_type*, *addressing_format*, *transmission_time*)

Bases: *uds.packet.abstract_can_packet_container.AbstractCanPacketContainer*, *uds.packet.abstract_packet.AbstractUdsPacketRecord*



Definition of a CAN packet record.

Objects of this class act as a storage for historic information about transmitted or received *CAN packet*.

Create a record of historic information about a CAN packet that was either received or transmitted.

Parameters

- **frame** (*CanFrameAlias*) – Either received or transmitted CAN frame that carried this CAN Packet.
- **direction** (*uds.transmission_attributes.TransmissionDirection*) – Information whether this packet was transmitted or received.
- **addressing_type** (*uds.transmission_attributes.AddressingType*) – Addressing type for which this CAN packet is relevant.
- **addressing_format** (*uds.can.CanAddressingFormat*) – CAN addressing format that this CAN packet used.
- **transmission_time** (*datetime.datetime*) – Time stamp when this packet was fully transmitted on a CAN bus.

property raw_frame_data: uds.utilities.RawBytesTupleAlias

Raw data bytes of a frame that carried this CAN packet.

Return type
uds.utilities.RawBytesTupleAlias

property can_id: int

CAN Identifier (CAN ID) of a CAN Frame that carries this CAN packet.

Return type
int

property addressing_format: uds.can.CanAddressingFormat

CAN addressing format used by this CAN packet.

Return type
uds.can.CanAddressingFormat

property addressing_type: uds.transmission_attributes.AddressingType

Addressing type over which this CAN packet was transmitted.

Return type
uds.transmission_attributes.AddressingType

property packet_type: *uds.packet.can_packet_type.CanPacketType*

CAN packet type value - N_PCI value of this N_PDU.

Return type
uds.packet.can_packet_type.CanPacketType

property target_address: int | None

Target Address (TA) value of this CAN Packet.

Target Address value is used with following *addressing formats*:

- *Normal Fixed Addressing*
- *Extended Addressing*
- *Mixed 29-bit Addressing*

None in other cases.

Return type
Optional[int]

property source_address: int | None

Source Address (SA) value of this CAN Packet.

Source Address value is used with following *addressing formats*:

- *Normal Fixed Addressing*
- *Mixed 29-bit Addressing*

None in other cases.

Return type
Optional[int]

property address_extension: int | None

Address Extension (AE) value of this CAN Packet.

Address Extension is used with following *addressing formats*:

- *Mixed Addressing* - either: - *Mixed 11-bit Addressing* - *Mixed 29-bit Addressing*

None in other cases.

Return type
Optional[int]

static _validate_frame(*value*)

Validate a CAN frame argument.

Parameters
value (*Any*) – Value to validate.

Raises
- **TypeError** – The frame argument has unsupported.
- **ValueError** – Some attribute of the frame argument is missing or its value is unexpected.

Return type
None

__assess_packet_type()

Assess and set value of Packet Type attribute.

Return type
None

__assess_ai_attributes()

Assess and set values of attributes with Addressing Information.

Raises
InconsistentArgumentsError – Value of Addressing Type that is already set does not match decoded one.

Return type
None

uds.packet.can_packet_type

CAN packet types definitions.

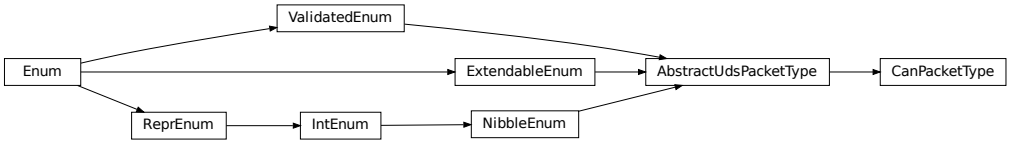
Module Contents

Classes

CanPacketType	Definition of CAN packet types.

class uds.packet.can_packet_type.**CanPacketType**

Bases: *uds.packet.abstract_packet_type.AbstractUdsPacketType*



Definition of CAN packet types.

CAN packet types are *Network Protocol Control Information (N_PCI)* values that are specific for CAN bus.

Initialize self. See help(type(self)) for accurate signature.

SINGLE_FRAME: *CanPacketType*

CAN packet type (N_PCI) value of *Single Frame (SF)*.

FIRST_FRAME: *CanPacketType*

CAN packet type (N_PCI) value of First Frame (FF) <knowledge-base-can-first-frame>`.

CONSECUTIVE_FRAME: *CanPacketType*

CAN packet type (N_PCI) value of *Consecutive Frame (CF)*.

FLOW_CONTROL: *CanPacketType*

CAN packet type (N_PCI) value of *Flow Control (FC)*.

classmethod is_initial_packet_type(*value*)

Check whether given argument is a CAN packet type that initiates a diagnostic message.

Parameters
value (CanPacketType) – Value to check.

Returns
True if given argument is a packet type that initiates a diagnostic message, else False.

Return type
bool

uds.segmentation

A subpackage with tools for handing segmentation.

***Segmentation* defines two processes:**

- *diagnostic message segmentation*
- *packets desegmentation*

This subpackage contains implementation of:

- common API interface for all segmentation duties
- classes that handles segmentation for each bus

Submodules

uds.segmentation.abstract_segmenter

Definition of API for segmentation and desegmentation strategies.

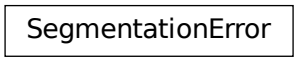
Module Contents

Classes

AbstractSegmenter	Abstract definition of a segmenter class.

exception uds.segmentation.abstract_segmenter.**SegmentationError**

Bases: ValueError

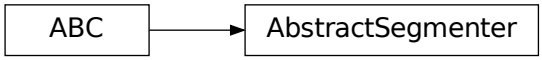


UDS segmentation or desegmentation process cannot be completed due to input data inconsistency.

Initialize self. See help(type(self)) for accurate signature.

class uds.segmentation.abstract_segmenter.**AbstractSegmenter**

Bases: abc.ABC



Abstract definition of a segmenter class.

Segmenter classes defines UDS segmentation and desegmentation duties. They contain helper methods that are essential for successful *segmentation* and *desegmentation* execution.

Note: Each concrete segmenter class handles exactly one bus.

abstract property supported_packet_class: Type[uds.packet.AbstractUdsPacket]

Class of UDS Packet supported by this segmenter.

Return type
Type[uds.packet.AbstractUdsPacket]

abstract property supported_packet_record_class: Type[uds.packet.AbstractUdsPacketRecord]

Class of UDS Packet Record supported by this segmenter.

Return type
Type[uds.packet.AbstractUdsPacketRecord]

is_supported_packet_type(*packet*)

Check if the argument value is a packet object of a supported type.

Parameters
packet (*uds.packet.AbstractUdsPacketContainer*) – Packet object to check.

Returns
True if provided value is an object of a supported packet type, False otherwise.

Return type
bool

abstract is_input_packet(***kwargs*)

Check if provided frame attributes belong to a UDS packet which is an input for this Segmenter.

Parameters
kwargs – Attributes of a frame to check.

Returns
Addressing Type used for transmission of this UDS packet according to the configuration of this Segmenter (if provided attributes belongs to an input UDS packet), otherwise None.

Return type
Optional[uds.transmission_attributes.AddressingType]

is_supported_packets_sequence_type(*packets*)

Check if the argument value is a packets sequence of a supported type.

Parameters
packets (*Sequence[uds.packet.AbstractUdsPacketContainer]*) – Packets sequence to check.

Returns
True if provided value is a packets sequence of a supported type, False otherwise.

Return type
bool

abstract is_desegmented_message(*packets*)

Check whether provided packets are full sequence of packets that form exactly one diagnostic message.

Parameters
packets (*uds.packet.PacketsContainersSequence*) – Packets sequence to check.

Returns
True if the packets form exactly one diagnostic message. False if there are missing, additional or inconsistent (e.g. two packets that initiate a message) packets.

Return type
bool

abstract desegmentation(*packets*)

Perform desegmentation of UDS packets.

Parameters
packets (*uds.packet.PacketsContainersSequence*) – UDS packets to desegment into UDS message.

Raises
SegmentationError – Provided packets are not a complete packet sequence that form a diagnostic message.

Returns
A diagnostic message that is carried by provided UDS packets.

Return type
Union[uds.message.UdsMessage, uds.message.UdsMessageRecord]

abstract segmentation(*message*)

Perform segmentation of a diagnostic message.

Parameters
message (*uds.message.UdsMessage*) – UDS message to divide into UDS packets.

Raises
SegmentationError – Provided diagnostic message cannot be segmented.

Returns
UDS packet(s) that carry provided diagnostic message.

Return type
uds.packet.PacketsTuple

uds.segmentation.can_segmenter

Segmentation specific for CAN bus.

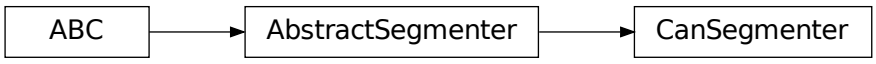
Module Contents

Classes

CanSegmenter	Segmenter class that provides utilities for segmentation and desegmentation specific for CAN bus.

class uds.segmentation.can_segmenter.**CanSegmenter**(*, *addressing_information*, *dlc=CanDlcHandler.MIN_BASE_UDS_DLC*, *use_data_optimization=False*, *filler_byte=DEFAULT_FILLER_BYTE*)

Bases: *uds.segmentation.abstract_segmenter.AbstractSegmenter*



Segmenter class that provides utilities for segmentation and desegmentation specific for CAN bus.

Configure CAN Segmenter.

Parameters

- **addressing_information** (*uds.can.AbstractCanAddressingInformation*) – Addressing Information configuration of a CAN node.

- **dlc** (*int*) – Base CAN DLC value to use for creating CAN Packets.
- **use_data_optimization** (*bool*) – Information whether to use CAN Frame Data Optimization in created CAN Packets during segmentation.
- **filler_byte** (*int*) – Filler byte value to use for CAN Frame Data Padding in created CAN Packets during segmentation.

property supported_packet_class: Type[uds.packet.AbstractUdsPacket]

Class of UDS Packet supported by CAN segmenter.

Return type
Type[uds.packet.AbstractUdsPacket]

property supported_packet_record_class: Type[uds.packet.AbstractUdsPacketRecord]

Class of UDS Packet Record supported by CAN segmenter.

Return type
Type[uds.packet.AbstractUdsPacketRecord]

property addressing_format: uds.can.CanAddressingFormat

CAN Addressing format used.

Return type
uds.can.CanAddressingFormat

property rx_packets_physical_ai: uds.can.PacketAIParamsAlias

Addressing Information parameters of incoming physically addressed CAN packets.

Return type
uds.can.PacketAIParamsAlias

property tx_packets_physical_ai: uds.can.PacketAIParamsAlias

Addressing Information parameters of outgoing physically addressed CAN packets.

Return type
uds.can.PacketAIParamsAlias

property rx_packets_functional_ai: uds.can.PacketAIParamsAlias

Addressing Information parameters of incoming functionally addressed CAN packets.

Return type
uds.can.PacketAIParamsAlias

property tx_packets_functional_ai: uds.can.PacketAIParamsAlias

Addressing Information parameters of outgoing functionally addressed CAN packets.

Return type
uds.can.PacketAIParamsAlias

property addressing_information: uds.can.AbstractCanAddressingInformation

Addressing Information configuration of a CAN node.

Return type
uds.can.AbstractCanAddressingInformation

property dlc: int

Value of base CAN DLC to use for CAN Packets.

Note: All output CAN Packets (created by *segmentation()*) will have this DLC value set unless *CAN Frame Data Optimization* is used.

Return type
int

`property use_data_optimization: bool`

Information whether to use CAN Frame Data Optimization during CAN Packet creation.

Return type
bool

`property filler_byte: int`

Filler byte value to use for CAN Frame Data Padding during segmentation.

Return type
int

`is_input_packet`(*can_id*, *data*)

Check if provided frame attributes belong to a UDS CAN packet which is an input for this CAN Segmenter.

Parameters

- **can_id** (*int*) – Identifier of CAN frame to check.
- **data** (*uds.utilities.RawBytesAlias*) – Data field of CAN frame to check.

Returns
Addressing Type used for transmission of this UDS CAN packet according to the configuration of this CAN Segmenter (if provided attributes belongs to an input UDS CAN packet), otherwise None.

Return type
Optional[uds.transmission_attributes.AddressingType]

`is_desegmented_message`(*packets*)

Check whether provided packets are full sequence of packets that form exactly one diagnostic message.

Parameters
packets (*uds.packet.PacketsContainersSequence*) – Packets sequence to check.

Raises

- **ValueError** – Provided value is not CAN packets sequence.
- **NotImplementedError** – There is missing implementation for the provided initial packet type. Please create an issue in our Issues Tracking System with detailed description if you face this error.

Returns
True if the packets form exactly one diagnostic message. False if there are missing, additional or inconsistent (e.g. two packets that initiate a message) packets.

Return type
bool

`desegmentation`(*packets*)

Perform desegmentation of CAN packets.

Parameters
packets (*uds.packet.PacketsContainersSequence*) – CAN packets to desegment into UDS message.

Raises

- *SegmentationError* – Provided packets are not a complete packets sequence that form a diagnostic message.
- **NotImplementedError** – There is missing implementation for the provided CAN Packets. Please create an issue in our Issues Tracking System with detailed description if you face this error.

Returns
A diagnostic message that is an outcome of CAN packets desegmentation.

Return type
Union[uds.message.UdsMessage, uds.message.UdsMessageRecord]

segmentation(*message*)

Perform segmentation of a diagnostic message.

Parameters
message (*uds.message.UdsMessage*) – UDS message to divide into UDS packets.

Raises
- **TypeError** – Provided value is not instance of UdsMessage class.
- **NotImplementedError** – There is missing implementation for the Addressing Type used by provided message. Please create an issue in our Issues Tracking System with detailed description if you face this error.

Returns
CAN packets that are an outcome of UDS message segmentation.

Return type
uds.packet.PacketsTuple

__physical_segmentation(*message*)

Segment physically addressed diagnostic message.

Parameters
message (*uds.message.UdsMessage*) – UDS message to divide into UDS packets.

Raises
SegmentationError – Provided diagnostic message cannot be segmented.

Returns
CAN packets that are an outcome of UDS message segmentation.

Return type
uds.packet.PacketsTuple

__functional_segmentation(*message*)

Segment functionally addressed diagnostic message.

Parameters
message (*uds.message.UdsMessage*) – UDS message to divide into UDS packets.

Raises
SegmentationError – Provided diagnostic message cannot be segmented.

Returns
CAN packets that are an outcome of UDS message segmentation.

Return type
uds.packet.PacketsTuple

uds.transmission_attributes

Definition of attributes that describes UDS communication.

Submodules

uds.transmission_attributes.addressing

Implementation of diagnostic messages addressing.

Addressing describes a communication model that is used during UDS communication.

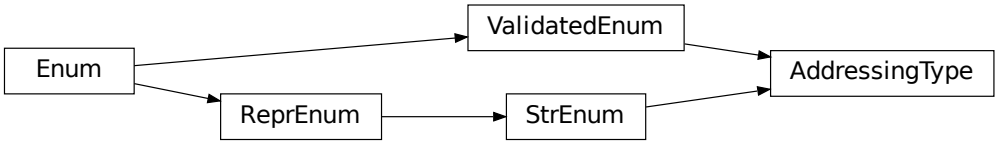
Module Contents

Classes

AddressingType	Addressing types values defined by UDS protocol.

class uds.transmission_attributes.addressing.**AddressingType**

Bases: aenum.StrEnum, uds.utilities.ValidatedEnum



Addressing types values defined by UDS protocol.

Addressing describes a communication model that is used during UDS communication.

Initialize self. See help(type(self)) for accurate signature.

PHYSICAL: *AddressingType* = 'Physical'

Physical addressing - 1 (client) to 1 (server) communication.

FUNCTIONAL: *AddressingType* = 'Functional'

Functional addressing - 1 (client) to many (servers) communication.

uds.transmission_attributes.transmission_direction

Definition of communication direction.

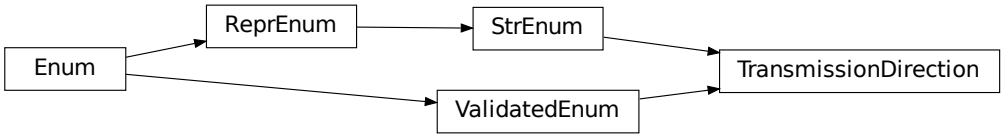
Module Contents

Classes

TransmissionDirection	Direction of a communication.

class uds.transmission_attributes.transmission_direction.**TransmissionDirection**

Bases: aenum.StrEnum, uds.utilities.ValidatedEnum



Direction of a communication.

Initialize self. See help(type(self)) for accurate signature.

RECEIVED: *TransmissionDirection* = **'Rx'**

Incoming transmission from the perspective of the code.

TRANSMITTED: *TransmissionDirection* = **'Tx'**

Outgoing transmission from the perspective of the code.

uds.transport_interface

A subpackage with implementation for UDS middle layers (Transport and Network).

It provides configurable Transport Interfaces for:

- transmitting and receiving UDS packets
- transmitting and receiving UDS messages
- storing historic information about transmitted and received UDS packets
- storing historic information about transmitted and received UDS messages
- managing Transport and Network layer errors

Submodules

uds.transport_interface.abstract_transport_interface

Abstract definition of UDS Transport Interface.

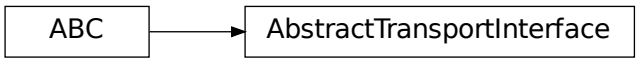
Module Contents

Classes

AbstractTransportInterface	Abstract definition of Transport Interface.

class uds.transport_interface.abstract_transport_interface.**AbstractTransportInterface**(*bus_manager*)

Bases: abc.ABC



Abstract definition of Transport Interface.

Transport Interfaces are meant to handle middle layers (Transport and Network) of UDS OSI Model.

Create Transport Interface (an object for handling UDS Transport and Network layers).

Parameters

bus_manager (*Any*) – An object that handles the bus (Physical and Data layers of OSI Model).

Raises

ValueError – Provided value of bus manager is not supported by this Transport Interface.

property bus_manager: Any

Value of the bus manager used by this Transport Interface.

Bus manager handles Physical and Data layers (OSI Model) of the bus.

Return type

Any

abstract property segmenter: uds.segmentation.AbstractSegmenter

Value of the segmenter used by this Transport Interface.

> **Warning:** Do not change any segmenter attributes as it might cause malfunction of the entire Transport Interface.

Return type

uds.segmentation.AbstractSegmenter

abstract static is_supported_bus_manager(*bus_manager*)

Check whether provided value is a bus manager that is supported by this Transport Interface.

Parameters
bus_manager (*Any*) – Value to check.

Returns
True if provided bus object is compatible with this Transport Interface, False otherwise.

Return type
bool

abstract send_packet(*packet*)

Transmit UDS packet.

Parameters
packet (*uds.packet.AbstractUdsPacket*) – A packet to send.

Returns
Record with historic information about transmitted UDS packet.

Return type
uds.packet.AbstractUdsPacketRecord

abstract receive_packet(*timeout=None*)

Receive UDS packet.

Parameters
timeout (*Optional[uds.utilities.TimeMillisecondsAlias]*) – Maximal time (in milliseconds) to wait.

Raises
TimeoutError – Timeout was reached.

Returns
Record with historic information about received UDS packet.

Return type
uds.packet.AbstractUdsPacketRecord

abstract async async_send_packet(*packet*, *loop=None*)

Transmit UDS packet asynchronously.

Parameters

- **packet** (*uds.packet.AbstractUdsPacket*) – A packet to send.
- **loop** (*Optional[asyncio.AbstractEventLoop]*) – An asyncio event loop to use for scheduling this task.

Returns
Record with historic information about transmitted UDS packet.

Return type
uds.packet.AbstractUdsPacketRecord

abstract async async_receive_packet(*timeout=None*, *loop=None*)

Receive UDS packet asynchronously.

Parameters

- **timeout** (*Optional[uds.utilities.TimeMillisecondsAlias]*) – Maximal time (in milliseconds) to wait.

- **loop** (*Optional[asyncio.AbstractEventLoop]*) – An asyncio event loop to use for scheduling this task.

Raises

- **TimeoutError** – Timeout was reached.
- **asyncio.TimeoutError** – Timeout was reached.

Returns
Record with historic information about received UDS packet.

Return type
uds.packet.AbstractUdsPacketRecord

uds.transport_interface.can_transport_interface

Definition and implementation of UDS Transport Interface for CAN bus.

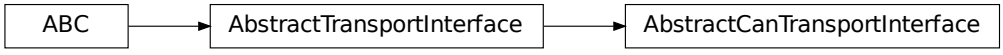
Module Contents

Classes

AbstractCanTransportInterface	Abstract definition of Transport Interface for managing UDS on CAN bus.
PyCanTransportInterface	Transport Interface for managing UDS on CAN with python-can package as bus handler.

class uds.transport_interface.can_transport_interface.**AbstractCanTransportInterface**(*can_bus_manager*, *addressing_information*, ***kwargs*)

Bases: *uds.transport_interface.abstract_transport_interface.AbstractTransportInterface*



Abstract definition of Transport Interface for managing UDS on CAN bus.

CAN Transport Interfaces are meant to handle UDS middle layers (Transport and Network) on CAN bus.

Create Transport Interface (an object for handling UDS Transport and Network layers).

Parameters

- **can_bus_manager** (*Any*) – An object that handles CAN bus (Physical and Data layers of OSI Model).
- **addressing_information** (*uds.can.AbstractCanAddressingInformation*) – Addressing Information of CAN Transport Interface.

- **kwargs** (*Any*) – Optional arguments that are specific for CAN bus.
  - **parameter n_as_timeout**
    Timeout value for *N_As* time parameter.
  - **parameter n_ar_timeout**
    Timeout value for *N_Ar* time parameter.
  - **parameter n_bs_timeout**
    Timeout value for *N_Bs* time parameter.
  - **parameter n_br**
    Value of *N_Br* time parameter to use in communication.
  - **parameter n_cs**
    Value of *N_Cs* time parameter to use in communication.
  - **parameter n_cr_timeout**
    Timeout value for *N_Cr* time parameter.
  - **parameter dlc**
    Base CAN DLC value to use for CAN Packets.
  - **parameter use_data_optimization**
    Information whether to use CAN Frame Data Optimization.
  - **parameter filler_byte**
    Filler byte value to use for CAN Frame Data Padding.

Raises
TypeError – Provided Addressing Information value has unexpected type.

`property segmenter: uds.segmentation.CanSegmenter`

Value of the segmenter used by this CAN Transport Interface.

Return type
uds.segmentation.CanSegmenter

`property n_as_timeout: uds.utilities.TimeMillisecondsAlias`

Timeout value for N_As time parameter.

Return type
uds.utilities.TimeMillisecondsAlias

`abstract property n_as_measured: uds.utilities.TimeMillisecondsAlias | None`

Get the last measured value of N_As time parameter.

Returns
Time in milliseconds or None if the value was never measured.

Return type
Optional[uds.utilities.TimeMillisecondsAlias]

`property n_ar_timeout: uds.utilities.TimeMillisecondsAlias`

Timeout value for N_Ar time parameter.

Return type
uds.utilities.TimeMillisecondsAlias

`abstract property n_ar_measured: uds.utilities.TimeMillisecondsAlias | None`

Get the last measured value of N_Ar time parameter.

Returns
Time in milliseconds or None if the value was never measured.

Return type
Optional[uds.utilities.TimeMillisecondsAlias]

property n_bs_timeout: uds.utilities.TimeMillisecondsAlias

Timeout value for N_Bs time parameter.

Return type
uds.utilities.TimeMillisecondsAlias

abstract property n_bs_measured: uds.utilities.TimeMillisecondsAlias | None

Get the last measured value of N_Bs time parameter.

Returns
Time in milliseconds or None if the value was never measured.

Return type
Optional[uds.utilities.TimeMillisecondsAlias]

property n_br: uds.utilities.TimeMillisecondsAlias

Get the value of N_Br time parameter which is currently set.

Note: The actual (observed on the bus) value will be slightly longer as it also includes computation and CAN Interface delays.

Return type
uds.utilities.TimeMillisecondsAlias

property n_br_max: uds.utilities.TimeMillisecondsAlias

Get the maximum valid value of N_Br time parameter.

Warning: To assess maximal value of *N_Br*, the actual value of *N_Ar* time parameter is required. Either the latest measured value of N_Ar would be used, or 0ms would be assumed (if there are no measurement result).

Return type
uds.utilities.TimeMillisecondsAlias

property n_cs: uds.utilities.TimeMillisecondsAlias | None

Get the value of N_Cs time parameter which is currently set.

Note: The actual (observed on the bus) value will be slightly longer as it also includes computation and CAN Interface delays.

Return type
Optional[uds.utilities.TimeMillisecondsAlias]

property n_cs_max: uds.utilities.TimeMillisecondsAlias

Get the maximum valid value of N_Cs time parameter.

> **Warning:** To assess maximal value of *N_Cs*, the actual value of *N_As* time parameter is required. Either the latest measured value of N_Ar would be used, or 0ms would be assumed (if there are no measurement result).

Return type
uds.utilities.TimeMillisecondsAlias

`property n_cr_timeout: uds.utilities.TimeMillisecondsAlias`

Timeout value for N_Cr time parameter.

Return type
uds.utilities.TimeMillisecondsAlias

`abstract property n_cr_measured: uds.utilities.TimeMillisecondsAlias | None`

Get the last measured value of N_Cr time parameter.

Returns
Time in milliseconds or None if the value was never measured.

Return type
Optional[uds.utilities.TimeMillisecondsAlias]

`property addressing_information: uds.can.AbstractCanAddressingInformation`

Addressing Information of Transport Interface.

> **Warning:** Once the value is set, it must not be changed as it might cause communication problems.

Return type
uds.can.AbstractCanAddressingInformation

`property dlc: int`

Value of base CAN DLC to use for output CAN Packets.

Note: All output CAN Packets will have this DLC value set unless *CAN Frame Data Optimization* is used.

Return type
int

`property use_data_optimization: bool`

Information whether to use CAN Frame Data Optimization during CAN Packets creation.

Return type
bool

`property filler_byte: int`

Filler byte value to use for output CAN Frame Data Padding during segmentation.

Return type
int

N_AS_TIMEOUT: uds.utilities.TimeMillisecondsAlias = 1000
Timeout value of *N_As* time parameter according to ISO 15765-2.

N_AR_TIMEOUT: uds.utilities.TimeMillisecondsAlias = 1000
Timeout value of *N_Ar* time parameter according to ISO 15765-2.

N_BS_TIMEOUT: uds.utilities.TimeMillisecondsAlias = 1000
Timeout value of *N_Bs* time parameter according to ISO 15765-2.

N_CR_TIMEOUT: uds.utilities.TimeMillisecondsAlias = 1000
Timeout value of *N_Cr* time parameter according to ISO 15765-2.

DEFAULT_N_BR: uds.utilities.TimeMillisecondsAlias = 0
Default value of *N_Br* time parameter.

DEFAULT_N_CS: uds.utilities.TimeMillisecondsAlias | None
Default value of *N_Cs* time parameter.

class uds.transport_interface.can_transport_interface.**PyCanTransportInterface**(*can_bus_manager*, *addressing_information*, ***kwargs*)

Bases: *AbstractCanTransportInterface*

Transport Interface for managing UDS on CAN with python-can package as bus handler.

Note: Documentation for python-can package: https://python-can.readthedocs.io/

Create python-can Transport Interface.

Parameters

- **can_bus_manager** (*can.BusABC*) – Python-can bus object for handling CAN.

  Warning: Bus must have capability of receiving transmitted frames (`receive_own_messages=True` set).

- **addressing_information** (*uds.can.AbstractCanAddressingInformation*) – Addressing Information of CAN Transport Interface.
- **kwargs** (*Any*) – Optional arguments that are specific for CAN bus.
  - **parameter n_as_timeout**
    Timeout value for *N_As* time parameter.
  - **parameter n_ar_timeout**
    Timeout value for *N_Ar* time parameter.
  - **parameter n_bs_timeout**
    Timeout value for *N_Bs* time parameter.

- **parameter n_br**
  Value of *N_Br* time parameter to use in communication.
- **parameter n_cs**
  Value of *N_Cs* time parameter to use in communication.
- **parameter n_cr_timeout**
  Timeout value for *N_Cr* time parameter.
- **parameter dlc**
  Base CAN DLC value to use for CAN Packets.
- **parameter use_data_optimization**
  Information whether to use CAN Frame Data Optimization.
- **parameter filler_byte**
  Filler byte value to use for CAN Frame Data Padding.

property n_as_measured: uds.utilities.TimeMillisecondsAlias | None

Get the last measured value of N_As time parameter.

Returns
Time in milliseconds or None if the value was never measured.

Return type
Optional[uds.utilities.TimeMillisecondsAlias]

property n_ar_measured: uds.utilities.TimeMillisecondsAlias | None

Get the last measured value of N_Ar time parameter.

Returns
Time in milliseconds or None if the value was never measured.

Return type
Optional[uds.utilities.TimeMillisecondsAlias]

property n_bs_measured: uds.utilities.TimeMillisecondsAlias | None

Get the last measured value of N_Bs time parameter.

Returns
Time in milliseconds or None if the value was never measured.

Return type
Optional[uds.utilities.TimeMillisecondsAlias]

property n_cr_measured: uds.utilities.TimeMillisecondsAlias | None

Get the last measured value of N_Cr time parameter.

Returns
Time in milliseconds or None if the value was never measured.

Return type
Optional[uds.utilities.TimeMillisecondsAlias]

_MAX_LISTENER_TIMEOUT: float = 4280000.0

Maximal timeout value accepted by python-can listeners.

_MIN_NOTIFIER_TIMEOUT: float = 1e-07

Minimal timeout for notifiers that does not cause malfunctioning of listeners.

__del__()

Safely close all threads open by this object.

_teardown_notifier(*suppress_warning=False*)

Stop and remove CAN frame notifier for synchronous communication.

Parameters
suppress_warning (*bool*) – Do not warn about mixing Synchronous and Asynchronous implementation.

Return type
None

_teardown_async_notifier(*suppress_warning=False*)

Stop and remove CAN frame notifier for asynchronous communication.

Parameters
suppress_warning (*bool*) – Do not warn about mixing Synchronous and Asynchronous implementation.

Return type
None

_setup_notifier()

Configure CAN frame notifier for synchronous communication.

Return type
None

_setup_async_notifier(*loop*)

Configure CAN frame notifier for asynchronous communication.

Parameters
loop (*asyncio.AbstractEventLoop*) – An `asyncio` event loop to use.

Return type
None

clear_frames_buffers()

Clear buffers with transmitted and received frames.

> **Warning:** This will cause that all CAN packets received in a past are no longer accessible.

Return type
None

static is_supported_bus_manager(*bus_manager*)

Check whether provided value is a bus manager that is supported by this Transport Interface.

Parameters
bus_manager (*Any*) – Value to check.

Returns
True if provided bus object is compatible with this Transport Interface, False otherwise.

Return type
bool

send_packet(*packet*)

Transmit CAN packet.

> **Warning:** Must not be called within an asynchronous function.

Parameters
packet (*uds.packet.CanPacket*) – CAN packet to send.

Returns
Record with historic information about transmitted CAN packet.

Return type
uds.packet.CanPacketRecord

receive_packet(*timeout=None*)

Receive CAN packet.

> **Warning:** Must not be called within an asynchronous function.

Parameters
timeout (*Optional[uds.utilities.TimeMillisecondsAlias]*) – Maximal time (in milliseconds) to wait.

Raises

- **TypeError** – Provided timeout value is not None neither int nor float type.
- **ValueError** – Provided timeout value is less or equal 0.
- **TimeoutError** – Timeout was reached.

Returns
Record with historic information about received CAN packet.

Return type
uds.packet.CanPacketRecord

async async_send_packet(*packet*, *loop=None*)

Transmit CAN packet.

Parameters

- **packet** (*uds.packet.CanPacket*) – CAN packet to send.
- **loop** (*Optional[asyncio.AbstractEventLoop]*) – An asyncio event loop used for observing messages.

Raises
TypeError – Provided packet is not CAN Packet.

Returns
Record with historic information about transmitted CAN packet.

Return type
uds.packet.CanPacketRecord

async async_receive_packet(*timeout=None*, *loop=None*)

Receive CAN packet.

Parameters

- **timeout** (*Optional[uds.utilities.TimeMillisecondsAlias]*) – Maximal time (in milliseconds) to wait.
- **loop** (*Optional[asyncio.AbstractEventLoop]*) – An asyncio event loop used for observing messages.

Raises

- **TypeError** – Provided timeout value is not None neither int nor float type.
- **ValueError** – Provided timeout value is less or equal 0.
- **TimeoutError** – Timeout was reached.

Returns
Record with historic information about received CAN packet.

Return type
uds.packet.CanPacketRecord

uds.utilities

Various helper functions, classes and variables that are shared and reused within the project.

Submodules

uds.utilities.bytes_operations

Module with bytes list operations implementation.

Module Contents

Classes

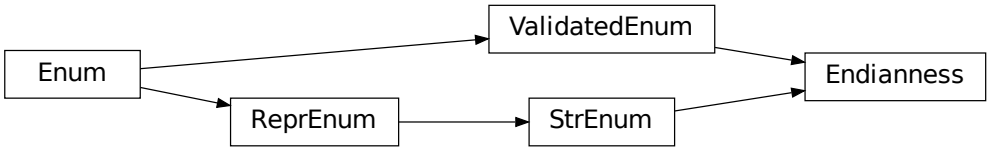
Endianness	Endianness values definitions.

Functions

bytes_list_to_int(bytes_list[, endianness])	Convert a list of bytes to integer value.
int_to_bytes_list(int_value[, list_size, endianness])	Convert integer value to a list of bytes.

class uds.utilities.bytes_operations.**Endianness**

Bases: aenum.StrEnum, *uds.utilities.enums.ValidatedEnum*



Endianness values definitions.

Endianness determines order of bytes in a bytes sequence.

Initialize self. See help(type(self)) for accurate signature.

LITTLE_ENDIAN: *Endianness* = 'little'

Little Endian stores the most significant byte at the largest memory address and the least significant byte at the smallest.

BIG_ENDIAN: *Endianness* = 'big'

Big Endian stores the most significant byte at the smallest memory address and the least significant byte at the largest.

uds.utilities.bytes_operations.**bytes_list_to_int**(*bytes_list*, *endianness=Endianness.BIG_ENDIAN*)

Convert a list of bytes to integer value.

Parameters

- **bytes_list** (*uds.utilities.common_types.RawBytesAlias*) – List of bytes to convert.
- **endianness** (Endianness) – Order of bytes to use.

Returns

The integer value represented by provided list of bytes.

Return type

int

uds.utilities.bytes_operations.**int_to_bytes_list**(*int_value*, *list_size=None*, *endianness=Endianness.BIG_ENDIAN*)

Convert integer value to a list of bytes.

Parameters

- **int_value** (*int*) – Integer value to convert.
- **list_size** (*Optional[int]*) – Size of the output list. Use None to use the smallest possible list size.
- **endianness** (Endianness) – Order of bytes to use.

Raises

- **TypeError** – At least one provided value has invalid type.
- **ValueError** – At least one provided value is out of range.
- ***InconsistentArgumentsError*** – Provided value of *list_size* is too small to contain entire *int_value*.

- **NotImplementedError** – There is missing implementation for the provided Endianness. Please create an issue in our Issues Tracking System with detailed description if you face this error.

Returns
The value of bytes list that represents the provided integer value.

Return type
uds.utilities.common_types.RawBytesListAlias

uds.utilities.common_types

Module with all common types (and its aliases) used in the package and helper functions for these types.

Module Contents

Functions

validate_nibble(value)	Validate whether provided value stores a nibble value.
validate_raw_byte(value)	Validate whether provided value stores a raw byte value.
validate_raw_bytes(value[, allow_empty])	Validate whether provided value stores raw bytes value.

Attributes

TimeMillisecondsAlias	Alias of a time value in milliseconds.
RawBytesTupleAlias	Alias of a tuple filled with byte values.
RawBytesSetAlias	Alias of a set filled with byte values.
RawBytesListAlias	Alias of a list filled with byte values.
RawBytesAlias	Alias of a sequence filled with byte values.

uds.utilities.common_types.**TimeMillisecondsAlias**
Alias of a time value in milliseconds.

uds.utilities.common_types.**RawBytesTupleAlias**
Alias of a tuple filled with byte values.

uds.utilities.common_types.**RawBytesSetAlias**
Alias of a set filled with byte values.

uds.utilities.common_types.**RawBytesListAlias**
Alias of a list filled with byte values.

uds.utilities.common_types.**RawBytesAlias**
Alias of a sequence filled with byte values.

uds.utilities.common_types.**validate_nibble**(*value*)
Validate whether provided value stores a nibble value.

Parameters
value (*int*) – Value to validate.

Raises

- **TypeError** – Value is not int type.
- **ValueError** – Value is out of byte range (0x0-0xF).

Return type
None

uds.utilities.common_types.**validate_raw_byte**(*value*)

Validate whether provided value stores a raw byte value.

Parameters
value (*int*) – Value to validate.

Raises

- **TypeError** – Value is not int type.
- **ValueError** – Value is out of byte range (0x00-0xFF).

Return type
None

uds.utilities.common_types.**validate_raw_bytes**(*value*, *allow_empty=False*)

Validate whether provided value stores raw bytes value.

Parameters

- **value** (*RawBytesAlias*) – Value to validate.
- **allow_empty** (*bool*) – True if empty list is allowed, False otherwise.

Raises

- **TypeError** – Value is not tuple or list type.
- **ValueError** – Value does not contain raw bytes (int value between 0x00-0xFF) only.

Return type
None

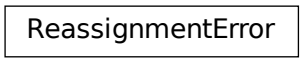
uds.utilities.custom_exceptions

Custom exception that are used within the project.

Module Contents

exception uds.utilities.custom_exceptions.**ReassignmentError**

Bases: Exception



An attempt to set a new value to an unchangeable attribute.

Example:

Objects of class X are initialized with an attribute const_x that must not be changed after the object creation (outside __init__ method).

ReassignmentError would be raised when a user tries to change the value of const_x attribute after the object is initialized.

Initialize self. See help(type(self)) for accurate signature.

exception uds.utilities.custom_exceptions.**InconsistentArgumentsError**

Bases: ValueError

Provided arguments values are not compatible with each other.

Example:

A function takes two parameters: *a*, *b*

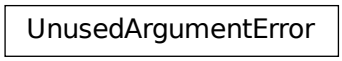
Let's assume that the function requires that: $a > b$

The function would raise InconsistentArgumentsError when values of *a* and *b* are not satisfying the requirement ($a > b$), e.g. $a = 0, b = 1$.

Initialize self. See help(type(self)) for accurate signature.

exception uds.utilities.custom_exceptions.**UnusedArgumentError**

Bases: ValueError



At least one argument (that was provided by user) will be ignored.

Example:

A function takes two parameters: a, b

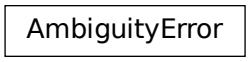
Let's assume that parameter *a* must always be provided. Parameter *b* is used only when $a == 1$.

The function would raise this exception when both parameters are provided but $a != 1$.

Initialize self. See help(type(self)) for accurate signature.

exception uds.utilities.custom_exceptions.**AmbiguityError**

Bases: ValueError



Operation cannot be executed because it is ambiguous.

Initialize self. See help(type(self)) for accurate signature.

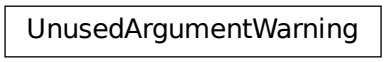
uds.utilities.custom_warnings

Custom warnings that are used within the project.

Module Contents

exception uds.utilities.custom_warnings.**UnusedArgumentWarning**

Bases: Warning



At least one argument (that was provided by user) will be ignored.

It is meant to be used in less strict situation than *UnusedArgumentError*.

Example:

A function takes two parameters: a, b

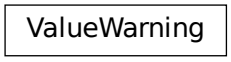
Let's assume that parameter *a* must always be provided. Parameter *b* is used only when *a == 1*.

The function would warn (using this warning) when both parameters are provided but *a != 1*.

Initialize self. See help(type(self)) for accurate signature.

exception uds.utilities.custom_warnings.**ValueWarning**

Bases: Warning



Value of the argument is out of typical range, but the package is able to handle it.

Initialize self. See help(type(self)) for accurate signature.

uds.utilities.enums

Module with common and reused implementation of enums.

Enumerated types (enums) are data types that consists of named values. This module provides extension to aenum package.

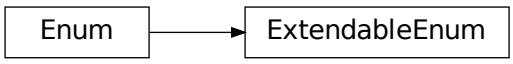
Module Contents

Classes

ExtendableEnum	Enum that supports new members adding.
ValidatedEnum	Enum that supports members validation.
ByteEnum	Enum which members are one byte integers (0x00-0xFF) only.
NibbleEnum	Enum which members are one nibble (4 bits) integers (0x0-0xF) only.

class uds.utilities.enums.**ExtendableEnum**

Bases: aenum.Enum



Enum that supports new members adding.

classmethod add_member(*name*, *value*)

Register a new member.

Parameters

- **name** (*str*) – Name of a new member.
- **value** (*Any*) – Value of a new member.

Raises

ValueError – Such name or value is already in use.

Returns

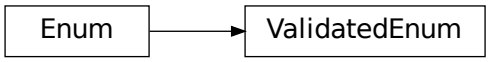
The new member that was just created.

Return type

aenum.Enum

class uds.utilities.enums.**ValidatedEnum**

Bases: aenum.Enum



Enum that supports members validation.

classmethod **is_member**(*value*)

Check whether given argument is a member or a value stored by this Enum.

Parameters
value (*Any*) – Value to check.

Returns
True if given argument is a member or a value of this Enum, else False.

Return type
bool

classmethod **validate_member**(*value*)

Validate whether given argument is a member or a value stored by this Enum.

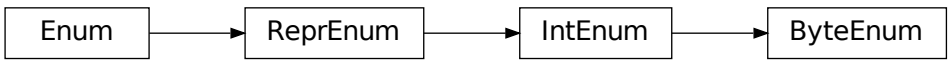
Parameters
value (*Any*) – Value to validate.

Raises
ValueError – Provided value is not a member neither a value of this Enum.

Return type
ValidatedEnum

class uds.utilities.enums.**ByteEnum**

Bases: aenum.IntEnum

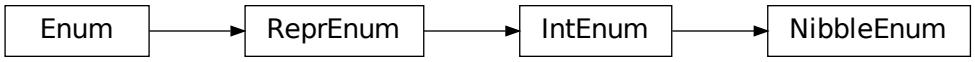


Enum which members are one byte integers (0x00-0xFF) only.

Initialize self. See help(type(self)) for accurate signature.

class uds.utilities.enums.**NibbleEnum**

Bases: aenum.IntEnum



Enum which members are one nibble (4 bits) integers (0x0-0xF) only.

Initialize self. See help(type(self)) for accurate signature.

4.1.2 Package Contents

uds.__version__ = '0.3.0'

uds.__author__ = 'Maciej Dąbrowski'

uds.__maintainer__ = 'Maciej Dąbrowski'

uds.__credits__ = ['Maciej Dąbrowski
(https://www.linkedin.com/in/maciej-dabrowski-test-engineer/)', 'Merit...

uds.__email__ = 'uds-package-development@googlegroups.com'

uds.__license__ = 'MIT'

CHAPTER

FIVE

UDS KNOWLEDGE BASE

If you are not an UDS expert, this part of documentation is created for you. It is meant to provide a technical support for every user of UDS package so you can better understand the code, but also UDS protocol itself.

5.1 UDS OSI Model

Overview of UDS OSI model.

5.1.1 UDS Standards

UDS is defined by multiple standards which are the main source of information and requirements about this protocol. Full list of standards is included in the table below:

OSI Layer	Common	CAN	FlexRay	Ethernet	K-Line	LIN
Layer 7 Application	ISO 14229-1 ISO 27145-3	ISO 14229-3	ISO 14229-4	ISO 14229-5	ISO 14229-6	ISO 14229-7
Layer 6 Presentation	ISO 27145-2					
Layer 5 Session	ISO 14229-2					
Layer 4 Transport	ISO 27145-4	ISO 15765-2	ISO 10681-2	ISO 13400-2	Not applicable	ISO 17987-2
Layer 3 Network						
Layer 2 Data		ISO 11898-1	ISO 17458-2	ISO 13400-3	ISO 14230-2	ISO 17987-3
Layer 1 Physical		ISO 11898-2 ISO 11898-3	ISO 17458-4		ISO 14230-1	ISO 17987-4

Where:

- OSI Layer - OSI Model Layer for which standards are relevant
- Common - standards mentioned in this column are always relevant for UDS communication regardless of bus used
- CAN - standards which are specific for UDS on CAN implementation
- FlexRay - standards which are specific for UDS on FlexRay implementation
- Ethernet - standards which are specific for UDS on IP implementation
- K-Line - standards which are specific for UDS on K-Line implementation

- LIN - standards which are specific for UDS on LIN implementation

5.1.2 UDS Functionalities

An overview of features that are required to fully implement UDS protocol is presented in the table below:

OSI Layer	Functionalities	Implementation
Layer 7 Application	• diagnostic messages support	• `uds.message`
Layer 6 Presentation	• diagnostic messages data interpretation • messaging database import from a file • messaging database export to a file	*To be provided with Database feature.*
Layer 5 Session	• Client simulation • Server simulation	*To be provided with Client feature.* *To be provided with Server feature.*
Layer 4 Transport	• UDS packet support • bus specific segmentation • bus specific packets transmission	• `uds.packet` • `uds.segmentation` • `uds.transport_interface` • `uds.can`
Layer 3 Network		*To be extended with support for:* • *Ethernet* • *LIN* • *K-Line* • *FlexRay*
Layer 2 Data	• frames transmission • frames receiving	External python packages for bus handling: • CAN: • python-can
Layer 1 Physical		*More packages handling other buses to be decided.*

Where:

- OSI Layer - considered OSI Model Layer
- Functionalities - functionalities required in the implementation to handle considered UDS OSI layer
- Implementation - UDS package implementation that provides mentioned functionalities

5.1.3 Protocol Data Units

Each layer of OSI Model defines their own Protocol Data Unit (PDU). To make things simpler for the users and our developers, in the implementation we distinguish following PDUs:

- Application Protocol Data Unit (A_PDU) - called *diagnostic message* or *UDS Message* in the implementation and documentation. More information about A_PDU can be found in:
  - *knowledge base section - diagnostic message*
  - *implementation - diagnostic message*
- Network Protocol Data Unit (N_PDU) - called *UDS packet* in the implementation and documentation. More information about N_PDU can be found in:
  - *knowledge base section - UDS packet*
  - implementation - `uds.packet`
- Data Protocol Data Unit (D_PDU) - called *frame* in the implementation and documentation. We do not have any internal frames documentation. Implementation of frames is usually provided by external packages.

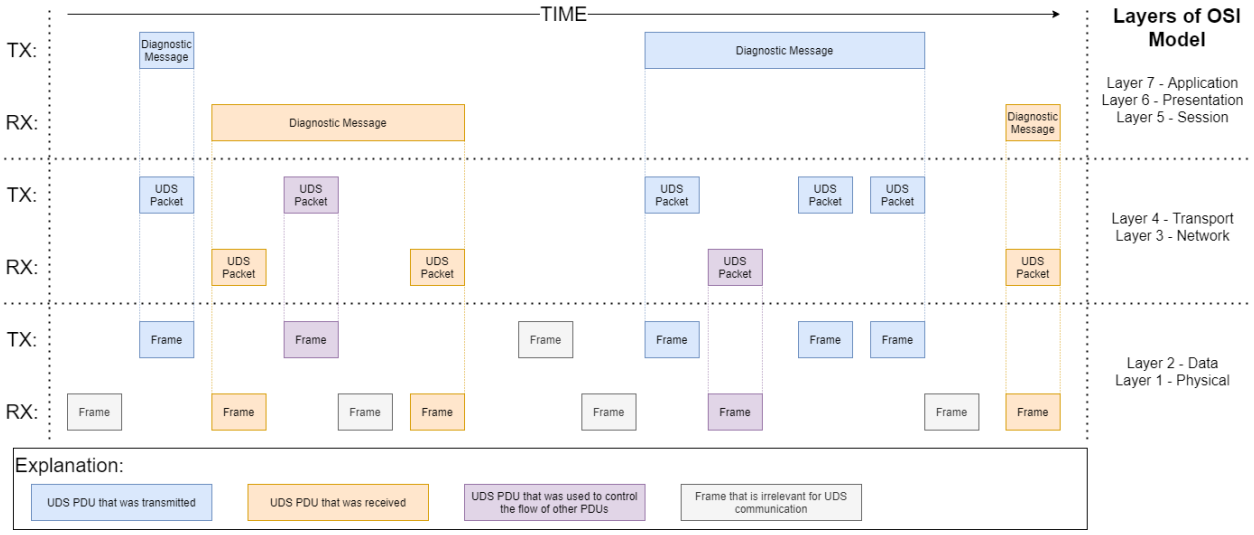


Fig. 1: UDS Protocol Data Units on different layers of OSI Model.

5.2 Diagnostic Message

Messages that are exchanged by clients and servers during UDS communications are usually called *diagnostic messages*. In the documentation and the implementation, *UDS message* name is also in use.

We distinguish two types of diagnostic messages depending on who is a transmitter:

- *diagnostic request*
- *diagnostic response*

UDS communication is always initiated by a client who sends a *diagnostic request* to a network that it has direct connection with. The client might not be directly connected to a desired recipient(s) of the request, therefore some servers might be forced to act as gateways and transmit the request to another sub-network(s). Servers' decision (whether to redirect a message to another sub-network) depends on a target(s) of the request i.e. server shall transmit the request to the sub-network if this is a route (not necessarily a direct one) to at least one recipient of the message.

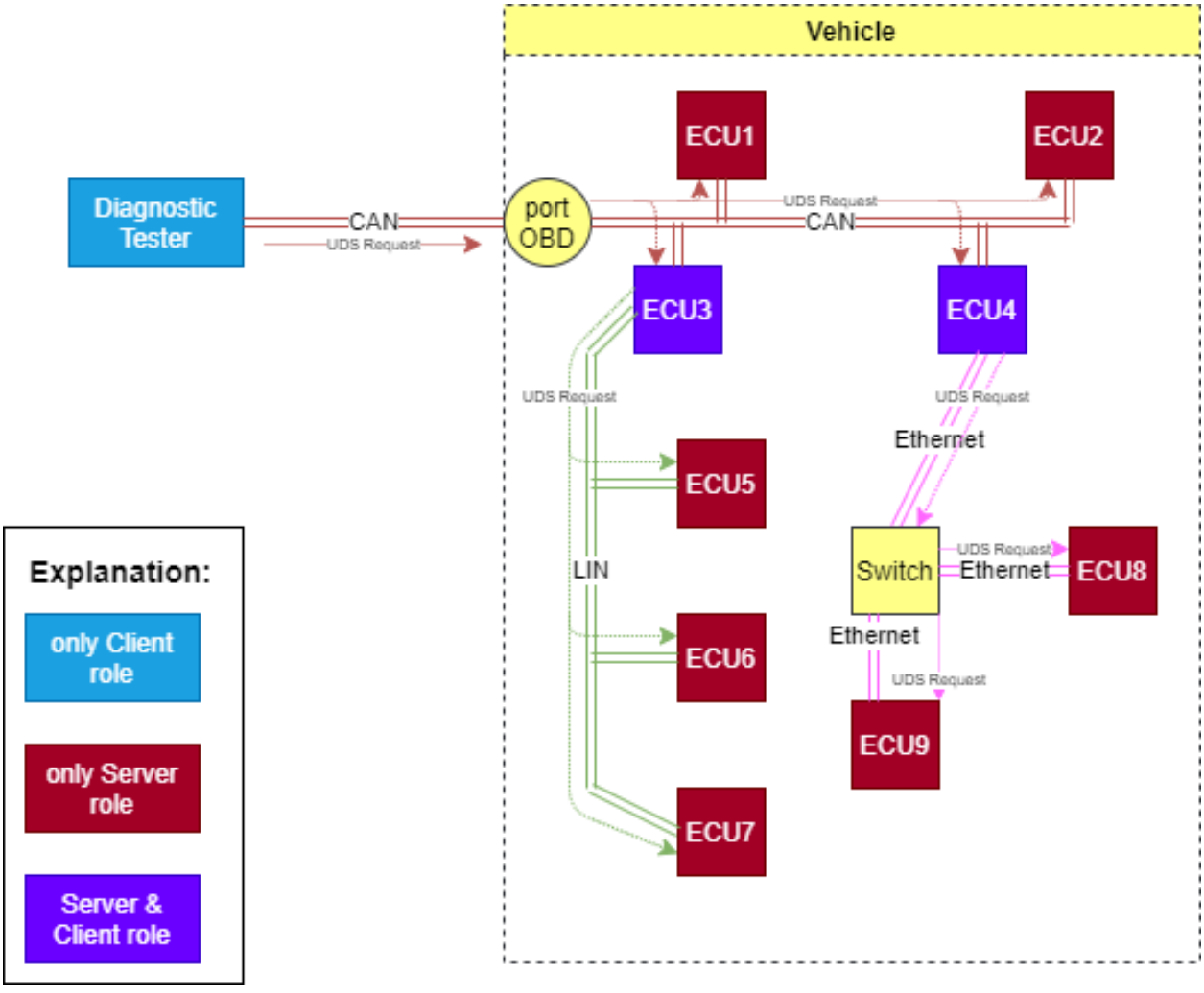


Fig. 2: Diagnostic request routing in example vehicle networks.
In this example all ECUs in the vehicle are the targets of the request - functionally addressed request was sent.

Each server which was the recipient of the request, might decide to send a response back to the nearest client (the one which previously transmitted the request in this sub-network). Then, the client shall act as a gateway again and redirect the response back until it reaches the request message originator (Diagnostic Tester).

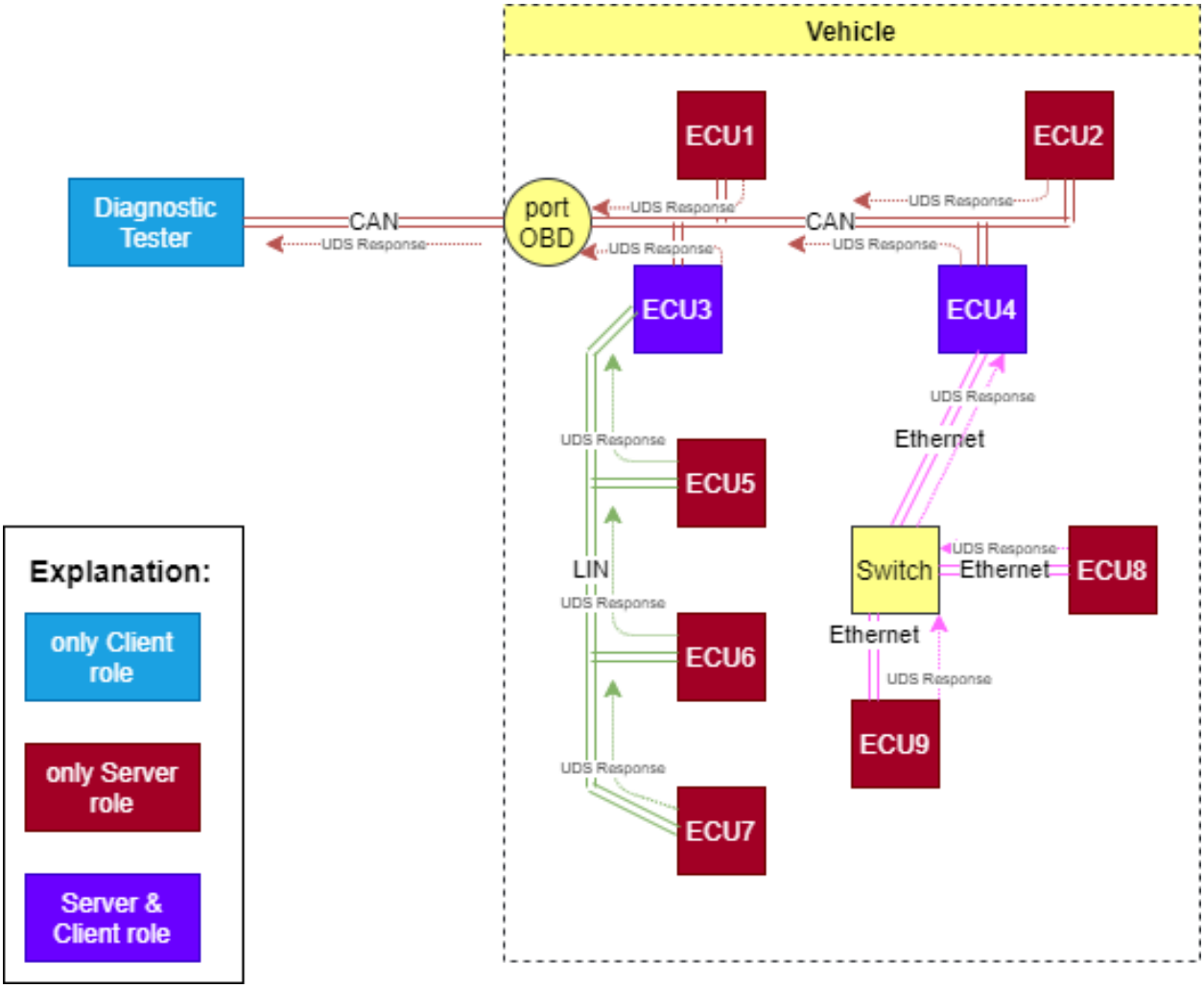


Fig. 3: Diagnostic responses routing in example vehicle networks.
In this example all ECUs in the vehicle responds to the request.

5.2.1 Diagnostic Request

Diagnostic request is a *diagnostic message* that was transmitted by a client and targets a server or group of servers. Diagnostic request can be identified by its *Service Identifier* (SID) value.

5.2.2 Diagnostic Response

Diagnostic response is a *diagnostic message* that was transmitted by a server and targets a client. Diagnostic response can be identified by its *Service Identifier* (SID) value.

UDS defines two formats of diagnostic responses:

- *positive response message*
- *negative response message*

Positive Response Message

If a server responds with a positive response message, it means that the server received the corresponding request message and executed actions requested by a client.

Format of positive response messages:

Byte	Description	Value
1	Response SID	SID + 0x40
2	data-parameter#1	XX
...	...	...
n	data-parameter#n	XX

Where:

- SID - *Service Identifier* value that was received in the request message to which the server responded
- XX - any byte value

Note: When positive diagnostic message is received, this equation is always true:

```
RSID = SID + 0x40
```

Negative Response Message

If a server responds with a negative response message, it means that (for some reason) the server could not execute actions requested by a client.

Format of negative response messages:

Byte	Description	Value
1	Negative Response SID	0x7F
2	Request SID	SID
3	NRC	XX

Where:

- SID - *Service Identifier* value that was received in the request message to which the server responded
- NRC - *Negative Response Code* value that identified the reason for negative response

5.2.3 Service Identifier

Service Identifier (SID) is data parameter that is always located in the first Application Data (A_Data) byte of each *diagnostic message* . SID value determines whether the message is *diagnostic request* or *diagnostic response*. General purpose (application) and format of *diagnostic message* is also by determined by SID value.

List of all Service Identifier (SID) values and their application:

- 0x00 - not applicable, reserved by ISO 14229-1
- 0x01-0x0F - ISO 15031-5/SAE J1979 specific services
- 0x10 - *DiagnosticSessionControl* service request
- 0x11 - *ECUReset* service request
- 0x12-0x13 - reserved by ISO 14229-1
- 0x14 - *ClearDiagnosticInformation* service request
- 0x15-0x18 - reserved by ISO 14229-1
- 0x19 - *ReadDTCInformation* service request
- 0x1A-0x21 - reserved by ISO 14229-1
- 0x22 - *ReadDataByIdentifier* service request
- 0x23 - *ReadMemoryByAddress* service request
- 0x24 - *ReadScalingDataByIdentifier* service request
- 0x25-0x26 - reserved by ISO 14229-1
- 0x27 - *SecurityAccess* service request
- 0x28 - *CommunicationControl* service request
- 0x29 - *Authentication* service request
- 0x2A - *ReadDataByPeriodicIdentifier* service request
- 0x2B - reserved by ISO 14229-1
- 0x2C - *DynamicallyDefineDataIdentifier* service request
- 0x2D - reserved by ISO 14229-1
- 0x2E - *WriteDataByIdentifier* service request
- 0x2F - *InputOutputControlByIdentifier* service request
- 0x30 - reserved by ISO 14229-1
- 0x31 - *RoutineControl* service request
- 0x32-0x33 - reserved by ISO 14229-1
- 0x34 - *RequestDownload* service request
- 0x35 - *RequestUpload* service request
- 0x36 - *TransferData* service request
- 0x37 - *RequestTransferExit* service request
- 0x38 - *RequestFileTransfer* service request
- 0x39-0x3C - reserved by ISO 14229-1

- 0x3D - *WriteMemoryByAddress* service request
- 0x3E - *TesterPresent* service request
- 0x3F - not applicable, reserved by ISO 14229-1
- 0x40 - not applicable, reserved by ISO 14229-1
- 0x41-0x4F - ISO 15031-5/SAE J1979 specific services
- 0x50 - positive response to *DiagnosticSessionControl* service
- 0x51 - positive response to *ECUReset* service
- 0x52-0x53 - reserved by ISO 14229-1
- 0x54 - positive response to *ClearDiagnosticInformation* service
- 0x55-0x58 - reserved by ISO 14229-1
- 0x59 - positive response to *ReadDTCInformation* service
- 0x5A-0x61 - reserved by ISO 14229-1
- 0x62 - positive response to *ReadDataByIdentifier* service
- 0x63 - positive response to *ReadMemoryByAddress* service
- 0x64 - positive response to *ReadScalingDataByIdentifier* service
- 0x65-0x66 - reserved by ISO 14229-1
- 0x67 - positive response to *SecurityAccess* service
- 0x68 - positive response to *CommunicationControl* service
- 0x69 - positive response to *Authentication* service
- 0x6A - positive response to *ReadDataByPeriodicIdentifier* service
- 0x6B - reserved by ISO 14229-1
- 0x6C - positive response to *DynamicallyDefineDataIdentifier* service
- 0x6D - reserved by ISO 14229-1
- 0x6E - positive response to *WriteDataByIdentifier* service
- 0x6F - positive response to *InputOutputControlByIdentifier* service
- 0x70 - reserved by ISO 14229-1
- 0x71 - positive response to *RoutineControl* service
- 0x72-0x73 - reserved by ISO 14229-1
- 0x74 - positive response to *RequestDownload* service
- 0x75 - positive response to *RequestUpload* service
- 0x76 - positive response to *TransferData* service
- 0x77 - positive response to *RequestTransferExit* service
- 0x78 - positive response to *RequestFileTransfer* service
- 0x79-0x7C - reserved by ISO 14229-1
- 0x7D - positive response to *WriteMemoryByAddress* service
- 0x7E - positive response to *TesterPresent* service

- 0x7F - negative response service identifier
- 0x80-0x82 - not applicable, reserved by ISO 14229-1
- 0x83 - reserved by ISO 14229-1
- 0x84 - *SecuredDataTransmission* service request
- 0x85 - *ControlDTCSetting* service request
- 0x86 - *ResponseOnEvent* service request
- 0x87 - *LinkControl* service request
- 0x88 - reserved by ISO 14229-1
- 0x89-0xB9 - not applicable, reserved by ISO 14229-1
- 0xBA-0xBE - system supplier specific service requests
- 0xBF-0xC2 - not applicable, reserved by ISO 14229-1
- 0xC3 - reserved by ISO 14229-1
- 0xC4 - positive response to *SecuredDataTransmission* service
- 0xC5 - positive response to *ControlDTCSetting* service
- 0xC6 - positive response to *ResponseOnEvent* service
- 0xC7 - positive response to *LinkControl* service
- 0xC8 - reserved by ISO 14229-1
- 0xC9-0xF9 - not applicable, reserved by ISO 14229-1
- 0xFA-0xFE - positive responses to system supplier specific requests
- 0xFF - not applicable, reserved by ISO 14229-1

DiagnosticSessionControl

DiagnosticSessionControl service is used to change diagnostic sessions in the server(s). In each diagnostic session a different set of diagnostic services (and/or functionalities) is enabled in the server. Server shall always be in exactly one diagnostic session.

ECUReset

ECUReset service is used by the client to request a server reset.

ClearDiagnosticInformation

ClearDiagnosticInformation service is used by the client to clear all diagnostic information (DTC and related data) in one or multiple servers' memory.

ReadDTCInformation

ReadDTCInformation service allows the client to read from any server or group of servers within a vehicle, current information about all Diagnostic Trouble Codes. This could be a status of reported Diagnostic Trouble Code (DTC), number of currently active DTCs or any other information returned by supported ReadDTCInformation SubFunctions.

ReadDataByIdentifier

ReadDataByIdentifier service allows the client to request data record values from the server identifier by one or more DataIdentifiers (DIDs).

ReadMemoryByAddress

ReadMemoryByAddress service allows the client to request server's memory data stored under provided memory address.

ReadScalingDataByIdentifier

ReadScalingDataByIdentifier service allows the client to request from the server a scaling data record identified by a DataIdentifier (DID). The scaling data contains information such as data record type (e.g. ASCII, signed float), formula and its coefficients used for value calculation, units, etc.

SecurityAccess

SecurityAccess service allows the client to unlock functions/services with restricted access.

CommunicationControl

CommunicationControl service allows the client to switch on/off the transmission and/or the reception of certain messages on a server(s).

Authentication

Authentication service provides a means for the client to prove its identity, allowing it to access data and/or diagnostic services, which have restricted access for, for example security, emissions, or safety reasons.

ReadDataByPeriodicIdentifier

ReadDataByPeriodicIdentifier service allows the client to request the periodic transmission of data record values from the server identified by one or more periodicDataIdentifiers.

DynamicallyDefineDataIdentifier

DynamicallyDefineDataIdentifier service allows the client to dynamically define in a server a DataIdentifier (DID) that can be read via the *ReadDataByIdentifier* service at a later time.

WriteDataByIdentifier

WriteDataByIdentifier service allows the client to write information into the server at an internal location specified by the provided DataIdentifier (DID).

InputOutputControlByIdentifier

InputOutputControlByIdentifier service allows the client to substitute a value for an input signal, internal server function and/or force control to a value for an output (actuator) of an electronic system.

RoutineControl

RoutineControl service allows the client to execute a defined sequence of steps to obtain any relevant result. There is a lot of flexibility with this service, but typical usage may include functionality such as erasing memory, resetting or learning adaptive data, running a self-test, overriding the normal server control strategy.

RequestDownload

RequestDownload service allows the client to initiate a data transfer from the client to the server (download).

RequestUpload

RequestUpload service allows the client to initiate a data transfer from the server to the client (upload).

TransferData

TransferData service is used by the client to transfer data either from the client to the server (download) or from the server to the client (upload).

RequestTransferExit

RequestTransferExit service is used by the client to terminate a data transfer between the client and server.

RequestFileTransfer

RequestFileTransfer service allows the client to initiate a file data transfer either from the server to the client (upload) or from the server to the client (upload).

WriteMemoryByAddress

WriteMemoryByAddress service allows the client to write information into server's memory data under provided memory address.

TesterPresent

TesterPresent service is used by the client to indicate to a server(s) that the client is still connected to a vehicle and certain diagnostic services and/or communication that have been previously activated are to remain active.

SecuredDataTransmission

SecuredDataTransmission service is applicable if a client intends to use diagnostic services defined in this document in a secured mode. It may also be used to transmit external data, which conform to some other application protocol, in a secured mode between a client and a server. A secured mode in this context means that the data transmitted is protected by cryptographic methods.

ControlDTCSetting

ControlDTCSetting service allows the client to stop or resume the updating of DTC status bits in the server(s) memory.

ResponseOnEvent

ResponseOnEvent service allows the client to request from the server to start ot stop transmission of responses on a specified event.

LinkControl

LinkControl service allows the client to control the communication between the client and the server(s) in order to gain bus bandwidth for diagnostic purposes (e.g. programming).

5.2.4 Negative Response Code

Negative Response Code (NRC) is one byte value which contains information why a server is not sending a positive response message.

List of NRC values:

- 0x00 - positiveResponse - This NRC shall not be used in a negative response message. This positiveResponse parameter value is reserved for server internal implementation.
- 0x00-0x0F - ISO Reserved - This range of values is reserved for future definition by ISO 14229 Standard.
- 0x10 - generalReject - This NRC indicates that the requested action has been rejected by the server.
- 0x11 - serviceNotSupported - This NRC indicates that the requested action will not be taken because the server does not support the requested service.
- 0x12 - SubFunctionNotSupported - This NRC indicates that the requested action will not be taken because the server does not support the service specific parameters of the request message.

- 0x13 - incorrectMessageLengthOrInvalidFormat - This NRC indicates that the requested action will not be taken because the length of the received request message does not match the prescribed length for the specified service or the format of the parameters do not match the prescribed format for the specified service.
- 0x14 - responseTooLong - This NRC shall be reported by the server if the response to be generated exceeds the maximum number of bytes available by the underlying network layer. This could occur if the response message exceeds the maximum size allowed by the underlying transport protocol or if the response message exceeds the server buffer size allocated for that purpose.
- 0x15-0x20 - ISO Reserved - This range of values is reserved for future definition by ISO 14229 Standard.
- 0x21 - busyRepeatRequest - This NRC indicates that the server is temporarily too busy to perform the requested operation. In this circumstance the client shall perform repetition of the "identical request message" or "another request message". The repetition of the request shall be delayed by a time specified in the respective implementation documents.
- 0x22 - conditionsNotCorrect - This NRC indicates that the requested action will not be taken because the server prerequisite conditions are not met.
- 0x23 - ISO Reserved - This value is reserved for future definition by ISO 14229 Standard.
- 0x24 - requestSequenceError - This NRC indicates that the requested action will not be taken because the server expects a different sequence of request messages or message as sent by the client. This may occur when sequence sensitive requests are issued in the wrong order.
- 0x25 - noResponseFromSubnetComponent - This NRC indicates that the server has received the request but the requested action could not be performed by the server as a subnet component which is necessary to supply the requested information did not respond within the specified time.
- 0x26 - FailurePreventsExecutionOfRequestedAction - This NRC indicates that the requested action will not be taken because a failure condition, identified by a DTC (with at least one DTC status bit for TestFailed, Pending, Confirmed or TestFailedSinceLastClear set to 1), has occurred and that this failure condition prevents the server from performing the requested action.
- 0x27-0x30 - ISO Reserved - This range of values is reserved for future definition by ISO 14229 Standard.
- 0x31 - requestOutOfRange - This NRC indicates that the requested action will not be taken because the server has detected that the request message contains a parameter which attempts to substitute a value beyond its range of authority (e.g. attempting to substitute a data byte of 111 when the data is only defined to 100), or which attempts to access a DataIdentifier/RoutineIdentifer that is not supported or not supported in active session.
- 0x32 - ISO Reserved - This value is reserved for future definition by ISO 14229 Standard.
- 0x33 - securityAccessDenied - This NRC indicates that the requested action will not be taken because the server's security strategy has not been satisfied by the client.
- 0x34 - authenticationRequired - This NRC indicates that the requested service will not be taken because the client has insufficient rights based on its Authentication state.
- 0x35 - invalidKey - This NRC indicates that the server has not given security access because the key sent by the client did not match with the key in the server's memory. This counts as an attempt to gain security.
- 0x36 - exceedNumberOfAttempts - This NRC indicates that the requested action will not be taken because the client has unsuccessfully attempted to gain security access more times than the server's security strategy will allow.
- 0x37 - requiredTimeDelayNotExpired - This NRC indicates that the requested action will not be taken because the client's latest attempt to gain security access was initiated before the server's required timeout period had elapsed.

- 0x38 - secureDataTransmissionRequired - This NRC indicates that the requested service will not be taken because the requested action is required to be sent using a secured communication channel.
- 0x39 - secureDataTransmissionNotAllowed - This NRC indicates that this message was received using the SecuredDataTransmission (SID 0x84) service. However, the requested action is not allowed to be sent using the SecuredDataTransmission (0x84) service.
- 0x3A - secureDataVerificationFailed - This NRC indicates that the message failed in the security sub-layer.
- 0x3B-0x4F - ISO Reserved - This range of values is reserved for future definition by ISO 14229 Standard.
- 0x50 - Certificate verification failed, Invalid Time Period - Date and time of the server does not match the validity period of the Certificate.
- 0x51 - Certificate verification failed, Invalid Signature - Signature of the Certificate could not be verified.
- 0x52 - Certificate verification failed, Invalid Chain of Trust - Certificate could not be verified against stored information about the issuing authority.
- 0x53 - Certificate verification failed, Invalid Type - Certificate does not match the current requested use case.
- 0x54 - Certificate verification failed, Invalid Format - Certificate could not be evaluated because the format requirement has not been met.
- 0x55 - Certificate verification failed, Invalid Content - Certificate could not be verified because the content does not match.
- 0x56 - Certificate verification failed, Invalid Scope - The scope of the Certificate does not match the contents of the server.
- 0x57 - Certificate verification failed, Invalid Certificate (revoked) - Certificate received from client is invalid, because the server has revoked access for some reason.
- 0x58 - Ownership verification failed - Delivered Ownership does not match the provided challenge or could not verified with the own private key.
- 0x59 - Challenge calculation failed - The challenge could not be calculated on the server side.
- 0x5A - Setting Access Rights failed - The server could not set the access rights.
- 0x5B - Session key creation/derivation failed - The server could not create or derive a session key.
- 0x5C - Configuration data usage failed - The server could not work with the provided configuration data.
- 0x5D - DeAuthentication failed - DeAuthentication was not successful, server could still be unprotected.
- 0x5E-0x6F - ISO Reserved - This range of values is reserved for future definition by ISO 14229 Standard.
- 0x70 - uploadDownloadNotAccepted - This NRC indicates that an attempt to upload/download to a server's memory cannot be accomplished due to some fault conditions.
- 0x71 - transferDataSuspended - This NRC indicates that a data transfer operation was halted due to some fault. The active transferData sequence shall be aborted.
- 0x72 - generalProgrammingFailure - This NRC indicates that the server detected an error when erasing or programming a memory location in the permanent memory device (e.g. Flash Memory).
- 0x73 - wrongBlockSequenceCounter - This NRC indicates that the server detected an error in the sequence of blockSequenceCounter values. Note that the repetition of a TransferData request message with a blockSequenceCounter equal to the one included in the previous TransferData request message shall be accepted by the server.
- 0x74-0x77 - ISO Reserved - This range of values is reserved for future definition by ISO 14229 Standard.

- 0x78 - requestCorrectlyReceived-ResponsePending - This NRC indicates that the request message was received correctly, and that all parameters in the request message were valid (these checks can be delayed until after sending this NRC if executing the boot software), but the action to be performed is not yet completed and the server is not yet ready to receive another request. As soon as the requested service has been completed, the server shall send a positive response message or negative response message with a response code different from this.
- 0x79-0x7D - ISO Reserved - This range of values is reserved for future definition by ISO 14229 Standard.
- 0x7E - SubFunctionNotSupportedInActiveSession - This NRC indicates that the requested action will not be taken because the server does not support the requested SubFunction in the session currently active. This NRC shall only be used when the requested SubFunction is known to be supported in another session, otherwise response code SubFunctionNotSupported shall be used.
- 0x7F - serviceNotSupportedInActiveSession - This NRC indicates that the requested action will not be taken because the server does not support the requested service in the session currently active. This NRC shall only be used when the requested service is known to be supported in another session, otherwise response code serviceNotSupported shall be used.
- 0x80 - ISO Reserved - This value is reserved for future definition by ISO 14229 Standard.
- 0x81 - rpmTooHigh - This NRC indicates that the requested action will not be taken because the server prerequisite condition for RPM is not met (current RPM is above a preprogrammed maximum threshold).
- 0x82 - rpmTooLow - This NRC indicates that the requested action will not be taken because the server prerequisite condition for RPM is not met (current RPM is below a preprogrammed minimum threshold).
- 0x83 - engineIsRunning - This NRC is required for those actuator tests which cannot be actuated while the Engine is running. This is different from RPM too high negative response, and shall be allowed.
- 0x84 - engineIsNotRunning - This NRC is required for those actuator tests which cannot be actuated unless the Engine is running. This is different from RPM too low negative response, and shall be allowed.
- 0x85 - engineRunTimeTooLow - This NRC indicates that the requested action will not be taken because the server prerequisite condition for engine run time is not met (current engine run time is below a preprogrammed limit).
- 0x86 - temperatureTooHigh - This NRC indicates that the requested action will not be taken because the server prerequisite condition for temperature is not met (current temperature is above a preprogrammed maximum threshold).
- 0x87 - temperatureTooLow - This NRC indicates that the requested action will not be taken because the server prerequisite condition for temperature is not met (current temperature is below a preprogrammed minimum threshold).
- 0x88 - vehicleSpeedTooHigh - This NRC indicates that the requested action will not be taken because the server prerequisite condition for vehicle speed is not met (current VS is above a preprogrammed maximum threshold).
- 0x89 - vehicleSpeedTooLow - This NRC indicates that the requested action will not be taken because the server prerequisite condition for vehicle speed is not met (current VS is below a preprogrammed minimum threshold).
- 0x8A - throttle/PedalTooHigh - This NRC indicates that the requested action will not be taken because the server prerequisite condition for throttle/pedal position is not met (current throttle/pedal position is above a preprogrammed maximum threshold).
- 0x8B - throttle/PedalTooLow - This NRC indicates that the requested action will not be taken because the server prerequisite condition for throttle/pedal position is not met (current throttle/pedal position is below a preprogrammed minimum threshold).

- 0x8C - transmissionRangeNotInNeutral - This NRC indicates that the requested action will not be taken because the server prerequisite condition for being in neutral is not met (current transmission range is not in neutral).
- 0x8D - transmissionRangeNotInGear - This NRC indicates that the requested action will not be taken because the server prerequisite condition for being in gear is not met (current transmission range is not in gear).
- 0x8E - ISO Reserved - This value is reserved for future definition by ISO 14229 Standard.
- 0x8F - brakeSwitch(es)NotClosed (Brake Pedal not pressed or not applied) - This NRC indicates that for safety reasons, this is required for certain tests before it begins, and shall be maintained for the entire duration of the test.
- 0x90 - shifterLeverNotInPark - This NRC indicates that for safety reasons, this is required for certain tests before it begins, and shall be maintained for the entire duration of the test.
- 0x91 - torqueConverterClutchLocked - This NRC indicates that the requested action will not be taken because the server prerequisite condition for torque converter clutch is not met (current torque converter clutch status above a preprogrammed limit or locked).
- 0x92 - voltageTooHigh - This NRC indicates that the requested action will not be taken because the server prerequisite condition for voltage at the primary pin of the server (ECU) is not met (current voltage is above a preprogrammed maximum threshold).
- 0x93 - voltageTooLow - This NRC indicates that the requested action will not be taken because the server prerequisite condition for voltage at the primary pin of the server (ECU) is not met (current voltage is below a preprogrammed minimum threshold).
- 0x94 - ResourceTemporarilyNotAvailable - This NRC indicates that the server has received the request but the requested action could not be performed by the server because an application which is necessary to supply the requested information is temporality not available. This NRC is in general supported by each diagnostic service, as not otherwise stated in the data link specific implementation document, therefore it is not listed in the list of applicable response codes of the diagnostic services.
- 0x95-0xEF - reservedForSpecificConditionsNotCorrect - This range of values is reserved for future definition condition not correct scenarios by ISO 14229 Standard.
- 0xF0-0xFE - vehicleManufacturerSpecificConditionsNotCorrect - This range of values is reserved for vehicle manufacturer specific condition not correct scenarios.
- 0xFF - ISO Reserved - This value is reserved for future definition by ISO 14229 Standard.

5.2.5 Addressing

Addressing determines model of UDS communication.

We distinguish following addressing types:

- *Physical*
- *Functional*

Physical

Physical addressing is used to send a dedicated message to a certain server (ECU). When physically addressed messages are sent, the direct (point-to-point) communication between the client and the server takes place. The server shall respond to a physically addressed request unless the request contains an information that a response is not required (further explained in *response behaviour to physically addressed request* chapter).

Note: You do not need a direct physical connection between a client and a server to have physically addressed communication as all messages shall be routed to a target of each message.

Response behaviour to physically addressed request

Expected server behaviour in case of receiving physically addressed request message with SubFunction parameter:

Client request		**Server capability**			**Server response**		**Comment**
Addressing	**SPRM**	**SID supported**	**SF supported**	**DataParam supported**	**Message**	**NRC**	
physical	False (bit = 0)	YES	YES	At least 1	Positive Response	—	Server supports the requests and sends positive response.
				At least 1	Negative Response	NRC = XX	Server sends negative response because an error occurred processing the data parameters of request message.
				None		NRC = ROOR	Servers sends negative response with NRC 0x31.
		NO	—	—		NRC = SNS or SNSIAS	Servers sends negative response with NRC 0x11 or 0x7F.
		YES	NO	—		NRC = SFNS or SFNSIAS	Servers sends negative response with NRC 0x12 or 0x7E.
	True (bit = 1)	YES	YES	At least 1	No Response	—	Server does not send a response.
				At least 1	Negative Response	NRC = XX	Server sends negative response because an error occurred processing the data parameters of request message.
				None		NRC = ROOR	Servers sends negative response with NRC 0x31.
		NO	—	—		NRC = SNS or SNSIAS	Servers sends negative response with NRC 0x11 or 0x7F.
		YES	NO	—		NRC = SFNS or SFNSIAS	Servers sends negative response with NRC 0x12 or 0x7E.

Expected server behaviour in case of receiving physically addressed request message without SubFunction parameter:

Client request	Server capability		Server response		Comment
Addressing	**SID supported**	**DataParam supported**	**Message**	**NRC**	
physical	YES	All	Positive Response	—	Server supports the requests and sends positive response.
		At least 1		—	Server supports the requests and sends positive response.
		At least 1	Negative Response	NRC = XX	Server sends negative response because an error occurred processing the data parameters of request message.
		None		NRC = ROOR	Servers sends negative response with NRC 0x31.
	NO	—		NRC = SNS or SNSIAS	Servers sends negative response with NRC 0x11 or 0x7F

Where:

- SPRMIB - flag informing whether Suppress Positive Response Message Indication Bit is set in the received request message
- SID supported - flag informing whether Service Identifier in the received request message is supported by the server
- SF supported - flag informing whether SubFunction in the received request message is supported by the server
- DataParam supported - information whether values of data parameters (e.g. DIDs, RIDs, DTCStatusMask) in the received request message are supported by the server
- NRC - Negative Response Code
- ROOR - NRC 0x31 (requestOutOfRange)
- SNS - NRC 0x11 (serviceNotSupported)
- SNSIAS - NRC 0x7F (serviceNotSupportedInActiveSession)
- SFNS - NRC 0x12 (SubFunctionNotSupported)
- SFNSIAS - NRC 0x7E (SubFunctionNotSupportedInActiveSession)
- XX - NRC code that is supported by the server and suitable to the current situation (e.g. NRC 0x21 busyRepeatRequest if server is currently overloaded and cannot process next request message)

Functional

Functional addressing is used to send messages to multiple servers (ECUs) in the network. When functionally addressed messages are sent, a one to many communication between a client and servers (ECUs) takes place. A server shall only respond to certain functionally addressed requests (further explained in *response behaviour to functionally addressed request* chapter.

Note: Some types of buses (e.g. LIN) might also support broadcast communication which slightly change expected server behaviour. When broadcast communication is used, then a server response is never expected by a client.

Response behaviour to functionally addressed request

Expected server behaviour in case of receiving functionally addressed request message with SubFunction parameter:

Client request		**Server capability**			**Server response**		**Comment**
Addressing	**SPRMI**	**SID supported**	**SF supported**	**DataParam supported**	**Message**	**NRC**	
functional	False (bit = 0)	YES	YES	At least 1	Positive Response	—	Server supports the requests and sends positive response.
				At least 1	Negative Response	NRC = XX	Server sends negative response because an error occurred processing the data parameters of request message.
				None	No Response	—	Server does not send a response.
		NO	—	—		—	Server does not send a response.
		YES	NO	—		—	Server does not send a response.
	True (bit = 1)	YES	YES	At least 1	No Response	—	Server does not send a response.
				At least 1	Negative Response	NRC = XX	Server sends negative response because an error occurred processing the data parameters of request message.
				None	No Response	—	Server does not send a response.
		NO	—	—		—	Server does not send a response.
		YES	NO	—		—	Server does not send a response.

Expected server behaviour in case of receiving functionally addressed request message without SubFunction parameter:

Client request	Server capability		Server response		Comment
Addressing	**SID supported**	**DataParam supported**	**Message**	**NRC**	
functional	YES	All	Positive Response	—	Server supports the requests and sends positive response.
		At least 1		—	Server supports the requests and sends positive response.
		At least 1	Negative Response	NRC = XX	Server sends negative response because an error occurred processing the data parameters of request message.
		None	No Response	—	Server does not send a response.
	NO	—		—	Server does not send a response.

Where:

- SPRMIB - flag informing whether Suppress Positive Response Message Indication Bit is set in the received request message
- SID supported - flag informing whether Service Identifier in the received request message is supported by the server
- SF supported - flag informing whether SubFunction in the received request message is supported by the server
- DataParam supported - information whether values of data parameters (e.g. DIDs, RIDs, DTCStatusMask) in the received request message are supported by the server
- NRC - Negative Response Code
- XX - NRC code that is supported by the server and suitable to the current situation (e.g. NRC 0x21 busyRepeatRequest if server is currently overloaded and cannot process next request message)

5.3 UDS Packet

UDS packet might also be called Network Protocol Data Unit (N_PDU). The packets are created during *segmentation* of a *diagnostic message*. Each *diagnostic message* consists of at least one UDS Packet (N_PDU). There are some packets which does not carry any diagnostic message data as they are used to manage the flow of other packets.

UDS packet consists of following fields:

- *Network Address Information* (N_AI) - packet addressing
- *Network Data Field* (N_Data) - packet data
- *Network Protocol Control Information* (N_PCI) - packet type

5.3.1 Network Address Information

Network Address Information (N_AI) contains address information which identifies the recipient(s) and the sender between whom data exchange takes place. It also describes communication model (e.g. whether response is required) for the message.

5.3.2 Network Data Field

Network Data Field (N_Data) carries diagnostic message data. It might be an entire diagnostic message data (if a *diagnostic message* fits into one packet) or just a part of it (if *segmentation* had to be used to divide a *diagnostic message* into smaller parts).

5.3.3 Network Protocol Control Information

Network Protocol Control Information (N_PCI) identifies the type of *UDS packet* (Network Protocol Data Unit). N_PCI values and their interpretation are bus specific.

5.3.4 UDS Packet on CAN

In this chapter you will find information about UDS packets that are specific for CAN bus, therefore **applicable only for UDS packets that are transmitted over CAN bus**.

CAN Frame

CAN data frames are the only type of CAN frames that are used during normal UDS communication. CAN data frames are made up of many different fields, but the key in our case (these influenced by UDS protocol) are listed below:

- CAN Identifier (CAN ID)

  CAN ID is a field that informs every receiving CAN node about a sender and a content of frames. CAN nodes shall filter out and ignore CAN frames that are not relevant for them. In a normal situation, a CAN node detects a transmission of incoming CAN frames and once the identifier value (of a CAN frame) is decoded, the CAN node shall stop further listening to the frame if the CAN node is not a recipient of the frame.

  There are two formats of CAN ID:

  - Standard (11-bit Identifier)
  - Extended (29-bit identifier)

- Data Length Code (DLC)

  Data Length Code (DLC) informs about number of CAN frame payload bytes that CAN Data Field contains.

- CAN Data Field

  CAN Data consists of CAN frame payload bytes. The number of bytes that CAN Data Field contains is determined by frame's DLC values as presented in the table:

DLC	Number of CAN Data bytes	Supported by CLASSICAL CAN	Supported by CAN FD
0x0	0	YES	YES
0x1	1	YES	YES
0x2	2	YES	YES
0x3	3	YES	YES
0x4	4	YES	YES
0x5	5	YES	YES
0x6	6	YES	YES
0x7	7	YES	YES
0x8	8	YES	YES
0x9	12	NO	YES
0xA	16	NO	YES
0xB	20	NO	YES
0xC	24	NO	YES
0xD	32	NO	YES
0xE	48	NO	YES
0xF	64	NO	YES

Note: To learn more about CAN bus and CAN frame structure, you are encouraged to visit e-learning portal of Vector Informatik GmbH.

CAN Packet Addressing Formats

Each CAN packet addressing format describes a different way of providing *Network Address Information* to all recipients of CAN packets.

The exchange of UDS Packets on CAN is supported by three addressing formats:

- *Normal addressing*
- *Extended addressing*
- *Mixed addressing*

Warning: Addressing format must be predefined and configured before any CAN packet is received as every CAN packet addressing format determines a different way of decoding CAN packets information (*Network Address Information*, *Network Data Field* and *Network Protocol Control Information*) that is not compatible with other addressing formats.

Note: Regardless of addressing format used, to transmit a *functionally addressed* message over CAN, a sender is allowed to use *Single Frame* packets only.

Normal Addressing

If normal addressing format is used, then the value of CAN Identifier carries an entire *Network Address Information*. Basing on CAN Identifier value, it is possible to distinguish *an addressing type*, a sender and a target/targets entities of a packet.

Following parameters specifies *Network Address Information* when Normal Addressing is used:

- CAN ID

Note: Correspondence between *Network Address Information* and the value of CAN Identifier is left open for a network designer unless *normal fixed addressing* subformat is used.

Note: *Network Protocol Control Information* is placed in the **first byte** of *CAN frame data field* if normal addressing format is used.

Normal Fixed Addressing

Normal fixed addressing format is a special case of *normal addressing* in which the mapping of the address information into the CAN identifier is further defined.

Note: For normal fixed addressing, only 29-bit (extended) CAN Identifiers are allowed.

Following parameters specifies *Network Address Information* when Normal Fixed Addressing is used:

- CAN ID (with embedded **Target Address** and **Source Address**)

CAN Identifier values used for UDS communication using normal fixed addressing:

- For *physical addressed* messages, CAN Identifier value is defined as presented below:

```
CAN_ID = 0x18DATTSS
```

- For *functional addressed* messages, CAN Identifier value is defined as presented below:

```
CAN_ID = 0x18DBTTSS
```

where:

- CAN_ID - value of **CAN Identifier**
- TT - two (hexadecimal) digits of a 8-bit **Target Address** value
- SS - two (hexadecimal) digits of a 8-bit **Source Address** value

Extended Addressing

If extended addressing format is used, then the value of **the first CAN frame byte informs about a target** of a UDS packet and remaining *Network Address Information* (a sending entity and *an addressing type*) are determined by CAN Identifier value.

Following parameters specifies *Network Address Information* when Extended Addressing is used:

- CAN ID
- Target Address (located in the first data byte of a *CAN Frame*)

Note: *Network Protocol Control Information* is placed in the **second byte** of *CAN frame data field* if extended addressing format is used.

Mixed Addressing

Mixed addressing format specifies that **the first byte of a CAN frame is an extension** of *Network Address Information*.

Note: *Network Protocol Control Information* is placed in the **second byte** of *CAN frame data field* if mixed addressing format is used.

Mixed Addressing - 11-bit CAN Identifier

If mixed addressing format is used with 11-bit CAN Identifiers, then the value of **the first CAN frame byte extends** the CAN Identifier and a combination of these data forms the entire *Network Address Information* of a CAN packet.

Following parameters specifies *Network Address Information* when Extended Addressing is used:

- CAN ID
- Addressing Extension (located in the first data byte of a *CAN Frame*)

Mixed Addressing - 29-bit CAN Identifier

If mixed addressing format is used with 29-bit CAN Identifiers, then the value of **the first CAN frame byte extends** the CAN Identifier (that contains **Target Address** and **Sender Address** values) and a combination of these data forms the entire *Network Address Information* of a CAN packet.

Following parameters specifies *Network Address Information* when Extended Addressing is used:

- CAN ID (with embedded **Target Address** and **Source Address**)
- Addressing Extension (located in the first data byte of a *CAN Frame*)

CAN Identifier values used for UDS communication using mixed 29-bit addressing:

- For *physical addressed* messages, CAN Identifier value is defined as presented below:

```
CAN_ID = 0x18CETTSS
```

- For *functional addressed* messages, CAN Identifier value is defined as presented below:

```
CAN_ID = 0x18CDTTSS
```

where:

- CAN_ID - value of **CAN Identifier**
- TT - two (hexadecimal) digits of a 8-bit **Target Address** value
- SS - two (hexadecimal) digits of a 8-bit **Source Address** value

CAN Data Field

CAN frames that are exchanged during UDS communication must have Data Length Code (DLC) equal to 8 (for CLASSICAL CAN and CAN FD) or greater (for CAN FD). The only exception is usage of *CAN Frame Data Optimization*.

DLC	Description
<8	*Valid only for CAN frames using data optimization* Values in this range are only valid for Single Frame, Flow Control and Consecutive Frame that use CAN frame data optimization.
8	*Configured CAN frame maximum payload length of 8 bytes* For the use with CLASSICAL CAN and CAN FD type frames.
>8	*Configured CAN frame maximum payload length greater than 8 bytes* For the use with CAN FD type frames only.

where:

- DLC - Data Length Code of a *CAN frame*

Note: Number of bytes that carry diagnostic message payload depends on a type and a format of a CAN packet as it is presented in *the table with CAN packets formats*.

CAN Frame Data Padding

If a number of bytes specified in a UDS Packet is shorter than a number of bytes in CAN frame's data field, then the sender has to pad any unused bytes in the frame. This can only be a case for *Single Frame*, *Flow Control* and the last *Consecutive Frame* of a segmented message. If not specified differently, the default value 0xCC shall be used for the frame padding to minimize the bit stuffing insertions and bit alteration on the wire.

Note: CAN frame data padding is mandatory for *CAN frames* with DLC>8 and optional for frames with DLC=8.

CAN Frame Data Optimization

CAN frame data optimization is an alternative to *CAN Frame Data Padding*. If a number of bytes specified in a CAN Packet is shorter than a number of bytes in CAN frame's data field, then the sender might decrease DLC value of the *CAN frame* to the minimal number that is required to sent a desired number of data bytes in a single CAN packet.

Note: CAN Frame Data Optimization might always be used for CAN Packets with less than 8 bytes of data to send.

Warning: CAN Frame Data Optimization might not always be able to replace *CAN Frame Data Padding* when CAN FD is used. This is a consequence of DLC values from 9 to 15 meaning as these values are mapped into CAN frame data bytes numbers in a non-linear way (e.g. DLC=9 represents 12 data bytes).

Example:

When a CAN Packet with 47 bytes of data is planned for a transmission, then DLC=14 can be used instead of DLC=15, to choose 48-byte instead of 64-byte long CAN frame. Unfortunately, the last byte of CAN Frame data has to be padded as there is no way to send over CAN a frame with exactly 47 bytes of data.

CAN Packet Types

According to ISO 15765-2, CAN bus supports 4 types of UDS packets.

List of all values of *Network Protocol Control Information* supported by CAN bus:

- 0x0 - *Single Frame*
- 0x1 - *First Frame*
- 0x2 - *Consecutive Frame*
- 0x3 - *Flow Control*
- 0x4-0xF - values range reserved for future extension by ISO 15765

The format of all CAN packets is presented in the table below.

CAN N_PDU	Byte #1 Bits 7-4	Bits 3-0	Byte #2	Byte #3	Byte #4	Byte #5	Byte #6	...
Single Frame *DLC 8*	0x0	SF_DL						
Single Frame *DLC > 8*	0x0	0x0	SF_DL					
First Frame *FF_DL 4095*	0x1	FF_DL						
First Frame *FF_DL > 4095*	0x1	0x0	0x00	FF_DL				
Consecutive Frame	0x2	SN						
Flow Control	0x3	FS	BS	ST_min	N/A	N/A	N/A	N/A

where:

- DLC - Data Length Code of a CAN frame, it is equal to number of data bytes carried by this CAN frame
- SF_DL - *Single Frame Data Length*

- FF_DL - *First Frame Data Length*
- SN - *Sequence Number*
- FS - *Flow Status*
- BS - *Block Size*
- ST_min - *Separation Time minimum*
- N/A - Not Applicable (byte does not carry any information)

Single Frame

Single Frame (SF) is used by CAN entities to transmit a diagnostic message with a payload short enough to fit it into a single CAN packet. In other words, Single Frame carries payload of an entire diagnostic message. Number of payload bytes carried by SF is specified by *Single Frame Data Length* value.

Single Frame Data Length

Single Frame Data Length (SF_DL) is 4-bit (for CAN packets with DLC<=8) or 8-bit (for CAN packets with DLC>8) value carried by every Single Frame as presented in *the table with CAN packet formats*. SF_DL specifies number of diagnostic message payload bytes transmitted in a Single Frame.

Note: Maximal value of SF_DL depends on Single Frame *addressing format* and *DLC of a CAN message* that carries this packet.

First Frame

First Frame (FF) is used by CAN entities to indicate start of a diagnostic message transmission. First Frames are only used during a transmission of a segmented diagnostic messages that could not fit into a *Single Frame*. Number of payload bytes carried by FF is specified by *First Frame Data Length* value.

First Frame Data Length

First Frame Data Length (FF_DL) is 12-bit (if FF_DL 4095) or 4-byte (if FF_DL > 4095) value carried by every First Frame. FF_DL specifies number of diagnostic message payload bytes of a diagnostic message which transmission was initiated by a First Frame.

Note: Maximal value of FF_DL is 4294967295 (0xFFFFFFFF). It means that CAN bus is capable of transmitting diagnostic messages that contains up to nearly 4,3 GB of payload bytes.

Consecutive Frame

Consecutive Frame (CF) is used by CAN entities to continue transmission of a diagnostic message. *First Frame* shall always precede (one or more) Consecutive Frames. Consecutive Frames carry payload bytes of a diagnostic message that was not transmitted in a *First Frame* that preceded them. To avoid ambiguity and to make sure that no Consecutive Frame is lost, the order of Consecutive Frames is determined by *Sequence Number* value.

Sequence Number

Sequence Number (SN) is 4-bit value used to specify the order of Consecutive Frames.

The rules of proper Sequence Number value assignment are following:

- SN value of the first *Consecutive Frame* that directly follows a *First Frame* shall be set to 1
- SN shall be incremented by 1 for each following *Consecutive Frame*
- SN value shall not be affected by *Flow Control* frames
- when SN reaches the value of 15, it shall wraparound and be set to 0 in the next *Consecutive Frame*

Flow Control

Flow Control (FC) is used by receiving CAN entities to instruct sending entities to stop, start, pause or resume transmission of *Consecutive Frames*.

Flow Control packet contains following parameters:

- *Flow Status*
- *Block Size*
- *Separation Time Minimum*

Flow Status

Flow Status (FS) is 4-bit value that is used to inform a sending network entity whether it can proceed with a Consecutive Frames transmission.

Values of Flow Status:

- 0x0 - ContinueToSend (CTS)

  ContinueToSend value of Flow Status informs a sender of a diagnostic message that receiving entity (that responded with CTS) is ready to receive a maximum of *Block Size* number of *Consecutive Frames*.

  Reception of a *Flow Control* frame with ContinueToSend value shall cause the sender to resume ConsecutiveFrames sending.

- 0x1 - wait (WAIT)

  Wait value of Flow Status informs a sender of a diagnostic message that receiving entity (that responded with WAIT) is not ready to receive another *Consecutive Frames*.

  Reception of a *Flow Control* frame with WAIT value shall cause the sender to pause ConsecutiveFrames sending and wait for another *Flow Control* frame.

Values of *Block Size* and *STmin* in the *Flow Control* frame (that contains WAIT value of Flow Status) are not relevant and shall be ignored.

- 0x2 - Overflow (OVFLW)

  Overflow value of Flow Status informs a sender of a diagnostic message that receiving entity (that responded with OVFLW) is not able to receive a full diagnostic message as it is too big and reception of the message would result in Buffer Overflow on receiving side. In other words, the value of *FF_DL* exceeds the buffer size of the receiving entity.

  Reception of a *Flow Control* frame with Overflow value shall cause the sender to abort the transmission of a diagnostic message.

  Overflow value shall only be sent in a *Flow Control* frame that directly follows a *First Frame*.

  Values of *Block Size* and *STmin* in the *Flow Control* frame (that contains OVFLW value of Flow Status) are not relevant and shall be ignored.

- 0x3-0xF - Reserved

  This range of values is reserved for future extension by ISO 15765.

Block Size

Block Size (BS) is a one byte value specified by receiving entity that informs about number of *Consecutive Frames* to be sent in a one block of packets.

Block Size values:

- 0x00

  The value 0 of the Block Size parameter informs a sender that no more *Flow Control* frames shall be sent during the transmission of the segmented message.

  Reception of Block Size = 0 shall cause the sender to send all remaining *Consecutive Frames* without any stop for further *Flow Control* frames from the receiving entity.

- 0x01-0xFF

  This range of Block Size values informs a sender the maximum number of *Consecutive Frames* that can be transmitted without an intermediate *Flow Control* frames from the receiving entity.

Separation Time Minimum

Separation Time minimum (STmin) is a one byte value specified by receiving entity that informs about minimum time gap between the transmission of two following *Consecutive Frames*.

STmin values:

- 0x00-0x7F - Separation Time minimum range 0-127 ms

  The value of STmin in this range represents the value in milliseconds (ms).

  0x00 = 0 ms

  0xFF = 127 ms

- 0x80-0xF0 - Reserved

  This range of values is reserved for future extension by ISO 15765.

- 0xF1-0xF9 - Separation Time minimum range 100-900 s

The value of STmin in this range represents the value in microseconds (s) according to the formula:

```
(STmin - 0xF0) * 100 s
```

Meaning of example values:

0xF1 -> 100 s

0xF5 -> 500 s

0xF9 -> 900 s

- 0xFA-0xFF - Reserved

  This range of values is reserved for future extension by ISO 15765.

5.4 Segmentation

5.4.1 Message Segmentation

To transmit a diagnostic message, its information (payload and addressing) must be unambiguously encoded into one or more segments (these segments are called *UDS Packets* by this documentation) that are specific for bus used.

Note: Segmentation process is specific for each bus due to various topologies supported by each bus, various communication models (e.g. Master/Slave) enforced by them, etc.

Segmentation on CAN

Unsegmented message transmission

When mentioning unsegmented message transmission, we mean a case when an entire *Diagnostic Message* can be fully transmitted by a single packet. *Single Frame (CAN Packet)* is the only type of CAN Packets that can be used in the described scenario.

Segmented message transmission

When a *Diagnostic Message* to be transmitted on CAN contains payload which size is greater than a *Single Frame* capacity, then the message payload must be divided and transmitted by many CAN packets. The first packet to carry such messages is *First Frame (CAN Packet)* and its transmission is followed by *Consecutive Frames (CAN Packets)*. A receiver controls the stream of incoming *Consecutive Frames* by sending *Flow Control (CAN Packet)* after *First Frame* and each complete transmission of *Consecutive Frames* block.

Note: The size of *Consecutive Frames* block is determined by *Block Size* parameter which value is carried by *Flow Control*.

Note: The minimum time between two *Consecutive Frames* is determined by *Separation Time Minimum* parameter which value is carried by *Flow Control*.

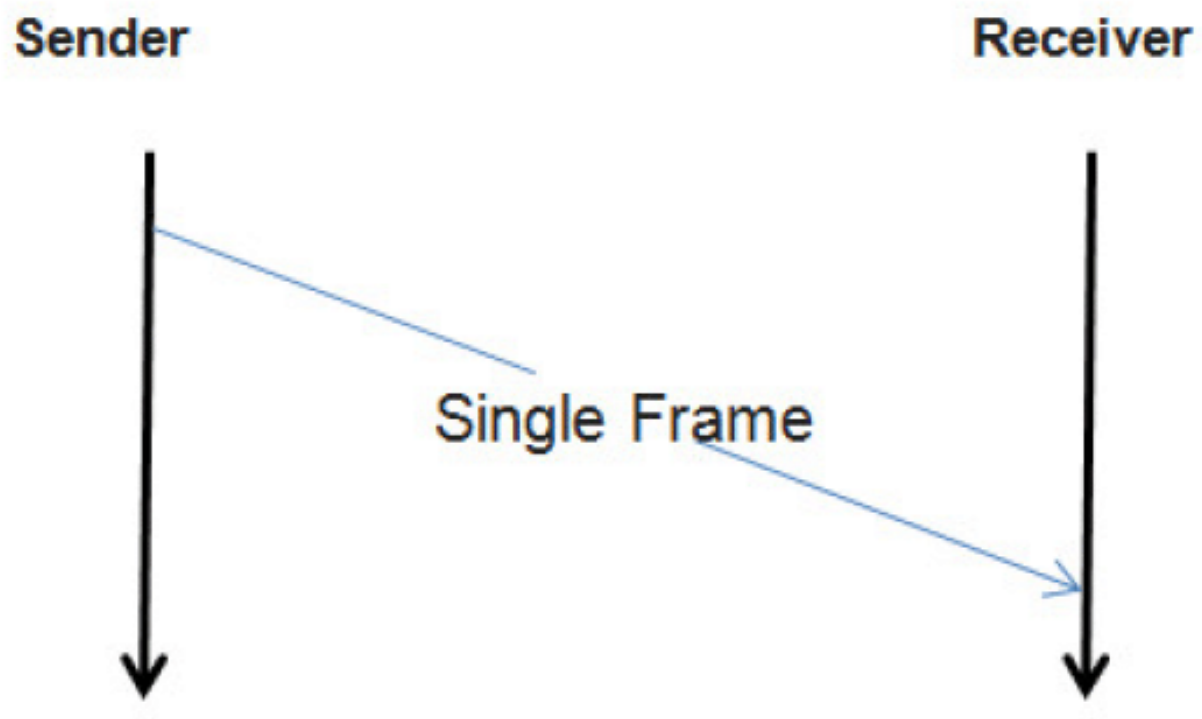


Fig. 4: Transmission of an unsegmented Diagnostic Message on CAN bus.
A sender transmits a *Single Frame (CAN Packet)* that contains an entire *Diagnostic Message*.

See also:

Only the typical use case of *Flow Control* was described here. Check *Flow Status* parameter and meaning of its values to study less likely scenarios.

5.4.2 Packets Desegmentation

Desegmentation is an unambiguous operation which is the reverse process to a *message segmentation*. It transforms one or more *UDS packets* into a *diagnostic message*.

Note: There are many ways to segment a diagnostic message into CAN packets, but there is always only one correct way to perform desegmentation and decode a diagnostic message out of CAN Packets.

5.5 Performance and Error Handling

In this chapter of the documentation, we would explain performance and timing requirements for UDS communication and how they are supposed to be handled by UDS entities.

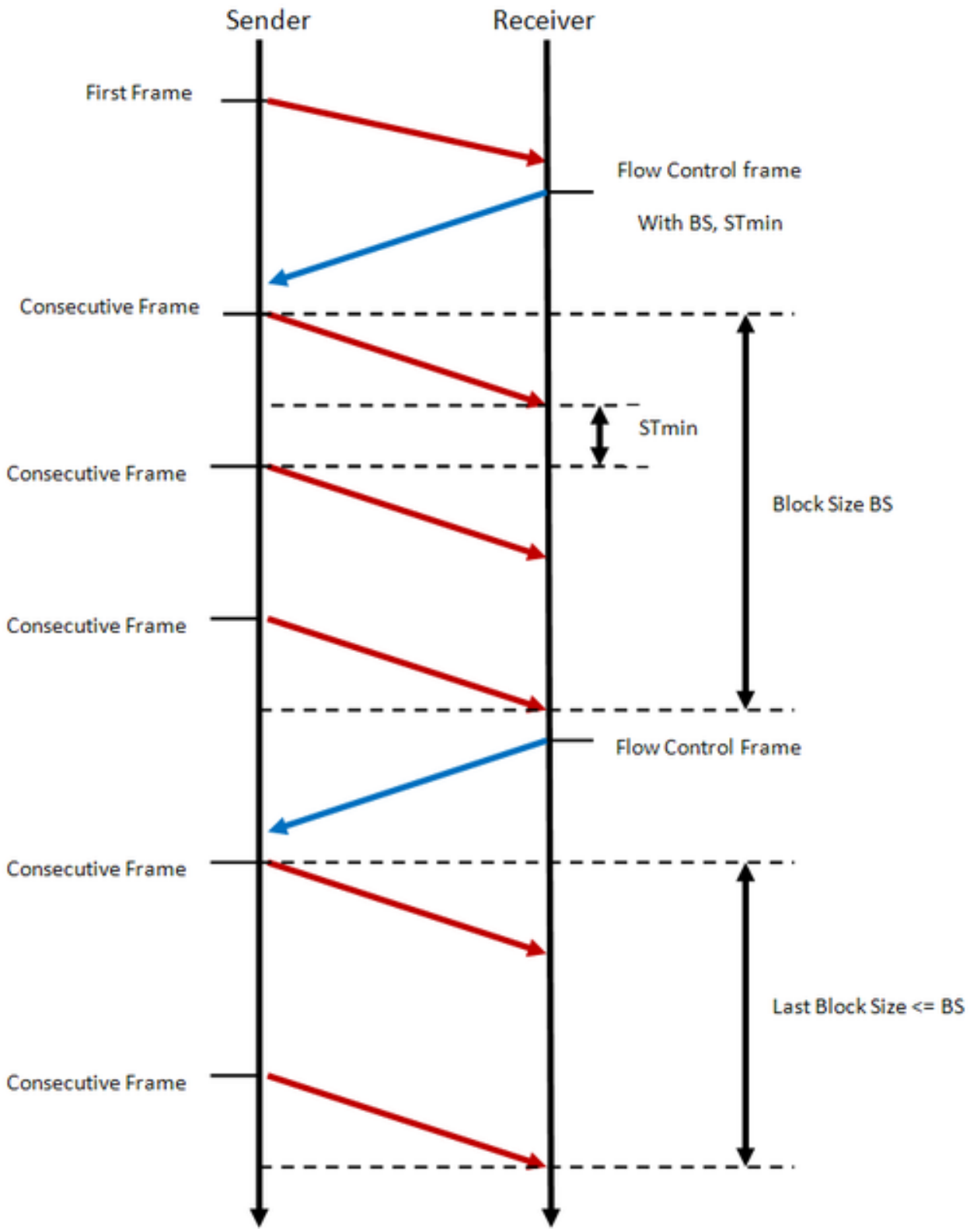


Fig. 5: Transmission of a segmented Diagnostic Message on CAN bus.
A sender initiates *Diagnostic Message* transmission with a *First Frame (CAN Packet)* Then, a receiver controls the stream of incoming *Consecutive Frames (CAN Packets)* by transmitting *Flow Controls (CAN Packets)*.

5.5.1 CAN specific

ISO standards defines following time values on the network layer of UDS on CAN communication:

- *N_As*
- *N_Ar*
- *N_Bs*
- *N_Br*
- *N_Cs*
- *N_Cr*

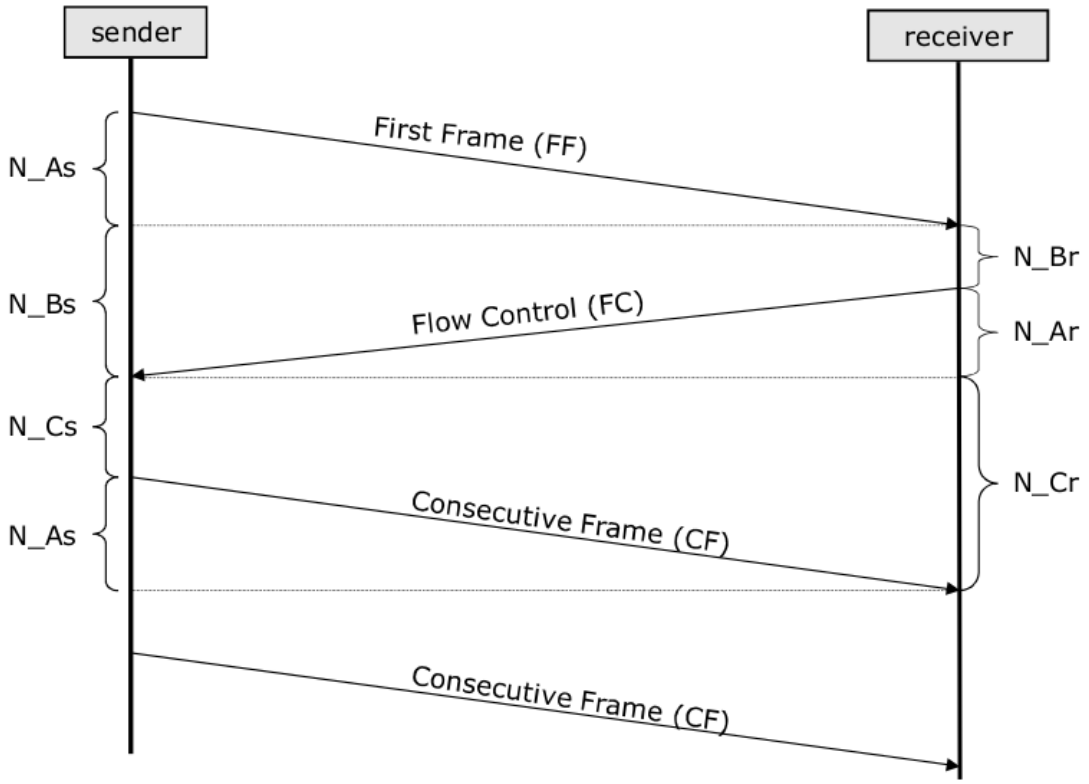


Fig. 6: Network layer time values (N_As, N_Ar, N_Bs, N_Br, N_Cs, N_Cr) present during UDS on CAN communication.

Note: The example uses *segmented diagnostic message transmission* as all CAN timings values can be presented there (all these times are applicable in this case). For *unsegmented diagnostic message transmission* though, the only applicable time parameter is *N_As*.

N_As

N_As is a time parameter related to transmission of any *CAN Packet* by a sender. It is measured from the beginning of the *CAN Frame* (that carries such CAN Packet) transmission till the reception of a confirmation that this CAN Frame was received by a receiver.

Timeout value:
1000 ms

Error handling:
If N_As timeout is exceeded, then the transmission of the *diagnostic message* shall be aborted.

Affected *CAN Packets*:

- *Single Frame*
- *First Frame*
- *Consecutive Frame*

N_Ar

N_Ar is a time parameter related to transmission of any *CAN Packet* by a receiver. It is measured from the beginning of the *CAN Frame* (that carries such CAN Packet) transmission till the reception of a confirmation that this CAN Frame was received by a sender.

Timeout value:
1000 ms

Error handling:
If N_Ar timeout is exceeded, then the reception of the *diagnostic message* shall be aborted.

Affected *CAN Packets*:

- *Flow Control*

N_Bs

N_Bs is a time parameter related to *Flow Control (CAN Packet)* reception by a sender. It is measured from the end of the last CAN Packet transmission (either transmitted *First Frame*, *Consecutive Frame* or received *Flow Control*), till the reception of *Flow Control*.

Timeout value:
1000 ms

Error handling:
If N_Bs timeout is exceeded, then the reception of the *diagnostic message* shall be aborted.

Affected *CAN Packets*:

- *Flow Control*

N_Br

N_Br is a time parameter related to *Flow Control (CAN Packet)* transmission by a receiver. It is measured from the end of the last CAN Packet transmission (either received *First Frame*, *Consecutive Frame* or transmitted *Flow Control*), till the start of *Flow Control* transmission.

Performance requirement:
A receiving entity is obliged to transmit *Flow Control* packet before value of N_Br achieves maximal value threshold.

```
[N_Br] + [N_Ar] < 0.9 * [N_Bs timeout]
[N_Br max] = 900ms - [N_Ar]
```

Affected *CAN Packets*:

- *Flow Control*

N_Cs

N_Cs is a time parameter related to *Consecutive Frame (CAN Packet)* transmission by a sender. It is measured from the end of the last CAN Packet transmission (either received *Flow Control* or transmitted *Consecutive Frame*), till the start of *Consecutive Frame* transmission.

Performance requirement:
A sending entity is obliged to transmit *Consecutive Frame* packet before value of N_Cs achieves maximal value threshold.

```
[N_Cs] + [N_As] < 0.9 * [N_Cr timeout]
[N_Cs max] = 900ms - [N_As]
```

Affected *CAN Packets*:

- *Consecutive Frame*

N_Cr

N_Cr is a time parameter related to *Consecutive Frame (CAN Packet)* reception by a receiver. It is measured from the end of the last CAN Packet transmission (either transmitted *Flow Control* or received *Consecutive Frame*), till the reception of *Consecutive Frame*.

Timeout value:
1000 ms

Error handling:
If N_Cr timeout is exceeded, then the reception of the *diagnostic message* shall be aborted.

Affected *CAN Packets*:

- *Consecutive Frame*

CHAPTER

SIX

CONTRIBUTION

6.1 How to contribute?

If you want to become a contributor, please visit UDS project page and read CONTRIBUTING.md file.

Contact us to find out what we have got to offer and to get more information.

6.2 Sponsoring

If you consider sponsoring our development team, please visit our wiki page with detailed information for sponsors. With a little help from you, we would be able to improve the quality of our code, speed up the development and provide more features. We also offer special treatment for all our sponsors. Please contact us for more details.

6.3 Reporting issues

To report issues, please use our issues tracking system.

6.4 Our Sponsors

Full list of our sponsors:

- Merit Automotive 

 - sponsoring since September 2021

CHAPTER

SEVEN

OVERVIEW

The purpose of this project is to provide python tools for simulation (on both sides - client and server) and monitoring of diagnostic communication defined by ISO-14229. It can be used with any bus type (e.g. CAN, Ethernet, LIN).

The most likely use cases of this package are:

- communication with your vehicle (e.g. reading Diagnostic Trouble Codes)
- monitoring and decoding ongoing UDS communication
- performing tests against on-board ECU (server)
- performing tests against OBD Tester (client)

CHAPTER

EIGHT

IMPLEMENTATION STATUS

The package is currently in the early development phase, therefore only a few features are currently available. If you want to speed up the development, please visit *contribution section* to find out what are your options.

8.1 Features

Current implementation status of package features:

Feature	Implementation Status
UDS Messages and Packets	Available since version 0.0.2
UDS Packets Reception and Transmission	Available since version 0.3.0
UDS Messages Reception and Transmission	Planned
Messages Segmentation	Available since version 0.2.0
UDS Packets Desegmentation	Available since version 0.2.0
Support for Services with multiple responses	Planned
Client Simulation	Planned
Server Simulation	Planned
Support for Messages Databases	Planned

8.2 Buses supported

Current implementation status of support for communication buses:

Bus	Implementation Status
CAN	Partial
FlexRay	Planned
Ethernet	Planned
K-Line	Planned
LIN	Planned

CHAPTER

NINE

LICENSE

The project is licensed under the MIT license - https://github.com/mdabrowski1990/uds/blob/main/LICENSE

CHAPTER

TEN

CONTACT

- e-mail: uds-package-development@googlegroups.com
- group: UDS package development
- discord: https://discord.gg/y3waVmR5PZ

Documentation generated

Mar 12, 2024

PYTHON MODULE INDEX

u

INDEX

Symbols

A

B

C

D

E

F

G

I

N

O

P

R

S

T

U

V

W