# Contents

1 Implementation 1
   1.1 Transport layer ................................................................. 1
   1.2 Parameters ............................................................................. 2
   1.3 Usage ..................................................................................... 4
   1.4 Errors ..................................................................................... 4

2 IsoTp Sockets 7
   2.1 Introduction ............................................................................ 7
   2.2 Troubleshooting ....................................................................... 7
   2.3 Examples ................................................................................ 8
   2.4 Usage ..................................................................................... 8

3 Addressing 11
   3.1 Definitions ............................................................................ 11
   3.2 Addressing modes ................................................................... 12

4 Examples 15
   4.1 Basic transmission with python-can ......................................... 15
   4.2 Threaded reception with python-can ......................................... 16
   4.3 Different type of addresses ...................................................... 17
   4.4 Sending with functional addressing (broadcast) ....................... 17
   4.5 Defining custom rxfn and txfn ................................................. 17
   4.6 Defining partial rxfn and txfn .................................................. 18

Index 21
This sections explains the python implementation of the IsoTP protocol.

## 1.1 Transport layer

Depending on your constraints, you may want to have the IsoTP protocol layer to run in Python (in the user space). For example, if you want to rely on python-can for the support of your CAN interface, you will need to run the IsoTP layer in Python.

In such case, the `isotp.TransportLayer` will be the right tool. One must first define functions to access the hardware and provide them to the `isotp.TransportLayer` as parameters named `rxfn` and `txfn`.

```python
class isotp.TransportLayer(rxfn, txfn, address=None, error_handler=None, params=None)
```

The IsoTP transport layer implementation

**Parameters**

- `rxfn (Callable)` – Function to be called by the transport layer to read the CAN layer. Must return a `isotp.CanMessage` or None if no message has been received.

- `txfn (Callable)` – Function to be called by the transport layer to send a message on the CAN layer. This function should receive a `isotp.CanMessage`

- `address (isotp.Address)` – The address information of CAN messages. Includes the addressing mode, txid/rxid, source/target address and address extension. See `isotp.Address` for more details.

- `error_handler (Callable)` – A function to be called when an error has been detected. An `isotp.IsoTpError` (inheriting Exception class) will be given as sole parameter. See the `Error section`

- `params (dict)` – List of parameters for the transport layer

If python-can must be used as CAN layer, one can use the `isotp.CanStack` which extends the TransportLayer object with predefined functions that calls python-can.
class `isotp.CanStack` (*bus*, *args*, **kwargs)

The IsoTP transport using python-can as CAN layer. python-can must be installed in order to use this class. All parameters except the `bus` parameter will be given to the `TransportLayer` constructor.

**Parameters**

- `bus` (*BusABC*) – A python-can bus object implementing `recv` and `send`
- `address` (*isotp.Address*) – The address information of CAN messages. Includes the addressing mode, txid/rxid, source/target address and address extension. See `isotp.Address` for more details.
- `error_handler` (*Callable*) – A function to be called when an error has been detected. An `isotp.protocol.IsoTpError` (inheriting `Exception` class) will be given as sole parameter.
- `params` (*dict*) – List of parameters for the transport layer

The CAN messages going in and out from the transport layer are defined with `isotp.CanMessage`.

class `isotp.CanMessage` (*arbitration_id=None*, *dlc=None*, *data=None*, *extended_id=False*, *is_fd=False*)

Represent a CAN message (ISO-11898)

**Parameters**

- `arbitration_id` (*int*) – The CAN arbitration ID. Must be a 11 bits value or a 29 bits value if `extended_id` is True
- `dlc` (*int*) – The Data Length Code representing the number of bytes in the data field
- `data` (*bytearray*) – The 8 bytes payload of the message
- `extended_id` (*bool*) – When True, the arbitration ID stands on 29 bits. 11 bits when False
- `is_fd` – When True, message has to be transmitted or has been received in a CAN FD frame. CAN frame when set to False

## 1.2 Parameters

The transport layer `params` parameter must be a dictionary with the following keys.

* `stmin` (*int*)
  
  default: 0
  
  The single-byte Separation Time to include in the flow control message that the layer will send when receiving data. Refer to ISO-15765-2 for specific values. From 1 to 127, represents milliseconds. From 0xF1 to 0xF9, represents hundreds of microseconds (100us, 200us, . . . , 900us). 0 Means no timing requirements

* `blocksize` (*int*)
  
  default: 8
  
  The single-byte Block Size to include in the flow control message that the layer will send when receiving data. Represents to number of consecutive frame that a sender should send before expecting the layer to send a flow control message. 0 Means infinitely large block size (implying no flow control message)

* `tx_data_length` (*int*)
  
  default: 8
The maximum number of bytes that the Link Layer (CAN layer) can transport. In other words, the biggest number of data bytes possible in a single CAN message. Valid values are: 8, 12, 16, 20, 24, 32, 48, 64.

Large frames will be transmitted in small CAN messages of this size except for the last CAN message that will be as small as possible, unless padding is used.

This parameter was formerly named `ll_data_length` but has been renamed to explicitly indicate that it affects transmitted messages only.

**squash_stmin_requirement (bool)**

**default:** False

Indicates if the layer should override the receiver separation time (stmin) when sending and try sending as fast as possible instead. This can be useful when the layer is running on an operating system giving low priority to your application; such as Windows that has a thread resolution of 16ms.

**rx_flowcontrol_timeout**

**default:** 1000

The number of milliseconds to wait for a flow control frame before stopping reception and triggering a `FlowControlTimeoutError`. Defined as $N_{BS}$ by ISO-15765-2.

**rx_consecutive_frame_timeout (int)**

**default:** 1000

The number of milliseconds to wait for a consecutive frame before stopping reception and triggering a `ConsecutiveFrameTimeoutError`. Defined as $N_{CS}$ by ISO-15765-2.

**tx_padding (int or None)**

**default:** None

When not `None` represents the byte used for padding messages sent. No padding applied when `None`.

**wftmax (int)**

**default:** 0

The single-byte Wait Frame Max to include in the flow control message that the layer will send when receiving data. When this limits is reached, reception will stop and trigger a `MaximumWaitFrameReachedError`.

A value of 0 that wait frames are not supported and none shall be sent.

**max_frame_size (int)**

**default:** 4095

The maximum frame length that the stack will accept to receive. ISO-15765-2:2016 allows frames as long as $2^{32}-1$ (4294967295 bytes). When a FirstFrame is sent with a length longer than `max_frame_size`, the message will be ignored, a FlowControl message with FlowStatus=2 (Overflow) will be sent and a `FrameTooLongError` will be triggered.

This parameter mainly is a protection to avoid crashes due to lack of memory (caused by an external device).

**can_fd (bool)**

**default:** False

When set to `True`, transmitted messages will be CAN FD. CAN 2.0 when `False`.

Setting this parameter to `True` does not change the behaviour of the `TransportLayer` except that outputted message will have their `is_fd` property set to `True`. This parameter is just a convenience to integrate more easily with python-can.
1.3 Usage

The `isotp.TransportLayer` object has the following methods:

* `TransportLayer.send(data, target_address_type=0)`
  Enqueue an IsoTP frame to be sent over CAN network

  **Parameters**
  
  - `data (bytearray)` – The data to be sent
  - `target_address_type (int)` – Optional parameter that can be Physical (0) for 1-to-1 communication or Functional (1) for 1-to-n. See `isotp.TargetAddressType`

  **Raises**
  
  - `ValueError` – Input parameter is not a bytearray or not convertible to bytearray
  - `RuntimeError` – Transmit queue is full

* `TransportLayer.recv()`
  Dequeue an IsoTP frame from the reception queue if available.

  **Returns** The next available IsoTP frame

  **Return type** bytearray or None

* `TransportLayer.available()`
  Returns True if an IsoTP frame is awaiting in the reception queue. False otherwise

* `TransportLayer.transmitting()`
  Returns True if an IsoTP frame is being transmitted. False otherwise

* `TransportLayer.set_address(address)`
  Sets the layer `Address`. Can be set after initialization if needed.

* `TransportLayer.reset()`
  Reset the layer: Empty all buffers, set the internal state machines to Idle

* `TransportLayer.process()`
  Function to be called periodically, as fast as possible. This function is non-blocking.

* `TransportLayer.sleep_time()`
  Returns a value in seconds that can be passed to `time.sleep()` when the stack is processed in a different thread.

  The value will change according to the internal state machine state, sleeping longer while idle and shorter when active.

1.4 Errors

When calling `TransportLayer.process`, no exception should raise. Still, errors are possible and are given to an error handler provided by the user. An error handler should be a callable function that expects an Exception as first parameter.

* `my_error_handler(error)`

  **Parameters**
  
  - `error (isotp.IsoTpError)` – The error

All errors inherit `isotp.IsoTpError` which itself inherits `Exception`
class isotp.FlowControlTimeoutError(*args, **kwargs)
    Happens when the sender fails to send a Flow Control message in time. Refer to TransportLayer parameter 
    `rx_flowcontrol_timeout`

class isotp.ConsecutiveFrameTimeoutError(*args, **kwargs)
    Happens when the sender fails to send a Consecutive Frame message in time. Refer to TransportLayer parameter 
    `rx_consecutive_frame_timeout`

class isotp.InvalidCanDataError(*args, **kwargs)
    Happens when a CAN message that cannot be decoded as valid First Frame, Consecutive Frame, Single Frame 
    or Flow Control PDU is received.

class isotp.UnexpectedFlowControlError(*args, **kwargs)
    Happens when a Flow Control message is received and was not expected

class isotp.UnexpectedConsecutiveFrameError(*args, **kwargs)
    Happens when a Consecutive Frame message is received and was not expected

class isotp.ReceptionInterruptedWithSingleFrameError(*args, **kwargs)
    Happens when the reception of a multi packet message reception is interrupted with a new Single Frame PDU.

class isotp.ReceptionInterruptedWithFirstFrameError(*args, **kwargs)
    Happens when the reception of a multi packet message reception is interrupted with a new First Frame PDU.

class isotp.WrongSequenceNumberError(*args, **kwargs)
    Happens when a consecutive frame is received with a wrong sequence number.

class isotp.UnsuportedWaitFrameError(*args, **kwargs)
    Happens when a Flow Control PDU with FlowStatus=Wait is received and `wftmax` is set to 0

class isotp.MaximumWaitFrameReachedError(*args, **kwargs)
    Happens when too much Flow Control PDU with FlowStatus=Wait is received. Refer to `wftmax`

class isotp.FrameTooLongError(*args, **kwargs)
    Happens when a FirstFrame with a length (FF_DL) longer than `max_frame_size` is received.

class isotp.ChangingInvalidRXDLError(*args, **kwargs)
    Happens when a ConsecutiveFrame is received with a length smaller than RX_DL (size of first frame) without 
    being the last message of the IsoTP frame.

class isotp.MissingEscapeSequenceError(*args, **kwargs)
    Happens when a SingleFrame with length (CAN_DL) greater than 8 bytes is received and the length of the 
    payload (SF_DL) is encoded in the first byte, which is forbidden by ISO-15765-2

class isotp.InvalidCanFdFirstFrameRXDL(*args, **kwargs)
    Happens when a FirstFrame is received with missing data; In other words when CAN_DL is smaller than the 
    deduced RX_DL. The sender did not optimized the capacity usage of the CAN message.
2.1 Introduction

Under Linux, CAN interfaces can be managed the same way as network interfaces. The support for each CAN driver is written directly in the Linux kernel, inside a module called SocketCAN. Therefore, it is possible to send data over CAN through sockets, which offer a unique interface for all CAN drivers.

SocketCAN allows a user to send data on the CAN layer only, which means that IsoTP protocol implementation has to be outside of the kernel, making timing requirements hard to meet. Fortunately, this third party loadable kernel module addresses that issue by making an implementation that can be loaded within the Linux kernel and accessed through a socket as well.

A socket is a standard interface for communication protocols and as anything generic, it can be complex to configure. The isotp.socket is a wrapper over a native IsoTP socket providing a friendly and pythonic interface for easy configuration. It does not offer any additional functionality that your operating system can’t provide, it makes the syntax clean and warns you in case of wrong usage.

2.2 Troubleshooting

- My socket module does not include the ‘CAN_ISOTP‘ constant
  That means that your Python version does not include support for IsoTP protocol. It should be included starting from Python 3.7, under Linux build only. See Python issue and Pull request

- When I create the socket, I get ‘OSError [errno XX] : Protocol not supported‘.
  The Loadable Kernel Module is not loaded in your Linux kernel. Follow the steps given in the module repository. You need to compile the module, install the .ko file and then run insmod can-isotp.ko as Super User. Then your OS will accept to create an ISO-TP sockets.
2.3 Examples

2.3.1 Without this project

```python
SOL_CAN_ISOTP = 106  # These constants exist in the module header, not in Python.
CAN_ISOTP_RECV_FC = 2
# Many more exists.

import socket
import struct

s = socket.socket(socket.AF_CAN, socket.SOCK_DGRAM, socket.CAN_ISOTP)
s2 = socket.socket(socket.AF_CAN, socket.SOCK_DGRAM, socket.CAN_ISOTP)
# Configuring the sockets with ugly struct.pack() that requires knowledge of the LKM structure
s.setsockopt(SOL_CAN_ISOTP, CAN_ISOTP_RECV_FC, struct.pack("=BBB", 0x10, 3,0))  # bs, stmin, wftmax
#s.setsockopt(SOL_CAN_ISOTP, CAN_ISOTP_OPTS, struct.pack(...))
#s.setsockopt(SOL_CAN_ISOTP, CAN_ISOTP_LL_OPTS, struct.pack(...))
s.bind(("vcan0", 0x123, 0x456))  # rxid, txid with confusing order.
s2.bind(("vcan0", 0x456, 0x123))  # rxid, txid
s2.send(b"Hello, this is a long payload sent in small chunks of 8 bytes.")
print(s.recv(4095))
```

2.3.2 With this project

```python
import isotp

s = isotp.socket()
s2 = isotp.socket()
# Configuring the sockets.
s.set_fc_opts(stmin=5, bs=10)
#s.set_general_opts(...)
#s.set_ll_opts(...)
s.bind("vcan0", isotp.Address(rxid=0x123 txid=0x456))
s2.bind("vcan0", isotp.Address(rxid=0x456, txid=0x123))
s2.send(b"Hello, this is a long payload sent in small chunks of 8 bytes.")
print(s.recv())
```

2.4 Usage

```python
class isotp.socket (timeout=0.1)
    A IsoTP socket wrapper for easy configuration

    Parameters timeout (int) – The underlying socket timeout set with settimeout. Makes the reception thread sleep

socket.bind (interface, address)
    Binds the socket to an address. If no address is provided, all additional parameters will be used to create an address. This is mainly to allow a syntax such as sock.bind('vcan0', rxid=0x123, txid=0x456) for backward compatibility.
```
Parameters

- **interface**(string) – The network interface to use
- **address** – The address to bind to.

Type: *isotp.Address*

To configure a socket, few methods are available:

```python
socket.set_opts(optflag=None, frame_txtime=None, ext_address=None, txpad=None, rxpad=None, rx_ext_address=None)
```

Sets the general options of the socket

Parameters

- **optflag**(int) – A list of flags modifying the protocol behaviour. Refer to `socket.flags`
- **frame_txtime**(int) – Sets the transmit separation time that will override the receiver requirement. If not None, `flags.FORCE_TXSTMIN` will be set.
- **ext_address**(int) – The extended address to use. If not None, `flags.EXTEND_ADDR` will be set.
- **txpad**(int) – The byte to use to pad the transmitted CAN messages. If not None, `flags.TX_PADDING` will be set.
- **rxpad**(int) – The byte to use to pad the transmitted CAN messages. If not None, `flags.RX_PADDING` will be set.
- **rx_ext_address**(int) – The extended address to use in reception. If not None, `flags.RX_EXT_ADDR` will be set.

```python
socket.set_fc_opts(bs=None, stmin=None, wftmax=None)
```

Sets the flow control options of the socket

Parameters

- **bs**(int) – The block size sent in the flow control message. Indicates the number of consecutive frame a sender can send before the socket sends a new flow control. A block size of 0 means that no additional flow control message will be sent (block size of infinity)
- **stmin**(int) – The minimum separation time sent in the flow control message. Indicates the amount of time to wait between 2 consecutive frame. This value will be sent as is over CAN. Values from 1 to 127 means milliseconds. Values from 0xF1 to 0xF9 means 100us to 900us. 0 Means no timing requirements
- **wftmax**(int) – Maximum number of wait frame (flow control message with flow status=1) allowed before dropping a message. 0 means that wait frame are not allowed

```python
socket.set_ll_opts(mtu=None, tx_dl=None, tx_flags=None)
```

Sets the link layer options. Default values are set to work with CAN 2.0. Link layer may be configure to work in CAN FD.

Parameters

- **mtu**(int) – The internal CAN frame structure size. Possible values are defined in `isotp.socket.LinkLayerProtocol`
- **tx_dl**(int) – The CAN message payload length. For CAN 2.0, this value should be 8. For CAN FD, possible values are 8,12,16,20,24,32,48,64
- **tx_flags**(int) – Link layer flags.

2.4. Usage
ISO-15765 defines several addressing modes and this module supports them all. Depending on the addressing mode, the source/target addresses will be defined in different way. An IsoTP address is represented by the `isotp.Address` object.

### 3.1 Definitions

```python
class isotp.Address(addressing_mode=0, txid=None, rxid=None, target_address=None, source_address=None, address_extension=None):
```

Represents the addressing information (N_AI) of the IsoTP layer. Will define what messages will be received and how to craft transmitted message to reach a specific party.

Parameters must be given according to the addressing mode. When not needed, a parameter may be left unset or set to `None`.

Both the `TransportLayer` and the `isotp.socket` expects this address object

**Parameters**

- **addressing_mode** *(int)* – The addressing mode. Valid values are defined by the `AddressingMode` class
- **txid** *(int or None)* – The CAN ID for transmission. Used for these addressing mode: Normal_11bits, Normal_29bits, Extended_11bits, Extended_29bits, Mixed_11bits
- **rxid** *(int or None)* – The CAN ID for reception. Used for these addressing mode: Normal_11bits, Normal_29bits, Extended_11bits, Extended_29bits, Mixed_11bits
- **target_address** *(int or None)* – Target address (N_TA) used in NormalFixed_29bits and Mixed_29bits addressing mode.
• **source_address** (*int or None*) – Source address (N_SA) used in NormalFixed_29bits and Mixed_29bits addressing mode.

• **address_extension** (*int or None*) – Address extension (N_AE) used in Mixed_11bits, Mixed_29bits addressing mode.

```python
class isotp.AddressingMode:
    Normal_11bits = 0
    Normal_29bits = 1
    NormalFixed_29bits = 2
    Extended_11bits = 3
    Extended_29bits = 4
    Mixed_11bits = 5
    Mixed_29bits = 6
```

```python
class isotp.TargetAddressType:
    Physical = 0
    Functional = 1
```

### 3.2 Addressing modes

#### 3.2.1 Normal addressing

In normal addressing, a CAN arbitration ID is selected for transmission (txid) and reception (rxid).

**Condition to receive a message (discarded if not met):**

- Message arbitration ID must match receiver rxid

This mode is possible in both legislated 11-bits and extended 29-bits CAN identifiers

**Example:**

- rxid : 0x123
- txid : 0x456

```plaintext
// Reception of a 10 bytes payload
0x123  [8] 10 0A 00 01 02 03 04 // First frame
0x456  [4] 03 00 08 00 // Flow control
0x123  [6] 21 05 06 07 08 09 // Consecutive frame
```
3.2.2 Normal fixed addressing

In normal fixed addressing, a target_address and a source_address is encoded in the CAN arbitration ID.

Condition to receive a message (discarded if not met):

- Message Target Address must match receiver source_address
- Message Source Address must match the receiver target_address

This mode is only possible with extended 29-bits CAN identifiers.

A message arbitration ID sent in normal fixed addressing is encoded like the following (with $<TA>$=Target Address and $<SA>$=Source Address)

- 1-to-1 communication (target_address_type = Physical) : 0x18DA<TA><SA>
- 1-to-n communication (target_address_type = Functional) : 0x18DB<TA><SA>

Example :

- source_address : 0x55
- target_address : 0xAA

```
// Reception of a 10 bytes payload
0x18DA55AA  [8]  10 0A 00 01 02 03 04 // First frame
0x18DAAA55  [4]  03 00 08 00       // Flow control
0x18DA55AA  [6]  21 05 06 07 08 09 // Consecutive frame
```

3.2.3 Extended addressing

In extended addressing, rxid and txid must be set just like normal addressing, but an additional source_address and target_address must be given. The additional addresses will be added as the first byte of each CAN message sent.

Condition to receive a message (discarded if not met):

- Message arbitration ID must match receiver rxid
- Payload first byte must match receiver source_address

This mode is possible in both legislated 11-bits and extended 29-bits CAN identifiers

Example :

- rxid : 0x123
- txid : 0x456
- source_address : 0x55
- target_address : 0xAA

```
// Reception of a 10 bytes payload
0x123   [8]  55 10 0A 00 01 02 03 // First frame
0x456   [5]  AA 03 00 08 00       // Flow control
0x123   [8]  55 21 04 05 06 07 08 // Consecutive frame
```
3.2.4 Mixed addressing - 11 bits

Mixed addressing (11 bits) is a mix of normal addressing and extended addressing. The payload prefix is called `address_extension`.

When used in legislated 11-bits CAN, Mixed addressing behave like extended addressing with both `source_address` and `target_address` being defined by `address_extension`.

**Condition to receive a message (discarded if not met):**

- Message arbitration ID must match receiver `rxid`.
- Payload first byte must match receiver `address_extension`.

**Example:**

- `rxid` : 0x123
- `txid` : 0x456
- `address_extension` : 0x99

```
// Reception of a 10 bytes payload
0x123 [8] 99 10 0A 00 01 02 03 // First frame
0x456 [5] 99 03 00 08 00 // Flow control
0x123 [8] 99 21 04 05 06 07 08 09 // consecutive frame
```

3.2.5 Mixed addressing - 29 bits

Mixed addressing (29 bits) is a mix of normal fixed addressing and extended addressing. The payload prefix is called `address_extension`.

A message arbitration ID sent in 29 bits mixed addressing is encoded like the following (with `<TA>`=Target Address and `<SA>`=Source Address)

- 1-to-1 communication (target_address_type = Physical) : 0x18CE<TA><SA>
- 1-to-n communication (target_address_type = Functional) : 0x18CD<TA><SA>

**Condition to receive a message (discarded if not met):**

- Message Target Address must match receiver `source_address`.
- Message Source Address must match the receiver `target_address`.
- Payload first byte must match receiver `address_extension`.

**Example:**

- `source_address` : 0x55
- `target_address` : 0xAA
- `address_extension` : 0x99

```
// Reception of a 10 bytes payload
0x18CE55AA [8] 99 10 0A 00 01 02 03 // First frame
0x18CEAA55 [5] 99 03 00 08 00 // Flow control
0x18CE55AA [8] 99 21 04 05 06 07 08 09 // consecutive frame
```
4.1 Basic transmission with python-can

```python
import isotp
import logging
import time

from can.interfaces.vector import VectorBus

def my_error_handler(error):
    logging.warning('IsoTp error happened: %s - %s' % (error.__class__.__name__, str(error)))

bus = VectorBus(channel=0, bitrate=500000)
addr = isotp.Address(isotp.AddressingMode.Normal_11bits, rxid=0x123, txid=0x456)

stack = isotp.CanStack(bus, address=addr, error_handler=my_error_handler)
stack.send(b'Hello, this is a long payload sent in small chunks')

while stack.transmitting():
    stack.process()
    time.sleep(stack.sleep_time())

print("Payload transmission done.")

bus.shutdown()
```
4.2 Threaded reception with python-can

```python
import isotp
import logging
import time
import threading

from can.interfaces.socketcan import SocketcanBus

class ThreadedApp:
    def __init__(self):
        self.exit_requested = False
        self.bus = SocketcanBus(channel='vcan0')
        addr = isotp.Address(isotp.AddressingMode.Normal_11bits, rxid=0x123, txid=0x456)
        self.stack = isotp.CanStack(self.bus, address=addr, error_handler=self.my_error_handler)

    def start(self):
        self.exit_requested = False
        self.thread = threading.Thread(target = self.thread_task)
        self.thread.start()

    def stop(self):
        self.exit_requested = True
        if self.thread.isAlive():
            self.thread.join()

    def my_error_handler(self, error):
        logging.warning('IsoTp error happened : %s - %s' % (error.__class__.__name__, str(error)))

    def thread_task(self):
        while self.exit_requested == False:
            self.stack.process()  # Non-blocking
            time.sleep(self.stack.sleep_time())  # Variable sleep time based on state

    def shutdown(self):
        self.stop()
        self.bus.shutdown()

if __name__ == '__main__':
    app = ThreadedApp()
    app.start()

    print('Waiting for payload - maximum 5 sec')
    t1 = time.time()
    while time.time() - t1 < 5:
        if app.stack.available():
            payload = app.stack.recv()
            print("Received payload : %s" % (payload))
            break
        time.sleep(0.2)

    print("Exiting")
    app.shutdown()
```
4.3 Different type of addresses

```python
import isotp

isotp.Address(isotp.AddressingMode.Normal_11bits, rxid=0x123, txid=0x456)
isotp.Address(isotp.AddressingMode.Normal_29bits, rxid=0x123456, txid=0x789ABC)
isotp.Address(isotp.AddressingMode.NormalFixed_29bits, source_address=0x11, target_address=0x22)
isotp.Address(isotp.AddressingMode.Extended_11bits, rxid=0x123, txid=0x456, source_address=0x55, target_address=0xAA)
isotp.Address(isotp.AddressingMode.Extended_29bits, rxid=0x123456, txid=0x789ABC, source_address=0x55, target_address=0xAA)
isotp.Address(isotp.AddressingMode.Mixed_11bits, rxid=0x123, txid=0x456, address_extension=0x99)
isotp.Address(isotp.AddressingMode.Mixed_29bits, source_address=0x11, target_address=0x22, address_extension=0x99)
```

4.4 Sending with functional addressing (broadcast)

```python
import isotp
import time

from can.interfaces.vector import VectorBus

bus = VectorBus(channel=0, bitrate=500000)
addr = isotp.Address(isotp.AddressingMode.Normal_11bits, rxid=0x123, txid=0x456)
stack = isotp.CanStack(bus, address=addr)
stack.send(b'Hello', isotp.TargetAddressType.Functional)  # Payload must fit a Single Frame. Functional addressing only works with Single Frames

while stack.transmitting():
    stack.process()
    time.sleep(stack.sleep_time())

bus.shutdown()
```

4.5 Defining custom rxfn and txfn

In this example, we see how to configure a `TransportLayer` to interact with a hardware different than python-can with a fictive API.

```python
import isotp

def my_rxfn():
    # All my_hardware_something and get_something() function are fictive of course.
    msg = my_hardware_api_recv()
    return isotp.CanMessage(arbitration_id=msg.get_id(), data=msg.get_data(), dlc=msg.get_dlc(), extended_id=msg.is_extended_id())
```

(continues on next page)
def my_txfn(isotp_msg):
    # all set_something functions and my_hardware_something are fictive.
    msg = my_hardware_api_make_msg()
    msg.set_id(isotp_msg.arbitration_id)
    msg.set_data(isotp_msg.data)
    msg.set_dlc(isotp_msg.dlc)
    msg.set_extended_id(isotp_msg.is_extended_id)
    my_hardware_api_send(msg)

addr = isotp.Address(isotp.AddressingMode.Normal_29bits, txid=0x123456, rxid = 0x123457)
layer = isotp.TransportLayer(rxfn=my_rxfn, txfn=my_txfn, address=addr)

# ... rest of programs
# ...
my_hardware_close()

4.6 Defining partial rxfn and txfn

If your hardware API requires some sort of handle to be given to its functions, you will need a way to pass this handle from your app down to rxfn and txfn. The TransportLayer will call rxfn and txfn with no additional parameters, which might be an issue.

A clean way to overcome this limitation is to use a functools.partial function.

```python
import isotp
from functools import partial  # Allow partial functions

# hardware_handle is passed through partial func
def my_rxfn(hardware_handle):
    msg = my_hardware_api_recv(hardware_handle)
    return isotp.CanMesage(arbitration_id=msg.get_id(), data=msg.get_data(), dlc=msg.get_dlc(), extended_id=msg.is_extended_id())

# hardware_handle is passed through partial func
def my_txfn(hardware_handle, isotp_msg):
    # all set_something functions and my_hardware_something are fictive.
    msg = my_hardware_api_make_msg()
    msg.set_id(isotp_msg.arbitration_id)
    msg.set_data(isotp_msg.data)
    msg.set_dlc(isotp_msg.dlc)
    msg.set_extended_id(isotp_msg.is_extended_id)
    my_hardware_api_send(hardware_handle, msg)

hardware_handle = my_hardware_open()  # Fictive handle mechanism
addr = isotp.Address(isotp.AddressingMode.Normal_29bits, txid=0x123456, rxid = 0x123457)
# This is where the magic happens
layer = isotp.TransportLayer(rxfn=partial(my_rxfn, hardware_handle), txfn=partial(my_txfn, hardware_handle), address=addr)
```

(continues on next page)
This project is a Python package meant to provide support for IsoTP (ISO-15765) protocol written in Python 3. The code is published under MIT license on GitHub (pylessard/python-can-isotp).

This package contains a Python implementation of the protocol that works in the user space that may or may not be coupled with python-can. It also contains a wrapper for a simplified usage of the Linux SocketCAN IsoTP kernel module.
UnsportedWaitFrameError (class in isotp), 5

W

wftmax, 3
WrongSequenceNumberError (class in isotp), 5