python-can

Release 4.3.1

unknown

Dec 12, 2023

CONTENTS

The **python-can** library provides Controller Area Network support for Python, providing common abstractions to different hardware devices, and a suite of utilities for sending and receiving messages on a CAN bus.

python-can runs any where Python runs; from high powered computers with commercial *CAN to USB* devices right down to low powered devices running linux such as a BeagleBone or RaspberryPi.

More concretely, some example uses of the library:

- Passively logging what occurs on a CAN bus. For example monitoring a commercial vehicle using its OBD-II port.
- Testing of hardware that interacts via CAN. Modules found in modern cars, motorcycles, boats, and even wheelchairs have had components tested from Python using this library.
- Prototyping new hardware modules or software algorithms in-the-loop. Easily interact with an existing bus.
- Creating virtual modules to prototype CAN bus communication.

Brief example of the library in action: connecting to a CAN bus, creating and sending a message:

```
#!/usr/bin/env python

"""
This example shows how sending a single message works.
"""

import can


def send_one():
    """Sends a single message."""

    # this uses the default configuration (for example from the config file)
    # see https://python-can.readthedocs.io/en/stable/configuration.html
    with can.Bus() as bus:
        # Using specific buses works similar:
        # bus = can.Bus(interface='socketcan', channel='vcan0', bitrate=250000)
        # bus = can.Bus(interface='pcan', channel='PCAN_USBBUS1', bitrate=250000)
        # bus = can.Bus(interface='ixxat', channel=0, bitrate=250000)
        # bus = can.Bus(interface='vector', app_name='CANalyzer', channel=0, bitrate=250000)
        # ...

        msg = can.Message(
            arbitration_id=0xC0FFEE, data=[0, 25, 0, 1, 3, 1, 4, 1], is_extended_id=True
        )

        try:
            bus.send(msg)
            print(f"Message sent on {bus.channel_info}")
        except can.CanError:
            print("Message NOT sent")


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

Contents:

CHAPTER

ONE

INSTALLATION

Install the `can` package from PyPi with `pip` or similar:

```
$ pip install python-can
```

Warning: As most likely you will want to interface with some hardware, you may also have to install platform dependencies. Be sure to check any other specifics for your hardware in *Hardware Interfaces*.

Many interfaces can install their dependencies at the same time as `python-can`, for instance the interface `serial` includes the `pyserial` dependency which can be installed with the `serial` extra:

```
$ pip install python-can[serial]
```

1.1 GNU/Linux dependencies

Reasonably modern Linux Kernels (2.6.25 or newer) have an implementation of `socketcan`. This version of python-can will directly use socketcan if called with Python 3.3 or greater, otherwise that interface is used via ctypes.

1.2 Windows dependencies

1.2.1 Kvaser

To install `python-can` using the Kvaser CANLib SDK as the backend:

1. Install Kvaser's latest Windows CANLib drivers.
2. Test that Kvaser's own tools work to ensure the driver is properly installed and that the hardware is working.

1.2.2 PCAN

Download and install the latest driver for your interface:

- Windows (also supported on *Cygwin*)
- Linux (also works without, see also *Linux installation*)
- macOS

Note that PCANBasic API timestamps count seconds from system startup. To convert these to epoch times, the uptime library is used. If it is not available, the times are returned as number of seconds from system startup. To install the uptime library, run `pip install python-can[pcan]`.

This library can take advantage of the Python for Windows Extensions library if installed. It will be used to get notified of new messages instead of the CPU intensive polling that will otherwise have be used.

1.2.3 IXXAT

To install `python-can` using the IXXAT VCI V3 or V4 SDK as the backend:

1. Install IXXAT's latest Windows VCI V3 SDK or VCI V4 SDK (Win10) drivers.
2. Test that IXXAT's own tools (i.e. MiniMon) work to ensure the driver is properly installed and that the hardware is working.

1.2.4 NI-CAN

Download and install the NI-CAN drivers from National Instruments.

Currently the driver only supports 32-bit Python on Windows.

1.2.5 neoVI

See *Intrepid Control Systems neoVI*.

1.2.6 Vector

To install `python-can` using the XL Driver Library as the backend:

1. Install the latest drivers for your Vector hardware interface.
2. Install the XL Driver Library or copy the `vxlapi.dll` and/or `vxlapi64.dll` into your working directory.
3. Use Vector Hardware Configuration to assign a channel to your application.

1.2.7 CANtact

CANtact is supported on Linux, Windows, and macOS. To install `python-can` using the CANtact driver backend:

```
python3 -m pip install "python-can[cantact]"
```

If `python-can` is already installed, the CANtact backend can be installed separately:

```
pip install cantact
```

Additional CANtact documentation is available at cantact.io.

1.2.8 CanViewer

`python-can` has support for showing a simple CAN viewer terminal application by running `python -m can.viewer`. On Windows, this depends on the windows-curses library which can be installed with:

```
python -m pip install "python-can[viewer]"
```

1.3 Installing python-can in development mode

A "development" install of this package allows you to make changes locally or pull updates from the Git repository and use them without having to reinstall. Download or clone the source repository then:

```
python setup.py develop
```

CHAPTER

TWO

CONFIGURATION

Usually this library is used with a particular CAN interface, this can be specified in code, read from configuration files or environment variables.

See *can.util.load_config()* for implementation.

2.1 In Code

The `can` object exposes an `rc` dictionary which can be used to set the **interface** and **channel**.

```
import can
can.rc['interface'] = 'socketcan'
can.rc['channel'] = 'vcan0'
can.rc['bitrate'] = 500000
from can.interface import Bus

bus = Bus()
```

You can also specify the interface and channel for each Bus instance:

```
import can

bus = can.interface.Bus(interface='socketcan', channel='vcan0', bitrate=500000)
```

2.2 Configuration File

On Linux and macOS systems the config file is searched in the following paths:

1. `~/can.conf`
2. `/etc/can.conf`
3. `$HOME/.can`
4. `$HOME/.canrc`

On Windows systems the config file is searched in the following paths:

1. `%USERPROFILE%/can.conf`
2. `can.ini` (current working directory)
3. `%APPDATA%/can.ini`

The configuration file sets the default interface and channel:

```
[default]
interface = <the name of the interface to use>
channel = <the channel to use by default>
bitrate = <the bitrate in bits/s to use by default>
```

The configuration can also contain additional sections (or context):

```
[default]
interface = <the name of the interface to use>
channel = <the channel to use by default>
bitrate = <the bitrate in bits/s to use by default>

[HS]
# All the values from the 'default' section are inherited
channel = <the channel to use>
bitrate = <the bitrate in bits/s to use. i.e. 500000>

[MS]
# All the values from the 'default' section are inherited
channel = <the channel to use>
bitrate = <the bitrate in bits/s to use. i.e. 125000>
```

```
from can.interface import Bus

hs_bus = Bus(context='HS')
ms_bus = Bus(context='MS')
```

2.3 Environment Variables

Configuration can be pulled from these environmental variables:

- CAN_INTERFACE
- CAN_CHANNEL
- CAN_BITRATE
- CAN_CONFIG

The `CAN_CONFIG` environment variable allows to set any bus configuration using JSON.

For example:

```
CAN_INTERFACE=socketcan CAN_CONFIG={"receive_own_messages":  true, "fd":  true}
```

2.4 Interface Names

Lookup table of interface names:

Name	Documentation
`"canalystii"`	*CANalyst-II*
`"cantact"`	*CANtact CAN Interface*
`"etas"`	*ETAS*
`"gs_usb"`	*Geschwister Schneider and candleLight*
`"iscan"`	*isCAN*
`"ixxat"`	*IXXAT Virtual Communication Interface*
`"kvaser"`	*Kvaser's CANLIB*
`"neousys"`	*Neousys CAN Interface*
`"neovi"`	*Intrepid Control Systems neoVI*
`"nican"`	*National Instruments NI-CAN*
`"nixnet"`	*National Instruments NI-XNET*
`"pcan"`	*PCAN Basic API*
`"robotell"`	*Robotell CAN-USB interface*
`"seeedstudio"`	*Seeed Studio USB-CAN Analyzer*
`"serial"`	*CAN over Serial*
`"slcan"`	*CAN over Serial / SLCAN*
`"socketcan"`	*SocketCAN*
`"socketcand"`	*socketcand Interface*
`"systec"`	*SYSTEC interface*
`"udp_multicast"`	*Multicast IP Interface*
`"usb2can"`	*USB2CAN Interface*
`"vector"`	*Vector*
`"virtual"`	*Virtual*

Additional interface types can be added via the *Plugin Interface*.

CHAPTER

THREE

LIBRARY API

The main objects are the *Bus* and the *Message*. A form of CAN interface is also required.

Hint: Check the backend specific documentation for any implementation specific details.

3.1 Bus

The *BusABC* class provides a wrapper around a physical or virtual CAN Bus.

An interface specific instance is created by calling the *Bus()* function with a particular `interface`, for example:

```
vector_bus = can.Bus(interface='vector', ...)
```

The created bus is then able to handle the interface specific software/hardware interactions while giving the user the same top level API.

A thread safe bus wrapper is also available, see *Thread safe bus*.

3.1.1 Transmitting

Writing individual messages to the bus is done by calling the *send()* method and passing a *Message* instance.

```
with can.Bus() as bus:
    msg = can.Message(
        arbitration_id=0xC0FFEE,
        data=[0, 25, 0, 1, 3, 1, 4, 1],
        is_extended_id=True
    )
    try:
        bus.send(msg)
        print(f"Message sent on {bus.channel_info}")
    except can.CanError:
        print("Message NOT sent")
```

Periodic sending is controlled by the *broadcast manager*.

3.1.2 Receiving

Reading from the bus is achieved by either calling the *recv()* method or by directly iterating over the bus:

```
with can.Bus() as bus:
    for msg in bus:
        print(msg.data)
```

Alternatively the *Listener* api can be used, which is a list of various *Listener* implementations that receive and handle messages from a *Notifier*.

3.1.3 Filtering

Message filtering can be set up for each bus. Where the interface supports it, this is carried out in the hardware or kernel layer - not in Python. All messages that match at least one filter are returned.

Example defining two filters, one to pass 11-bit ID `0x451`, the other to pass 29-bit ID `0xA0000`:

```
filters = [
    {"can_id": 0x451, "can_mask": 0x7FF, "extended": False},
    {"can_id": 0xA0000, "can_mask": 0x1FFFFFFF, "extended": True},
]
bus = can.interface.Bus(channel="can0", interface="socketcan", can_filters=filters)
```

See *set_filters()* for the implementation.

3.1.4 Bus API

can.**Bus**(*channel=None*, *interface=None*, *config_context=None*, *ignore_config=False*, ***kwargs*)

Create a new bus instance with configuration loading.

Instantiates a CAN Bus of the given `interface`, falls back to reading a configuration file from default locations.

Note: Please note that while the arguments provided to this class take precedence over any existing values from configuration, it is possible that other parameters from the configuration may be added to the bus instantiation. This could potentially have unintended consequences. To prevent this, you may use the *ignore_config* parameter to ignore any existing configurations.

Parameters

- **channel** (*str | int | None*) – Channel identification. Expected type is backend dependent. Set to `None` to let it be resolved automatically from the default *Configuration*.
- **interface** (*str | None*) – See *Interface Names* for a list of supported interfaces. Set to `None` to let it be resolved automatically from the default *Configuration*.
- **config_context** (*str | None*) – Extra 'context', that is passed to config sources. This can be used to select a section other than 'default' in the configuration file.
- **ignore_config** (*bool*) – If `True`, only the given arguments will be used for the bus instantiation. Existing configuration sources will be ignored.
- **kwargs** (*Any*) – `interface` specific keyword arguments.

Raises

- *CanInterfaceNotImplementedError* – if the `interface` isn't recognized or cannot be loaded
- *CanInitializationError* – if the bus cannot be instantiated
- `ValueError` – if the `channel` could not be determined

Return type
BusABC

class can.**BusABC**(*channel*, *can_filters=None*, ***kwargs*)

The CAN Bus Abstract Base Class that serves as the basis for all concrete interfaces.

This class may be used as an iterator over the received messages and as a context manager for auto-closing the bus when done using it.

Please refer to *Error Handling* for possible exceptions that may be thrown by certain operations on this bus.

Parameters

- **channel** (*Any*) –
- **can_filters** (*Sequence[CanFilter | CanFilterExtended] | None*) –
- **kwargs** (*object*) –

RECV_LOGGING_LEVEL = 9

Log level for received messages

channel_info = 'unknown'

a string describing the underlying bus and/or channel

property filters: Sequence[CanFilter | CanFilterExtended] | None

Modify the filters of this bus. See *set_filters()* for details.

flush_tx_buffer()

Discard every message that may be queued in the output buffer(s).

Return type
None

property protocol: *CanProtocol*

Return the CAN protocol used by this bus instance.

This value is set at initialization time and does not change during the lifetime of a bus instance.

recv(*timeout=None*)

Block waiting for a message from the Bus.

Parameters
timeout (*float | None*) – seconds to wait for a message or None to wait indefinitely

Returns
`None` on timeout or a *Message* object.

Raises
CanOperationError – If an error occurred while reading

Return type
Message | None

abstract send(*msg*, *timeout=None*)

Transmit a message to the CAN bus.

Override this method to enable the transmit path.

Parameters

- **msg** (Message) – A message object.
- **timeout** (*float | None*) – If > 0, wait up to this many seconds for message to be ACK'ed or for transmit queue to be ready depending on driver implementation. If timeout is exceeded, an exception will be raised. Might not be supported by all interfaces. None blocks indefinitely.

Raises

CanOperationError – If an error occurred while sending

Return type

None

send_periodic(*msgs*, *period*, *duration=None*, *store_task=True*, *modifier_callback=None*)

Start sending messages at a given period on this bus.

The task will be active until one of the following conditions are met:

- the (optional) duration expires
- the Bus instance goes out of scope
- the Bus instance is shutdown
- *stop_all_periodic_tasks()* is called
- the task's *stop()* method is called.

Parameters

- **msgs** (Message | *Sequence*[Message]) – Message(s) to transmit
- **period** (*float*) – Period in seconds between each message
- **duration** (*float | None*) – Approximate duration in seconds to continue sending messages. If no duration is provided, the task will continue indefinitely.
- **store_task** (*bool*) – If True (the default) the task will be attached to this Bus instance. Disable to instead manage tasks manually.
- **modifier_callback** (*Callable*[[Message], *None*] | *None*) – Function which should be used to modify each message's data before sending. The callback modifies the *data* of the message and returns None.

Returns

A started task instance. Note the task can be stopped (and depending on the backend modified) by calling the task's *stop()* method.

Return type

CyclicSendTaskABC

Note: Note the duration before the messages stop being sent may not be exactly the same as the duration specified by the user. In general the message will be sent at the given rate until at least **duration** seconds.

Note: For extremely long-running Bus instances with many short-lived tasks the default api with `store_task==True` may not be appropriate as the stopped tasks are still taking up memory as they are associated with the Bus instance.

set_filters(*filters=None*)

Apply filtering to all messages received by this Bus.

All messages that match at least one filter are returned. If *filters* is *None* or a zero length sequence, all messages are matched.

Calling without passing any filters will reset the applied filters to `None`.

Parameters

filters (*Sequence[CanFilter | CanFilterExtended] | None*) – A iterable of dictionaries each containing a "can_id", a "can_mask", and an optional "extended" key:

```
[{"can_id": 0x11, "can_mask": 0x21, "extended": False}]
```

A filter matches, when `<received_can_id> & can_mask == can_id & can_mask`. If `extended` is set as well, it only matches messages where `<received_is_extended> == extended`. Else it matches every messages based only on the arbitration ID and mask.

Return type

None

shutdown()

Called to carry out any interface specific cleanup required in shutting down a bus.

This method can be safely called multiple times.

Return type

None

property state: *BusState*

Return the current state of the hardware

stop_all_periodic_tasks(*remove_tasks=True*)

Stop sending any messages that were started using *send_periodic()*.

Note: The result is undefined if a single task throws an exception while being stopped.

Parameters

remove_tasks (*bool*) – Stop tracking the stopped tasks.

Return type

None

class can.**BusState**(*value*)

The state in which a *can.BusABC* can be.

ACTIVE = 1

ERROR = 3

PASSIVE = 2

class can.bus.**CanProtocol**(*value*)

The CAN protocol type supported by a *can.BusABC* instance

CAN_20 = 1

CAN_FD = 2

CAN_XL = 3

3.1.5 Thread safe bus

This thread safe version of the *BusABC* class can be used by multiple threads at once. Sending and receiving is locked separately to avoid unnecessary delays. Conflicting calls are executed by blocking until the bus is accessible.

It can be used exactly like the normal *BusABC*:

```
# 'socketcan' is only an example interface, it works with all the others too
my_bus = can.ThreadSafeBus(interface='socketcan', channel='vcan0')
my_bus.send(...)
my_bus.recv(...)
```

class can.**ThreadSafeBus**(**args*, ***kwargs*)

Contains a thread safe *can.BusABC* implementation that wraps around an existing interface instance. All public methods of that base class are now safe to be called from multiple threads. The send and receive methods are synchronized separately.

Use this as a drop-in replacement for *BusABC*.

Note: This approach assumes that both *send()* and *_recv_internal()* of the underlying bus instance can be called simultaneously, and that the methods use *_recv_internal()* instead of *recv()* directly.

3.2 Message

class can.**Message**(*timestamp=0.0*, *arbitration_id=0*, *is_extended_id=True*, *is_remote_frame=False*, *is_error_frame=False*, *channel=None*, *dlc=None*, *data=None*, *is_fd=False*, *is_rx=True*, *bitrate_switch=False*, *error_state_indicator=False*, *check=False*)

The *Message* object is used to represent CAN messages for sending, receiving and other purposes like converting between different logging formats.

Messages can use extended identifiers, be remote or error frames, contain data and may be associated to a channel.

Messages are always compared by identity and never by value, because that may introduce unexpected behaviour. See also *equals()*.

copy()/deepcopy() is supported as well.

Messages do not support "dynamic" attributes, meaning any others than the documented ones, since it uses __slots__.

To create a message object, simply provide any of the below attributes together with additional parameters as keyword arguments to the constructor.

Parameters

- **check** (*bool*) – By default, the constructor of this class does not strictly check the input. Thus, the caller must prevent the creation of invalid messages or set this parameter to *True*, to raise an Error on invalid inputs. Possible problems include the *dlc* field not matching the length of *data* or creating a message with both *is_remote_frame* and *is_error_frame* set to *True*.
- **timestamp** (*float*) –
- **arbitration_id** (*int*) –
- **is_extended_id** (*bool*) –
- **is_remote_frame** (*bool*) –
- **is_error_frame** (*bool*) –
- **channel** (*str | int | None*) –
- **dlc** (*int | None*) –
- **data** (*bytes | bytearray | int | Iterable[int] | None*) –
- **is_fd** (*bool*) –
- **is_rx** (*bool*) –
- **bitrate_switch** (*bool*) –
- **error_state_indicator** (*bool*) –

Raises

ValueError – If and only if *check* is set to *True* and one or more arguments were invalid

One can instantiate a *Message* defining data, and optional arguments for all attributes such as arbitration ID, flags, and timestamp.

```
>>> from can import Message
>>> test = Message(data=[1, 2, 3, 4, 5])
>>> test.data
bytearray(b'\x01\x02\x03\x04\x05')
>>> test.dlc
5
>>> print(test)
Timestamp:        0.000000    ID: 00000000    X Rx                DL:  5    01 02␣
↪03 04 05
```

The *arbitration_id* field in a CAN message may be either 11 bits (standard addressing, CAN 2.0A) or 29 bits (extended addressing, CAN 2.0B) in length, and `python-can` exposes this difference with the *is_extended_id* attribute.

timestamp

Type
float

The timestamp field in a CAN message is a floating point number representing when the message was received since the epoch in seconds. Where possible this will be timestamped in hardware.

arbitration_id

Type
int

The frame identifier used for arbitration on the bus.

The arbitration ID can take an int between 0 and the maximum value allowed depending on the `is_extended_id` flag (either 2^{11} - 1 for 11-bit IDs, or 2^{29} - 1 for 29-bit identifiers).

```
>>> print(Message(is_extended_id=False, arbitration_id=100))
Timestamp:        0.000000    ID:      064    S Rx                DL:  0
```

data

Type
bytearray

The data parameter of a CAN message is exposed as a **bytearray** with length between 0 and 8.

```
>>> example_data = bytearray([1, 2, 3])
>>> print(Message(data=example_data))
Timestamp:        0.000000    ID: 00000000    X Rx                DL:  3    01␣
↪02 03
```

A *Message* can also be created with bytes, or lists of ints:

```
>>> m1 = Message(data=[0x64, 0x65, 0x61, 0x64, 0x62, 0x65, 0x65, 0x66])
>>> print(m1.data)
bytearray(b'deadbeef')
>>> m2 = Message(data=b'deadbeef')
>>> m2.data
bytearray(b'deadbeef')
```

dlc

Type
int

The DLC (Data Length Code) parameter of a CAN message is an integer between 0 and 8 representing the frame payload length.

In the case of a CAN FD message, this indicates the data length in number of bytes.

```
>>> m = Message(data=[1, 2, 3])
>>> m.dlc
3
```

Note: The DLC value does not necessarily define the number of bytes of data in a message.

Its purpose varies depending on the frame type - for data frames it represents the amount of data contained in the message, in remote frames it represents the amount of data being requested.

---

channel

Type
str or int or None

This might store the channel from which the message came.

is_extended_id

Type
bool

This flag controls the size of the *arbitration_id* field. Previously this was exposed as *id_type*.

```
>>> print(Message(is_extended_id=False))
Timestamp:        0.000000    ID:      000    S Rx                DL:  0
>>> print(Message(is_extended_id=True))
Timestamp:        0.000000    ID: 00000000    X Rx                DL:  0
```

Note: The initializer argument and attribute `extended_id` has been deprecated in favor of `is_extended_id`, but will continue to work for the `3.x` release series.

`is_error_frame`

Type
bool

This boolean parameter indicates if the message is an error frame or not.

```
>>> print(Message(is_error_frame=True))
Timestamp:        0.000000    ID: 00000000    X Rx E              DL:  0
```

`is_remote_frame`

Type
bool

This boolean attribute indicates if the message is a remote frame or a data frame, and modifies the bit in the CAN message's flags field indicating this.

```
>>> print(Message(is_remote_frame=True))
Timestamp:        0.000000    ID: 00000000    X Rx   R            DL:  0
```

`is_fd`

Type
bool

Indicates that this message is a CAN FD message.

`is_rx`

Type
bool

Indicates whether this message is a transmitted (Tx) or received (Rx) frame

`bitrate_switch`

Type
bool

If this is a CAN FD message, this indicates that a higher bitrate was used for the data transmission.

`error_state_indicator`

Type
bool

If this is a CAN FD message, this indicates an error active state.

__str__()

A string representation of a CAN message:

```
>>> from can import Message
>>> test = Message()
>>> print(test)
Timestamp:        0.000000    ID: 00000000    X Rx                DL:  0
>>> test2 = Message(data=[1, 2, 3, 4, 5])
>>> print(test2)
Timestamp:        0.000000    ID: 00000000    X Rx                DL:  5    01␣
↪02 03 04 05
```

The fields in the printed message are (in order):

- timestamp,
- arbitration ID,
- flags,
- data length (DL),
- and data.

The flags field is represented as one, two or three letters:

- X if the *is_extended_id* attribute is set, otherwise S,
- E if the *is_error_frame* attribute is set,
- R if the *is_remote_frame* attribute is set.

The arbitration ID field is represented as either a four or eight digit hexadecimal number depending on the length of the arbitration ID (11-bit or 29-bit).

Each of the bytes in the data field (when present) are represented as two-digit hexadecimal numbers.

equals(*other*, *timestamp_delta=1e-06*, *check_channel=True*, *check_direction=True*)

Compares a given message with this one.

Parameters

- **other** (Message) – the message to compare with
- **timestamp_delta** (*float | None*) – the maximum difference in seconds at which two timestamps are still considered equal or *None* to not compare timestamps
- **check_channel** (*bool*) – whether to compare the message channel
- **check_direction** (*bool*) – whether to compare the messages' directions (Tx/Rx)

Returns

True if and only if the given message equals this one

Return type

bool

3.3 Reading and Writing Messages

3.3.1 Notifier

The Notifier object is used as a message distributor for a bus. Notifier creates a thread to read messages from the bus and distributes them to listeners.

class can.**Notifier**(*bus*, *listeners*, *timeout=1.0*, *loop=None*)

Manages the distribution of *Message* instances to listeners.

Supports multiple buses and listeners.

Note: Remember to call *stop()* after all messages are received as many listeners carry out flush operations to persist data.

Parameters

- **bus** (BusABC | *List*[BusABC]) – A *Bus* or a list of buses to listen to.
- **listeners** (*Iterable*[Listener | *Callable*[[Message], *Awaitable*[None] | None]]) – An iterable of *Listener* or callables that receive a *Message* and return nothing.
- **timeout** (*float*) – An optional maximum number of seconds to wait for any *Message*.
- **loop** (*AbstractEventLoop* | *None*) – An asyncio event loop to schedule the `listeners` in.

add_bus(*bus*)

Add a bus for notification.

Parameters
bus (BusABC) – CAN bus instance.

Return type
None

add_listener(*listener*)

Add new Listener to the notification list. If it is already present, it will be called two times each time a message arrives.

Parameters
listener (Listener | *Callable*[[Message], *Awaitable*[None] | None]) – Listener to be added to the list to be notified

Return type
None

exception: Exception | None

Exception raised in thread

remove_listener(*listener*)

Remove a listener from the notification list. This method throws an exception if the given listener is not part of the stored listeners.

Parameters
listener (Listener | *Callable*[[Message], *Awaitable*[None] | None]) – Listener to be removed from the list to be notified

Raises
ValueError – if *listener* was never added to this notifier

Return type
None

stop(*timeout=5*)

Stop notifying Listeners when new *Message* objects arrive and call *stop()* on each Listener.

Parameters
timeout (*float*) – Max time in seconds to wait for receive threads to finish. Should be longer than timeout given at instantiation.

Return type
None

3.3.2 Listener

The Listener class is an "abstract" base class for any objects which wish to register to receive notifications of new messages on the bus. A Listener can be used in two ways; the default is to **call** the Listener with a new message, or by calling the method **on_message_received**.

Listeners are registered with *Notifier* object(s) which ensure they are notified whenever a new message is received.

```
#!/usr/bin/env python

import time
import can


def main():
    with can.Bus(receive_own_messages=True) as bus:
        print_listener = can.Printer()
        can.Notifier(bus, [print_listener])

        bus.send(can.Message(arbitration_id=1, is_extended_id=True))
        bus.send(can.Message(arbitration_id=2, is_extended_id=True))
        bus.send(can.Message(arbitration_id=1, is_extended_id=False))

        time.sleep(1.0)


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

Subclasses of Listener that do not override **on_message_received** will cause NotImplementedError to be thrown when a message is received on the CAN bus.

class can.Listener(**args*, ***kwargs*)

The basic listener that can be called directly to handle some CAN message:

```
listener = SomeListener()
msg = my_bus.recv()

# now either call
```

(continues on next page)

(continued from previous page)

```
listener(msg)
# or
listener.on_message_received(msg)

# Important to ensure all outputs are flushed
listener.stop()
```

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

on_error(*exc*)

This method is called to handle any exception in the receive thread.

Parameters
exc (*Exception*) – The exception causing the thread to stop

Return type
None

abstract on_message_received(*msg*)

This method is called to handle the given message.

Parameters
msg (Message) – the delivered message

Return type
None

stop()

Stop handling new messages, carry out any final tasks to ensure data is persisted and cleanup any open resources.

Concrete implementations override.

Return type
None

There are some listeners that already ship together with *python-can* and are listed below. Some of them allow messages to be written to files, and the corresponding file readers are also documented here.

Note: Please note that writing and the reading a message might not always yield a completely unchanged message again, since some properties are not (yet) supported by some file formats.

Note: Additional file formats for both reading/writing log files can be added via a plugin reader/writer. An external package can register a new reader by using the `can.io.message_reader` entry point. Similarly, a writer can be added using the `can.io.message_writer` entry point.

The format of the entry point is `reader_name=module:classname` where `classname` is a *can.io.generic.BaseIOHandler* concrete implementation.

```
entry_points={
    'can.io.message_reader': [
       '.asc = my_package.io.asc:ASCReader'
   ]
},
```

3.3.3 BufferedReader

class can.**BufferedReader**(**args*, ***kwargs*)

A BufferedReader is a subclass of *Listener* which implements a **message buffer**: that is, when the *can.BufferedReader* instance is notified of a new message it pushes it into a queue of messages waiting to be serviced. The messages can then be fetched with *get_message()*.

Putting in messages after *stop()* has been called will raise an exception, see *on_message_received()*.

Attr is_stopped
True if the reader has been stopped

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

get_message(*timeout=0.5*)

Attempts to retrieve the message that has been in the queue for the longest amount of time (FIFO). If no message is available, it blocks for given timeout or until a message is received (whichever is shorter), or else returns None. This method does not block after *can.BufferedReader.stop()* has been called.

Parameters
timeout (*float*) – The number of seconds to wait for a new message.

Returns
the received *can.Message* or *None*, if the queue is empty.

Return type
Message | None

on_message_received(*msg*)

Append a message to the buffer.

Raises
BufferError if the reader has already been stopped

Parameters
msg (Message) –

Return type
None

stop()

Prohibits any more additions to this reader.

Return type
None

class can.**AsyncBufferedReader**(**args*, ***kwargs*)

A message buffer for use with asyncio.

See *Asyncio support* for how to use with *can.Notifier*.

Can also be used as an asynchronous iterator:

```
async for msg in reader:
    print(msg)
```

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

async **get_message**()

Retrieve the latest message when awaited for:

```
msg = await reader.get_message()
```

Returns
The CAN message.

Return type
Message

on_message_received(*msg*)

Append a message to the buffer.

Must only be called inside an event loop!

Parameters
msg (Message) –

Return type
None

3.3.4 RedirectReader

class can.**RedirectReader**(*bus*, **args*, ***kwargs*)

A RedirectReader sends all received messages to another Bus.

Parameters

- **bus** (BusABC) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

on_message_received(*msg*)

This method is called to handle the given message.

Parameters
msg (Message) – the delivered message

Return type
None

3.3.5 Logger

The *can.Logger* uses the following *can.Listener* types to create log files with different file types of the messages received.

class can.**Logger**(*filename*, ***kwargs*)

Logs CAN messages to a file.

The format is determined from the file suffix which can be one of:

- .asc: *can.ASCWriter*
- .blf *can.BLFWriter*
- .csv: *can.CSVWriter*
- .db: *can.SqliteWriter*
- .log *can.CanutilsLogWriter*
- .trc *can.TRCWriter*
- .txt *can.Printer*
- .mf4 *can.MF4Writer* (optional, depends on asammdf)

Any of these formats can be used with gzip compression by appending the suffix .gz (e.g. filename.asc.gz). However, third-party tools might not be able to read these files.

The **filename** may also be *None*, to fall back to *can.Printer*.

The log files may be incomplete until *stop()* is called due to buffering.

Note: This class itself is just a dispatcher, and any positional and keyword arguments are passed on to the returned instance.

Parameters

- **file** – a path-like object to open a file, a file-like object to be used as a file or *None* to not use a file at all
- **mode** – the mode that should be used to open the file, see open(), ignored if *file* is *None*
- **filename** (*str | os.PathLike[str] | None*) –
- **kwargs** (*Any*) –

Return type
MessageWriter

classmethod compress(*filename*, ***kwargs*)

Return the suffix and io object of the decompressed file. File will automatically recompress upon close.

Parameters

- **filename** (*str | PathLike[str]*) –
- **kwargs** (*Any*) –

Return type
Tuple[*Type*[MessageWriter], *TextIO* | *BinaryIO* | *GzipFile*]

on_message_received(*msg*)

This method is called to handle the given message.

Parameters
msg (Message) – the delivered message

Return type
None

class can.io.**BaseRotatingLogger**(***kwargs*)

Base class for rotating CAN loggers. This class is not meant to be instantiated directly. Subclasses must implement the *should_rollover()* and *do_rollover()* methods according to their rotation strategy.

The rotation behavior can be further customized by the user by setting the *namer* and *rotator* attributes after instantiating the subclass.

These attributes as well as the methods *rotation_filename()* and *rotate()* and the corresponding docstrings are carried over from the python builtin BaseRotatingHandler.

Subclasses must set the *_writer* attribute upon initialization.

Parameters

- **file** – a path-like object to open a file, a file-like object to be used as a file or *None* to not use a file at all
- **mode** – the mode that should be used to open the file, see open(), ignored if *file* is *None*
- **kwargs** (*Any*) –

abstract do_rollover()

Perform rollover.

Return type
None

namer: Callable[[str | os.PathLike[str]], str | os.PathLike[str]] | None = None

If this attribute is set to a callable, the *rotation_filename()* method delegates to this callable. The parameters passed to the callable are those passed to *rotation_filename()*.

on_message_received(*msg*)

This method is called to handle the given message.

Parameters
msg (Message) – the delivered message

Return type
None

rollover_count: int = 0

An integer counter to track the number of rollovers.

rotate(*source*, *dest*)

When rotating, rotate the current log.

The default implementation calls the *rotator* attribute of the handler, if it's callable, passing the *source* and *dest* arguments to it. If the attribute isn't callable (the default is None), the source is simply renamed to the destination.

Parameters

- **source** (*str | PathLike[str]*) – The source filename. This is normally the base filename, e.g. *"test.log"*

- **dest** (*str | PathLike[str]*) – The destination filename. This is normally what the source is rotated to, e.g. *"test_#001.log"*.

Return type
None

rotation_filename(*default_name*)

Modify the filename of a log file when rotating.

This is provided so that a custom filename can be provided. The default implementation calls the *namer* attribute of the handler, if it's callable, passing the default name to it. If the attribute isn't callable (the default is None), the name is returned unchanged.

Parameters
default_name (*str | PathLike[str]*) – The default name for the log file.

Return type
str | *PathLike*[str]

rotator: Callable[[str | os.PathLike[str], str | os.PathLike[str]], None] | None = None

If this attribute is set to a callable, the *rotate()* method delegates to this callable. The parameters passed to the callable are those passed to *rotate()*.

abstract should_rollover(*msg*)

Determine if the rollover conditions are met.

Parameters
msg (Message) –

Return type
bool

stop()

Stop handling new messages.

Carry out any final tasks to ensure data is persisted and cleanup any open resources.

Return type
None

property writer: *FileIOMessageWriter*

This attribute holds an instance of a writer class which manages the actual file IO.

class can.**SizedRotatingLogger**(*base_filename*, *max_bytes=0*, ***kwargs*)

Log CAN messages to a sequence of files with a given maximum size.

The logger creates a log file with the given *base_filename*. When the size threshold is reached the current log file is closed and renamed by adding a timestamp and the rollover count. A new log file is then created and written to.

This behavior can be customized by setting the *namer* and *rotator* attribute.

Example:

```
from can import Notifier, SizedRotatingLogger
from can.interfaces.vector import VectorBus

bus = VectorBus(channel=[0], app_name="CANape", fd=True)
```

(continues on next page)

(continued from previous page)

```
logger = SizedRotatingLogger(
    base_filename="my_logfile.asc",
    max_bytes=5 * 1024 ** 2,  # =5MB
)
logger.rollover_count = 23  # start counter at 23

notifier = Notifier(bus=bus, listeners=[logger])
```

The SizedRotatingLogger currently supports the formats

- .asc: *can.ASCWriter*
- .blf *can.BLFWriter*
- .csv: *can.CSVWriter*
- .log *can.CanutilsLogWriter*
- .txt *can.Printer* (if pointing to a file)

Note: The *can.SqliteWriter* is not supported yet.

The log files on disk may be incomplete due to buffering until *stop()* is called.

Parameters

- **base_filename** (*str | os.PathLike[str]*) – A path-like object for the base filename. The log file format is defined by the suffix of *base_filename*.
- **max_bytes** (*int*) – The size threshold at which a new log file shall be created. If set to 0, no rollover will be performed.
- **kwargs** (*Any*) –

do_rollover()

Perform rollover.

Return type
None

should_rollover(*msg*)

Determine if the rollover conditions are met.

Parameters
msg (Message) –

Return type
bool

3.3.6 Printer

class can.**Printer**(*file=None*, *append=False*, ***kwargs*)

The Printer class is a subclass of *Listener* which simply prints any messages it receives to the terminal (stdout). A message is turned into a string using *__str__()*.

Attr write_to_file
True if this instance prints to a file instead of standard out

Parameters

- **file** (*TextIO | None*) – An optional path-like object or a file-like object to "print" to instead of writing to standard out (stdout). If this is a file-like object, is has to be opened in text write mode, not binary write mode.
- **append** (*bool*) – If set to *True* messages, are appended to the file, else the file is truncated
- **kwargs** (*Any*) –

file_size()

Return an estimate of the current file size in bytes.

Return type
int

on_message_received(*msg*)

This method is called to handle the given message.

Parameters
msg (Message) – the delivered message

Return type
None

3.3.7 CSVWriter

class can.**CSVWriter**(*file*, *append=False*, ***kwargs*)

Writes a comma separated text file with a line for each message. Includes a header line.

The columns are as follows:

name of column	format description	example
timestamp	decimal float	1483389946.197
arbitration_id	hex	0x00dadada
extended	1 == True, 0 == False	1
remote	1 == True, 0 == False	0
error	1 == True, 0 == False	0
dlc	int	6
data	base64 encoded	WzQyLCA5XQ==

Each line is terminated with a platform specific line separator.

Parameters

- **file** (*TextIO*) – a path-like object or a file-like object to write to. If this is a file-like object, is has to open in text write mode, not binary write mode.

- **append** (*bool*) – if set to *True* messages are appended to the file and no header line is written, else the file is truncated and starts with a newly written header line
- **kwargs** (*Any*) –

on_message_received(*msg*)

This method is called to handle the given message.

Parameters
msg (Message) – the delivered message

Return type
None

class can.**CSVReader**(*file*, ***kwargs*)

Iterator over CAN messages from a .csv file that was generated by *CSVWriter* or that uses the same format as described there. Assumes that there is a header and thus skips the first line.

Any line separator is accepted.

Parameters

- **file** (*TextIO*) – a path-like object or as file-like object to read from If this is a file-like object, is has to opened in text read mode, not binary read mode.
- **kwargs** (*Any*) –

3.3.8 SqliteWriter

class can.**SqliteWriter**(*file*, *table_name='messages'*, ***kwargs*)

Logs received CAN data to a simple SQL database.

The sqlite database may already exist, otherwise it will be created when the first message arrives.

Messages are internally buffered and written to the SQL file in a background thread. Ensures that all messages that are added before calling *stop()* are actually written to the database after that call returns. Thus, calling *stop()* may take a while.

Attr str table_name
the name of the database table used for storing the messages

Attr int num_frames
the number of frames actually written to the database, this excludes messages that are still buffered

Attr float last_write
the last time a message war actually written to the database, as given by `time.time()`

Parameters

- **file** (*TextIO | BinaryIO | GzipFile | None*) –
- **table_name** (*str*) –
- **kwargs** (*Any*) –

Note: When the listener's *stop()* method is called the thread writing to the database will continue to receive and internally buffer messages if they continue to arrive before the *GET_MESSAGE_TIMEOUT*.

If the *GET_MESSAGE_TIMEOUT* expires before a message is received, the internal buffer is written out to the database file.

However if the bus is still saturated with messages, the Listener will continue receiving until the *MAX_TIME_BETWEEN_WRITES* timeout is reached or more than *MAX_BUFFER_SIZE_BEFORE_WRITES* messages are buffered.

Note: The database schema is given in the documentation of the loggers.

Parameters

- **file** (*TextIO* | *BinaryIO* | *GzipFile* | *None*) – a *str* or path like object that points to the database file to use
- **table_name** (*str*) – the name of the table to store messages in
- **kwargs** (*Any*) –

Warning: In contrary to all other readers/writers the Sqlite handlers do not accept file-like objects as the *file* parameter.

GET_MESSAGE_TIMEOUT = 0.25

Number of seconds to wait for messages from internal queue

MAX_BUFFER_SIZE_BEFORE_WRITES = 500

Maximum number of messages to buffer before writing to the database

MAX_TIME_BETWEEN_WRITES = 5.0

Maximum number of seconds to wait between writes to the database

stop()

Stops the reader an writes all remaining messages to the database. Thus, this might take a while and block.

class can.**SqliteReader**(*file*, *table_name='messages'*, ***kwargs*)

Reads recorded CAN messages from a simple SQL database.

This class can be iterated over or used to fetch all messages in the database with *read_all()*.

Calling `len()` on this object might not run in constant time.

Attr str table_name
the name of the database table used for storing the messages

Parameters

- **file** (*TextIO* | *BinaryIO* | *GzipFile* | *None*) –
- **table_name** (*str*) –
- **kwargs** (*Any*) –

Note: The database schema is given in the documentation of the loggers.

Parameters

- **file** (*TextIO* | *BinaryIO* | *GzipFile* | *None*) – a *str* path like object that points to the database file to use

- **table_name** (*str*) – the name of the table to look for the messages
- **kwargs** (*Any*) –

Warning: In contrary to all other readers/writers the Sqlite handlers do not accept file-like objects as the *file* parameter. It also runs in `append=True` mode all the time.

read_all()

Fetches all messages in the database.

Return type
Generator[*can.Message*]

stop()

Closes the connection to the database.

Database table format

The messages are written to the table `messages` in the sqlite database by default. The table is created if it does not already exist.

The entries are as follows:

Name	Data type	Note
ts	REAL	The timestamp of the message
arbitration_id	INTEGER	The arbitration id, might use the extended format
extended	INTEGER	1 if the arbitration id uses the extended format, else `0`
remote	INTEGER	1 if the message is a remote frame, else `0`
error	INTEGER	1 if the message is an error frame, else `0`
dlc	INTEGER	The data length code (DLC)
data	BLOB	The content of the message

3.3.9 ASC (.asc Logging format)

ASCWriter logs CAN data to an ASCII log file compatible with other CAN tools such as Vector CANalyzer/CANoe and other. Since no official specification exists for the format, it has been reverse- engineered from existing log files. One description of the format can be found here.

Note: Channels will be converted to integers.

class can.**ASCWriter**(*file*, *channel=1*, ***kwargs*)

Logs CAN data to an ASCII log file (.asc).

The measurement starts with the timestamp of the first registered message. If a message has a timestamp smaller than the previous one or None, it gets assigned the timestamp that was written for the last message. It the first message does not have a timestamp, it is set to zero.

Parameters

- **file** (*TextIO*) – a path-like object or as file-like object to write to If this is a file-like object, is has to opened in text write mode, not binary write mode.

- **channel** (*int*) – a default channel to use when the message does not have a channel set
- **kwargs** (*Any*) –

log_event(*message*, *timestamp=None*)

Add a message to the log file.

Parameters

- **message** (*str*) – an arbitrary message
- **timestamp** (*float | None*) – the absolute timestamp of the event

Return type
None

on_message_received(*msg*)

This method is called to handle the given message.

Parameters
msg (Message) – the delivered message

Return type
None

stop()

Closes the underlying file-like object and flushes it, if it was opened in write mode.

Return type
None

ASCReader reads CAN data from ASCII log files .asc, as further references can-utils can be used: asc2log, log2asc.

class can.**ASCReader**(*file*, *base='hex'*, *relative_timestamp=True*, ***kwargs*)

Iterator of CAN messages from a ASC logging file. Meta data (comments, bus statistics, J1939 Transport Protocol messages) is ignored.

Parameters

- **file** (*TextIO*) – a path-like object or as file-like object to read from If this is a file-like object, is has to opened in text read mode, not binary read mode.
- **base** (*str*) – Select the base(hex or dec) of id and data. If the header of the asc file contains base information, this value will be overwritten. Default "hex".
- **relative_timestamp** (*bool*) – Select whether the timestamps are *relative* (starting at 0.0) or *absolute* (starting at the system time). Default *True* = *relative*.
- **kwargs** (*Any*) –

3.3.10 Log (.log can-utils Logging format)

CanutilsLogWriter logs CAN data to an ASCII log file compatible with *can-utils <https://github.com/linux-can/can-utils>* As specification following references can-utils can be used: asc2log, log2asc.

class can.**CanutilsLogWriter**(*file*, *channel='vcan0'*, *append=False*, ***kwargs*)

Logs CAN data to an ASCII log file (.log). This class is is compatible with "candump -L".

If a message has a timestamp smaller than the previous one (or 0 or None), it gets assigned the timestamp that was written for the last message. It the first message does not have a timestamp, it is set to zero.

Parameters

- **file** (*TextIO*) – a path-like object or as file-like object to write to If this is a file-like object, is has to opened in text write mode, not binary write mode.
- **channel** (*str*) – a default channel to use when the message does not have a channel set
- **append** (*bool*) – if set to *True* messages are appended to the file, else the file is truncated
- **kwargs** (*Any*) –

on_message_received(*msg*)

This method is called to handle the given message.

Parameters
msg – the delivered message

CanutilsLogReader reads CAN data from ASCII log files .log

class can.**CanutilsLogReader**(*file*, ***kwargs*)

Iterator over CAN messages from a .log Logging File (candump -L).

Note: .log-format looks for example like this:

```
(0.0) vcan0 001#8d00100100820100
```

Parameters

- **file** (*TextIO*) – a path-like object or as file-like object to read from If this is a file-like object, is has to opened in text read mode, not binary read mode.
- **kwargs** (*Any*) –

3.3.11 BLF (Binary Logging Format)

Implements support for BLF (Binary Logging Format) which is a proprietary CAN log format from Vector Informatik GmbH.

The data is stored in a compressed format which makes it very compact.

Note: Channels will be converted to integers.

---

class can.**BLFWriter**(*file*, *append=False*, *channel=1*, *compression_level=-1*, ***kwargs*)

Logs CAN data to a Binary Logging File compatible with Vector's tools.

Parameters

- **file** (*BinaryIO*) – a path-like object or as file-like object to write to If this is a file-like object, is has to opened in mode "wb+".
- **channel** (*int*) – Default channel to log as if not specified by the interface.
- **append** (*bool*) – Append messages to an existing log file.
- **compression_level** (*int*) – An integer from 0 to 9 or -1 controlling the level of compression. 1 (Z_BEST_SPEED) is fastest and produces the least compression. 9 (Z_BEST_COMPRESSION) is slowest and produces the most. 0 means that data will be stored without processing. The default value is -1 (Z_DEFAULT_COMPRESSION). Z_DEFAULT_COMPRESSION represents a default compromise between speed and compression (currently equivalent to level 6).

- **kwargs** (*Any*) –

application_id = 5

Application identifier for the log writer

file_size()

Return an estimate of the current file size in bytes.

Return type
int

log_event(*text*, *timestamp=None*)

Add an arbitrary message to the log file as a global marker.

Parameters

- **text** (*str*) – The group name of the marker.
- **timestamp** (*float*) – Absolute timestamp in Unix timestamp format. If not given, the marker will be placed along the last message.

max_container_size = 131072

Max log container size of uncompressed data

on_message_received(*msg*)

This method is called to handle the given message.

Parameters
msg – the delivered message

stop()

Stops logging and closes the file.

The following class can be used to read messages from BLF file:

class can.**BLFReader**(*file*, ***kwargs*)

Iterator of CAN messages from a Binary Logging File.

Only CAN messages and error frames are supported. Other object types are silently ignored.

Parameters

- **file** (*BinaryIO*) – a path-like object or as file-like object to read from If this is a file-like object, is has to opened in binary read mode, not text read mode.
- **kwargs** (*Any*) –

3.3.12 MF4 (Measurement Data Format v4)

Implements support for MF4 (Measurement Data Format v4) which is a proprietary format from ASAM (Association for Standardization of Automation and Measuring Systems), widely used in many automotive software (Vector CANape, ETAS INCA, dSPACE ControlDesk, etc.).

The data is stored in a compressed format which makes it compact.

Note: MF4 support has to be installed as an extra with for example `pip install python-can[mf4]`.

Note: Channels will be converted to integers.

Note: MF4Writer does not suppport the append mode.

class can.**MF4Writer**(*file*, *database=None*, *compression_level=2*, ***kwargs*)

Logs CAN data to an ASAM Measurement Data File v4 (.mf4).

MF4Writer does not support append mode.

If a message has a timestamp smaller than the previous one or None, it gets assigned the timestamp that was written for the last message. It the first message does not have a timestamp, it is set to zero.

Parameters

- **file** (*BinaryIO | GzipFile*) – A path-like object or as file-like object to write to. If this is a file-like object, is has to be opened in binary write mode, not text write mode.
- **database** (*str | os.PathLike[str] | None*) – optional path to a DBC or ARXML file that contains message description.
- **compression_level** (*int*) – compression option as integer (default 2) * 0 - no compression * 1 - deflate (slower, but produces smaller files) * 2 - transposition + deflate (slowest, but produces the smallest files)
- **kwargs** (*Any*) –

file_size()

Return an estimate of the current file size in bytes.

Return type
int

on_message_received(*msg*)

This method is called to handle the given message.

Parameters
msg (Message) – the delivered message

Return type
None

stop()

Closes the underlying file-like object and flushes it, if it was opened in write mode.

Return type
None

The MDF format is very flexible regarding the internal structure and it is used to handle data from multiple sources, not just CAN bus logging. MDF4Writer will always create a fixed internal file structure where there will be three channel groups (for standard, error and remote frames). Using this fixed file structure allows for a simple implementation of MDF4Writer and MF4Reader classes. Therefor MF4Reader can only replay files created with MF4Writer.

The following class can be used to read messages from MF4 file:

class can.**MF4Reader**(*file*, ***kwargs*)

Iterator of CAN messages from a MF4 logging file.

The MF4Reader only supports MF4 files that were recorded with python-can.

Parameters

- **file** (*BinaryIO | GzipFile*) – a path-like object or as file-like object to read from If this is a file-like object, is has to be opened in binary read mode, not text read mode.

- **kwargs** (*Any*) –

stop()

Closes the underlying file-like object and flushes it, if it was opened in write mode.

Return type
None

3.3.13 TRC

Implements basic support for the TRC file format.

Note: Comments and contributions are welcome on what file versions might be relevant.

class can.**TRCWriter**(*file*, *channel=1*, ***kwargs*)

Logs CAN data to text file (.trc).

The measurement starts with the timestamp of the first registered message. If a message has a timestamp smaller than the previous one or None, it gets assigned the timestamp that was written for the last message. If the first message does not have a timestamp, it is set to zero.

Parameters

- **file** (*TextIO*) – a path-like object or as file-like object to write to If this is a file-like object, is has to opened in text write mode, not binary write mode.
- **channel** (*int*) – a default channel to use when the message does not have a channel set
- **kwargs** (*Any*) –

on_message_received(*msg*)

This method is called to handle the given message.

Parameters
msg (Message) – the delivered message

Return type
None

The following class can be used to read messages from TRC file:

class can.**TRCReader**(*file*, ***kwargs*)

Iterator of CAN messages from a TRC logging file.

Parameters

- **file** (*TextIO*) – a path-like object or as file-like object to read from If this is a file-like object, is has to opened in text read mode, not binary read mode.
- **kwargs** (*Any*) –

3.4 Asyncio support

The `asyncio` module built into Python 3.4 and later can be used to write asynchronous code in a single thread. This library supports receiving messages asynchronously in an event loop using the *can.Notifier* class.

There will still be one thread per CAN bus but the user application will execute entirely in the event loop, allowing simpler concurrency without worrying about threading issues. Interfaces that have a valid file descriptor will however be supported natively without a thread.

You can also use the *can.AsyncBufferedReader* listener if you prefer to write coroutine based code instead of using callbacks.

3.4.1 Example

Here is an example using both callback and coroutine based code:

```
#!/usr/bin/env python

"""
This example demonstrates how to use async IO with python-can.
"""

import asyncio
from typing import List

import can
from can.notifier import MessageRecipient


def print_message(msg: can.Message) -> None:
    """Regular callback function. Can also be a coroutine."""
    print(msg)


async def main() -> None:
    """The main function that runs in the loop."""

    with can.Bus(
        interface="virtual", channel="my_channel_0", receive_own_messages=True
    ) as bus:
        reader = can.AsyncBufferedReader()
        logger = can.Logger("logfile.asc")

        listeners: List[MessageRecipient] = [
            print_message,  # Callback function
            reader,  # AsyncBufferedReader() listener
            logger,  # Regular Listener object
        ]
        # Create Notifier with an explicit loop to use for scheduling of callbacks
        loop = asyncio.get_running_loop()
        notifier = can.Notifier(bus, listeners, loop=loop)
        # Start sending first message
        bus.send(can.Message(arbitration_id=0))
```

(continues on next page)

(continued from previous page)

```
        print("Bouncing 10 messages...")
        for _ in range(10):
            # Wait for next message from AsyncBufferedReader
            msg = await reader.get_message()
            # Delay response
            await asyncio.sleep(0.5)
            msg.arbitration_id += 1
            bus.send(msg)

        # Wait for last message to arrive
        await reader.get_message()
        print("Done!")

        # Clean-up
        notifier.stop()


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

3.5 Broadcast Manager

The broadcast manager allows the user to setup periodic message jobs. For example sending a particular message at a given period. The broadcast manager supported natively by several interfaces and a software thread based scheduler is used as a fallback.

This example shows the socketcan backend using the broadcast manager:

```
#!/usr/bin/env python

"""
This example exercises the periodic sending capabilities.

Expects a vcan0 interface:

    python3 -m examples.cyclic

"""

import logging
import time

import can

logging.basicConfig(level=logging.INFO)


def simple_periodic_send(bus):
    """
    Sends a message every 20ms with no explicit timeout
```

(continues on next page)

(continued from previous page)

```
    Sleeps for 2 seconds then stops the task.
    """
    print("Starting to send a message every 200ms for 2s")
    msg = can.Message(
        arbitration_id=0x123, data=[1, 2, 3, 4, 5, 6], is_extended_id=False
    )
    task = bus.send_periodic(msg, 0.20)
    assert isinstance(task, can.CyclicSendTaskABC)
    time.sleep(2)
    task.stop()
    print("stopped cyclic send")


def limited_periodic_send(bus):
    """Send using LimitedDurationCyclicSendTaskABC."""
    print("Starting to send a message every 200ms for 1s")
    msg = can.Message(
        arbitration_id=0x12345678, data=[0, 0, 0, 0, 0, 0], is_extended_id=True
    )
    task = bus.send_periodic(msg, 0.20, 1, store_task=False)
    if not isinstance(task, can.LimitedDurationCyclicSendTaskABC):
        print("This interface doesn't seem to support LimitedDurationCyclicSendTaskABC")
        task.stop()
        return

    time.sleep(2)
    print("Cyclic send should have stopped as duration expired")
    # Note the (finished) task will still be tracked by the Bus
    # unless we pass `store_task=False` to bus.send_periodic
    # alternatively calling stop removes the task from the bus
    # task.stop()


def test_periodic_send_with_modifying_data(bus):
    """Send using ModifiableCyclicTaskABC."""
    print("Starting to send a message every 200ms. Initial data is four consecutive 1s")
    msg = can.Message(arbitration_id=0x0CF02200, data=[1, 1, 1, 1])
    task = bus.send_periodic(msg, 0.20)
    if not isinstance(task, can.ModifiableCyclicTaskABC):
        print("This interface doesn't seem to support modification")
        task.stop()
        return
    time.sleep(2)
    print("Changing data of running task to begin with 99")
    msg.data[0] = 0x99
    task.modify_data(msg)
    time.sleep(2)

    task.stop()
    print("stopped cyclic send")
    print("Changing data of stopped task to single ff byte")
    msg.data = bytearray([0xFF])
```

(continues on next page)

(continued from previous page)

```
    msg.dlc = 1
    task.modify_data(msg)
    time.sleep(1)
    print("starting again")
    task.start()
    time.sleep(1)
    task.stop()
    print("done")


# Will have to consider how to expose items like this. The socketcan
# interfaces will continue to support it... but the top level api won't.
# def test_dual_rate_periodic_send():
#     """Send a message 10 times at 1ms intervals, then continue to send every 500ms"""
#     msg = can.Message(arbitration_id=0x123, data=[0, 1, 2, 3, 4, 5])
#     print("Creating cyclic task to send message 10 times at 1ms, then every 500ms")
#     task = can.interface.MultiRateCyclicSendTask('vcan0', msg, 10, 0.001, 0.50)
#     time.sleep(2)
#
#     print("Changing data[0] = 0x42")
#     msg.data[0] = 0x42
#     task.modify_data(msg)
#     time.sleep(2)
#
#     task.stop()
#     print("stopped cyclic send")
#
#     time.sleep(2)
#
#     task.start()
#     print("starting again")
#     time.sleep(2)
#     task.stop()
#     print("done")


def main():
    """Test different cyclic sending tasks."""
    reset_msg = can.Message(
        arbitration_id=0x00, data=[0, 0, 0, 0, 0, 0], is_extended_id=False
    )

    # this uses the default configuration (for example from environment variables, or a
    # config file) see https://python-can.readthedocs.io/en/stable/configuration.html
    with can.Bus() as bus:
        bus.send(reset_msg)

        simple_periodic_send(bus)

        bus.send(reset_msg)

        limited_periodic_send(bus)
```

(continues on next page)

(continued from previous page)

```
        test_periodic_send_with_modifying_data(bus)

        # print("Carrying out multirate cyclic test for {} interface".format(interface))
        # can.rc['interface'] = interface
        # test_dual_rate_periodic_send()

    time.sleep(2)


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

3.5.1 Message Sending Tasks

The class based api for the broadcast manager uses a series of mixin classes. All mixins inherit from *CyclicSendTaskABC* which inherits from *CyclicTask*.

class can.broadcastmanager.**CyclicTask**

Abstract Base for all cyclic tasks.

abstract stop()

Cancel this periodic task.

Raises
CanError – If stop is called on an already stopped task.

Return type
None

class can.broadcastmanager.**CyclicSendTaskABC**(*messages*, *period*)

Message send task with defined period

Parameters

- **messages** (*Sequence[Message] | Message*) – The messages to be sent periodically.
- **period** (*float*) – The rate in seconds at which to send the messages.

Raises
ValueError – If the given messages are invalid

class can.broadcastmanager.**LimitedDurationCyclicSendTaskABC**(*messages*, *period*, *duration*)

Message send task with a defined duration and period.

Parameters

- **messages** (*Sequence[Message] | Message*) – The messages to be sent periodically.
- **period** (*float*) – The rate in seconds at which to send the messages.
- **duration** (*float | None*) – Approximate duration in seconds to continue sending messages. If no duration is provided, the task will continue indefinitely.

Raises
ValueError – If the given messages are invalid

class can.broadcastmanager.**MultiRateCyclicSendTaskABC**(*channel*, *messages*, *count*, *initial_period*, *subsequent_period*)

A Cyclic send task that supports switches send frequency after a set time.

Transmits a message *count* times at *initial_period* then continues to transmit messages at *subsequent_period*.

Parameters

- **channel** (*int | str*) – See interface specific documentation.
- **messages** (*Sequence[Message] | Message*) –
- **count** (*int*) –
- **initial_period** (*float*) –
- **subsequent_period** (*float*) –

Raises
ValueError – If the given messages are invalid

class can.**ModifiableCyclicTaskABC**(*messages*, *period*)

Parameters

- **messages** (*Sequence[Message] | Message*) – The messages to be sent periodically.
- **period** (*float*) – The rate in seconds at which to send the messages.

Raises
ValueError – If the given messages are invalid

modify_data(*messages*)

Update the contents of the periodically sent messages, without altering the timing.

Parameters
messages (*Sequence[Message] | Message*) – The messages with the new *Message.data*.

Note: The arbitration ID cannot be changed.

Note: The number of new cyclic messages to be sent must be equal to the original number of messages originally specified for this task.

Raises
ValueError – If the given messages are invalid

Return type
None

class can.**RestartableCyclicTaskABC**(*messages*, *period*)

Adds support for restarting a stopped cyclic task

Parameters

- **messages** (*Sequence[Message] | Message*) – The messages to be sent periodically.
- **period** (*float*) – The rate in seconds at which to send the messages.

Raises
ValueError – If the given messages are invalid

abstract start()

Restart a stopped periodic task.

Return type
None

class can.broadcastmanager.**ThreadBasedCyclicSendTask**(*bus*, *lock*, *messages*, *period*, *duration=None*, *on_error=None*, *modifier_callback=None*)

Fallback cyclic send task using daemon thread.

Transmits *messages* with a *period* seconds for *duration* seconds on a *bus*.

The *on_error* is called if any error happens on *bus* while sending *messages*. If *on_error* present, and returns `False` when invoked, thread is stopped immediately, otherwise, thread continuously tries to send *messages* ignoring errors on a *bus*. Absence of *on_error* means that thread exits immediately on error.

Parameters

- **on_error** (*Callable[[Exception], bool] | None*) – The callable that accepts an exception if any error happened on a *bus* while sending *messages*, it shall return either `True` or `False` depending on desired behaviour of *ThreadBasedCyclicSendTask*.
- **bus** (BusABC) –
- **lock** (*allocate_lock*) –
- **messages** (*Sequence[Message] | Message*) –
- **period** (*float*) –
- **duration** (*float | None*) –
- **modifier_callback** (*Callable[[Message], None] | None*) –

Raises
ValueError – If the given messages are invalid

start()

Restart a stopped periodic task.

Return type
None

stop()

Cancel this periodic task.

Raises
CanError – If stop is called on an already stopped task.

Return type
None

3.6 Error Handling

There are several specific `Exception` classes to allow user code to react to specific scenarios related to CAN busses:

```
Exception (Python standard library)
 +-- ...
 +-- CanError (python-can)
     +-- CanInterfaceNotImplementedError
     +-- CanInitializationError
     +-- CanOperationError
     +-- CanTimeoutError
```

Keep in mind that some functions and methods may raise different exceptions. For example, validating typical arguments and parameters might result in a `ValueError`. This should always be documented for the function at hand.

exception can.exceptions.**CanError**(*message=''*, *error_code=None*)

Bases: `Exception`

Base class for all CAN related exceptions.

If specified, the error code is automatically appended to the message:

```
>>> # With an error code (it also works with a specific error):
>>> error = CanOperationError(message="Failed to do the thing", error_code=42)
>>> str(error)
'Failed to do the thing [Error Code 42]'
>>>
>>> # Missing the error code:
>>> plain_error = CanError(message="Something went wrong ...")
>>> str(plain_error)
'Something went wrong ...'
```

Parameters

- **error_code** (*int | None*) – An optional error code to narrow down the cause of the fault
- **error_code** – An optional error code to narrow down the cause of the fault
- **message** (*str*) –

Return type
None

exception can.exceptions.**CanInitializationError**(*message=''*, *error_code=None*)

Bases: *CanError*

Indicates an error the occurred while initializing a *can.BusABC*.

If initialization fails due to a driver or platform missing/being unsupported, a *CanInterfaceNotImplementedError* is raised instead. If initialization fails due to a value being out of range, a `ValueError` is raised.

Example scenarios:

- Try to open a non-existent device and/or channel
- Try to use an invalid setting, which is ok by value, but not ok for the interface

- The device or other resources are already used

Parameters

- **message** (*str*) –
- **error_code** (*int | None*) –

Return type
None

exception can.exceptions.**CanInterfaceNotImplementedError**(*message=''*, *error_code=None*)

Bases: *CanError*, NotImplementedError

Indicates that the interface is not supported on the current platform.

Example scenarios:

- No interface with that name exists
- The interface is unsupported on the current operating system or interpreter
- The driver could not be found or has the wrong version

Parameters

- **message** (*str*) –
- **error_code** (*int | None*) –

Return type
None

exception can.exceptions.**CanOperationError**(*message=''*, *error_code=None*)

Bases: *CanError*

Indicates an error while in operation.

Example scenarios:

- A call to a library function results in an unexpected return value
- An invalid message was received
- The driver rejected a message that was meant to be sent
- Cyclic redundancy check (CRC) failed
- A message remained unacknowledged
- A buffer is full

Parameters

- **message** (*str*) –
- **error_code** (*int | None*) –

Return type
None

exception can.exceptions.**CanTimeoutError**(*message=''*, *error_code=None*)

Bases: *CanError*, TimeoutError

Indicates the timeout of an operation.

Example scenarios:

- Some message could not be sent after the timeout elapsed
- No message was read within the given time

Parameters

- **message** (*str*) –
- **error_code** (*int | None*) –

Return type
None

can.exceptions.**error_check**(*error_message=None*, *exception_type=<class 'can.exceptions.CanOperationError'>*)

Catches any exceptions and turns them into the new type while preserving the stack trace.

Parameters

- **error_message** (*str | None*) –
- **exception_type** (*Type[CanError]*) –

Return type
Generator[None, None, None]

3.7 Bit Timing Configuration

Attention: This feature is experimental. The implementation might change in future versions.

The CAN protocol, specified in ISO 11898, allows the bitrate, sample point and number of samples to be optimized for a given application. These parameters, known as bit timings, can be adjusted to meet the requirements of the communication system and the physical communication channel.

These parameters include:

- **tseg1**: The time segment 1 (TSEG1) is the amount of time from the end of the sync segment until the sample point. It is expressed in time quanta (TQ).
- **tseg2**: The time segment 2 (TSEG2) is the amount of time from the sample point until the end of the bit. It is expressed in TQ.
- **sjw**: The synchronization jump width (SJW) is the maximum number of TQ that the controller can resynchronize every bit.
- **sample point**: The sample point is defined as the point in time within a bit where the bus controller samples the bus for dominant or recessive levels. It is typically expressed as a percentage of the bit time. The sample point depends on the bus length and propagation time as well as the information processing time of the nodes.

Fig. 1: Bit Timing and Sample Point

For example, consider a bit with a total duration of 8 TQ and a sample point at 75%. The values for TSEG1, TSEG2 and SJW would be 5, 2, and 2, respectively. The sample point would be 6 TQ after the start of the bit, leaving 2 TQ for the information processing by the bus nodes.

Note: The values for TSEG1, TSEG2 and SJW are chosen such that the sample point is at least 50% of the total bit time. This ensures that there is sufficient time for the signal to stabilize before it is sampled.

Note: In CAN FD, the arbitration (nominal) phase and the data phase can have different bit rates. As a result, there are two separate sample points to consider.

Another important parameter is **f_clock**: The CAN system clock frequency in Hz. This frequency is used to derive the TQ size from the bit rate. The relationship is `f_clock = (tseg1+tseg2+1) * bitrate * brp`. The bit rate prescaler value **brp** is usually determined by the controller and is chosen to ensure that the resulting bit time is an integer value. Typical CAN clock frequencies are 8-80 MHz.

In most cases, the recommended settings for a predefined set of common bit rates will work just fine. In some cases, however, it may be necessary to specify custom bit timings. The *BitTiming* and *BitTimingFd* classes can be used for this purpose to specify bit timings in a relatively interface agnostic manner.

BitTiming or *BitTimingFd* can also help you to produce an overview of possible bit timings for your desired bit rate:

```
>>> import contextlib
>>> import can
...
>>> timings = set()
>>> for sample_point in range(50, 100):
...     with contextlib.suppress(ValueError):
...         timings.add(
...             can.BitTiming.from_sample_point(
...                 f_clock=8_000_000,
...                 bitrate=250_000,
...                 sample_point=sample_point,
...             )
...         )
...
>>> for timing in sorted(timings, key=lambda x: x.sample_point):
...     print(timing)
BR: 250_000 bit/s, SP: 50.00%, BRP: 2, TSEG1: 7, TSEG2: 8, SJW: 4, BTR: C176h, CLK: 8MHz
BR: 250_000 bit/s, SP: 56.25%, BRP: 2, TSEG1: 8, TSEG2: 7, SJW: 4, BTR: C167h, CLK: 8MHz
BR: 250_000 bit/s, SP: 62.50%, BRP: 2, TSEG1: 9, TSEG2: 6, SJW: 4, BTR: C158h, CLK: 8MHz
BR: 250_000 bit/s, SP: 68.75%, BRP: 2, TSEG1: 10, TSEG2: 5, SJW: 4, BTR: C149h, CLK: 8MHz
BR: 250_000 bit/s, SP: 75.00%, BRP: 2, TSEG1: 11, TSEG2: 4, SJW: 4, BTR: C13Ah, CLK: 8MHz
BR: 250_000 bit/s, SP: 81.25%, BRP: 2, TSEG1: 12, TSEG2: 3, SJW: 3, BTR: 812Bh, CLK: 8MHz
BR: 250_000 bit/s, SP: 87.50%, BRP: 2, TSEG1: 13, TSEG2: 2, SJW: 2, BTR: 411Ch, CLK: 8MHz
BR: 250_000 bit/s, SP: 93.75%, BRP: 2, TSEG1: 14, TSEG2: 1, SJW: 1, BTR: 010Dh, CLK: 8MHz
```

It is possible to specify CAN 2.0 bit timings using the config file:

```
[default]
f_clock=8000000
brp=1
tseg1=5
tseg2=2
sjw=1
nof_samples=1
```

The same is possible for CAN FD:

```
[default]
f_clock=80000000
nom_brp=1
nom_tseg1=119
nom_tseg2=40
nom_sjw=40
data_brp=1
data_tseg1=29
data_tseg2=10
data_sjw=10
```

A `dict` of the relevant config parameters can be easily obtained by calling `dict(timing)` or `{**timing}` where `timing` is the *BitTiming* or *BitTimingFd* instance.

Check *Configuration* for more information about saving and loading configurations.

class can.**BitTiming**(*f_clock*, *brp*, *tseg1*, *tseg2*, *sjw*, *nof_samples=1*, *strict=False*)

Bases: `Mapping`

Representation of a bit timing configuration for a CAN 2.0 bus.

The class can be constructed in multiple ways, depending on the information available. The preferred way is using CAN clock frequency, prescaler, tseg1, tseg2 and sjw:

```
can.BitTiming(f_clock=8_000_000, brp=1, tseg1=5, tseg2=1, sjw=1)
```

Alternatively you can set the bitrate instead of the bit rate prescaler:

```
can.BitTiming.from_bitrate_and_segments(
    f_clock=8_000_000, bitrate=1_000_000, tseg1=5, tseg2=1, sjw=1
)
```

It is also possible to specify BTR registers:

```
can.BitTiming.from_registers(f_clock=8_000_000, btr0=0x00, btr1=0x14)
```

or to calculate the timings for a given sample point:

```
can.BitTiming.from_sample_point(f_clock=8_000_000, bitrate=1_000_000, sample_
↪point=75.0)
```

Parameters

- **f_clock** (*int*) – The CAN system clock frequency in Hz.
- **brp** (*int*) – Bit rate prescaler.

- **tseg1** (*int*) – Time segment 1, that is, the number of quanta from (but not including) the Sync Segment to the sampling point.
- **tseg2** (*int*) – Time segment 2, that is, the number of quanta from the sampling point to the end of the bit.
- **sjw** (*int*) – The Synchronization Jump Width. Decides the maximum number of time quanta that the controller can resynchronize every bit.
- **nof_samples** (*int*) – Either 1 or 3. Some CAN controllers can also sample each bit three times. In this case, the bit will be sampled three quanta in a row, with the last sample being taken in the edge between TSEG1 and TSEG2. Three samples should only be used for relatively slow baudrates.
- **strict** (*bool*) – If True, restrict bit timings to the minimum required range as defined in ISO 11898. This can be used to ensure compatibility across a wide variety of CAN hardware.

Raises
ValueError – if the arguments are invalid.

classmethod from_bitrate_and_segments(*f_clock*, *bitrate*, *tseg1*, *tseg2*, *sjw*, *nof_samples=1*, *strict=False*)

Create a *BitTiming* instance from bitrate and segment lengths.

Parameters

- **f_clock** (*int*) – The CAN system clock frequency in Hz.
- **bitrate** (*int*) – Bitrate in bit/s.
- **tseg1** (*int*) – Time segment 1, that is, the number of quanta from (but not including) the Sync Segment to the sampling point.
- **tseg2** (*int*) – Time segment 2, that is, the number of quanta from the sampling point to the end of the bit.
- **sjw** (*int*) – The Synchronization Jump Width. Decides the maximum number of time quanta that the controller can resynchronize every bit.
- **nof_samples** (*int*) – Either 1 or 3. Some CAN controllers can also sample each bit three times. In this case, the bit will be sampled three quanta in a row, with the last sample being taken in the edge between TSEG1 and TSEG2. Three samples should only be used for relatively slow baudrates.
- **strict** (*bool*) – If True, restrict bit timings to the minimum required range as defined in ISO 11898. This can be used to ensure compatibility across a wide variety of CAN hardware.

Raises
ValueError – if the arguments are invalid.

Return type
BitTiming

classmethod from_registers(*f_clock*, *btr0*, *btr1*)

Create a *BitTiming* instance from registers btr0 and btr1.

Parameters

- **f_clock** (*int*) – The CAN system clock frequency in Hz.
- **btr0** (*int*) – The BTR0 register value used by many CAN controllers.

- **btr1** (*int*) – The BTR1 register value used by many CAN controllers.

Raises
ValueError – if the arguments are invalid.

Return type
BitTiming

classmethod iterate_from_sample_point(*f_clock*, *bitrate*, *sample_point=69.0*)

Create a *BitTiming* iterator with all the solutions for a sample point.

Parameters

- **f_clock** (*int*) – The CAN system clock frequency in Hz.
- **bitrate** (*int*) – Bitrate in bit/s.
- **sample_point** (*int*) – The sample point value in percent.

Raises
ValueError – if the arguments are invalid.

Return type
Iterator[BitTiming]

classmethod from_sample_point(*f_clock*, *bitrate*, *sample_point=69.0*)

Create a *BitTiming* instance for a sample point.

This function tries to find bit timings, which are close to the requested sample point. It does not take physical bus properties into account, so the calculated bus timings might not work properly for you.

The *oscillator_tolerance()* function might be helpful to evaluate the bus timings.

Parameters

- **f_clock** (*int*) – The CAN system clock frequency in Hz.
- **bitrate** (*int*) – Bitrate in bit/s.
- **sample_point** (*int*) – The sample point value in percent.

Raises
ValueError – if the arguments are invalid.

Return type
BitTiming

property f_clock: int

The CAN system clock frequency in Hz.

property bitrate: int

Bitrate in bits/s.

property brp: int

Bit Rate Prescaler.

property tq: int

Time quantum in nanoseconds

property nbt: int

Nominal Bit Time.

property tseg1: int

Time segment 1.

The number of quanta from (but not including) the Sync Segment to the sampling point.

property tseg2: int

Time segment 2.

The number of quanta from the sampling point to the end of the bit.

property sjw: int

Synchronization Jump Width.

property nof_samples: int

Number of samples (1 or 3).

property sample_point: float

Sample point in percent.

property btr0: int

Bit timing register 0 for SJA1000.

property btr1: int

Bit timing register 1 for SJA1000.

oscillator_tolerance(*node_loop_delay_ns=250.0*, *bus_length_m=10.0*)

Oscillator tolerance in percent according to ISO 11898-1.

Parameters

- **node_loop_delay_ns** (*float*) – Transceiver loop delay in nanoseconds.
- **bus_length_m** (*float*) – Bus length in meters.

Return type
float

recreate_with_f_clock(*f_clock*)

Return a new *BitTiming* instance with the given *f_clock* but the same bit rate and sample point.

Parameters
f_clock (*int*) – The CAN system clock frequency in Hz.

Raises
ValueError – if no suitable bit timings were found.

Return type
BitTiming

class can.**BitTimingFd**(*f_clock*, *nom_brp*, *nom_tseg1*, *nom_tseg2*, *nom_sjw*, *data_brp*, *data_tseg1*, *data_tseg2*, *data_sjw*, *strict=False*)

Bases: Mapping

Representation of a bit timing configuration for a CAN FD bus.

The class can be constructed in multiple ways, depending on the information available. The preferred way is using CAN clock frequency, bit rate prescaler, tseg1, tseg2 and sjw for both the arbitration (nominal) and data phase:

```
can.BitTimingFd(
    f_clock=80_000_000,
    nom_brp=1,
    nom_tseg1=59,
    nom_tseg2=20,
    nom_sjw=10,
    data_brp=1,
    data_tseg1=6,
    data_tseg2=3,
    data_sjw=2,
)
```

Alternatively you can set the bit rates instead of the bit rate prescalers:

```
can.BitTimingFd.from_bitrate_and_segments(
    f_clock=80_000_000,
    nom_bitrate=1_000_000,
    nom_tseg1=59,
    nom_tseg2=20,
    nom_sjw=10,
    data_bitrate=8_000_000,
    data_tseg1=6,
    data_tseg2=3,
    data_sjw=2,
)
```

It is also possible to calculate the timings for a given pair of arbitration and data sample points:

```
can.BitTimingFd.from_sample_point(
    f_clock=80_000_000,
    nom_bitrate=1_000_000,
    nom_sample_point=75.0,
    data_bitrate=8_000_000,
    data_sample_point=70.0,
)
```

Initialize a BitTimingFd instance with the specified parameters.

Parameters

- **f_clock** (*int*) – The CAN system clock frequency in Hz.
- **nom_brp** (*int*) – Nominal (arbitration) phase bitrate prescaler.
- **nom_tseg1** (*int*) – Nominal phase Time segment 1, that is, the number of quanta from (but not including) the Sync Segment to the sampling point.
- **nom_tseg2** (*int*) – Nominal phase Time segment 2, that is, the number of quanta from the sampling point to the end of the bit.
- **nom_sjw** (*int*) – The Synchronization Jump Width for the nominal phase. This value determines the maximum number of time quanta that the controller can resynchronize every bit.
- **data_brp** (*int*) – Data phase bitrate prescaler.
- **data_tseg1** (*int*) – Data phase Time segment 1, that is, the number of quanta from (but not including) the Sync Segment to the sampling point.

- **data_tseg2** (*int*) – Data phase Time segment 2, that is, the number of quanta from the sampling point to the end of the bit.
- **data_sjw** (*int*) – The Synchronization Jump Width for the data phase. This value determines the maximum number of time quanta that the controller can resynchronize every bit.
- **strict** (*bool*) – If True, restrict bit timings to the minimum required range as defined in ISO 11898. This can be used to ensure compatibility across a wide variety of CAN hardware.

Raises
ValueError – if the arguments are invalid.

classmethod from_bitrate_and_segments(*f_clock*, *nom_bitrate*, *nom_tseg1*, *nom_tseg2*, *nom_sjw*, *data_bitrate*, *data_tseg1*, *data_tseg2*, *data_sjw*, *strict=False*)

Create a *BitTimingFd* instance with the bitrates and segments lengths.

Parameters

- **f_clock** (*int*) – The CAN system clock frequency in Hz.
- **nom_bitrate** (*int*) – Nominal (arbitration) phase bitrate in bit/s.
- **nom_tseg1** (*int*) – Nominal phase Time segment 1, that is, the number of quanta from (but not including) the Sync Segment to the sampling point.
- **nom_tseg2** (*int*) – Nominal phase Time segment 2, that is, the number of quanta from the sampling point to the end of the bit.
- **nom_sjw** (*int*) – The Synchronization Jump Width for the nominal phase. This value determines the maximum number of time quanta that the controller can resynchronize every bit.
- **data_bitrate** (*int*) – Data phase bitrate in bit/s.
- **data_tseg1** (*int*) – Data phase Time segment 1, that is, the number of quanta from (but not including) the Sync Segment to the sampling point.
- **data_tseg2** (*int*) – Data phase Time segment 2, that is, the number of quanta from the sampling point to the end of the bit.
- **data_sjw** (*int*) – The Synchronization Jump Width for the data phase. This value determines the maximum number of time quanta that the controller can resynchronize every bit.
- **strict** (*bool*) – If True, restrict bit timings to the minimum required range as defined in ISO 11898. This can be used to ensure compatibility across a wide variety of CAN hardware.

Raises
ValueError – if the arguments are invalid.

Return type
BitTimingFd

classmethod iterate_from_sample_point(*f_clock*, *nom_bitrate*, *nom_sample_point*, *data_bitrate*, *data_sample_point*)

Create an *BitTimingFd* iterator with all the solutions for a sample point.

Parameters

- **f_clock** (*int*) – The CAN system clock frequency in Hz.
- **nom_bitrate** (*int*) – Nominal bitrate in bit/s.

- **nom_sample_point** (*int*) – The sample point value of the arbitration phase in percent.
- **data_bitrate** (*int*) – Data bitrate in bit/s.
- **data_sample_point** (*int*) – The sample point value of the data phase in percent.

Raises
ValueError – if the arguments are invalid.

Return type
Iterator[BitTimingFd]

classmethod from_sample_point(*f_clock*, *nom_bitrate*, *nom_sample_point*, *data_bitrate*, *data_sample_point*)

Create a *BitTimingFd* instance for a sample point.

This function tries to find bit timings, which are close to the requested sample points. It does not take physical bus properties into account, so the calculated bus timings might not work properly for you.

The *oscillator_tolerance()* function might be helpful to evaluate the bus timings.

Parameters

- **f_clock** (*int*) – The CAN system clock frequency in Hz.
- **nom_bitrate** (*int*) – Nominal bitrate in bit/s.
- **nom_sample_point** (*int*) – The sample point value of the arbitration phase in percent.
- **data_bitrate** (*int*) – Data bitrate in bit/s.
- **data_sample_point** (*int*) – The sample point value of the data phase in percent.

Raises
ValueError – if the arguments are invalid.

Return type
BitTimingFd

property f_clock: int

The CAN system clock frequency in Hz.

property nom_bitrate: int

Nominal (arbitration phase) bitrate.

property nom_brp: int

Prescaler value for the arbitration phase.

property nom_tq: int

Nominal time quantum in nanoseconds

property nbt: int

Number of time quanta in a bit of the arbitration phase.

property nom_tseg1: int

Time segment 1 value of the arbitration phase.

This is the sum of the propagation time segment and the phase buffer segment 1.

property nom_tseg2: int

Time segment 2 value of the arbitration phase. Also known as phase buffer segment 2.

property nom_sjw: int

Synchronization jump width of the arbitration phase.

The phase buffer segments may be shortened or lengthened by this value.

property nom_sample_point: float

Sample point of the arbitration phase in percent.

property data_bitrate: int

Bitrate of the data phase in bit/s.

property data_brp: int

Prescaler value for the data phase.

property data_tq: int

Data time quantum in nanoseconds

property dbt: int

Number of time quanta in a bit of the data phase.

property data_tseg1: int

TSEG1 value of the data phase.

This is the sum of the propagation time segment and the phase buffer segment 1.

property data_tseg2: int

TSEG2 value of the data phase. Also known as phase buffer segment 2.

property data_sjw: int

Synchronization jump width of the data phase.

The phase buffer segments may be shortened or lengthened by this value.

property data_sample_point: float

Sample point of the data phase in percent.

oscillator_tolerance(*node_loop_delay_ns=250.0*, *bus_length_m=10.0*)

Oscillator tolerance in percent according to ISO 11898-1.

Parameters

- **node_loop_delay_ns** (*float*) – Transceiver loop delay in nanoseconds.
- **bus_length_m** (*float*) – Bus length in meters.

Return type
float

recreate_with_f_clock(*f_clock*)

Return a new *BitTimingFd* instance with the given *f_clock* but the same bit rates and sample points.

Parameters
f_clock (*int*) – The CAN system clock frequency in Hz.

Raises
ValueError – if no suitable bit timings were found.

Return type
BitTimingFd

3.8 Utilities

can.**detect_available_configs**(*interfaces=None*)

Detect all configurations/channels that the interfaces could currently connect with.

This might be quite time-consuming.

Automated configuration detection may not be implemented by every interface on every platform. This method will not raise an error in that case, but with rather return an empty list for that interface.

Parameters
interfaces (*None | str | Iterable[str]*) – either - the name of an interface to be searched in as a string, - an iterable of interface names to search in, or - *None* to search in all known interfaces.

Return type
list[dict]

Returns
an iterable of dicts, each suitable for usage in the constructor of *can.BusABC*.

3.9 Internal API

Here we document the odds and ends that are more helpful for creating your own interfaces or listeners but generally shouldn't be required to interact with python-can.

3.9.1 BusABC

The *BusABC* class, as the name suggests, provides an abstraction of a CAN bus. The bus provides a wrapper around a physical or virtual CAN Bus.

An interface specific instance of the *BusABC* is created by the *Bus* class, see *Bus* for the user facing API.

3.9.2 Extending the BusABC class

Concrete implementations must implement the following:

- *send()* to send individual messages
- *_recv_internal()* to receive individual messages (see note below!)
- set the *channel_info* attribute to a string describing the underlying bus and/or channel

They might implement the following:

- *flush_tx_buffer()* to allow discarding any messages yet to be sent
- *shutdown()* to override how the bus should shut down
- *_send_periodic_internal()* to override the software based periodic sending and push it down to the kernel or hardware.
- *_apply_filters()* to apply efficient filters to lower level systems like the OS kernel or hardware.
- *_detect_available_configs()* to allow the interface to report which configurations are currently available for new connections.

- *state()* property to allow reading and/or changing the bus state.

Note: *TL;DR*: Only override *_recv_internal()*, never *recv()* directly.

Previously, concrete bus classes had to override *recv()* directly instead of *_recv_internal()*, but that has changed to allow the abstract base class to handle in-software message filtering as a fallback. All internal interfaces now implement that new behaviour. Older (custom) interfaces might still be implemented like that and thus might not provide message filtering:

Concrete instances are usually created by *can.Bus()* which takes the users configuration into account.

Bus Internals

Several methods are not documented in the main *can.BusABC* as they are primarily useful for library developers as opposed to library users.

abstract BusABC.**__init__**(*channel*, *can_filters=None*, ***kwargs*)

Construct and open a CAN bus instance of the specified type.

Subclasses should call though this method with all given parameters as it handles generic tasks like applying filters.

Parameters

- **channel** (*Any*) – The can interface identifier. Expected type is backend dependent.
- **can_filters** (*Sequence[CanFilter | CanFilterExtended] | None*) – See *set_filters()* for details.
- **kwargs** (*dict*) – Any backend dependent configurations are passed in this dictionary

Raises

- **ValueError** – If parameters are out of range
- ***CanInterfaceNotImplementedError*** – If the driver cannot be accessed
- ***CanInitializationError*** – If the bus cannot be initialized

BusABC.**__iter__**()

Allow iteration on messages as they are received.

```
for msg in bus:
    print(msg)
```

Yields
Message msg objects.

Return type
Iterator[Message]

BusABC.**__str__**()

Return str(self).

Return type
str

BusABC.__weakref__

list of weak references to the object (if defined)

BusABC._recv_internal(*timeout*)

Read a message from the bus and tell whether it was filtered. This methods may be called by *recv()* to read a message multiple times if the filters set by *set_filters()* do not match and the call has not yet timed out.

New implementations should always override this method instead of *recv()*, to be able to take advantage of the software based filtering provided by *recv()* as a fallback. This method should never be called directly.

Note: This method is not an *@abstractmethod* (for now) to allow older external implementations to continue using their existing *recv()* implementation.

Note: The second return value (whether filtering was already done) may change over time for some interfaces, like for example in the Kvaser interface. Thus it cannot be simplified to a constant value.

Parameters

timeout (*float*) – seconds to wait for a message, see *send()*

Returns

1. a message that was read or None on timeout
2. a bool that is True if message filtering has already been done and else False

Raises

- *CanOperationError* – If an error occurred while reading
- **NotImplementedError** – if the bus provides it's own *recv()* implementation (legacy implementation)

Return type

Tuple[Message | None, bool]

BusABC._apply_filters(*filters*)

Hook for applying the filters to the underlying kernel or hardware if supported/implemented by the interface.

Parameters

filters (*Sequence[CanFilter | CanFilterExtended] | None*) – See *set_filters()* for details.

Return type

None

BusABC._send_periodic_internal(*msgs*, *period*, *duration=None*, *modifier_callback=None*)

Default implementation of periodic message sending using threading.

Override this method to enable a more efficient backend specific approach.

Parameters

- **msgs** (*Sequence[Message] | Message*) – Messages to transmit
- **period** (*float*) – Period in seconds between each message
- **duration** (*float | None*) – The duration between sending each message at the given rate. If no duration is provided, the task will continue indefinitely.

- **modifier_callback** (*Callable[[Message], None] | None*) –

Returns
A started task instance. Note the task can be stopped (and depending on the backend modified) by calling the *stop()* method.

Return type
CyclicSendTaskABC

static BusABC.**_detect_available_configs**()

Detect all configurations/channels that this interface could currently connect with.

This might be quite time consuming.

May not to be implemented by every interface on every platform.

Returns
an iterable of dicts, each being a configuration suitable for usage in the interface's bus constructor.

Return type
List[*AutoDetectedConfig*]

3.9.3 About the IO module

Handling of the different file formats is implemented in `can.io`. Each file/IO type is within a separate module and ideally implements both a *Reader* and a *Writer*. The reader usually extends *can.io.generic.BaseIOHandler*, while the writer often additionally extends *can.Listener*, to be able to be passed directly to a *can.Notifier*.

Adding support for new file formats

This assumes that you want to add a new file format, called *canstore*. Ideally add both reading and writing support for the new file format, although this is not strictly required.

1. Create a new module: *can/io/canstore.py* (*or* simply copy some existing one like *can/io/csv.py*)
2. Implement a reader `CanstoreReader` (which often extends *can.io.generic.BaseIOHandler*, but does not have to). Besides from a constructor, only `__iter__(self)` needs to be implemented.
3. Implement a writer `CanstoreWriter` (which often extends *can.io.generic.BaseIOHandler* and *can.Listener*, but does not have to). Besides from a constructor, only `on_message_received(self, msg)` needs to be implemented.
4. Add a case to `can.io.player.LogReader`'s `__new__()`.
5. Document the two new classes (and possibly additional helpers) with docstrings and comments. Please mention features and limitations of the implementation.
6. Add a short section to the bottom of *doc/listeners.rst*.
7. Add tests where appropriate, for example by simply adding a test case called *class TestCanstoreFileFormat(ReaderWriterTest)* to *test/logformats_test.py*. That should already handle all of the general testing. Just follow the way the other tests in there do it.
8. Add imports to *can/__init__py* and *can/io/__init__py* so that the new classes can be simply imported as *from can import CanstoreReader, CanstoreWriter*.

IO Utilities

Contains generic base classes for file IO.

class can.io.generic.**BaseIOHandler**(*file*, *mode='rt'*, ***kwargs*)

A generic file handler that can be used for reading and writing.

Can be used as a context manager.

Attr file
the file-like object that is kept internally, or *None* if none was opened

Parameters

- **file** (*TextIO | BinaryIO | GzipFile | None*) – a path-like object to open a file, a file-like object to be used as a file or *None* to not use a file at all
- **mode** (*str*) – the mode that should be used to open the file, see open(), ignored if *file* is *None*
- **kwargs** (*Any*) –

stop()

Closes the underlying file-like object and flushes it, if it was opened in write mode.

Return type
None

class can.io.generic.**BinaryIOMessageReader**(*file*, *mode='rt'*, ***kwargs*)

Parameters

- **file** (*BinaryIO | GzipFile*) – a path-like object to open a file, a file-like object to be used as a file or *None* to not use a file at all
- **mode** (*str*) – the mode that should be used to open the file, see open(), ignored if *file* is *None*
- **kwargs** (*Any*) –

class can.io.generic.**BinaryIOMessageWriter**(*file*, *mode='wt'*, ***kwargs*)

Parameters

- **file** (*BinaryIO | GzipFile*) – a path-like object to open a file, a file-like object to be used as a file or *None* to not use a file at all
- **mode** (*str*) – the mode that should be used to open the file, see open(), ignored if *file* is *None*
- **kwargs** (*Any*) –

class can.io.generic.**FileIOMessageWriter**(*file*, *mode='wt'*, ***kwargs*)

A specialized base class for all writers with file descriptors.

Parameters

- **file** (*TextIO | BinaryIO | GzipFile*) – a path-like object to open a file, a file-like object to be used as a file or *None* to not use a file at all
- **mode** (*str*) – the mode that should be used to open the file, see open(), ignored if *file* is *None*
- **kwargs** (*Any*) –

file_size()

Return an estimate of the current file size in bytes.

Return type
int

class can.io.generic.**MessageReader**(*file*, *mode='rt'*, ***kwargs*)

The base class for all readers.

Parameters

- **file** (*TextIO | BinaryIO | GzipFile | None*) – a path-like object to open a file, a file-like object to be used as a file or *None* to not use a file at all
- **mode** (*str*) – the mode that should be used to open the file, see open(), ignored if *file* is *None*
- **kwargs** (*Any*) –

class can.io.generic.**MessageWriter**(*file*, *mode='rt'*, ***kwargs*)

The base class for all writers.

Parameters

- **file** (*TextIO | BinaryIO | GzipFile | None*) – a path-like object to open a file, a file-like object to be used as a file or *None* to not use a file at all
- **mode** (*str*) – the mode that should be used to open the file, see open(), ignored if *file* is *None*
- **kwargs** (*Any*) –

class can.io.generic.**TextIOMessageReader**(*file*, *mode='rt'*, ***kwargs*)

Parameters

- **file** (*TextIO*) – a path-like object to open a file, a file-like object to be used as a file or *None* to not use a file at all
- **mode** (*str*) – the mode that should be used to open the file, see open(), ignored if *file* is *None*
- **kwargs** (*Any*) –

class can.io.generic.**TextIOMessageWriter**(*file*, *mode='wt'*, ***kwargs*)

Parameters

- **file** (*TextIO*) – a path-like object to open a file, a file-like object to be used as a file or *None* to not use a file at all
- **mode** (*str*) – the mode that should be used to open the file, see open(), ignored if *file* is *None*
- **kwargs** (*Any*) –

3.9.4 Other Utilities

Utilities and configuration file parsing.

can.util.**cast_from_string**(*string_val*)

Perform trivial type conversion from `str` values.

Parameters
string_val (*str*) – the string, that shall be converted

Return type
str | int | float | bool

can.util.**channel2int**(*channel*)

Try to convert the channel to an integer.

Parameters
channel (*str | int | None*) – Channel string (e.g. *"can0"*, *"CAN1"*) or an integer

Returns
Channel integer or `None` if unsuccessful

Return type
int | None

can.util.**check_or_adjust_timing_clock**(*timing*, *valid_clocks*)

Adjusts the given timing instance to have an *f_clock* value that is within the allowed values specified by *valid_clocks*. If the *f_clock* value of timing is already within *valid_clocks*, then *timing* is returned unchanged.

Parameters

- **timing** (*T2*) – The *BitTiming* or *BitTimingFd* instance to adjust.
- **valid_clocks** (*Iterable[int]*) – An iterable of integers representing the valid *f_clock* values that the timing instance can be changed to. The order of the values in *valid_clocks* determines the priority in which they are tried, with earlier values being tried before later ones.

Returns
A new *BitTiming* or *BitTimingFd* instance with an *f_clock* value within *valid_clocks*.

Raises
CanInitializationError – If no compatible *f_clock* value can be found within *valid_clocks*.

Return type
T2

can.util.**deprecated_args_alias**(*deprecation_start*, *deprecation_end=None*, ***aliases*)

Allows to rename/deprecate a function kwarg(s) and optionally have the deprecated kwarg(s) set as alias(es)

Example:

```
@deprecated_args_alias("1.2.0", oldArg="new_arg", anotherOldArg="another_new_arg")
def library_function(new_arg, another_new_arg):
    pass

@deprecated_args_alias(
    deprecation_start="1.2.0",
    deprecation_end="3.0.0",
    oldArg="new_arg",
```

(continues on next page)

(continued from previous page)

```
    obsoleteOldArg=None,
)
def library_function(new_arg):
    pass
```

Parameters

- **deprecation_start** (*str*) – The *python-can* version, that introduced the DeprecationWarning.
- **deprecation_end** (*str | None*) – The *python-can* version, that marks the end of the deprecation period.
- **aliases** (*str | None*) – keyword arguments, that map the deprecated argument names to the new argument names or None.

Return type
Callable[[*Callable*[[~P1], *T1*]], *Callable*[[~P1], *T1*]]

can.util.**dlc2len**(*dlc*)

Calculate the data length from DLC.

Parameters
dlc (*int*) – DLC (0-15)

Returns
Data length in number of bytes (0-64)

Return type
int

can.util.**len2dlc**(*length*)

Calculate the DLC from data length.

Parameters
length (*int*) – Length in number of bytes (0-64)

Returns
DLC (0-15)

Return type
int

can.util.**load_config**(*path=None*, *config=None*, *context=None*)

Returns a dict with configuration details which is loaded from (in this order):

- config
- can.rc
- Environment variables CAN_INTERFACE, CAN_CHANNEL, CAN_BITRATE
- Config files `/etc/can.conf` or `~/.can` or `~/.canrc` where the latter may add or replace values of the former.

Interface can be any of the strings from `can.VALID_INTERFACES` for example: kvaser, socketcan, pcan, usb2can, ixxat, nican, virtual.

Note: The key `bustype` is copied to `interface` if that one is missing and does never appear in the result.

Parameters

- **path** (*TextIO | BinaryIO | GzipFile | str | PathLike[str] | None*) – Optional path to config file.
- **config** (*Dict[str, Any] | None*) – A dict which may set the 'interface', and/or the 'channel', or neither. It may set other values that are passed through.
- **context** (*str | None*) – Extra 'context' pass to config sources. This can be used to section other than 'default' in the configuration file.

Returns

A config dictionary that should contain 'interface' & 'channel':

```
{
    'interface': 'python-can backend interface to use',
    'channel': 'default channel to use',
    # possibly more
}
```

Note `None` will be used if all the options are exhausted without finding a value.

All unused values are passed from `config` over to this.

Raises

CanInterfaceNotImplementedError if the `interface` name isn't recognized

Return type

BusConfig

`can.util.`**load_environment_config**(*context=None*)

Loads config dict from environmental variables (if set):

- CAN_INTERFACE
- CAN_CHANNEL
- CAN_BITRATE
- CAN_CONFIG

if context is supplied, "_{context}" is appended to the environment variable name we will look at. For example if context="ABC":

- CAN_INTERFACE_ABC
- CAN_CHANNEL_ABC
- CAN_BITRATE_ABC
- CAN_CONFIG_ABC

Parameters

context (*str | None*) –

Return type

Dict[str, str]

`can.util.`**load_file_config**(*path=None*, *section='default'*)

Loads configuration from file with following content:

```
[default]
interface = socketcan
channel = can0
```

Parameters

- **path** (*TextIO | BinaryIO | GzipFile | str | PathLike[str] | None*) – path to config file. If not specified, several sensible default locations are tried depending on platform.
- **section** (*str*) – name of the section to read configuration from.

Return type
Dict[str, str]

can.util.**set_logging_level**(*level_name*)

Set the logging level for the *"can"* logger.

Parameters
level_name (*str*) – One of: *'critical'*, *'error'*, *'warning'*, *'info'*, *'debug'*, *'subdebug'*, or the value None (=default). Defaults to *'debug'*.

Return type
None

can.util.**time_perfcounter_correlation**()

Get the *perf_counter* value nearest to when time.time() is updated

Computed if the default timer used by *time.time* on this platform has a resolution higher than 10s, otherwise the current time and perf_counter is directly returned. This was chosen as typical timer resolution on Linux/macOS is ~1s, and the Windows platform can vary from ~500s to 10ms.

Note this value is based on when *time.time()* is observed to update from Python, it is not directly returned by the operating system.

Returns
(t, performance_counter) time.time value and time.perf_counter value when the time.time is updated

Return type
Tuple[float, float]

CHAPTER

FOUR

HARDWARE INTERFACES

python-can hides the low-level, device-specific interfaces to controller area network adapters in interface dependant modules. However as each hardware device is different, you should carefully go through your interface's documentation.

Note: The *Interface Names* are listed in *Configuration*.

The following hardware interfaces are included in python-can:

4.1 CANalyst-II

CANalyst-II is a USB to CAN Analyzer device produced by Chuangxin Technology.

Install: `pip install "python-can[canalystii]"`

4.1.1 Supported platform

Windows, Linux and Mac.

Note: The backend driver depends on pyusb so a `pyusb` backend driver library such as `libusb` must be installed. On Windows a tool such as Zadig can be used to set the Canalyst-II USB device driver to `libusb-win32`.

4.1.2 Limitations

Multiple Channels

The USB protocol transfers messages grouped by channel. Messages received on channel 0 and channel 1 may be returned by software out of order between the two channels (although inside each channel, all messages are in order). The timestamp field of each message comes from the hardware and shows the exact time each message was received. To compare ordering of messages on channel 0 vs channel 1, sort the received messages by the timestamp field first.

4.1.3 Backend Driver

The backend driver module *canalystii <https://pypi.org/project/canalystii>* must be installed to use this interface. This open source driver is unofficial and based on reverse engineering. Earlier versions of python-can required a binary library from the vendor for this functionality.

4.1.4 Bus

class can.interfaces.canalystii.**CANalystIIBus**(*channel=(0, 1)*, *device=0*, *bitrate=None*, *timing=None*, *can_filters=None*, *rx_queue_size=None*, ***kwargs*)

Parameters

- **channel** (*int | Sequence[int] | str*) – Optional channel number, list/tuple of multiple channels, or comma separated string of channels. Default is to configure both channels.
- **device** (*int*) – Optional USB device number. Default is 0 (first device found).
- **bitrate** (*int | None*) – CAN bitrate in bits/second. Required unless the bit_timing argument is set.
- **timing** (BitTiming | BitTimingFd | *None*) – Optional *BitTiming* instance to use for custom bit timing setting. If this argument is set then it overrides the bitrate argument. The *f_clock* value of the timing instance must be set to 8_000_000 (8MHz) for standard CAN. CAN FD and the *BitTimingFd* class are not supported.
- **can_filters** (*Sequence[CanFilter | CanFilterExtended] | None*) – Optional filters for received CAN messages.
- **rx_queue_size** (*int | None*) – If set, software received message queue can only grow to this many messages (for all channels) before older messages are dropped
- **kwargs** (*Dict[str, Any]*) –

4.2 CANtact CAN Interface

Interface for CANtact devices from Linklayer Labs

class can.interfaces.cantact.**CantactBus**(*channel*, *bitrate=500000*, *poll_interval=0.01*, *monitor=False*, *timing=None*, ***kwargs*)

Bases: *BusABC*

CANtact interface

Parameters

- **channel** (*int*) – Channel number (zero indexed, labeled on multi-channel devices)
- **bitrate** (*int*) – Bitrate in bits/s
- **monitor** (*bool*) – If true, operate in listen-only monitoring mode
- **timing** (BitTiming | BitTimingFd | *None*) – Optional *BitTiming* instance to use for custom bit timing setting. If this argument is set then it overrides the bitrate argument. The *f_clock* value of the timing instance must be set to 24_000_000 (24MHz) for standard CAN. CAN FD and the *BitTimingFd* class are not supported.
- **poll_interval** (*float*) –

- **kwargs** (*Any*) –

send(*msg*, *timeout=None*)

Transmit a message to the CAN bus.

Override this method to enable the transmit path.

Parameters

- **msg** (Message) – A message object.
- **timeout** – If > 0, wait up to this many seconds for message to be ACK'ed or for transmit queue to be ready depending on driver implementation. If timeout is exceeded, an exception will be raised. Might not be supported by all interfaces. None blocks indefinitely.

Raises

CanOperationError – If an error occurred while sending

shutdown()

Called to carry out any interface specific cleanup required in shutting down a bus.

This method can be safely called multiple times.

4.3 ETAS

This interface adds support for CAN interfaces by ETAS. The ETAS BOA (Basic Open API) is used.

4.3.1 Installation

Install the "ETAS ECU and Bus Interfaces – Distribution Package".

Warning: Only Windows is supported by this interface.

The Linux kernel v5.13 (and greater) natively supports ETAS ES581.4, ES582.1 and ES584.1 USB modules. To use these under Linux, please refer to the *SocketCAN* interface documentation.

4.3.2 Configuration

The simplest configuration file would be:

```
[default]
interface = etas
channel = ETAS://ETH/ES910:abcd/CAN:1
```

Channels are the URIs used by the underlying API.

To find available URIs, use *detect_available_configs()*:

```
configs = can.interface.detect_available_configs(interfaces="etas")
for c in configs:
    print(c)
```

4.3.3 Bus

class can.interfaces.etas.**EtasBus**(*channel*, *can_filters=None*, *receive_own_messages=False*, *bitrate=1000000*, *fd=True*, *data_bitrate=2000000*, ***kwargs*)

Construct and open a CAN bus instance of the specified type.

Subclasses should call though this method with all given parameters as it handles generic tasks like applying filters.

Parameters

- **channel** (*str*) – The can interface identifier. Expected type is backend dependent.
- **can_filters** (*Sequence[CanFilter | CanFilterExtended] | None*) – See *set_filters()* for details.
- **kwargs** (*dict*) – Any backend dependent configurations are passed in this dictionary
- **receive_own_messages** (*bool*) –
- **bitrate** (*int*) –
- **fd** (*bool*) –
- **data_bitrate** (*int*) –

Raises

- **ValueError** – If parameters are out of range
- ***CanInterfaceNotImplementedError*** – If the driver cannot be accessed
- ***CanInitializationError*** – If the bus cannot be initialized

flush_tx_buffer()

Discard every message that may be queued in the output buffer(s).

Return type
None

send(*msg*, *timeout=None*)

Transmit a message to the CAN bus.

Override this method to enable the transmit path.

Parameters

- **msg** (Message) – A message object.
- **timeout** (*float | None*) – If > 0, wait up to this many seconds for message to be ACK'ed or for transmit queue to be ready depending on driver implementation. If timeout is exceeded, an exception will be raised. Might not be supported by all interfaces. None blocks indefinitely.

Raises
CanOperationError – If an error occurred while sending

Return type
None

shutdown()

Called to carry out any interface specific cleanup required in shutting down a bus.

This method can be safely called multiple times.

Return type
None

property state: *BusState*
Return the current state of the hardware

4.4 Geschwister Schneider and candleLight

Windows/Linux/Mac CAN driver based on usbfs or WinUSB WCID for Geschwister Schneider USB/CAN devices and candleLight USB CAN interfaces.

Install: `pip install "python-can[gs_usb]"`

Usage: pass device `index` (starting from 0) if using automatic device detection:

```
import can

bus = can.Bus(interface="gs_usb", channel=dev.product, index=0, bitrate=250000)
```

Alternatively, pass `bus` and `address` to open a specific device. The parameters can be got by `pyusb` as shown below:

```
import usb
import can

dev = usb.core.find(idVendor=0x1D50, idProduct=0x606F)
bus = can.Bus(
    interface="gs_usb",
    channel=dev.product,
    bus=dev.bus,
    address=dev.address,
    bitrate=250000
)
```

4.4.1 Supported devices

Geschwister Schneider USB/CAN devices and bytewerk.org candleLight USB CAN interfaces such as candleLight, canable, cantact, etc.

4.4.2 Supported platform

Windows, Linux and Mac.

Note: The backend driver depends on pyusb so a `pyusb` backend driver library such as `libusb` must be installed.

On Windows a tool such as Zadig can be used to set the USB device driver to `libusb-win32`.

4.4.3 Supplementary Info

The firmware implementation for Geschwister Schneider USB/CAN devices and candleLight USB CAN can be found in candle-usb/candleLight_fw. The Linux kernel driver can be found in linux/drivers/net/can/usb/gs_usb.c.

The `gs_usb` interface in `python-can` relies on upstream `gs_usb` package, which can be found in https://pypi.org/project/gs-usb/ or https://github.com/jxltom/gs_usb.

The `gs_usb` package uses `pyusb` as backend, which brings better cross-platform compatibility.

Note: The bitrate `10K`, `20K`, `50K`, `83.333K`, `100K`, `125K`, `250K`, `500K`, `800K` and `1M` are supported in this interface, as implemented in the upstream `gs_usb` package's `set_bitrate` method.

Warning: Message filtering is not supported in Geschwister Schneider USB/CAN devices and bytewerk.org candleLight USB CAN interfaces.

4.4.4 Bus

class can.interfaces.gs_usb.**GsUsbBus**(*channel*, *bitrate*, *index=None*, *bus=None*, *address=None*, *can_filters=None*, ***kwargs*)

Parameters

- **channel** – usb device name
- **index** – device number if using automatic scan, starting from 0. If specified, bus/address shall not be provided.
- **bus** – number of the bus that the device is connected to
- **address** – address of the device on the bus it is connected to
- **can_filters** – not supported
- **bitrate** – CAN network bandwidth (bits/s)

send(*msg*, *timeout=None*)

Transmit a message to the CAN bus.

Parameters

- **msg** (Message) – A message object.
- **timeout** (*float | None*) – timeout is not supported. The function won't return until message is sent or exception is raised.

Raises

CanOperationError – if the message could not be sent

shutdown()

Called to carry out any interface specific cleanup required in shutting down a bus.

This method can be safely called multiple times.

4.5 isCAN

Interface for isCAN from Thorsis Technologies GmbH, former ifak system GmbH.

4.5.1 Bus

class can.interfaces.iscan.**IscanBus**(*channel*, *bitrate=500000*, *poll_interval=0.01*, ***kwargs*)

isCAN interface

Parameters

- **channel** (*str* | *int*) – Device number
- **bitrate** (*int*) – Bitrate in bits/s
- **poll_interval** (*float*) – Poll interval in seconds when reading messages

exception can.interfaces.iscan.**IscanError**(*function*, *error_code*, *arguments*)

Parameters
error_code (*int*) –

Return type
None

4.6 IXXAT Virtual Communication Interface

Interface to IXXAT Virtual Communication Interface V3 SDK. Works on Windows.

The Linux ECI SDK is currently unsupported, however on Linux some devices are supported with *SocketCAN*.

The *send_periodic()* method is supported natively through the on-board cyclic transmit list. Modifying cyclic messages is not possible. You will need to stop it, and then start a new periodic message.

4.6.1 Configuration

The simplest configuration file would be:

```
[default]
interface = ixxat
channel = 0
```

Python-can will search for the first IXXAT device available and open the first channel. `interface` and `channel` parameters are interpreted by frontend `can.interfaces.interface` module, while the following parameters are optional and are interpreted by IXXAT implementation.

- `receive_own_messages` (default False) Enable self-reception of sent messages.
- `unique_hardware_id` (default first device) Unique hardware ID of the IXXAT device.
- `extended` (default True) Allow usage of extended IDs.
- `fd` (default False) Enable CAN-FD capabilities.
- `rx_fifo_size` (default 16 for CAN, 1024 for CAN-FD) Number of RX mailboxes.
- `tx_fifo_size` (default 16 for CAN, 128 for CAN-FD) Number of TX mailboxes.

- `bitrate` (default 500000) Channel bitrate.
- `data_bitrate` (defaults to 2Mbps) Channel data bitrate (only canfd, to use when message bitrate_switch is used).
- `sjw_abr` (optional, only canfd) Bus timing value sample jump width (arbitration).
- `tseg1_abr` (optional, only canfd) Bus timing value tseg1 (arbitration).
- `tseg2_abr` (optional, only canfd) Bus timing value tseg2 (arbitration).
- `sjw_dbr` (optional, only used if baudrate switch enabled) Bus timing value sample jump width (data).
- `tseg1_dbr` (optional, only used if baudrate switch enabled) Bus timing value tseg1 (data).
- `tseg2_dbr` (optional, only used if baudrate switch enabled) Bus timing value tseg2 (data).
- `ssp_dbr` (optional, only used if baudrate switch enabled) Secondary sample point (data).

4.6.2 Filtering

The CAN filters act as an allow list in IXXAT implementation, that is if you supply a non-empty filter list you must explicitly state EVERY frame you want to receive (including RTR field). The can_id/mask must be specified according to IXXAT behaviour, that is bit 0 of can_id/mask parameters represents the RTR field in CAN frame. See IXXAT VCI documentation, section "Message filters" for more info.

4.6.3 List available devices

In case you have connected multiple IXXAT devices, you have to select them by using their unique hardware id. The function *detect_available_configs()* can be used to generate a list of *BusABC* constructors (including the channel number and unique hardware ID number for the connected devices).

```
>>> import can
>>> configs = can.detect_available_configs("ixxat")
>>> for config in configs:
...     print(config)
{'interface': 'ixxat', 'channel': 0, 'unique_hardware_id': 'HW441489'}
{'interface': 'ixxat', 'channel': 0, 'unique_hardware_id': 'HW107422'}
{'interface': 'ixxat', 'channel': 1, 'unique_hardware_id': 'HW107422'}
```

You may also get a list of all connected IXXAT devices using the function `get_ixxat_hwids()` as demonstrated below:

```
>>> from can.interfaces.ixxat import get_ixxat_hwids
>>> for hwid in get_ixxat_hwids():
...     print("Found IXXAT with hardware id '%s'." % hwid)
Found IXXAT with hardware id 'HW441489'.
Found IXXAT with hardware id 'HW107422'.
```

4.6.4 Bus

class can.interfaces.ixxat.**IXXATBus**(*channel*, *can_filters=None*, *receive_own_messages=False*, *unique_hardware_id=None*, *extended=True*, *fd=False*, *rx_fifo_size=None*, *tx_fifo_size=None*, *bitrate=500000*, *data_bitrate=2000000*, *sjw_abr=None*, *tseg1_abr=None*, *tseg2_abr=None*, *sjw_dbr=None*, *tseg1_dbr=None*, *tseg2_dbr=None*, *ssp_dbr=None*, ***kwargs*)

The CAN Bus implemented for the IXXAT interface.

Based on the C implementation of IXXAT, two different dlls are provided by IXXAT, one to work with CAN, the other with CAN-FD.

This class only delegates to related implementation (in calib_vcinpl or canlib_vcinpl2) class depending on fd user option.

Parameters

- **channel** (*int*) – The Channel id to create this bus with.
- **can_filters** – See *can.BusABC.set_filters()*.
- **receive_own_messages** (*bool*) – Enable self-reception of sent messages.
- **unique_hardware_id** (*int* / *None*) – UniqueHardwareId to connect (optional, will use the first found if not supplied)
- **extended** (*bool*) – Default True, enables the capability to use extended IDs.
- **fd** (*bool*) – Default False, enables CAN-FD usage.
- **rx_fifo_size** (*int* / *None*) – Receive fifo size (default 1024 for fd, else 16)
- **tx_fifo_size** (*int* / *None*) – Transmit fifo size (default 128 for fd, else 16)
- **bitrate** (*int*) – Channel bitrate in bit/s
- **data_bitrate** (*int*) – Channel bitrate in bit/s (only in CAN-Fd if baudrate switch enabled).
- **sjw_abr** (*int* / *None*) – Bus timing value sample jump width (arbitration). Only takes effect with fd enabled.
- **tseg1_abr** (*int* / *None*) – Bus timing value tseg1 (arbitration). Only takes effect with fd enabled.
- **tseg2_abr** (*int* / *None*) – Bus timing value tseg2 (arbitration). Only takes effect with fd enabled.
- **sjw_dbr** (*int* / *None*) – Bus timing value sample jump width (data). Only takes effect with fd and baudrate switch enabled.
- **tseg1_dbr** (*int* / *None*) – Bus timing value tseg1 (data). Only takes effect with fd and bitrate switch enabled.
- **tseg2_dbr** (*int* / *None*) – Bus timing value tseg2 (data). Only takes effect with fd and bitrate switch enabled.
- **ssp_dbr** (*int* / *None*) – Secondary sample point (data). Only takes effect with fd and bitrate switch enabled.

flush_tx_buffer()

Flushes the transmit buffer on the IXXAT

send(*msg*, *timeout=None*)

Transmit a message to the CAN bus.

Override this method to enable the transmit path.

Parameters

- **msg** (Message) – A message object.
- **timeout** (*float | None*) – If > 0, wait up to this many seconds for message to be ACK'ed or for transmit queue to be ready depending on driver implementation. If timeout is exceeded, an exception will be raised. Might not be supported by all interfaces. None blocks indefinitely.

Raises

CanOperationError – If an error occurred while sending

Return type

None

shutdown()

Called to carry out any interface specific cleanup required in shutting down a bus.

This method can be safely called multiple times.

Return type

None

property state: *BusState*

Return the current state of the hardware

Implementation based on vcinpl.dll

class can.interfaces.ixxat.canlib_vcinpl.**IXXATBus**(*channel*, *can_filters=None*, *receive_own_messages=False*, *unique_hardware_id=None*, *extended=True*, *rx_fifo_size=16*, *tx_fifo_size=16*, *bitrate=500000*, ***kwargs*)

The CAN Bus implemented for the IXXAT interface.

Warning: This interface does implement efficient filtering of messages, but the filters have to be set in `__init__` using the `can_filters` parameter. Using *set_filters()* does not work.

Parameters

- **channel** (*int*) – The Channel id to create this bus with.
- **can_filters** – See *can.BusABC.set_filters()*.
- **receive_own_messages** (*bool*) – Enable self-reception of sent messages.
- **unique_hardware_id** (*int | None*) – unique_hardware_id to connect (optional, will use the first found if not supplied)
- **extended** (*bool*) – Default True, enables the capability to use extended IDs.
- **rx_fifo_size** (*int*) – Receive fifo size (default 16)
- **tx_fifo_size** (*int*) – Transmit fifo size (default 16)

- **bitrate** (*int*) – Channel bitrate in bit/s

flush_tx_buffer()

Flushes the transmit buffer on the IXXAT

send(*msg*, *timeout=None*)

Sends a message on the bus. The interface may buffer the message.

Parameters

- **msg** (Message) – The message to send.
- **timeout** (*float | None*) – Timeout after some time.

Raise
:class:CanTimeoutError :class:CanOperationError

Return type
None

shutdown()

Called to carry out any interface specific cleanup required in shutting down a bus.

This method can be safely called multiple times.

property state: *BusState*

Return the current state of the hardware

class can.interfaces.ixxat.canlib_vcinpl.**CyclicSendTask**(*scheduler*, *msgs*, *period*, *duration*, *resolution*)

A message in the cyclic transmit list.

Message send task with a defined duration and period.

Parameters

- **messages** – The messages to be sent periodically.
- **period** – The rate in seconds at which to send the messages.
- **duration** – Approximate duration in seconds to continue sending messages. If no duration is provided, the task will continue indefinitely.

Raises
ValueError – If the given messages are invalid

pause()

Pause transmitting message (keep it in the list).

start()

Start transmitting message (add to list if needed).

stop()

Stop transmitting message (remove from list).

Implementation based on vcinpl2.dll

class can.interfaces.ixxat.canlib_vcinpl2.**IXXATBus**(*channel*, *can_filters=None*, *receive_own_messages=False*, *unique_hardware_id=None*, *extended=True*, *rx_fifo_size=1024*, *tx_fifo_size=128*, *bitrate=500000*, *data_bitrate=2000000*, *sjw_abr=None*, *tseg1_abr=None*, *tseg2_abr=None*, *sjw_dbr=None*, *tseg1_dbr=None*, *tseg2_dbr=None*, *ssp_dbr=None*, ***kwargs*)

The CAN Bus implemented for the IXXAT interface.

Warning: This interface does implement efficient filtering of messages, but the filters have to be set in `__init__` using the `can_filters` parameter. Using *set_filters()* does not work.

Parameters

- **channel** (*int*) – The Channel id to create this bus with.
- **can_filters** – See *can.BusABC.set_filters()*.
- **receive_own_messages** (*int*) – Enable self-reception of sent messages.
- **unique_hardware_id** (*int | None*) – unique_hardware_id to connect (optional, will use the first found if not supplied)
- **extended** (*bool*) – Default True, enables the capability to use extended IDs.
- **rx_fifo_size** (*int*) – Receive fifo size (default 1024)
- **tx_fifo_size** (*int*) – Transmit fifo size (default 128)
- **bitrate** (*int*) – Channel bitrate in bit/s
- **data_bitrate** (*int*) – Channel bitrate in bit/s (only in CAN-Fd if baudrate switch enabled).
- **sjw_abr** (*int | None*) – Bus timing value sample jump width (arbitration).
- **tseg1_abr** (*int | None*) – Bus timing value tseg1 (arbitration)
- **tseg2_abr** (*int | None*) – Bus timing value tseg2 (arbitration)
- **sjw_dbr** (*int | None*) – Bus timing value sample jump width (data)
- **tseg1_dbr** (*int | None*) – Bus timing value tseg1 (data). Only takes effect with fd and bitrate switch enabled.
- **tseg2_dbr** (*int | None*) – Bus timing value tseg2 (data). Only takes effect with fd and bitrate switch enabled.
- **ssp_dbr** (*int | None*) – Secondary sample point (data). Only takes effect with fd and bitrate switch enabled.

flush_tx_buffer()

Flushes the transmit buffer on the IXXAT

send(*msg*, *timeout=None*)

Sends a message on the bus. The interface may buffer the message.

Parameters

- **msg** (Message) – The message to send.
- **timeout** (*float | None*) – Timeout after some time.

Raise
:class:CanTimeoutError :class:CanOperationError

Return type
None

shutdown()

Called to carry out any interface specific cleanup required in shutting down a bus.

This method can be safely called multiple times.

class can.interfaces.ixxat.canlib_vcinpl2.**CyclicSendTask**(*scheduler*, *msgs*, *period*, *duration*, *resolution*)

A message in the cyclic transmit list.

Message send task with a defined duration and period.

Parameters

- **messages** – The messages to be sent periodically.
- **period** – The rate in seconds at which to send the messages.
- **duration** – Approximate duration in seconds to continue sending messages. If no duration is provided, the task will continue indefinitely.

Raises
ValueError – If the given messages are invalid

pause()

Pause transmitting message (keep it in the list).

start()

Start transmitting message (add to list if needed).

stop()

Stop transmitting message (remove from list).

4.6.5 Internals

The IXXAT *BusABC* object is a fairly straightforward interface to the IXXAT VCI library. It can open a specific device ID or use the first one found.

The frame exchange *does not involve threads* in the background but is explicitly instantiated by the caller.

- `recv()` is a blocking call with optional timeout.
- `send()` is not blocking but may raise a VCIError if the TX FIFO is full

RX and TX FIFO sizes are configurable with `rx_fifo_size` and `tx_fifo_size` options, defaulting to 16 for both.

4.7 Kvaser's CANLIB

Kvaser's CANLib SDK for Windows (also available on Linux).

4.7.1 Bus

class can.interfaces.kvaser.canlib.**KvaserBus**(*channel*, *can_filters=None*, *timing=None*, ***kwargs*)

The CAN Bus implemented for the Kvaser interface.

Parameters

- **channel** (*int*) – The Channel id to create this bus with.
- **can_filters** (*list*) – See *can.BusABC.set_filters()*.
- **timing** (BitTiming | BitTimingFd | *None*) –

Backend Configuration

Parameters

- **timing** (BitTiming | BitTimingFd | *None*) – An instance of *BitTiming* or *BitTimingFd* to specify the bit timing parameters for the Kvaser interface. If provided, it takes precedence over the all other timing-related parameters. Note that the *f_clock* property of the *timing* instance must be 16_000_000 (16MHz) for standard CAN or 80_000_000 (80MHz) for CAN FD.
- **bitrate** (*int*) – Bitrate of channel in bit/s
- **accept_virtual** (*bool*) – If virtual channels should be accepted.
- **tseg1** (*int*) – Time segment 1, that is, the number of quanta from (but not including) the Sync Segment to the sampling point. If this parameter is not given, the Kvaser driver will try to choose all bit timing parameters from a set of defaults.
- **tseg2** (*int*) – Time segment 2, that is, the number of quanta from the sampling point to the end of the bit.
- **sjw** (*int*) – The Synchronization Jump Width. Decides the maximum number of time quanta that the controller can resynchronize every bit.
- **no_samp** (*int*) – Either 1 or 3. Some CAN controllers can also sample each bit three times. In this case, the bit will be sampled three quanta in a row, with the last sample being taken in the edge between TSEG1 and TSEG2. Three samples should only be used for relatively slow baudrates.
- **driver_mode** (*bool*) – Silent or normal.
- **single_handle** (*bool*) – Use one Kvaser CANLIB bus handle for both reading and writing. This can be set if reading and/or writing is done from one thread.
- **receive_own_messages** (*bool*) – If messages transmitted should also be received back. Only works if single_handle is also False. If you want to receive messages from other applications on the same computer, set this to True or set single_handle to True.
- **fd** (*bool*) – If CAN-FD frames should be supported.
- **exclusive** (*bool*) – Don't allow sharing of this CANlib channel.
- **override_exclusive** (*bool*) – Open the channel even if it is opened for exclusive access already.

- **data_bitrate** (*int*) – Which bitrate to use for data phase in CAN FD. Defaults to arbitration bitrate.
- **channel** (*int*) –
- **can_filters** (*Sequence[CanFilter | CanFilterExtended] | None*) –

flash(*flash=True*)

Turn on or off flashing of the device's LED for physical identification purposes.

flush_tx_buffer()

Wipeout the transmit buffer on the Kvaser.

send(*msg*, *timeout=None*)

Transmit a message to the CAN bus.

Override this method to enable the transmit path.

Parameters

- **msg** (*Message*) – A message object.
- **timeout** – If > 0, wait up to this many seconds for message to be ACK'ed or for transmit queue to be ready depending on driver implementation. If timeout is exceeded, an exception will be raised. Might not be supported by all interfaces. None blocks indefinitely.

Raises

CanOperationError – If an error occurred while sending

shutdown()

Called to carry out any interface specific cleanup required in shutting down a bus.

This method can be safely called multiple times.

4.7.2 Internals

The Kvaser *Bus* object with a physical CAN Bus can be operated in two modes; `single_handle` mode with one shared bus handle used for both reading and writing to the CAN bus, or with two separate bus handles. Two separate handles are needed if receiving and sending messages in different threads (see Kvaser documentation).

Warning: Any objects inheriting from *Bus* should *not* directly use the interface handle(/s).

Message filtering

The Kvaser driver and hardware only supports setting one filter per handle. If one filter is requested, this is will be handled by the Kvaser driver. If more than one filter is needed, these will be handled in Python code in the `recv` method. If a message does not match any of the filters, `recv()` will return None.

Custom methods

This section contains Kvaser driver specific methods.

KvaserBus.**get_stats**()

Retrieves the bus statistics.

Use like so:

```
>>> stats = bus.get_stats()
>>> print(stats)
std_data: 0, std_remote: 0, ext_data: 0, ext_remote: 0, err_frame: 0, bus_load: 0.0
↪%, overruns: 0
```

Returns
bus statistics.

Return type
BusStatistics

class can.interfaces.kvaser.structures.**BusStatistics**

This structure is used with the method *get_stats()*.

property bus_load

The bus load, expressed as an integer in the interval 0 - 10000 representing 0.00% - 100.00% bus load.

property err_frame

Number of error frames.

property ext_data

Number of received extended (29-bit identifiers) data frames.

property ext_remote

Number of received extended (29-bit identifiers) remote frames.

property overruns

Number of overruns.

property std_data

Number of received standard (11-bit identifiers) data frames.

property std_remote

Number of received standard (11-bit identifiers) remote frames.

4.8 Neousys CAN Interface

This kind of interface can be found for example on Neousys POC-551VTC One needs to have correct drivers and DLL (Share object for Linux) from Neousys.

Beware this is only tested on Linux kernel higher than v5.3. This should be drop in with Windows but you have to replace with correct named DLL

class can.interfaces.neousys.**NeousysBus**(*channel*, *device=0*, *bitrate=500000*, ***kwargs*)

Bases: *BusABC*

Neousys CAN bus Class

Parameters

- **channel** – channel number
- **device** – device number
- **bitrate** – bit rate.

send(*msg*, *timeout=None*)

Parameters

- **msg** – message to send
- **timeout** – timeout is not used here

Return type
None

shutdown()

Called to carry out any interface specific cleanup required in shutting down a bus.

This method can be safely called multiple times.

4.9 Intrepid Control Systems neoVI

Note: This `ICS neoVI` documentation is a work in progress. Feedback and revisions are most welcome!

Interface to Intrepid Control Systems neoVI API range of devices via python-ics wrapper on Windows.

4.9.1 Installation

This neoVI interface requires the installation of the ICS neoVI DLL and `python-ics` package.

- **Download and install the Intrepid Product Drivers**
  Intrepid Product Drivers
- **Install `python-can` with the `neovi` extras:**

```
pip install python-can[neovi]
```

4.9.2 Configuration

An example *can.ini* file for windows 7:

```
[default]
interface = neovi
channel = 1
```

4.9.3 Bus

class can.interfaces.ics_neovi.**NeoViBus**(*channel*, *can_filters=None*, ***kwargs*)

The CAN Bus implemented for the python_ics interface https://github.com/intrepidcs/python_ics

Parameters

- **channel** (*int or str or list(int) or list(str)*) – The channel ids to create this bus with. Can also be a single integer, netid name or a comma separated string.
- **can_filters** (*list*) – See *can.BusABC.set_filters()* for details.
- **receive_own_messages** (*bool*) – If transmitted messages should also be received by this bus.
- **use_system_timestamp** (*bool*) – Use system timestamp for can messages instead of the hardware time stamp
- **serial** (*str*) – Serial to connect (optional, will use the first found if not supplied)
- **bitrate** (*int*) – Channel bitrate in bit/s. (optional, will enable the auto bitrate feature if not supplied)
- **fd** (*bool*) – If CAN-FD frames should be supported.
- **data_bitrate** (*int*) – Which bitrate to use for data phase in CAN FD. Defaults to arbitration bitrate.
- **override_library_name** – Absolute path or relative path to the library including filename.

Raises

- **ImportError** – If *python-ics* is not available
- ***CanInitializationError*** – If the bus could not be set up. May or may not be a *ICSInitializationError*.

exception can.interfaces.ics_neovi.**ICSApiError**(*error_code*, *description_short*, *description_long*, *severity*, *restart_needed*)

Indicates an error with the ICS API.

Parameters

- **error_code** (*int*) –
- **description_short** (*str*) –
- **description_long** (*str*) –
- **severity** (*int*) –
- **restart_needed** (*int*) –

exception can.interfaces.ics_neovi.**ICSInitializationError**(*error_code*, *description_short*, *description_long*, *severity*, *restart_needed*)

Parameters

- **error_code** (*int*) –
- **description_short** (*str*) –
- **description_long** (*str*) –
- **severity** (*int*) –

- **restart_needed** (*int*) –

exception can.interfaces.ics_neovi.**ICSOperationError**(*error_code*, *description_short*, *description_long*, *severity*, *restart_needed*)

Parameters

- **error_code** (*int*) –
- **description_short** (*str*) –
- **description_long** (*str*) –
- **severity** (*int*) –
- **restart_needed** (*int*) –

4.10 National Instruments NI-CAN

This interface adds support for NI-CAN controllers by National Instruments.

Warning: NI-CAN only seems to support 32-bit architectures so if the driver can't be loaded on a 64-bit Python, try using a 32-bit version instead.

Warning: CAN filtering has not been tested thoroughly and may not work as expected.

4.10.1 Bus

class can.interfaces.nican.**NicanBus**(*channel*, *can_filters=None*, *bitrate=None*, *log_errors=True*, ***kwargs*)

The CAN Bus implemented for the NI-CAN interface.

Warning: This interface does implement efficient filtering of messages, but the filters have to be set in `__init__` using the `can_filters` parameter. Using *set_filters()* does not work.

Parameters

- **channel** (*str*) – Name of the object to open (e.g. *"CAN0"*)
- **bitrate** (*int | None*) – Bitrate in bit/s
- **can_filters** (*Sequence[CanFilter | CanFilterExtended] | None*) – See *can.BusABC.set_filters()*.
- **log_errors** (*bool*) – If True, communication errors will appear as CAN messages with `is_error_frame` set to True and `arbitration_id` will identify the error (default True)

Raises

- *CanInterfaceNotImplementedError* – If the current operating system is not supported or the driver could not be loaded.
- *NicanInitializationError* – If the bus could not be set up.

exception can.interfaces.nican.**NicanError**(*function*, *error_code*, *arguments*)

Error from NI-CAN driver.

Parameters
error_code (*int*) –

Return type
None

exception can.interfaces.nican.**NicanInitializationError**(*function*, *error_code*, *arguments*)

Parameters
error_code (*int*) –

Return type
None

4.11 National Instruments NI-XNET

This interface adds support for NI-XNET CAN controllers by National Instruments.

Note: NI-XNET only supports windows platforms.

4.11.1 Bus

class can.interfaces.nixnet.**NiXNETcanBus**(*channel='CAN1'*, *bitrate=500000*, *timing=None*, *can_filters=None*, *receive_own_messages=False*, *can_termination=False*, *fd=False*, *fd_bitrate=None*, *poll_interval=0.001*, ***kwargs*)

Bases: *BusABC*

The CAN Bus implemented for the NI-XNET interface.

Parameters

- **channel** (*str*) – Name of the object to open (e.g. 'CAN0')
- **bitrate** (*int*) – Bitrate in bits/s
- **timing** (*BitTiming | BitTimingFd | None*) – Optional *BitTiming* or *BitTimingFd* instance to use for custom bit timing setting. The *f_clock* value of the timing instance must be set to 40_000_000 (40MHz). If this parameter is provided, it takes precedence over all other timing-related parameters like *bitrate*, *fd_bitrate* and *fd*.
- **can_filters** (*list*) – See *can.BusABC.set_filters()*.
- **receive_own_messages** (*bool*) – Enable self-reception of sent messages.
- **poll_interval** (*float*) – Poll interval in seconds.
- **can_termination** (*bool*) –
- **fd** (*bool*) –
- **fd_bitrate** (*int | None*) –
- **kwargs** (*Any*) –

Raises
CanInitializationError – If starting communication fails

send(*msg*, *timeout=None*)

Send a message using NI-XNET.

Parameters

- **msg** (can.Message) – Message to send
- **timeout** (*float*) – Max time to wait for the device to be ready in seconds, None if time is infinite

Raises
can.exceptions.CanOperationError – If writing to transmit buffer fails. It does not wait for message to be ACKed currently.

Return type
None

reset()

Resets network interface. Stops network interface, then resets the CAN chip to clear the CAN error counters (clear error passive state). Resetting includes clearing all entries from read and write queues.

Return type
None

shutdown()

Close object.

Return type
None

4.12 PCAN Basic API

Interface to Peak-System's PCAN-Basic API.

4.12.1 Configuration

Here is an example configuration file for using PCAN-USB:

```
[default]
interface = pcan
channel = PCAN_USBBUS1
state = can.bus.BusState.PASSIVE
bitrate = 500000
```

channel (default "PCAN_USBBUS1")
CAN interface name. Valid channel values:

```
PCAN_ISABUSx
PCAN_DNGBUSx
PCAN_PCIBUSx
PCAN_USBBUSx
PCAN_PCCBUSx
PCAN_LANBUSx
```

Where `x` should be replaced with the desired channel number starting at `1`.

state (default `can.bus.BusState.ACTIVE`)
BusState of the channel

bitrate (default `500000`)
Channel bitrate

4.12.2 Linux installation

Beginning with version 3.4, Linux kernels support the PCAN adapters natively via *SocketCAN*, refer to: *PCAN*.

4.12.3 Bus

class can.interfaces.pcan.**PcanBus**(*channel='PCAN_USBBUS1'*, *device_id=None*, *state=BusState.ACTIVE*, *timing=None*, *bitrate=500000*, ***kwargs*)

A PCAN USB interface to CAN.

On top of the usual *Bus* methods provided, the PCAN interface includes the *flash()* and *status()* methods.

Parameters

- **channel** (*str*) – The can interface name. An example would be 'PCAN_USBBUS1'. Alternatively the value can be an int with the numerical value. Default is 'PCAN_USBBUS1'
- **device_id** (*int*) – Select the PCAN interface based on its ID. The device ID is a 8/32bit value that can be configured for each PCAN device. If you set the device_id parameter, it takes precedence over the channel parameter. The constructor searches all connected interfaces and initializes the first one that matches the parameter value. If no device is found, an exception is raised.
- **state** (*can.bus.BusState*) – BusState of the channel. Default is ACTIVE
- **timing** (*BitTiming* | *BitTimingFd* | *None*) – An instance of *BitTiming* or *BitTimingFd* to specify the bit timing parameters for the PCAN interface. If this parameter is provided, it takes precedence over all other timing-related parameters. If this parameter is not provided, the bit timing parameters can be specified using the *bitrate* parameter for standard CAN or the *fd*, *f_clock*, *f_clock_mhz*, *nom_brp*, *nom_tseg1*, *nom_tseg2*, *nom_sjw*, *data_brp*, *data_tseg1*, *data_tseg2*, and *data_sjw* parameters for CAN FD. Note that the *f_clock* value of the *timing* instance must be 8_000_000 for standard CAN or any of the following values for CAN FD: 20_000_000, 24_000_000, 30_000_000, 40_000_000, 60_000_000, 80_000_000.
- **bitrate** (*int*) – Bitrate of channel in bit/s. Default is 500 kbit/s. Ignored if using CanFD.
- **fd** (*bool*) – Should the Bus be initialized in CAN-FD mode.
- **f_clock** (*int*) – Clock rate in Hz. Any of the following: 20000000, 24000000, 30000000, 40000000, 60000000, 80000000. Ignored if not using CAN-FD. Pass either f_clock or f_clock_mhz.
- **f_clock_mhz** (*int*) – Clock rate in MHz. Any of the following: 20, 24, 30, 40, 60, 80. Ignored if not using CAN-FD. Pass either f_clock or f_clock_mhz.
- **nom_brp** (*int*) – Clock prescaler for nominal time quantum. In the range (1..1024) Ignored if not using CAN-FD.

- **nom_tseg1** (*int*) – Time segment 1 for nominal bit rate, that is, the number of quanta from (but not including) the Sync Segment to the sampling point. In the range (1..256). Ignored if not using CAN-FD.
- **nom_tseg2** (*int*) – Time segment 2 for nominal bit rate, that is, the number of quanta from the sampling point to the end of the bit. In the range (1..128). Ignored if not using CAN-FD.
- **nom_sjw** (*int*) – Synchronization Jump Width for nominal bit rate. Decides the maximum number of time quanta that the controller can resynchronize every bit. In the range (1..128). Ignored if not using CAN-FD.
- **data_brp** (*int*) – Clock prescaler for fast data time quantum. In the range (1..1024) Ignored if not using CAN-FD.
- **data_tseg1** (*int*) – Time segment 1 for fast data bit rate, that is, the number of quanta from (but not including) the Sync Segment to the sampling point. In the range (1..32). Ignored if not using CAN-FD.
- **data_tseg2** (*int*) – Time segment 2 for fast data bit rate, that is, the number of quanta from the sampling point to the end of the bit. In the range (1..16). Ignored if not using CAN-FD.
- **data_sjw** (*int*) – Synchronization Jump Width for fast data bit rate. Decides the maximum number of time quanta that the controller can resynchronize every bit. In the range (1..16). Ignored if not using CAN-FD.
- **auto_reset** (*bool*) – Enable automatic recovery in bus off scenario. Resetting the driver takes ~500ms during which it will not be responsive.
- **kwargs** (*Any*) –

flash(*flash*)

Turn on or off flashing of the device's LED for physical identification purposes.

get_device_number()

Return the PCAN device number.

Return type
int

Returns
PCAN device number

reset()

Command the PCAN driver to reset the bus after an error.

send(*msg*, *timeout=None*)

Transmit a message to the CAN bus.

Override this method to enable the transmit path.

Parameters

- **msg** (Message) – A message object.
- **timeout** – If > 0, wait up to this many seconds for message to be ACK'ed or for transmit queue to be ready depending on driver implementation. If timeout is exceeded, an exception will be raised. Might not be supported by all interfaces. None blocks indefinitely.

Raises
CanOperationError – If an error occurred while sending

`set_device_number`(*device_number*)

Set the PCAN device number.

Parameters
`device_number` – new PCAN device number

Return type
bool

Returns
True if device number set successfully

`shutdown`()

Called to carry out any interface specific cleanup required in shutting down a bus.

This method can be safely called multiple times.

`property state`

Return the current state of the hardware

`status`()

Query the PCAN bus status.

Return type
int

Returns
The status code. See values in **basic.PCAN_ERROR_**

`status_is_ok`()

Convenience method to check that the bus status is OK

`status_string`()

Query the PCAN bus status.

Returns
The status description, if any was found.

Return type
str | None

4.13 Robotell CAN-USB interface

An USB to CAN adapter sold on Aliexpress, etc. with the manufacturer name Robotell printed on the case. There is also a USB stick version with a clear case. If the description or screenshots refer to `EmbededDebug` or `EmbededConfig` the device should be compatible with this driver. These USB devices are based on a STM32 controller with a CH340 serial interface and use a binary protocol - NOT compatible with SLCAN

See https://www.amobbs.com/thread-4651667-1-1.html for some background on these devices.

This driver directly uses either the local or remote (not tested) serial port. Remote serial ports will be specified via special URL. Both raw TCP sockets as also RFC2217 ports are supported.

Usage: use `port or URL[@baurate]` to open the device. For example use `/dev/ttyUSB0@115200` or `COM4@9600` for local serial ports and `socket://192.168.254.254:5000` or `rfc2217://192.168.254.254:5000` for remote ports.

4.13.1 Bus

class can.interfaces.robotell.**robotellBus**(*channel*, *ttyBaudrate=115200*, *bitrate=None*, *rtscts=False*, ***kwargs*)

robotell interface

Parameters

- **channel** (*str*) – port of underlying serial or usb device (e.g. `/dev/ttyUSB0`, `COM8`, ...) Must not be empty. Can also end with `@115200` (or similarly) to specify the baudrate.
- **ttyBaudrate** (*int*) – baudrate of underlying serial or usb device (Ignored if set via the `channel` parameter)
- **bitrate** (*int*) – CAN Bitrate in bit/s. Value is stored in the adapter and will be used as default if no bitrate is specified
- **rtscts** (*bool*) – turn hardware handshake (RTS/CTS) on and off

get_serial_number(*timeout*)

Get serial number of the slcan interface.

Parameters
timeout (*int | None*) – seconds to wait for serial number or None to wait indefinitely

Returns
None on timeout or a str object.

Return type
str | None

send(*msg*, *timeout=None*)

Transmit a message to the CAN bus.

Override this method to enable the transmit path.

Parameters

- **msg** (Message) – A message object.
- **timeout** – If > 0, wait up to this many seconds for message to be ACK'ed or for transmit queue to be ready depending on driver implementation. If timeout is exceeded, an exception will be raised. Might not be supported by all interfaces. None blocks indefinitely.

Raises
CanOperationError – If an error occurred while sending

set_auto_bus_management(*auto_man*)

Parameters
auto_man (*bool*) – Enable/disable automatic bus management

set_auto_retransmit(*retrans_flag*)

Parameters
retrans_flag (*bool*) – Enable/disable automatic retransmission of unacknowledged CAN frames

set_bitrate(*bitrate*)

Raises
ValueError – if *bitrate* is greater than 1000000

Parameters
bitrate (*int*) – Bitrate in bit/s

`set_hw_filter`(*filterid*, *enabled*, *msgid_value*, *msgid_mask*, *extended_msg*)

Raises
ValueError – if *filterid* is not between 1 and 14

Parameters

- **filterid** (*int*) – ID of filter (1-14)
- **enabled** (*bool*) – This filter is enabled
- **msgid_value** (*int*) – CAN message ID to filter on. The test unit does not accept an extented message ID unless bit 31 of the ID was set.
- **msgid_mask** (*int*) – Mask to apply to CAN messagge ID
- **extended_msg** (*bool*) – Filter operates on extended format messages

`set_serial_rate`(*serial_bps*)

Parameters
serial_bps (*int*) – Set the baud rate of the serial port (not CAN) interface

`shutdown`()

Called to carry out any interface specific cleanup required in shutting down a bus.

This method can be safely called multiple times.

4.14 Seeed Studio USB-CAN Analyzer

SKU: 114991193

Links:

- https://www.seeedstudio.com/USB-CAN-Analyzer-p-2888.html
- https://github.com/SeeedDocument/USB-CAN_Analyzer
- https://copperhilltech.com/blog/usbcan-analyzer-usb-to-can-bus-serial-protocol-definition/

4.14.1 Installation

This interface has additional dependencies which can be installed using pip and the optional extra `seeedstudio`. That will include the dependency `pyserial`:

```
pip install python-can[seeedstudio]
```

4.14.2 Interface

```
can.interfaces.seeedstudio.SeeedBus
```

A bus example:

```
bus = can.interface.Bus(interface='seeedstudio', channel='/dev/ttyUSB0', bitrate=500000)
```

4.14.3 Configuration

```
SeeedBus(channel,
         baudrate=2000000,
         timeout=0.1,
         frame_type='STD',
         operation_mode='normal',
         bitrate=500000)
```

CHANNEL
The serial port created by the USB device when connected.

TIMEOUT
Only used by the underling serial port, it probably should not be changed. The serial port baudrate=2000000 and rtscts=false are also matched to the device so are not added here.

FRAMETYPE

- "STD"
- "EXT"

OPERATIONMODE

- "normal"
- "loopback"
- "silent"
- "loopback_and_silent"

BITRATE

- 1000000
- 800000
- 500000
- 400000
- 250000
- 200000
- 125000
- 100000
- 50000
- 20000
- 10000

- 5000

4.15 CAN over Serial

A text based interface. For example use over serial ports like `/dev/ttyS1` or `/dev/ttyUSB0` on Linux machines or `COM1` on Windows. Remote ports can be also used via a special URL. Both raw TCP sockets as also RFC2217 ports are supported: `socket://192.168.254.254:5000` or `rfc2217://192.168.254.254:5000`. In addition a virtual loopback can be used via `loop://` URL. The interface is a simple implementation that has been used for recording CAN traces.

Note: The properties **extended_id**, **is_remote_frame** and **is_error_frame** from the class:*~can.Message* are not in use. This interface will not send or receive flags for this properties.

4.15.1 Bus

class can.interfaces.serial.serial_can.**SerialBus**(*channel*, *baudrate=115200*, *timeout=0.1*, *rtscts=False*, **args*, ***kwargs*)

Enable basic can communication over a serial device.

Note: See *_recv_internal()* for some special semantics.

Parameters

- **channel** (*str*) – The serial device to open. For example "/dev/ttyS1" or "/dev/ttyUSB0" on Linux or "COM1" on Windows systems.
- **baudrate** (*int*) – Baud rate of the serial device in bit/s (default 115200).

  Warning: Some serial port implementations don't care about the baudrate.

- **timeout** (*float*) – Timeout for the serial device in seconds (default 0.1).
- **rtscts** (*bool*) – turn hardware handshake (RTS/CTS) on and off

Raises

- *CanInitializationError* – If the given parameters are invalid.
- *CanInterfaceNotImplementedError* – If the serial module is not installed.

_recv_internal(*timeout*)

Read a message from the serial device.

Parameters

timeout (*float | None*) –

Warning: This parameter will be ignored. The timeout value of the channel is used.

Returns

Received message and `False` (because no filtering as taken place).

> **Warning:** Flags like `is_extended_id`, `is_remote_frame` and `is_error_frame` will not be set over this function, the flags in the return message are the default values.

Return type

Tuple[Message | None, bool]

4.15.2 Internals

The frames that will be sent and received over the serial interface consist of six parts. The start and the stop byte for the frame, the timestamp, DLC, arbitration ID and the payload. The payload has a variable length of between 0 and 8 bytes, the other parts are fixed. Both, the timestamp and the arbitration ID will be interpreted as 4 byte unsigned integers. The DLC is also an unsigned integer with a length of 1 byte.

Serial frame format

	Start of frame	Timestamp	DLC	Arbitration ID	Payload	End of frame
Length (Byte)	1	4	1	4	0 - 8	1
Data type	Byte	Unsigned 4 byte integer	Unsigned 1 byte integer	Unsigned 4 byte integer	Byte	Byte
Byte order	-	Little-Endian	Little-Endian	Little-Endian	-	-
Description	Must be 0xAA	Usually s, ms or µs since start of the device	Length in byte of the payload	-	-	Must be 0xBB

Examples of serial frames

CAN message with 8 byte payload

CAN message Arbitration ID	Payload
1	0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88

Serial frame Start of frame	Timestamp	DLC	Arbitration ID	Payload	End of frame
0xAA	0x66 0x73 0x00 0x00	0x08	0x01 0x00 0x00 0x00	0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88	0xBB

CAN message with 1 byte payload

CAN message Arbitration ID	Payload
1	0x11

Serial frame Start of frame	Timestamp	DLC	Arbitration ID	Payload	End of frame
0xAA	0x66 0x73 0x00 0x00	0x01	0x01 0x00 0x00 0x00	0x11	0xBB

CAN message with 0 byte payload

CAN message Arbitration ID	Payload
1	None

Serial frame Start of frame	Timestamp	DLC	Arbitration ID	End of frame
0xAA	0x66 0x73 0x00 0x00	0x00	0x01 0x00 0x00 0x00	0xBB

4.16 CAN over Serial / SLCAN

A text based interface: compatible to slcan-interfaces (slcan ASCII protocol) should also support LAWICEL direct. These interfaces can also be used with socketcan and slcand with Linux. This driver directly uses either the local or remote serial port, it makes slcan-compatible interfaces usable with Windows also. Remote serial ports will be specified via special URL. Both raw TCP sockets as also RFC2217 ports are supported.

Usage: use `port or URL[@baurate]` to open the device. For example use `/dev/ttyUSB0@115200` or `COM4@9600` for local serial ports and `socket://192.168.254.254:5000` or `rfc2217://192.168.254.254:5000` for remote ports.

4.16.1 Supported devices

Todo: Document this.

4.16.2 Bus

class can.interfaces.slcan.**slcanBus**(*channel*, *ttyBaudrate=115200*, *bitrate=None*, *btr=None*, *sleep_after_open=2*, *rtscts=False*, *timeout=0.001*, ***kwargs*)

slcan interface

Parameters

- **channel** (*str*) – port of underlying serial or usb device (e.g. `/dev/ttyUSB0`, `COM8`, ...) Must not be empty. Can also end with `@115200` (or similarly) to specify the baudrate.
- **ttyBaudrate** (*int*) – baudrate of underlying serial or usb device (Ignored if set via the `channel` parameter)
- **bitrate** (*int | None*) – Bitrate in bit/s
- **btr** (*str | None*) – BTR register value to set custom can speed
- **poll_interval** – Poll interval in seconds when reading messages
- **sleep_after_open** (*float*) – Time to wait in seconds after opening serial connection
- **rtscts** (*bool*) – turn hardware handshake (RTS/CTS) on and off
- **timeout** (*float*) – Timeout for the serial or usb device in seconds (default 0.001)
- **kwargs** (*Any*) –

Raises

- **ValueError** – if both `bitrate` and `btr` are set or the channel is invalid
- ***CanInterfaceNotImplementedError*** – if the serial module is missing
- ***CanInitializationError*** – if the underlying serial connection could not be established

get_serial_number(*timeout*)

Get serial number of the slcan interface.

Parameters
timeout (*float | None*) – seconds to wait for serial number or `None` to wait indefinitely

Returns
`None` on timeout or a `str` object.

Return type
str | None

get_version(*timeout*)

Get HW and SW version of the slcan interface.

Parameters
timeout (*float | None*) – seconds to wait for version or None to wait indefinitely

Returns
tuple (hw_version, sw_version) WHERE int hw_version is the hardware version or None on timeout int sw_version is the software version or None on timeout

Return type
Tuple[int | None, int | None]

send(*msg*, *timeout=None*)

Transmit a message to the CAN bus.

Override this method to enable the transmit path.

Parameters

- **msg** (Message) – A message object.
- **timeout** (*float | None*) – If > 0, wait up to this many seconds for message to be ACK'ed or for transmit queue to be ready depending on driver implementation. If timeout is exceeded, an exception will be raised. Might not be supported by all interfaces. None blocks indefinitely.

Raises

CanOperationError – If an error occurred while sending

Return type

None

`set_bitrate`(*bitrate*)

Parameters

bitrate (*int*) – Bitrate in bit/s

Raises

ValueError – if `bitrate` is not among the possible values

Return type

None

`set_bitrate_reg`(*btr*)

Parameters

btr (*str*) – BTR register value to set custom can speed

Return type

None

`shutdown`()

Called to carry out any interface specific cleanup required in shutting down a bus.

This method can be safely called multiple times.

Return type

None

4.16.3 Internals

Todo: Document the internals of slcan interface.

4.17 SocketCAN

The SocketCAN documentation can be found in the Linux kernel docs in the `networking` directory. Quoting from the SocketCAN Linux documentation:

> The socketcan package is an implementation of CAN protocols (Controller Area Network) for Linux. CAN is a networking technology which has widespread use in automation, embedded devices, and automotive fields. While there have been other CAN implementations for Linux based on character devices, SocketCAN uses the Berkeley socket API, the Linux network stack and implements the CAN device drivers as

network interfaces. The CAN socket API has been designed as similar as possible to the TCP/IP protocols to allow programmers, familiar with network programming, to easily learn how to use CAN sockets.

Important: *python-can* versions before 2.2 had two different implementations named `socketcan_ctypes` and `socketcan_native`. These were removed in version 4.0.0 after a deprecation period.

4.17.1 Socketcan Quickstart

The CAN network driver provides a generic interface to setup, configure and monitor CAN devices. To configure bit-timing parameters use the program `ip`.

The virtual CAN driver (vcan)

The virtual CAN interfaces allow the transmission and reception of CAN frames without real CAN controller hardware. Virtual CAN network devices are usually named 'vcanX', like vcan0 vcan1 vcan2.

To create a virtual can interface using socketcan run the following:

```
sudo modprobe vcan
# Create a vcan network interface with a specific name
sudo ip link add dev vcan0 type vcan
sudo ip link set vcan0 up
```

Real Device

`vcan` should be substituted for `can` and `vcan0` should be substituted for `can0` if you are using real hardware. Setting the bitrate can also be done at the same time, for example to enable an existing `can0` interface with a bitrate of 1MB:

```
sudo ip link set can0 up type can bitrate 1000000
```

CAN over Serial / SLCAN

SLCAN adapters can be used directly via *CAN over Serial / SLCAN*, or via *SocketCAN* with some help from the `slcand` utility which can be found in the can-utils package.

To create a socketcan interface for an SLCAN adapter run the following:

```
slcand -f -o -c -s5 /dev/ttyAMA0
ip link set up slcan0
```

Names of the interfaces created by `slcand` match the `slcan\d+` regex. If a custom name is required, it can be specified as the last argument. E.g.:

```
slcand -f -o -c -s5 /dev/ttyAMA0 can0
ip link set up can0
```

PCAN

Kernels >= 3.4 supports the PCAN adapters natively via *SocketCAN*, so there is no need to install any drivers. The CAN interface can be brought like so:

```
sudo modprobe peak_usb
sudo modprobe peak_pci
sudo ip link set can0 up type can bitrate 500000
```

Intrepid

The Intrepid Control Systems, Inc provides several devices (e.g. ValueCAN) as well as Linux module and user-space daemon to make it possible to use them via SocketCAN.

Refer to below repositories for installation instructions:

- Intrepid kernel module
- Intrepid user-space daemon

Send Test Message

The can-utils library for Linux includes a *cansend* tool which is useful to send known payloads. For example to send a message on *vcan0*:

```
cansend vcan0 123#DEADBEEF
```

CAN Errors

A device may enter the "bus-off" state if too many errors occurred on the CAN bus. Then no more messages are received or sent. An automatic bus-off recovery can be enabled by setting the "restart-ms" to a non-zero value, e.g.:

```
sudo ip link set canX type can restart-ms 100
```

Alternatively, the application may realize the "bus-off" condition by monitoring CAN error frames and do a restart when appropriate with the command:

```
ip link set canX type can restart
```

Note that a restart will also create a CAN error frame.

List network interfaces

To reveal the newly created `can0` or a `vcan0` interface:

```
ifconfig
```

Display CAN statistics

```
ip -details -statistics link show vcan0
```

Network Interface Removal

To remove the network interface:

```
sudo ip link del vcan0
```

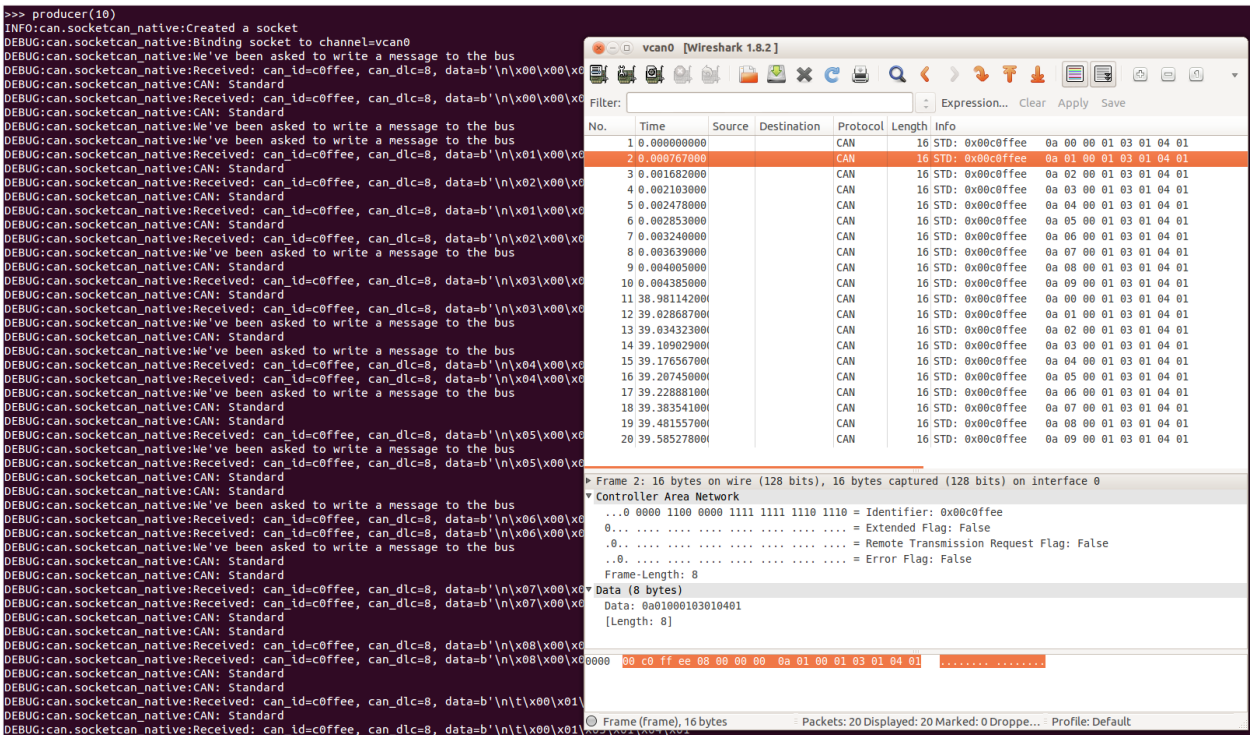
4.17.2 Wireshark

Wireshark supports socketcan and can be used to debug *python-can* messages. Fire it up and watch your new interface.

To spam a bus:

```
import time
import can

interface = 'socketcan'
channel = 'vcan0'

def producer(id):
    """:param id: Spam the bus with messages including the data id."""
    bus = can.Bus(channel=channel, interface=interface)
    for i in range(10):
        msg = can.Message(arbitration_id=0xc0ffee, data=[id, i, 0, 1, 3, 1, 4, 1], is_
↪extended_id=False)
        bus.send(msg)

    time.sleep(1)

producer(10)
```

With debugging turned right up this looks something like this:



The process to follow bus traffic is even easier:

```
for message in Bus(can_interface):
    print(message)
```

4.17.3 Reading and Timeouts

Reading a single CAN message off of the bus is simple with the `bus.recv()` function:

```
import can

bus = can.Bus(channel='vcan0', interface='socketcan')
message = bus.recv()
```

By default, this performs a blocking read, which means `bus.recv()` won't return until a CAN message shows up on the socket. You can optionally perform a blocking read with a timeout like this:

```
message = bus.recv(1.0)  # Timeout in seconds.

if message is None:
    print('Timeout occurred, no message.')
```

If you set the timeout to `0.0`, the read will be executed as non-blocking, which means `bus.recv(0.0)` will return immediately, either with a `Message` object or `None`, depending on whether data was available on the socket.

4.17.4 Filtering

The implementation features efficient filtering of can_id's. That filtering occurs in the kernel and is much much more efficient than filtering messages in Python.

4.17.5 Broadcast Manager

The `socketcan` interface implements thin wrappers to the linux *broadcast manager* socket api. This allows the cyclic transmission of CAN messages at given intervals. The overhead for periodic message sending is extremely low as all the heavy lifting occurs within the linux kernel.

The *BusABC* initialized for *socketcan* interface transparently handles scheduling of CAN messages to Linux BCM via *send_periodic()*:

```
with can.interface.Bus(interface="socketcan", channel="can0") as bus:
    task = bus.send_periodic(...)
```

More examples that uses *send_periodic()* are included in `python-can/examples/cyclic.py`.

The *task* object returned by *send_periodic()* can be used to halt, alter or cancel the periodic message task:

class can.interfaces.socketcan.**CyclicSendTask**(*bcm_socket*, *task_id*, *messages*, *period*, *duration=None*)

A SocketCAN cyclic send task supports:

- setting of a task duration
- modifying the data
- stopping then subsequent restarting of the task

Construct and *start()* a task.

Parameters

- **bcm_socket** (*socket*) – An open BCM socket on the desired CAN channel.
- **task_id** (*int*) – The identifier used to uniquely reference particular cyclic send task within Linux BCM.
- **messages** (*Sequence[Message] | Message*) – The messages to be sent periodically.
- **period** (*float*) – The rate in seconds at which to send the messages.
- **duration** (*float | None*) – Approximate duration in seconds to send the messages for.

modify_data(*messages*)

Update the contents of the periodically sent CAN messages by sending TX_SETUP message to Linux kernel.

The number of new cyclic messages to be sent must be equal to the original number of messages originally specified for this task.

Note: The messages must all have the same *arbitration_id* like the first message.

Parameters

messages (*Sequence[Message] | Message*) – The messages with the new *can.Message.data*.

Return type
None

`start()`

Restart a periodic task by sending TX_SETUP message to Linux kernel.

It verifies presence of the particular BCM task through sending TX_READ message to Linux kernel prior to scheduling.

Raises
`ValueError` – If the task referenced by `task_id` is already running.

Return type
None

`stop()`

Stop a task by sending TX_DELETE message to Linux kernel.

This will delete the entry for the transmission of the CAN-message with the specified `task_id` identifier. The message length for the command TX_DELETE is {[bcm_msg_head]} (only the header).

Return type
None

4.17.6 Buffer Sizes

Currently, the sending buffer size cannot be adjusted by this library. However, this issue describes how to change it via the command line/shell.

4.17.7 Bus

The *SocketcanBus* specializes *BusABC* to ensure usage of SocketCAN Linux API. The most important differences are:

- usage of SocketCAN BCM for periodic messages scheduling;
- filtering of CAN messages on Linux kernel level;
- usage of nanosecond timings from the kernel.

`class can.interfaces.socketcan.SocketcanBus(`*channel=''*, *receive_own_messages=False*, *local_loopback=True*, *fd=False*, *can_filters=None*, *ignore_rx_error_frames=False*, ***kwargs*`)`

A SocketCAN interface to CAN.

It implements *can.BusABC._detect_available_configs()* to search for available interfaces.

Creates a new socketcan bus.

If setting some socket options fails, an error will be printed but no exception will be thrown. This includes enabling:

- that own messages should be received,
- CAN-FD frames and
- error frames.

Parameters

- **channel** (*str*) – The can interface name with which to create this bus. An example channel would be 'vcan0' or 'can0'. An empty string '' will receive messages from all channels. In that case any sent messages must be explicitly addressed to a channel using *can.Message.channel*.
- **receive_own_messages** (*bool*) – If transmitted messages should also be received by this bus.
- **local_loopback** (*bool*) – If local loopback should be enabled on this bus. Please note that local loopback does not mean that messages sent on a socket will be readable on the same socket, they will only be readable on other open sockets on the same machine. More info can be read on the socketcan documentation: See https://www.kernel.org/doc/html/latest/networking/can.html#socketcan-local-loopback1
- **fd** (*bool*) – If CAN-FD frames should be supported.
- **can_filters** (*Sequence[CanFilter | CanFilterExtended] | None*) – See *can.BusABC.set_filters()*.
- **ignore_rx_error_frames** – If incoming error frames should be discarded.

RECV_LOGGING_LEVEL = 9

Log level for received messages

channel_info = 'unknown'

a string describing the underlying bus and/or channel

property filters: Sequence[CanFilter | CanFilterExtended] | None

Modify the filters of this bus. See *set_filters()* for details.

flush_tx_buffer()

Discard every message that may be queued in the output buffer(s).

Return type
None

property protocol: *CanProtocol*

Return the CAN protocol used by this bus instance.

This value is set at initialization time and does not change during the lifetime of a bus instance.

recv(*timeout=None*)

Block waiting for a message from the Bus.

Parameters
timeout (*float | None*) – seconds to wait for a message or None to wait indefinitely

Returns
None on timeout or a *Message* object.

Raises
CanOperationError – If an error occurred while reading

Return type
Message | None

send(*msg*, *timeout=None*)

Transmit a message to the CAN bus.

Parameters

- **msg** (Message) – A message object.

- **timeout** (*float* | *None*) – Wait up to this many seconds for the transmit queue to be ready. If not given, the call may fail immediately.

Raises
CanError – if the message could not be written.

Return type
None

send_periodic(*msgs*, *period*, *duration=None*, *store_task=True*, *modifier_callback=None*)

Start sending messages at a given period on this bus.

The task will be active until one of the following conditions are met:

- the (optional) duration expires
- the Bus instance goes out of scope
- the Bus instance is shutdown
- *stop_all_periodic_tasks()* is called
- the task's *stop()* method is called.

Parameters

- **msgs** (Message | *Sequence*[Message]) – Message(s) to transmit
- **period** (*float*) – Period in seconds between each message
- **duration** (*float* | *None*) – Approximate duration in seconds to continue sending messages. If no duration is provided, the task will continue indefinitely.
- **store_task** (*bool*) – If True (the default) the task will be attached to this Bus instance. Disable to instead manage tasks manually.
- **modifier_callback** (*Callable*[[Message], *None*] | *None*) – Function which should be used to modify each message's data before sending. The callback modifies the *data* of the message and returns None.

Returns
A started task instance. Note the task can be stopped (and depending on the backend modified) by calling the task's *stop()* method.

Return type
CyclicSendTaskABC

Note: Note the duration before the messages stop being sent may not be exactly the same as the duration specified by the user. In general the message will be sent at the given rate until at least **duration** seconds.

Note: For extremely long-running Bus instances with many short-lived tasks the default api with store_task==True may not be appropriate as the stopped tasks are still taking up memory as they are associated with the Bus instance.

---

set_filters(*filters=None*)

Apply filtering to all messages received by this Bus.

All messages that match at least one filter are returned. If *filters* is *None* or a zero length sequence, all messages are matched.

Calling without passing any filters will reset the applied filters to None.

Parameters

filters (*Sequence[CanFilter | CanFilterExtended] | None*) – A iterable of dictionaries each containing a "can_id", a "can_mask", and an optional "extended" key:

```
[{"can_id": 0x11, "can_mask": 0x21, "extended": False}]
```

A filter matches, when `<received_can_id> & can_mask == can_id & can_mask`. If `extended` is set as well, it only matches messages where `<received_is_extended> == extended`. Else it matches every messages based only on the arbitration ID and mask.

Return type

None

shutdown()

Stops all active periodic tasks and closes the socket.

Return type

None

property state: *BusState*

Return the current state of the hardware

stop_all_periodic_tasks(*remove_tasks=True*)

Stop sending any messages that were started using *send_periodic()*.

Note: The result is undefined if a single task throws an exception while being stopped.

Parameters

remove_tasks (*bool*) – Stop tracking the stopped tasks.

Return type

None

4.18 socketcand Interface

Socketcand is part of the Linux-CAN project, providing a Network-to-CAN bridge as a Linux damon. It implements a specific TCP/IP based communication protocol to transfer CAN frames and control commands.

The main advantage compared to UDP-based protocols (e.g. virtual interface) is, that TCP guarantees delivery and that the message order is kept.

Here is a small example dumping all can messages received by a socketcand daemon running on a remote Raspberry Pi:

```
import can

bus = can.interface.Bus(interface='socketcand', host="10.0.16.15", port=29536, channel=
↪"can0")

# loop until Ctrl-C
try:
  while True:
```

(continues on next page)

(continued from previous page)

```
    msg = bus.recv()
    print(msg)
except KeyboardInterrupt:
  pass
```

The output may look like this:

```
Timestamp: 1637791111.209224    ID: 000006fd    X Rx                DLC:  8    c4 10 e3␣
↪2d 96 ff 25 6b
Timestamp: 1637791111.233951    ID: 000001ad    X Rx                DLC:  4    4d 47 c7␣
↪64
Timestamp: 1637791111.409415    ID: 000005f7    X Rx                DLC:  8    86 de e6␣
↪0f 42 55 5d 39
Timestamp: 1637791111.434377    ID: 00000665    X Rx                DLC:  8    97 96 51␣
↪0f 23 25 fc 28
Timestamp: 1637791111.609763    ID: 0000031d    X Rx                DLC:  8    16 27 d8␣
↪3d fe d8 31 24
Timestamp: 1637791111.634630    ID: 00000587    X Rx                DLC:  8    4e 06 85␣
↪23 6f 81 2b 65
```

4.18.1 Bus

class can.interfaces.socketcand.**SocketCanDaemonBus**(*channel*, *host*, *port*, *tcp_tune=False*, *can_filters=None*, ***kwargs*)

Bases: *BusABC*

Connects to a CAN bus served by socketcand.

It will attempt to connect to the server for up to 10s, after which a TimeoutError exception will be thrown.

If the handshake with the socketcand server fails, a CanError exception is thrown.

Parameters

- **channel** – The can interface name served by socketcand. An example channel would be 'vcan0' or 'can0'.
- **host** – The host address of the socketcand server.
- **port** – The port of the socketcand server.
- **tcp_tune** – This tunes the TCP socket for low latency (TCP_NODELAY, and TCP_QUICKACK). This option is not available under windows.
- **can_filters** – See *can.BusABC.set_filters()*.

send(*msg*, *timeout=None*)

Transmit a message to the CAN bus.

Parameters

- **msg** – A message object.
- **timeout** – Ignored

shutdown()

Stops all active periodic tasks and closes the socket.

4.18.2 Socketcand Quickstart

The following section will show how to get the stuff installed on a Raspberry Pi with a MCP2515-based CAN interface, e.g. available from Waveshare. However, it will also work with any other socketcan device.

Install CAN Interface for a MCP2515 based interface on a Raspberry Pi

Add the following lines to `/boot/config.txt`. Please take care on the frequency of the crystal on your MCP2515 board:

```
dtparam=spi=on
dtoverlay=mcp2515-can0,oscillator=12000000,interrupt=25,spimaxfrequency=1000000
```

Reboot after `/boot/config.txt` has been modified.

Enable socketcan for can0

Create config file for systemd-networkd to start the socketcan interface automatically:

```
cat >/etc/systemd/network/80-can.network <<'EOT'
[Match]
Name=can0
[CAN]
BitRate=250K
RestartSec=100ms
EOT
```

Enable `systemd-networkd` on reboot and start it immediately (if it was not already startet):

```
sudo systemctl enable systemd-networkd
sudo systemctl start systemd-networkd
```

Build socketcand from source

```
# autoconf is needed to build socketcand
sudo apt-get install -y autoconf
# clone & build sources
git clone https://github.com/linux-can/socketcand.git
cd socketcand
./autogen.sh
./configure
make
```

Install socketcand

```
make install
```

Run socketcand

```
./socketcand -v -i can0
```

During start, socketcand will prompt its IP address and port it listens to:

```
Verbose output activated

Using network interface 'eth0'
Listen adress is 10.0.16.15
Broadcast adress is 10.0.255.255
creating broadcast thread...
binding socket to 10.0.16.15:29536
```

4.19 SYSTEC interface

Windows interface for the USBCAN devices supporting up to 2 channels based on the particular product. There is support for the devices also on Linux through the *SocketCAN* interface and for Windows using this `systec` interface.

4.19.1 Installation

The interface requires installation of the **USBCAN32.dll** library. Download and install the driver for specific SYSTEC device.

4.19.2 Supported devices

The interface supports following devices:

- GW-001 (obsolete),
- GW-002 (obsolete),
- Multiport CAN-to-USB G3,
- USB-CANmodul1 G3,
- USB-CANmodul2 G3,
- USB-CANmodul8 G3,
- USB-CANmodul16 G3,
- USB-CANmodul1 G4,
- USB-CANmodul2 G4.

4.19.3 Configuration

The simplest configuration would be:

```
interface = systec
channel = 0
```

Python-can will search for the first device found if not specified explicitly by the `device_number` parameter. The `interface` and `channel` are the only mandatory parameters. The interface supports two channels 0 and 1. The maximum number of entries in the receive and transmit buffer can be set by the parameters `rx_buffer_entries` and `tx_buffer_entries`, with default value 4096 set for both.

Optional parameters:

- `bitrate` (default 500000) Channel bitrate in bit/s
- `device_number` (default first device) The device number of the USB-CAN
- `rx_buffer_entries` (default 4096) The maximum number of entries in the receive buffer
- `tx_buffer_entries` (default 4096) The maximum number of entries in the transmit buffer
- `state` (default BusState.ACTIVE) BusState of the channel
- `receive_own_messages` (default False) If messages transmitted should also be received back

4.19.4 Bus

class can.interfaces.systec.ucanbus.**UcanBus**(*channel*, *can_filters=None*, ***kwargs*)

The CAN Bus implemented for the SYSTEC interface.

Parameters

- **channel** (*int*) – The Channel id to create this bus with.
- **can_filters** (*list*) – See *can.BusABC.set_filters()*.

Backend Configuration

Parameters

- **bitrate** (*int*) – Channel bitrate in bit/s. Default is 500000.
- **device_number** (*int*) – The device number of the USB-CAN. Valid values: 0 through 254. Special value 255 is reserved to detect the first connected device (should only be used, in case only one module is connected to the computer). Default is 255.
- **state** (can.bus.BusState) – BusState of the channel. Default is ACTIVE.
- **receive_own_messages** (*bool*) – If messages transmitted should also be received back. Default is False.
- **rx_buffer_entries** (*int*) – The maximum number of entries in the receive buffer. Default is 4096.
- **tx_buffer_entries** (*int*) – The maximum number of entries in the transmit buffer. Default is 4096.

Raises

- **ValueError** – If invalid input parameter were passed.
- ***CanInterfaceNotImplementedError*** – If the platform is not supported.

- *CanInitializationError* – If hardware or CAN interface initialization failed.

static create_filter(*extended*, *from_id*, *to_id*, *rtr_only*, *rtr_too*)

Calculates AMR and ACR using CAN-ID as parameter.

Parameters

- **extended** (*bool*) – if True parameters from_id and to_id contains 29-bit CAN-ID
- **from_id** (*int*) – first CAN-ID which should be received
- **to_id** (*int*) – last CAN-ID which should be received
- **rtr_only** (*bool*) – if True only RTR-Messages should be received, and rtr_too will be ignored
- **rtr_too** (*bool*) – if True CAN data frames and RTR-Messages should be received

Returns

Returns list with one filter containing a "can_id", a "can_mask" and "extended" key.

flush_tx_buffer()

Flushes the transmit buffer.

Raises

CanError – If flushing of the transmit buffer failed.

send(*msg*, *timeout=None*)

Sends one CAN message.

When a transmission timeout is set the firmware tries to send a message within this timeout. If it could not be sent the firmware sets the "auto delete" state. Within this state all transmit CAN messages for this channel will be deleted automatically for not blocking the other channel.

Parameters

- **msg** (can.Message) – The CAN message.
- **timeout** (*float*) – Transmit timeout in seconds (value 0 switches off the "auto delete")

Raises

CanOperationError – If the message could not be sent.

shutdown()

Shuts down all CAN interfaces and hardware interface.

property state

Return the current state of the hardware

4.19.5 Internals

Message filtering

The interface and driver supports only setting of one filter per channel. If one filter is requested, this is will be handled by the driver itself. If more than one filter is needed, these will be handled in Python code in the `recv` method. If a message does not match any of the filters, `recv()` will return None.

Periodic tasks

The driver supports periodic message sending but without the possibility to set the interval between messages. Therefore the handling of the periodic messages is done by the interface using the *ThreadBasedCyclicSendTask*.

4.20 USB2CAN Interface

The USB2CAN is a cheap CAN interface based on an ARM7 chip (STR750FV2). There is support for this device on Linux through the *SocketCAN* interface and for Windows using this `usb2can` interface.

Support though windows is achieved through a DLL very similar to the way the PCAN functions. The API is called CANAL (CAN Abstraction Layer) which is a separate project designed to be used with VSCP which is a socket like messaging system that is not only cross platform but also supports other types of devices.

4.20.1 Installation

1. To install on Windows download the USB2CAN Windows driver. It is compatible with XP, Vista, Win7, Win8/8.1. (Written against driver version v1.0.2.1)
2. Install the appropriate version of pywin32 (win32com)
3. Download the USB2CAN CANAL DLL from the USB2CAN website. Place this in either the same directory you are running usb2can.py from or your DLL folder in your python install. Note that only a 32-bit version is currently available, so this only works in a 32-bit Python environment.

4.20.2 Internals

This interface originally written against CANAL DLL version `v1.0.6`.

Interface Layout

- **`usb2canabstractionlayer.py`**
  This file is only a wrapper for the CANAL API that the interface expects. There are also a couple of constants here to try and make dealing with the bitwise operations for flag setting a little easier. Other than that this is only the CANAL API. If a programmer wanted to work with the API directly this is the file that allows you to do this. The CANAL project does not provide this wrapper and normally must be accessed with C.
- **`usb2canInterface.py`**
  This file provides the translation to and from the python-can library to the CANAL API. This is where all the logic is and setup code is. Most issues if they are found will be either found here or within the DLL that is provided
- **`serial_selector.py`**
  See the section below for the reason for adding this as it is a little odd. What program does is if a serial number is not provided to the usb2canInterface file this program does WMI (Windows Management Instrumentation) calls to try and figure out what device to connect to. It then returns the serial number of the device. Currently it is not really smart enough to figure out what to do if there are multiple devices. This needs to be changed if people are using more than one interface.

4.20.3 Interface Specific Items

There are a few things that are kinda strange about this device and are not overly obvious about the code or things that are not done being implemented in the DLL.

1. You need the Serial Number to connect to the device under Windows. This is part of the "setup string" that configures the device. There are a few options for how to get this.
    1. Use `usb2canWin.py` to find the serial number.
    2. Look on the device and enter it either through a prompt/barcode scanner/hardcode it. (Not recommended)
    3. Reprogram the device serial number to something and do that for all the devices you own. (Really Not Recommended, can no longer use multiple devices on one computer)
2. In `usb2canabstractionlayer.py` there is a structure called `CanalMsg` which has a unsigned byte array of size 8. In the `usb2canInterface` file it passes in an unsigned byte array of size 8 also which if you pass less than 8 bytes in it stuffs it with extra zeros. So if the data `"01020304"` is sent the message would look like `"0102030400000000"`.

    There is also a part of this structure called `sizeData` which is the actual length of the data that was sent not the stuffed message (in this case would be 4). What then happens is although a message of size 8 is sent to the device only the first 4 bytes of information would be sent. This is done because the DLL expects a length of 8 and nothing else. So to make it compatible that has to be sent through the wrapper. If `usb2canInterface` sent an array of length 4 with sizeData of 4 as well the array would throw an incompatible data type error.
3. The masking features have not been implemented currently in the CANAL interface in the version currently on the USB2CAN website.

Warning: Currently message filtering is not implemented. Contributions are most welcome!

4.20.4 Bus

class can.interfaces.usb2can.**Usb2canBus**(*channel=None*, *dll='usb2can.dll'*, *flags=8*, **_*, *bitrate=500000*, *serial=None*, ***kwargs*)

Interface to a USB2CAN Bus.

This interface only works on Windows. Please use socketcan on Linux.

Parameters

- **channel** (*str | None*) – The device's serial number. If not provided, Windows Management Instrumentation will be used to identify the first such device.
- **bitrate** (*int*) – Bitrate of channel in bit/s. Values will be limited to a maximum of 1000 Kb/s. Default is 500 Kbs
- **flags** (*int*) – Flags to directly pass to open function of the usb2can abstraction layer.
- **dll** (*str*) – Path to the DLL with the CANAL API to load Defaults to 'usb2can.dll'
- **serial** (*str | None*) – Alias for *channel* that is provided for legacy reasons. If both *serial* and *channel* are set, *serial* will be used and channel will be ignored.

Construct and open a CAN bus instance of the specified type.

Subclasses should call though this method with all given parameters as it handles generic tasks like applying filters.

Parameters

- **channel** (*str | None*) – The can interface identifier. Expected type is backend dependent.
- **can_filters** – See *set_filters()* for details.
- **kwargs** (*dict*) – Any backend dependent configurations are passed in this dictionary
- **dll** (*str*) –
- **flags** (*int*) –
- **bitrate** (*int*) –
- **serial** (*str | None*) –

Raises

- **ValueError** – If parameters are out of range
- ***CanInterfaceNotImplementedError*** – If the driver cannot be accessed
- ***CanInitializationError*** – If the bus cannot be initialized

4.20.5 Exceptions

exception can.interfaces.usb2can.usb2canabstractionlayer.**CanalError**(*value*)

An enumeration.

4.20.6 Miscellaneous

class can.interfaces.usb2can.**Usb2CanAbstractionLayer**(*dll='usb2can.dll'*)

A low level wrapper around the usb2can library.

Documentation: http://www.8devices.com/media/products/usb2can/downloads/CANAL_API.pdf

Parameters
dll (*str | os.PathLike[str]*) – the path to the usb2can DLL to load

Raises
CanInterfaceNotImplementedError – if the DLL could not be loaded

blocking_receive(*handle*, *msg*, *timeout*)

Return type
CanalError

blocking_send(*handle*, *msg*, *timeout*)

Return type
CanalError

close(*handle*)

Return type
CanalError

get_library_version()

get_statistics(*handle*, *statistics*)

Return type
CanalError

get_status(*handle*, *status*)

Return type
CanalError

get_vendor_string()

get_version()

open(*configuration*, *flags*)

Opens a CAN connection using *CanalOpen()*.

Parameters

- **configuration** (*str*) – the configuration: "device_id; baudrate"
- **flags** (*int*) – the flags to be set

Returns
Valid handle for CANAL API functions on success

Raises
CanInterfaceNotImplementedError – if any error occurred

receive(*handle*, *msg*)

Return type
CanalError

send(*handle*, *msg*)

Return type
CanalError

4.21 Vector

This interface adds support for CAN controllers by Vector. Only Windows is supported.

4.21.1 Configuration

By default this library uses the channel configuration for CANalyzer. To use a different application, open **Vector Hardware Configuration** program and create a new application and assign the channels you may want to use. Specify the application name as `app_name='Your app name'` when constructing the bus or in a config file.

Channel should be given as a list of channels starting at 0.

Here is an example configuration file connecting to CAN 1 and CAN 2 for an application named "python-can":

```
[default]
interface = vector
channel = 0, 1
app_name = python-can
```

4.21.2 VectorBus

class can.interfaces.vector.**VectorBus**(*channel*, *can_filters=None*, *poll_interval=0.01*, *receive_own_messages=False*, *timing=None*, *bitrate=None*, *rx_queue_size=16384*, *app_name='CANalyzer'*, *serial=None*, *fd=False*, *data_bitrate=None*, *sjw_abr=2*, *tseg1_abr=6*, *tseg2_abr=3*, *sjw_dbr=2*, *tseg1_dbr=6*, *tseg2_dbr=3*, ***kwargs*)

Bases: *BusABC*

The CAN Bus implemented for the Vector interface.

Parameters

- **channel** (*int | Sequence[int] | str*) – The channel indexes to create this bus with. Can also be a single integer or a comma separated string.
- **can_filters** (*Sequence[CanFilter | CanFilterExtended] | None*) – See *can.BusABC*.
- **receive_own_messages** (*bool*) – See *can.BusABC*.
- **timing** (*BitTiming | BitTimingFd | None*) – An instance of *BitTiming* or *BitTimingFd* to specify the bit timing parameters for the VectorBus interface. The *f_clock* value of the timing instance must be set to 8_000_000 (8MHz) or 16_000_000 (16MHz) for CAN 2.0 or 80_000_000 (80MHz) for CAN FD. If this parameter is provided, it takes precedence over all other timing-related parameters. Otherwise, the bit timing can be specified using the following parameters: *bitrate* for standard CAN or *fd*, *data_bitrate*, *sjw_abr*, *tseg1_abr*, *tseg2_abr*, *sjw_dbr*, *tseg1_dbr*, and *tseg2_dbr* for CAN FD.
- **poll_interval** (*float*) – Poll interval in seconds.
- **bitrate** (*int | None*) – Bitrate in bits/s.
- **rx_queue_size** (*int*) – Number of messages in receive queue (power of 2). CAN: range *16…32768* CAN-FD: range *8192…524288*
- **app_name** (*str | None*) – Name of application in *Vector Hardware Config*. If set to *None*, the channel should be a global channel index.
- **serial** (*int | None*) – Serial number of the hardware to be used. If set, the channel parameter refers to the channels ONLY on the specified hardware. If set, the *app_name* does not have to be previously defined in *Vector Hardware Config*.
- **fd** (*bool*) – If CAN-FD frames should be supported.
- **data_bitrate** (*int | None*) – Which bitrate to use for data phase in CAN FD. Defaults to arbitration bitrate.
- **sjw_abr** (*int*) – Bus timing value sample jump width (arbitration).
- **tseg1_abr** (*int*) – Bus timing value tseg1 (arbitration)
- **tseg2_abr** (*int*) – Bus timing value tseg2 (arbitration)
- **sjw_dbr** (*int*) – Bus timing value sample jump width (data)
- **tseg1_dbr** (*int*) – Bus timing value tseg1 (data)
- **tseg2_dbr** (*int*) – Bus timing value tseg2 (data)
- **kwargs** (*Any*) –

Raises

- *CanInterfaceNotImplementedError* – If the current operating system is not supported or the driver could not be loaded.
- *CanInitializationError* – If the bus could not be set up. This may or may not be a *VectorInitializationError*.

send(*msg*, *timeout=None*)

Transmit a message to the CAN bus.

Override this method to enable the transmit path.

Parameters

- **msg** (*Message*) – A message object.
- **timeout** (*float | None*) – If > 0, wait up to this many seconds for message to be ACK'ed or for transmit queue to be ready depending on driver implementation. If timeout is exceeded, an exception will be raised. Might not be supported by all interfaces. None blocks indefinitely.

Raises

CanOperationError – If an error occurred while sending

Return type

None

flush_tx_buffer()

Flush the TX buffer of the bus.

Implementation does not use function `xlCanFlushTransmitQueue` of the XL driver, as it works only for XL family devices.

Warning: Using this function will flush the queue and send a high voltage message (ID = 0, DLC = 0, no data).

Return type

None

shutdown()

Called to carry out any interface specific cleanup required in shutting down a bus.

This method can be safely called multiple times.

Return type

None

static popup_vector_hw_configuration(*wait_for_finish=0*)

Open vector hardware configuration window.

Parameters

wait_for_finish (*int*) – Time to wait for user input in milliseconds.

Return type

None

static get_application_config(*app_name*, *app_channel*)

Retrieve information for an application in Vector Hardware Configuration.

Parameters

- **app_name** (*str*) – The name of the application.
- **app_channel** (*int*) – The channel of the application.

Returns
Returns a tuple of the hardware type, the hardware index and the hardware channel.

Raises
can.interfaces.vector.VectorInitializationError – If the application name does not exist in the Vector hardware configuration.

Return type
Tuple[int | XL_HardwareType, int, int]

static set_application_config(*app_name*, *app_channel*, *hw_type*, *hw_index*, *hw_channel*, ***kwargs*)

Modify the application settings in Vector Hardware Configuration.

This method can also be used with a channel config dictionary:

```
import can
from can.interfaces.vector import VectorBus

configs = can.detect_available_configs(interfaces=['vector'])
cfg = configs[0]
VectorBus.set_application_config(app_name="MyApplication", app_channel=0, **cfg)
```

Parameters

- **app_name** (*str*) – The name of the application. Creates a new application if it does not exist yet.
- **app_channel** (*int*) – The channel of the application.
- **hw_type** (*int* | XL_HardwareType) – The hardware type of the interface. E.g XL_HardwareType.XL_HWTYPE_VIRTUAL
- **hw_index** (*int*) – The index of the interface if multiple interface with the same hardware type are present.
- **hw_channel** (*int*) – The channel index of the interface.
- **kwargs** (*Any*) –

Raises
can.interfaces.vector.VectorInitializationError – If the application name does not exist in the Vector hardware configuration.

Return type
None

set_filters(*filters=None*)

Apply filtering to all messages received by this Bus.

All messages that match at least one filter are returned. If *filters* is *None* or a zero length sequence, all messages are matched.

Calling without passing any filters will reset the applied filters to `None`.

Parameters
filters (*Sequence[CanFilter | CanFilterExtended] | None*) – A iterable of dictionaries each containing a "can_id", a "can_mask", and an optional "extended" key:

```
[{"can_id": 0x11, "can_mask": 0x21, "extended": False}]
```

A filter matches, when `<received_can_id> & can_mask == can_id & can_mask`. If `extended` is set as well, it only matches messages where `<received_is_extended> == extended`. Else it matches every messages based only on the arbitration ID and mask.

Return type
None

recv(*timeout=None*)

Block waiting for a message from the Bus.

Parameters
timeout (*float | None*) – seconds to wait for a message or None to wait indefinitely

Returns
None on timeout or a *Message* object.

Raises
CanOperationError – If an error occurred while reading

Return type
Message | None

send_periodic(*msgs*, *period*, *duration=None*, *store_task=True*, *modifier_callback=None*)

Start sending messages at a given period on this bus.

The task will be active until one of the following conditions are met:

- the (optional) duration expires
- the Bus instance goes out of scope
- the Bus instance is shutdown
- *stop_all_periodic_tasks()* is called
- the task's *stop()* method is called.

Parameters

- **msgs** (*Message | Sequence[Message]*) – Message(s) to transmit
- **period** (*float*) – Period in seconds between each message
- **duration** (*float | None*) – Approximate duration in seconds to continue sending messages. If no duration is provided, the task will continue indefinitely.
- **store_task** (*bool*) – If True (the default) the task will be attached to this Bus instance. Disable to instead manage tasks manually.
- **modifier_callback** (*Callable[[Message], None] | None*) – Function which should be used to modify each message's data before sending. The callback modifies the *data* of the message and returns None.

Returns
A started task instance. Note the task can be stopped (and depending on the backend modified) by calling the task's *stop()* method.

Return type
CyclicSendTaskABC

Note: Note the duration before the messages stop being sent may not be exactly the same as the duration specified by the user. In general the message will be sent at the given rate until at least **duration** seconds.

Note: For extremely long-running Bus instances with many short-lived tasks the default api with `store_task==True` may not be appropriate as the stopped tasks are still taking up memory as they are associated with the Bus instance.

stop_all_periodic_tasks(*remove_tasks=True*)

Stop sending any messages that were started using *send_periodic()*.

Note: The result is undefined if a single task throws an exception while being stopped.

Parameters
remove_tasks (*bool*) – Stop tracking the stopped tasks.

Return type
None

4.21.3 Exceptions

exception can.interfaces.vector.**VectorError**(*error_code*, *error_string*, *function*)

Bases: *CanError*

exception can.interfaces.vector.**VectorInitializationError**(*error_code*, *error_string*, *function*)

Bases: *VectorError*, *CanInitializationError*

exception can.interfaces.vector.**VectorOperationError**(*error_code*, *error_string*, *function*)

Bases: *VectorError*, *CanOperationError*

4.21.4 Miscellaneous

can.interfaces.vector.**get_channel_configs**()

Read channel properties from Vector XL API.

Return type
List[VectorChannelConfig]

class can.interfaces.vector.**VectorChannelConfig**(*name*, *hw_type*, *hw_index*, *hw_channel*, *channel_index*, *channel_mask*, *channel_capabilities*, *channel_bus_capabilities*, *is_on_bus*, *connected_bus_type*, *bus_params*, *serial_number*, *article_number*, *transceiver_name*)

Bases: NamedTuple

NamedTuple which contains the channel properties from Vector XL API.

Parameters

- **name** (*str*) –

- **hw_type** (*int* | XL_HardwareType) –
- **hw_index** (*int*) –
- **hw_channel** (*int*) –
- **channel_index** (*int*) –
- **channel_mask** (*int*) –
- **channel_capabilities** (XL_ChannelCapabilities) –
- **channel_bus_capabilities** (XL_BusCapabilities) –
- **is_on_bus** (*bool*) –
- **connected_bus_type** (XL_BusTypes) –
- **bus_params** (VectorBusParams | *None*) –
- **serial_number** (*int*) –
- **article_number** (*int*) –
- **transceiver_name** (*str*) –

class can.interfaces.vector.canlib.**VectorBusParams**(*bus_type*, *can*, *canfd*)

Bases: NamedTuple

Parameters

- **bus_type** (XL_BusTypes) –
- **can** (VectorCanParams) –
- **canfd** (VectorCanFdParams) –

class can.interfaces.vector.canlib.**VectorCanParams**(*bitrate*, *sjw*, *tseg1*, *tseg2*, *sam*, *output_mode*, *can_op_mode*)

Bases: NamedTuple

Parameters

- **bitrate** (*int*) –
- **sjw** (*int*) –
- **tseg1** (*int*) –
- **tseg2** (*int*) –
- **sam** (*int*) –
- **output_mode** (XL_OutputMode) –
- **can_op_mode** (XL_CANFD_BusParams_CanOpMode) –

class can.interfaces.vector.canlib.**VectorCanFdParams**(*bitrate*, *data_bitrate*, *sjw_abr*, *tseg1_abr*, *tseg2_abr*, *sam_abr*, *sjw_dbr*, *tseg1_dbr*, *tseg2_dbr*, *output_mode*, *can_op_mode*)

Bases: NamedTuple

Parameters

- **bitrate** (*int*) –
- **data_bitrate** (*int*) –
- **sjw_abr** (*int*) –

- **tseg1_abr** (*int*) –
- **tseg2_abr** (*int*) –
- **sam_abr** (*int*) –
- **sjw_dbr** (*int*) –
- **tseg1_dbr** (*int*) –
- **tseg2_dbr** (*int*) –
- **output_mode** (XL_OutputMode) –
- **can_op_mode** (XL_CANFD_BusParams_CanOpMode) –

class can.interfaces.vector.xldefine.**XL_HardwareType**(*value*)

Bases: IntEnum

An enumeration.

XL_HWTYPE_NONE = 0

XL_HWTYPE_VIRTUAL = 1

XL_HWTYPE_CANCARDX = 2

XL_HWTYPE_CANAC2PCI = 6

XL_HWTYPE_CANCARDY = 12

XL_HWTYPE_CANCARDXL = 15

XL_HWTYPE_CANCASEXL = 21

XL_HWTYPE_CANCASEXL_LOG_OBSOLETE = 23

XL_HWTYPE_CANBOARDXL = 25

XL_HWTYPE_CANBOARDXL_PXI = 27

XL_HWTYPE_VN2600 = 29

XL_HWTYPE_VN2610 = 29

XL_HWTYPE_VN3300 = 37

XL_HWTYPE_VN3600 = 39

XL_HWTYPE_VN7600 = 41

XL_HWTYPE_CANCARDXLE = 43

XL_HWTYPE_VN8900 = 45

XL_HWTYPE_VN8950 = 47

XL_HWTYPE_VN2640 = 53

XL_HWTYPE_VN1610 = 55

XL_HWTYPE_VN1630 = 57

```
XL_HWTYPE_VN1640 = 59

XL_HWTYPE_VN8970 = 61

XL_HWTYPE_VN1611 = 63

XL_HWTYPE_VN5240 = 64

XL_HWTYPE_VN5610 = 65

XL_HWTYPE_VN5620 = 66

XL_HWTYPE_VN7570 = 67

XL_HWTYPE_VN5650 = 68

XL_HWTYPE_IPCLIENT = 69

XL_HWTYPE_VN5611 = 70

XL_HWTYPE_IPSERVER = 71

XL_HWTYPE_VN5612 = 72

XL_HWTYPE_VX1121 = 73

XL_HWTYPE_VN5601 = 74

XL_HWTYPE_VX1131 = 75

XL_HWTYPE_VT6204 = 77

XL_HWTYPE_VN1630_LOG = 79

XL_HWTYPE_VN7610 = 81

XL_HWTYPE_VN7572 = 83

XL_HWTYPE_VN8972 = 85

XL_HWTYPE_VN0601 = 87

XL_HWTYPE_VN5640 = 89

XL_HWTYPE_VX0312 = 91

XL_HWTYPE_VH6501 = 94

XL_HWTYPE_VN8800 = 95

XL_HWTYPE_IPCL8800 = 96

XL_HWTYPE_IPSRV8800 = 97

XL_HWTYPE_CSMCAN = 98

XL_HWTYPE_VN5610A = 101

XL_HWTYPE_VN7640 = 102

XL_HWTYPE_VX1135 = 104
```

XL_HWTYPE_VN4610 = 105

XL_HWTYPE_VT6306 = 107

XL_HWTYPE_VT6104A = 108

XL_HWTYPE_VN5430 = 109

XL_HWTYPE_VTSSERVICE = 110

XL_HWTYPE_VN1530 = 112

XL_HWTYPE_VN1531 = 113

XL_HWTYPE_VX1161A = 114

XL_HWTYPE_VX1161B = 115

XL_HWTYPE_VGNSS = 116

XL_HWTYPE_VXLAPINIC = 118

XL_MAX_HWTYPE = 120

class can.interfaces.vector.xldefine.**XL_ChannelCapabilities**(*value*)

Bases: IntFlag

An enumeration.

XL_CHANNEL_FLAG_TIME_SYNC_RUNNING = 1

XL_CHANNEL_FLAG_NO_HWSYNC_SUPPORT = 1024

XL_CHANNEL_FLAG_SPDIF_CAPABLE = 16384

XL_CHANNEL_FLAG_CANFD_BOSCH_SUPPORT = 536870912

XL_CHANNEL_FLAG_CMACTLICENSE_SUPPORT = 1073741824

XL_CHANNEL_FLAG_CANFD_ISO_SUPPORT = 2147483648

class can.interfaces.vector.xldefine.**XL_BusCapabilities**(*value*)

Bases: IntFlag

An enumeration.

XL_BUS_COMPATIBLE_CAN = 1

XL_BUS_ACTIVE_CAP_CAN = 65536

XL_BUS_COMPATIBLE_LIN = 2

XL_BUS_ACTIVE_CAP_LIN = 131072

XL_BUS_COMPATIBLE_FLEXRAY = 4

XL_BUS_ACTIVE_CAP_FLEXRAY = 262144

XL_BUS_COMPATIBLE_MOST = 16

XL_BUS_ACTIVE_CAP_MOST = 1048576

XL_BUS_COMPATIBLE_DAIO = 64

XL_BUS_ACTIVE_CAP_DAIO = 4194304

XL_BUS_COMPATIBLE_J1708 = 256

XL_BUS_ACTIVE_CAP_J1708 = 16777216

XL_BUS_COMPATIBLE_KLINE = 2048

XL_BUS_ACTIVE_CAP_KLINE = 134217728

XL_BUS_COMPATIBLE_ETHERNET = 4096

XL_BUS_ACTIVE_CAP_ETHERNET = 268435456

XL_BUS_COMPATIBLE_A429 = 8192

XL_BUS_ACTIVE_CAP_A429 = 536870912

class can.interfaces.vector.xldefine.**XL_BusTypes**(*value*)

Bases: IntFlag

An enumeration.

XL_BUS_TYPE_NONE = 0

XL_BUS_TYPE_CAN = 1

XL_BUS_TYPE_LIN = 2

XL_BUS_TYPE_FLEXRAY = 4

XL_BUS_TYPE_AFDX = 8

XL_BUS_TYPE_MOST = 16

XL_BUS_TYPE_DAIO = 64

XL_BUS_TYPE_J1708 = 256

XL_BUS_TYPE_KLINE = 2048

XL_BUS_TYPE_ETHERNET = 4096

XL_BUS_TYPE_A429 = 8192

class can.interfaces.vector.xldefine.**XL_OutputMode**(*value*)

Bases: IntEnum

An enumeration.

XL_OUTPUT_MODE_SILENT = 0

XL_OUTPUT_MODE_NORMAL = 1

XL_OUTPUT_MODE_TX_OFF = 2

XL_OUTPUT_MODE_SJA_1000_SILENT = 3

class can.interfaces.vector.xldefine.**XL_CANFD_BusParams_CanOpMode**(*value*)

Bases: IntFlag

An enumeration.

XL_BUS_PARAMS_CANOPMODE_CAN20 = 1

XL_BUS_PARAMS_CANOPMODE_CANFD = 2

XL_BUS_PARAMS_CANOPMODE_CANFD_NO_ISO = 8

class can.interfaces.vector.xldefine.**XL_Status**(*value*)

Bases: IntEnum

An enumeration.

XL_SUCCESS = 0

XL_PENDING = 1

XL_ERR_QUEUE_IS_EMPTY = 10

XL_ERR_QUEUE_IS_FULL = 11

XL_ERR_TX_NOT_POSSIBLE = 12

XL_ERR_NO_LICENSE = 14

XL_ERR_WRONG_PARAMETER = 101

XL_ERR_TWICE_REGISTER = 110

XL_ERR_INVALID_CHAN_INDEX = 111

XL_ERR_INVALID_ACCESS = 112

XL_ERR_PORT_IS_OFFLINE = 113

XL_ERR_CHAN_IS_ONLINE = 116

XL_ERR_NOT_IMPLEMENTED = 117

XL_ERR_INVALID_PORT = 118

XL_ERR_HW_NOT_READY = 120

XL_ERR_CMD_TIMEOUT = 121

XL_ERR_CMD_HANDLING = 122

XL_ERR_HW_NOT_PRESENT = 129

XL_ERR_NOTIFY_ALREADY_ACTIVE = 131

XL_ERR_INVALID_TAG = 132

XL_ERR_INVALID_RESERVED_FLD = 133

XL_ERR_INVALID_SIZE = 134

XL_ERR_INSUFFICIENT_BUFFER = 135

```
XL_ERR_ERROR_CRC = 136

XL_ERR_BAD_EXE_FORMAT = 137

XL_ERR_NO_SYSTEM_RESOURCES = 138

XL_ERR_NOT_FOUND = 139

XL_ERR_INVALID_ADDRESS = 140

XL_ERR_REQ_NOT_ACCEP = 141

XL_ERR_INVALID_LEVEL = 142

XL_ERR_NO_DATA_DETECTED = 143

XL_ERR_INTERNAL_ERROR = 144

XL_ERR_UNEXP_NET_ERR = 145

XL_ERR_INVALID_USER_BUFFER = 146

XL_ERR_INVALID_PORT_ACCESS_TYPE = 147

XL_ERR_NO_RESOURCES = 152

XL_ERR_WRONG_CHIP_TYPE = 153

XL_ERR_WRONG_COMMAND = 154

XL_ERR_INVALID_HANDLE = 155

XL_ERR_RESERVED_NOT_ZERO = 157

XL_ERR_INIT_ACCESS_MISSING = 158

XL_ERR_WRONG_VERSION = 160

XL_ERR_CANNOT_OPEN_DRIVER = 201

XL_ERR_WRONG_BUS_TYPE = 202

XL_ERR_DLL_NOT_FOUND = 203

XL_ERR_INVALID_CHANNEL_MASK = 204

XL_ERR_NOT_SUPPORTED = 205

XL_ERR_CONNECTION_BROKEN = 210

XL_ERR_CONNECTION_CLOSED = 211

XL_ERR_INVALID_STREAM_NAME = 212

XL_ERR_CONNECTION_FAILED = 213

XL_ERR_STREAM_NOT_FOUND = 214

XL_ERR_STREAM_NOT_CONNECTED = 215

XL_ERR_QUEUE_OVERRUN = 216
```

XL_ERROR = 255

XL_ERR_INVALID_DLC = 513

XL_ERR_INVALID_CANID = 514

XL_ERR_INVALID_FDFLAG_MODE20 = 515

XL_ERR_EDL_RTR = 516

XL_ERR_EDL_NOT_SET = 517

XL_ERR_UNKNOWN_FLAG = 518

Additional interface types can be added via the *Plugin Interface*, or by installing a third party package that utilises the *Plugin Interface*.

CHAPTER

FIVE

VIRTUAL INTERFACES

There are quite a few implementations for CAN networks that do not require physical CAN hardware. The built in virtual interfaces are:

5.1 Virtual

The virtual interface can be used as a way to write OS and driver independent tests. Any *VirtualBus* instances connecting to the same channel (from within the same Python process) will receive each others messages.

If messages shall be sent across process or host borders, consider using the *Multicast IP Interface* and refer to *Virtual Interfaces* for a comparison and general discussion of different virtual interfaces.

5.1.1 Example

```
import can

bus1 = can.interface.Bus('test', interface='virtual')
bus2 = can.interface.Bus('test', interface='virtual')

msg1 = can.Message(arbitration_id=0xabcde, data=[1,2,3])
bus1.send(msg1)
msg2 = bus2.recv()

#assert msg1 == msg2
assert msg1.arbitration_id == msg2.arbitration_id
assert msg1.data == msg2.data
assert msg1.timestamp != msg2.timestamp
```

```
import can

bus1 = can.interface.Bus('test', interface='virtual', preserve_timestamps=True)
bus2 = can.interface.Bus('test', interface='virtual')

msg1 = can.Message(timestamp=1639740470.051948, arbitration_id=0xabcde, data=[1,2,3])

# Messages sent on bus1 will have their timestamps preserved when received
# on bus2
bus1.send(msg1)
msg2 = bus2.recv()
```

(continues on next page)

(continued from previous page)

```
assert msg1.arbitration_id == msg2.arbitration_id
assert msg1.data == msg2.data
assert msg1.timestamp == msg2.timestamp

# Messages sent on bus2 will not have their timestamps preserved when
# received on bus1
bus2.send(msg1)
msg3 = bus1.recv()

assert msg1.arbitration_id == msg3.arbitration_id
assert msg1.data == msg3.data
assert msg1.timestamp != msg3.timestamp
```

5.1.2 Bus Class Documentation

class can.interfaces.virtual.**VirtualBus**(*channel=None*, *receive_own_messages=False*, *rx_queue_size=0*, *preserve_timestamps=False*, *protocol=CanProtocol.CAN_20*, ***kwargs*)

A virtual CAN bus using an internal message queue. It can be used for example for testing.

In this interface, a channel is an arbitrary object used as an identifier for connected buses.

Implements *can.BusABC._detect_available_configs()*; see *_detect_available_configs()* for how it behaves here.

Note: The timeout when sending a message applies to each receiver individually. This means that sending can block up to 5 seconds if a message is sent to 5 receivers with the timeout set to 1.0.

Warning: This interface guarantees reliable delivery and message ordering, but does *not* implement rate limiting or ID arbitration/prioritization under high loads. Please refer to the section *Virtual Interfaces* for more information on this and a comparison to alternatives.

The constructed instance has access to the bus identified by the channel parameter. It is able to see all messages transmitted on the bus by virtual instances constructed with the same channel identifier.

Parameters

- **channel** (*Any*) – The channel identifier. This parameter can be an arbitrary value. The bus instance will be able to see messages from other virtual bus instances that were created with the same value.
- **receive_own_messages** (*bool*) – If set to True, sent messages will be reflected back on the input queue.
- **rx_queue_size** (*int*) – The size of the reception queue. The reception queue stores messages until they are read. If the queue reaches its capacity, it will start dropping the oldest messages to make room for new ones. If set to 0, the queue has an infinite capacity. Be aware that this can cause memory leaks if messages are read with a lower frequency than they arrive on the bus.

- **preserve_timestamps** (*bool*) – If set to True, messages transmitted via *send()* will keep the timestamp set in the *Message* instance. Otherwise, the timestamp value will be replaced with the current system time.
- **protocol** (CanProtocol) – The protocol implemented by this bus instance. The value does not affect the operation of the bus instance and can be set to an arbitrary value for testing purposes.
- **kwargs** (*Any*) – Additional keyword arguments passed to the parent constructor.

static _detect_available_configs()

Returns all currently used channels as well as one other currently unused channel.

Note: This method will run into problems if thousands of autodetected buses are used at once.

Return type
List[*AutoDetectedConfig*]

send(*msg*, *timeout=None*)

Transmit a message to the CAN bus.

Override this method to enable the transmit path.

Parameters

- **msg** (Message) – A message object.
- **timeout** (*float | None*) – If > 0, wait up to this many seconds for message to be ACK'ed or for transmit queue to be ready depending on driver implementation. If timeout is exceeded, an exception will be raised. Might not be supported by all interfaces. None blocks indefinitely.

Raises
CanOperationError – If an error occurred while sending

Return type
None

shutdown()

Called to carry out any interface specific cleanup required in shutting down a bus.

This method can be safely called multiple times.

Return type
None

5.2 Multicast IP Interface

This module implements transport of CAN and CAN FD messages over UDP via Multicast IPv4 and IPv6. This virtual interface allows for communication between multiple processes and even hosts. This differentiates it from the *Virtual* interface, which can only passes messages within a single process but does not require a network stack.

It runs on UDP to have the lowest possible latency (as opposed to using TCP), and because normal IP multicast is inherently unreliable, as the recipients are unknown. This enables ad-hoc networks that do not require a central server but is also a so-called *unreliable network*. In practice however, local area networks (LANs) should most often be sufficiently reliable for this interface to function properly.

Note: For an overview over the different virtual buses in this library and beyond, please refer to the section *Virtual Interfaces*. It also describes important limitations of this interface.

Please refer to the *Bus class documentation* below for configuration options and useful resources for specifying multicast IP addresses.

5.2.1 Supported Platforms

It should work on most Unix systems (including Linux with kernel 2.6.22+ and macOS) but currently not on Windows.

5.2.2 Example

This example should print a single line indicating that a CAN message was successfully sent from `bus_1` to `bus_2`:

```
import time
import can
from can.interfaces.udp_multicast import UdpMulticastBus

# The bus can be created using the can.Bus wrapper class or using UdpMulticastBus␣
↪directly
with can.Bus(channel=UdpMulticastBus.DEFAULT_GROUP_IPv6, interface='udp_multicast') as␣
↪bus_1, \
        UdpMulticastBus(channel=UdpMulticastBus.DEFAULT_GROUP_IPv6) as bus_2:

    # register a callback on the second bus that prints messages to the standard out
    notifier = can.Notifier(bus_2, [can.Printer()])

    # create and send a message with the first bus, which should arrive at the second one
    message = can.Message(arbitration_id=0x123, data=[1, 2, 3])
    bus_1.send(message)

    # give the notifier enough time to get triggered by the second bus
    time.sleep(2.0)
```

5.2.3 Bus Class Documentation

class can.interfaces.udp_multicast.**UdpMulticastBus**(*channel='ff15:7079:7468:6f6e:6465:6d6f:6d63:6173'*, *port=43113*, *hop_limit=1*, *receive_own_messages=False*, *fd=True*, ***kwargs*)

A virtual interface for CAN communications between multiple processes using UDP over Multicast IP.

It supports IPv4 and IPv6, specified via the channel (which really is just a multicast IP address as a string). You can also specify the port and the IPv6 *hop limit*/the IPv4 *time to live* (TTL).

This bus does not support filtering based on message IDs on the kernel level but instead provides it in user space (in Python) as a fallback.

Both default addresses should allow for multi-host CAN networks in a normal local area network (LAN) where multicast is enabled.

Note: The auto-detection of available interfaces (see) is implemented using heuristic that checks if the required socket operations are available. It then returns two configurations, one based on the *DEFAULT_GROUP_IPv6* address and another one based on the *DEFAULT_GROUP_IPv4* address.

Warning: The parameter *receive_own_messages* is currently unsupported and setting it to *True* will raise an exception.

Warning: This interface does not make guarantees on reliable delivery and message ordering, and also does not implement rate limiting or ID arbitration/prioritization under high loads. Please refer to the section *Virtual Interfaces* for more information on this and a comparison to alternatives.

Parameters

- **channel** (*str*) – A multicast IPv4 address (in *224.0.0.0/4*) or an IPv6 address (in *ff00::/8*). This defines which version of IP is used. See Wikipedia ("Multicast address") for more details on the addressing schemes. Defaults to *DEFAULT_GROUP_IPv6*.
- **port** (*int*) – The IP port to read from and write to.
- **hop_limit** (*int*) – The hop limit in IPv6 or in IPv4 the time to live (TTL).
- **receive_own_messages** (*bool*) – If transmitted messages should also be received by this bus. CURRENTLY UNSUPPORTED.
- **fd** (*bool*) – If CAN-FD frames should be supported. If set to false, an error will be raised upon sending such a frame and such received frames will be ignored.
- **can_filters** – See *set_filters()*.

Raises

- **RuntimeError** – If the *msgpack*-dependency is not available. It should be installed on all non Windows platforms via the *setup.py* requirements.
- **NotImplementedError** – If the *receive_own_messages* is passed as *True*.

Construct and open a CAN bus instance of the specified type.

Subclasses should call though this method with all given parameters as it handles generic tasks like applying filters.

Parameters

- **channel** (*str*) – The can interface identifier. Expected type is backend dependent.
- **can_filters** – See *set_filters()* for details.
- **kwargs** (*dict*) – Any backend dependent configurations are passed in this dictionary
- **port** (*int*) –
- **hop_limit** (*int*) –
- **receive_own_messages** (*bool*) –
- **fd** (*bool*) –

Raises

- `ValueError` – If parameters are out of range
- *`CanInterfaceNotImplementedError`* – If the driver cannot be accessed
- *`CanInitializationError`* – If the bus cannot be initialized

`DEFAULT_GROUP_IPv4 = '239.74.163.2'`

An arbitrary IPv4 multicast address with "administrative" scope, i.e. only to be routed within administrative organizational boundaries and not beyond it. It should allow for multi-host CAN networks in a normal IPv4 LAN. This is provided as a default fallback channel if IPv6 is (still) not supported.

`DEFAULT_GROUP_IPv6 = 'ff15:7079:7468:6f6e:6465:6d6f:6d63:6173'`

An arbitrary IPv6 multicast address with "site-local" scope, i.e. only to be routed within the local physical network and not beyond it. It should allow for multi-host CAN networks in a normal IPv6 LAN. This is the default channel and should work with most modern routers if multicast is allowed.

`fileno``()`

Provides the internally used file descriptor of the socket or *-1* if not available.

Return type
int

`shutdown``()`

Close all sockets and free up any resources.

Never throws errors and only logs them.

Return type
None

5.3 Comparison

The following table compares some known virtual interfaces:

Name	Availability	Applicability			Implementation		
		Within Process	Between Processes	Via (IP) Networks	Without Central Server	Transport Technology	Serialization Format
`virtual` (this)	*included*	✓			✓	Singleton & Mutex (reliable)	none
`udp_multicast` (*doc*)	*included*	✓	✓	✓	✓	UDP via IP multicast (unreliable)	custom using msgpack
christiansandberg/python-can-remote	external	✓	✓	✓		Websockets via TCP/IP (reliable)	custom binary
windelbouwman/virtualcan	external	✓	✓	✓		ZeroMQ via TCP/IP (reliable)	custom binary[1]

[1] The only option in this list that implements interoperability with other languages out of the box. For the others (except the first intra-process one), other programs written in potentially different languages could effortlessly interface with the bus once they mimic the serialization format. The last one, however, has already implemented the entire bus functionality in *C++* and *Rust*, besides the Python variant.

5.4 Common Limitations

Guaranteed delivery and **message ordering** is one major point of difference: While in a physical CAN network, a message is either sent or in queue (or an explicit error occurred), this may not be the case for virtual networks. The `udp_multicast` bus for example, drops this property for the benefit of lower latencies by using unreliable UDP/IP instead of reliable TCP/IP (and because normal IP multicast is inherently unreliable, as the recipients are unknown by design). The other three buses faithfully model a physical CAN network in this regard: They ensure that all recipients actually receive (and acknowledge each message), much like in a physical CAN network. They also ensure that messages are relayed in the order they have arrived at the central server and that messages arrive at the recipients exactly once. Both is not guaranteed to hold for the best-effort `udp_multicast` bus as it uses UDP/IP as a transport layer.

Central servers are, however, required by interfaces 3 and 4 (the external tools) to provide these guarantees of message delivery and message ordering. The central servers receive and distribute the CAN messages to all other bus participants, unlike in a real physical CAN network. The first intra-process `virtual` interface only runs within one Python process, effectively the Python instance of *VirtualBus* acts as a central server. Notably the `udp_multicast` bus does not require a central server.

Arbitration and throughput are two interrelated functions/properties of CAN networks which are typically abstracted in virtual interfaces. In all four interfaces, an unlimited amount of messages can be sent per unit of time (given the computational power of the machines and networks that are involved). In a real CAN/CAN FD networks, however, throughput is usually much more restricted and prioritization of arbitration IDs is thus an important feature once the bus is starting to get saturated. None of the interfaces presented above support any sort of throttling or ID arbitration under high loads.

CHAPTER

SIX

PLUGIN INTERFACE

External packages can register new interfaces by using the `can.interface` entry point in its project configuration. The format of the entry point depends on your project configuration format (*pyproject.toml*, *setup.cfg* or *setup.py*).

In the following example `module` defines the location of your bus class inside your package e.g. `my_package.subpackage.bus_module` and `classname` is the name of your *can.BusABC* subclass.

```
# Note the quotes around can.interface in order to escape the dot .
[project.entry-points."can.interface"]
interface_name = "module:classname"
```

```
[options.entry_points]
can.interface =
    interface_name = module:classname
```

```
from setuptools import setup

setup(
    # ...,
    entry_points = {
        'can.interface': [
            'interface_name = module:classname'
        ]
    }
)
```

The `interface_name` can be used to create an instance of the bus in the **python-can** API:

```
import can

bus = can.Bus(interface="interface_name", channel=0)
```

6.1 Example Interface Plugins

The table below lists interface drivers that can be added by installing additional packages that utilise the plugin API. These modules are optional dependencies of python-can.

Note: The packages listed below are maintained by other authors. Any issues should be reported in their corresponding repository and **not** in the python-can repository.

Name	Description
python-can-canine	CAN Driver for the CANine CAN interface
python-can-cvector	Cython based version of the 'VectorBus'
python-can-remote	CAN over network bridge
python-can-sontheim	CAN Driver for Sontheim CAN interfaces (e.g. CANfox)

CHAPTER

SEVEN

OTHER CAN BUS TOOLS

In order to keep the project maintainable, the scope of the package is limited to providing common abstractions to different hardware devices, and a basic suite of utilities for sending and receiving messages on a CAN bus. Other tools are available that either extend the functionality of python-can, or provide complementary features that python-can users might find useful.

Some of these tools are listed below for convenience.

7.1 CAN Message protocols (implemented in Python)

1. **SAE J1939 Message Protocol**
    - The can-j1939 module provides an implementation of the CAN SAE J1939 standard for Python, including J1939-22. can-j1939 uses python-can to provide support for multiple hardware interfaces.
2. **CIA CANopen**
    - The canopen module provides an implementation of the CIA CANopen protocol, aiming to be used for automation and testing purposes
3. **ISO 15765-2 (ISO TP)**
    - The can-isotp module provides an implementation of the ISO TP CAN protocol for sending data packets via a CAN transport layer.
4. **UDS**
    - The python-uds module is a communication protocol agnostic implementation of the Unified Diagnostic Services (UDS) protocol defined in ISO 14229-1, although it does have extensions for performing UDS over CAN utilising the ISO TP protocol. This module has not been updated for some time.
    - The uds module is another tool that implements the UDS protocol, although it does have extensions for performing UDS over CAN utilising the ISO TP protocol. This module has not been updated for some time.
5. **XCP**
    - The pyxcp module implements the Universal Measurement and Calibration Protocol (XCP). The purpose of XCP is to adjust parameters and acquire current values of internal variables in an ECU.

7.2 CAN Frame Parsing tools etc. (implemented in Python)

1. **CAN Message / Database scripting**
    - The cantools package provides multiple methods for interacting with can message database files, and using these files to monitor live busses with a command line monitor tool.
2. **CAN Message / Log Decoding**
    - The canmatrix module provides methods for converting between multiple popular message frame definition file formats (e.g. .DBC files, .KCD files, .ARXML files etc.).
    - The pretty_j1939 module can be used to post-process CAN logs of J1939 traffic into human readable terminal prints or into a JSON file for consumption elsewhere in your scripts.

7.3 Other CAN related tools, programs etc.

1. **Micropython CAN class**
    - A CAN class is available for the original micropython pyboard, with much of the same functionality as is available with python-can (but with a different API!).
2. **ASAM MDF Files**
    - The asammdf module provides many methods for processing ASAM (Association for Standardization of Automation and Measuring Systems) MDF (Measurement Data Format) files.

Note: See also the available plugins for python-can in *Plugin Interface*.

CHAPTER

EIGHT

SCRIPTS

The following modules are callable from `python-can`.

They can be called for example by `python -m can.logger` or `can_logger` (if installed using pip).

8.1 can.logger

Command line help, called with `--help`:

```
$ python -m can.logger -h
ldf is not supported
xls is not supported
xlsx is not supported
yaml is not supported
usage: logger.py [-h] [-c CHANNEL]
                 [-i {canalystii,cantact,etas,gs_usb,iscan,ixxat,kvaser,neousys,neovi,
↪nican,nixnet,pcan,robotell,seeedstudio,serial,slcan,socketcan,socketcand,systec,udp_
↪multicast,usb2can,vector,virtual}]
                 [-b BITRATE] [--fd] [--data_bitrate DATA_BITRATE]
                 [-f LOG_FILE] [-a] [-s FILE_SIZE] [-v]
                 [--filter {<can_id>:<can_mask>,<can_id>~<can_mask>} [{<can_id>:<can_
↪mask>,<can_id>~<can_mask>} ...]]
                 [--active | --passive]
                 ...

Log CAN traffic, printing messages to stdout or to a given file.

positional arguments:
  extra_args            The remaining arguments will be used for the interface
                        and logger/player initialisation. For example, `-i
                        vector -c 1 --app-name=MyCanApp` is the equivalent to
                        opening the bus with `Bus('vector', channel=1,
                        app_name='MyCanApp')

options:
  -h, --help            show this help message and exit
  -c CHANNEL, --channel CHANNEL
                        Most backend interfaces require some sort of channel.
                        For example with the serial interface the channel
                        might be a rfcomm device: "/dev/rfcomm0". With the
```

(continues on next page)

(continued from previous page)

```
                        socketcan interface valid channel examples include:
                        "can0", "vcan0".
  -i {canalystii,cantact,etas,gs_usb,iscan,ixxat,kvaser,neousys,neovi,nican,nixnet,pcan,
↪robotell,seeedstudio,serial,slcan,socketcan,socketcand,systec,udp_multicast,usb2can,
↪vector,virtual}, --interface {canalystii,cantact,etas,gs_usb,iscan,ixxat,kvaser,
↪neousys,neovi,nican,nixnet,pcan,robotell,seeedstudio,serial,slcan,socketcan,socketcand,
↪systec,udp_multicast,usb2can,vector,virtual}
                        Specify the backend CAN interface to use. If left
                        blank, fall back to reading from configuration files.
  -b BITRATE, --bitrate BITRATE
                        Bitrate to use for the CAN bus.
  --fd                  Activate CAN-FD support
  --data_bitrate DATA_BITRATE
                        Bitrate to use for the data phase in case of CAN-FD.
  -f LOG_FILE, --file_name LOG_FILE
                        Path and base log filename, for supported types see
                        can.Logger.
  -a, --append          Append to the log file if it already exists.
  -s FILE_SIZE, --file_size FILE_SIZE
                        Maximum file size in bytes. Rotate log file when size
                        threshold is reached. (The resulting file sizes will
                        be consistent, but are not guaranteed to be exactly
                        what is specified here due to the rollover conditions
                        being logger implementation specific.)
  -v                    How much information do you want to see at the command
                        line? You can add several of these e.g., -vv is DEBUG
  --filter {<can_id>:<can_mask>,<can_id>~<can_mask>} [{<can_id>:<can_mask>,<can_id>~<can_
↪mask>} ...]
                        R|Space separated CAN filters for the given CAN
                        interface: <can_id>:<can_mask> (matches when
                        <received_can_id> & mask == can_id & mask)
                        <can_id>~<can_mask> (matches when <received_can_id> &
                        mask != can_id & mask) Fx to show only frames with ID
                        0x100 to 0x103 and 0x200 to 0x20F: python -m
                        can.viewer -f 100:7FC 200:7F0 Note that the ID and
                        mask are always interpreted as hex values
  --active              Start the bus as active, this is applied by default.
  --passive             Start the bus as passive.
```

8.2 can.player

```
$ python -m can.player -h
ldf is not supported
xls is not supported
xlsx is not supported
yaml is not supported
usage: player.py [-h] [-c CHANNEL]
                 [-i {canalystii,cantact,etas,gs_usb,iscan,ixxat,kvaser,neousys,neovi,
↪nican,nixnet,pcan,robotell,seeedstudio,serial,slcan,socketcan,socketcand,systec,udp_
↪multicast,usb2can,vector,virtual}]
```

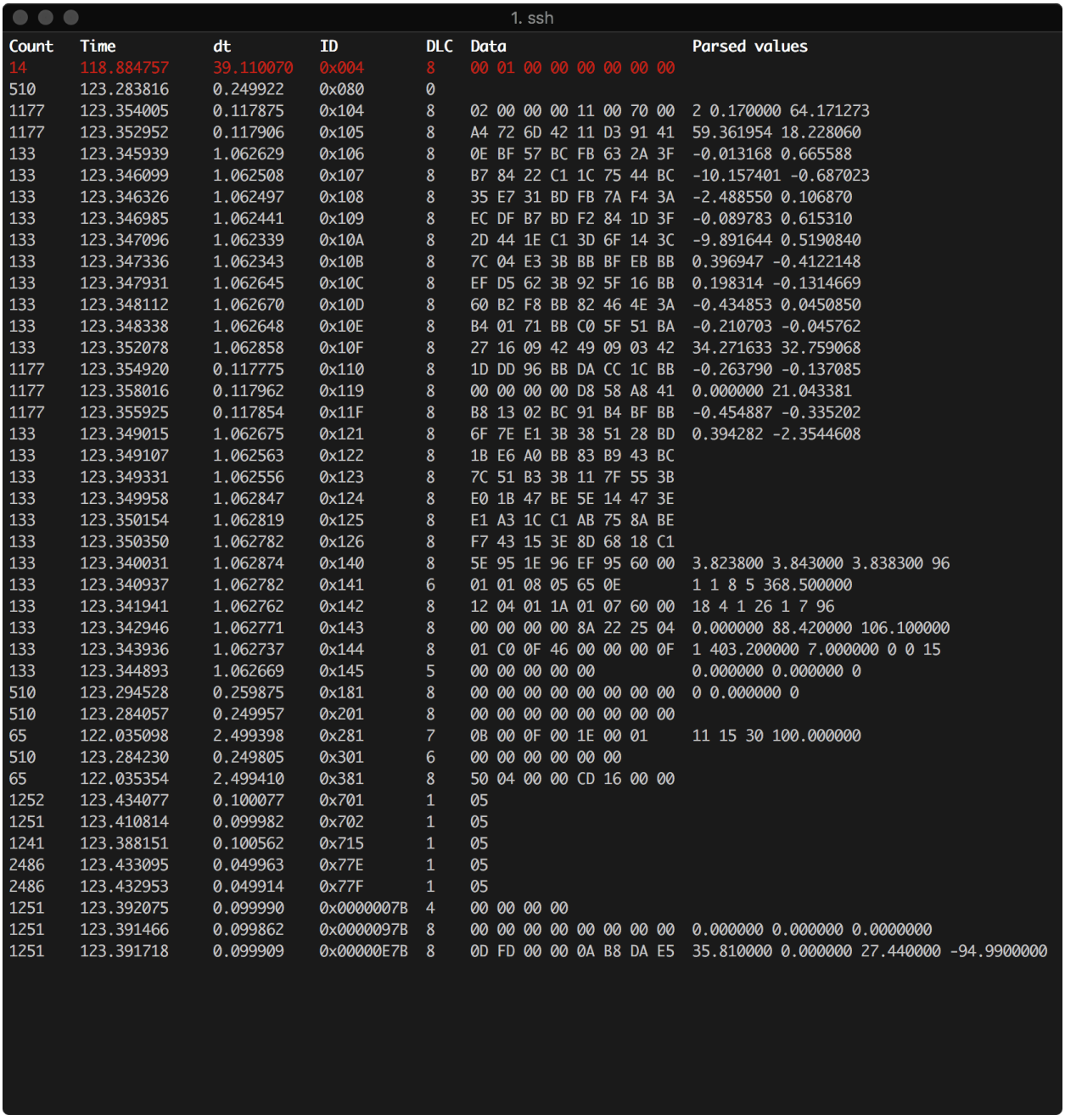
(continues on next page)

(continued from previous page)

```
                [-b BITRATE] [--fd] [--data_bitrate DATA_BITRATE]
                [-f LOG_FILE] [-v] [--ignore-timestamps] [--error-frames]
                [-g GAP] [-s SKIP]
                ... input-file

Replay CAN traffic.

positional arguments:
  extra_args            The remaining arguments will be used for the interface
                        and logger/player initialisation. For example, `-i
                        vector -c 1 --app-name=MyCanApp` is the equivalent to
                        opening the bus with `Bus('vector', channel=1,
                        app_name='MyCanApp')
  input-file            The file to replay. For supported types see
                        can.LogReader.

options:
  -h, --help            show this help message and exit
  -c CHANNEL, --channel CHANNEL
                        Most backend interfaces require some sort of channel.
                        For example with the serial interface the channel
                        might be a rfcomm device: "/dev/rfcomm0". With the
                        socketcan interface valid channel examples include:
                        "can0", "vcan0".
  -i {canalystii,cantact,etas,gs_usb,iscan,ixxat,kvaser,neousys,neovi,nican,nixnet,pcan,
↪robotell,seeedstudio,serial,slcan,socketcan,socketcand,systec,udp_multicast,usb2can,
↪vector,virtual}, --interface {canalystii,cantact,etas,gs_usb,iscan,ixxat,kvaser,
↪neousys,neovi,nican,nixnet,pcan,robotell,seeedstudio,serial,slcan,socketcan,socketcand,
↪systec,udp_multicast,usb2can,vector,virtual}
                        Specify the backend CAN interface to use. If left
                        blank, fall back to reading from configuration files.
  -b BITRATE, --bitrate BITRATE
                        Bitrate to use for the CAN bus.
  --fd                  Activate CAN-FD support
  --data_bitrate DATA_BITRATE
                        Bitrate to use for the data phase in case of CAN-FD.
  -f LOG_FILE, --file_name LOG_FILE
                        Path and base log filename, for supported types see
                        can.LogReader.
  -v                    Also print can frames to stdout. You can add several
                        of these to enable debugging
  --ignore-timestamps   Ignore timestamps (send all frames immediately with
                        minimum gap between frames)
  --error-frames        Also send error frames to the interface.
  -g GAP, --gap GAP     <s> minimum time between replayed frames
  -s SKIP, --skip SKIP  <s> skip gaps greater than 's' seconds
```

8.3 can.viewer

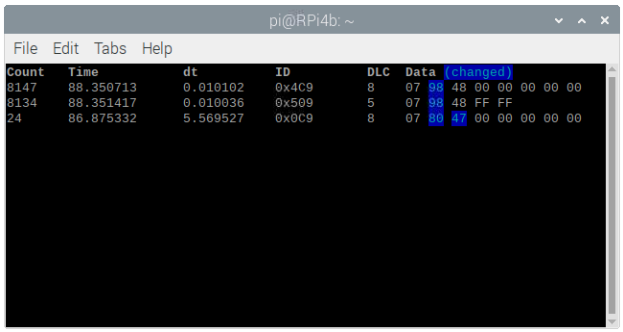
A screenshot of the application can be seen below:



The first column is the number of times a frame with the particular ID that has been received, next is the timestamp of the frame relative to the first received message. The third column is the time between the current frame relative to the previous one. Next is the length of the frame, the data and then the decoded data converted according to the `-d` argument. The top red row indicates an error frame. There are several keyboard shortcuts that can be used with the viewer script, they function as follows:

- ESCAPE - Quit the viewer script
- q - as ESCAPE

- c - Clear the stored frames
- s - Sort the stored frames
- h - Toggle highlighting of changed bytes in the data field - see the below image
- SPACE - Pause the viewer
- UP/DOWN - Scroll the viewer



A byte in the data field is highlighted blue if the value is different from the last time the message was received.

8.3.1 Command line arguments

By default the `can.viewer` uses the *SocketCAN* interface. All interfaces are supported and can be specified using the `-i` argument or configured following *Configuration*.

The full usage page can be seen below:

```
$ python -m can.viewer -h
ldf is not supported
xls is not supported
xlsx is not supported
yaml is not supported
Usage: python -m can.viewer [-c CHANNEL]
                            [-i {canalystii,cantact,etas,gs_usb,iscan,ixxat,kvaser,
↪neousys,neovi,nican,nixnet,pcan,robotell,seeedstudio,serial,slcan,socketcan,socketcand,
↪systec,udp_multicast,usb2can,vector,virtual}]
                            [-b BITRATE] [--fd] [--data_bitrate DATA_BITRATE]
                            [-h] [--version]
                            [-d ('{<id>:<format>,<id>:<format>:<scaling1>:...:<scalingN>,
↪file.txt}',)]
                            [-f ('{<can_id>:<can_mask>,<can_id>~<can_mask>}',)]
                            [-v]
                            ('extra_args',)

A simple CAN viewer terminal application written in Python

positional arguments:
  extra_args            The remaining arguments will be used for the interface
                        and logger/player initialisation. For example, `-i
                        vector -c 1 --app-name=MyCanApp` is the equivalent to
                        opening the bus with `Bus('vector', channel=1,
                        app_name='MyCanApp')
```

(continues on next page)

(continued from previous page)

```
options:
  -c, --channel CHANNEL
                        Most backend interfaces require some sort of channel.
                        For example with the serial interface the channel
                        might be a rfcomm device: "/dev/rfcomm0". With the
                        socketcan interface valid channel examples include:
                        "can0", "vcan0".
  -i, --interface {canalystii,cantact,etas,gs_usb,iscan,ixxat,kvaser,neousys,neovi,nican,
↪nixnet,pcan,robotell,seeedstudio,serial,slcan,socketcan,socketcand,systec,udp_
↪multicast,usb2can,vector,virtual}
                        Specify the backend CAN interface to use. If left
                        blank, fall back to reading from configuration files.
  -b, --bitrate BITRATE
                        Bitrate to use for the CAN bus.
  --fd                  Activate CAN-FD support
  --data_bitrate DATA_BITRATE
                        Bitrate to use for the data phase in case of CAN-FD.

Optional arguments:
  -h, --help            Show this help message and exit
  --version             Show program's version number and exit
  -d, --decode ('{<id>:<format>,<id>:<format>:<scaling1>:...:<scalingN>,file.txt}',)
                        Specify how to convert the raw bytes into real values.
                        The ID of the frame is given as the first argument and the␣
↪format as the second.
                        The Python struct package is used to unpack the received data
                        where the format characters have the following meaning:
                              < = little-endian, > = big-endian
                              x = pad byte
                              c = char
                              ? = bool
                              b = int8_t, B = uint8_t
                              h = int16, H = uint16
                              l = int32_t, L = uint32_t
                              q = int64_t, Q = uint64_t
                              f = float (32-bits), d = double (64-bits)
                        Fx to convert six bytes with ID 0x100 into uint8_t, uint16 and␣
↪uint32_t:
                          $ python -m can.viewer -d "100:<BHL"
                        Note that the IDs are always interpreted as hex values.
                        An optional conversion from integers to real units can be given
                        as additional arguments. In order to convert from raw integer
                        values the values are divided with the corresponding scaling␣
↪value,
                        similarly the values are multiplied by the scaling value in order
                        to convert from real units to raw integer values.
                        Fx lets say the uint8_t needs no conversion, but the uint16 and␣
↪the uint32_t
                        needs to be divided by 10 and 100 respectively:
                          $ python -m can.viewer -d "101:<BHL:1:10.0:100.0"
                        Be aware that integer division is performed if the scaling value␣
↪is an integer.
```

(continues on next page)

(continued from previous page)

```
                        Multiple arguments are separated by spaces:
                          $ python -m can.viewer -d "100:<BHL" "101:<BHL:1:10.0:100.0"
                        Alternatively a file containing the conversion strings separated␣
↪by new lines
                        can be given as input:
                          $ cat file.txt
                              100:<BHL
                              101:<BHL:1:10.0:100.0
                          $ python -m can.viewer -d file.txt
  -f, --filter ('{<can_id>:<can_mask>,<can_id>~<can_mask>}',)
                        Space separated CAN filters for the given CAN interface:
                              <can_id>:<can_mask> (matches when <received_can_id> & mask␣
↪== can_id & mask)
                              <can_id>~<can_mask> (matches when <received_can_id> & mask␣
↪!= can_id & mask)
                        Fx to show only frames with ID 0x100 to 0x103 and 0x200 to 0x20F:
                              python -m can.viewer -f 100:7FC 200:7F0
                        Note that the ID and mask are always interpreted as hex values
  -v                    How much information do you want to see at the command
                        line? You can add several of these e.g., -vv is DEBUG

Shortcuts:
        +---------+--------------------------------+
        |   Key   |       Description              |
        +---------+--------------------------------+
        | ESQ/q   | Exit the viewer                |
        | c       | Clear the stored frames        |
        | s       | Sort the stored frames         |
        | h       | Toggle highlight byte changes  |
        | SPACE   | Pause the viewer               |
        | UP/DOWN | Scroll the viewer              |
        +---------+--------------------------------+
```

8.4 can.logconvert

```
$ python -m can.logconvert -h
ldf is not supported
xls is not supported
xlsx is not supported
yaml is not supported
usage: logconvert.py [-h] [-s FILE_SIZE] INFILE OUTFILE

Convert a log file from one format to another.

positional arguments:
  INFILE                Input filename. The type is dependent on the suffix,
                        see can.LogReader.
  OUTFILE               Output filename. The type is dependent on the suffix,
                        see can.Logger.
```

(continues on next page)

(continued from previous page)

```
options:
  -h, --help            show this help message and exit
  -s FILE_SIZE, --file_size FILE_SIZE
                        Maximum file size in bytes. Rotate log file when size
                        threshold is reached.
```

CHAPTER

NINE

DEVELOPER'S OVERVIEW

9.1 Contributing

Contribute to source code, documentation, examples and report issues: https://github.com/hardbyte/python-can

Note that the latest released version on PyPi may be significantly behind the `develop` branch. Please open any feature requests against the `develop` branch

There is also a python-can mailing list for development discussion.

Some more information about the internals of this library can be found in the chapter *Internal API*. There is also additional information on extending the `can.io` module.

9.2 Pre-releases

The latest pre-release can be installed with:

```
pip install --upgrade --pre python-can
```

9.3 Building & Installing

The following assumes that the commands are executed from the root of the repository:

The project can be built with:

```
pipx run build
pipx run twine check dist/*
```

The project can be installed in editable mode with:

```
pip install -e .
```

The unit tests can be run with:

```
pipx run tox -e py
```

The documentation can be built with:

```
pip install -r doc/doc-requirements.txt
python -m sphinx -an doc build
```

The linters can be run with:

```
pip install -e .[lint]
black --check can
mypy can
ruff check can
pylint --rcfile=.pylintrc can/**.py
```

9.4 Creating a new interface/backend

These steps are a guideline on how to add a new backend to python-can.

- Create a module (either a `*.py` or an entire subdirectory depending on the complexity) inside `can.interfaces`
- Implement the central part of the backend: the bus class that extends *can.BusABC*. See *Extending the BusABC class* for more info on this one!
- Register your backend bus class in `BACKENDS` in the file `can.interfaces.__init__.py`.
- Add docs where appropriate. At a minimum add to `doc/interfaces.rst` and add a new interface specific document in `doc/interface/*`. It should document the supported platforms and also the hardware/software it requires. A small snippet of how to install the dependencies would also be useful to get people started without much friction.
- Also, don't forget to document your classes, methods and function with docstrings.
- Add tests in `test/*` where appropriate. To get started, have a look at `back2back_test.py`: Simply add a test case like `BasicTestSocketCan` and some basic tests will be executed for the new interface.

Attention: We strongly recommend using the *Plugin Interface* to extend python-can. Publish a python package that contains your *can.BusABC* subclass and use it within the python-can API. We will mention your package inside this documentation and add it as an optional dependency.

9.5 Code Structure

The modules in `python-can` are:

Module	Description
interfaces	Contains interface dependent code.
bus	Contains the interface independent Bus object.
message	Contains the interface independent Message object.
io	Contains a range of file readers and writers.
broadcastmanager	Contains interface independent broadcast manager code.

9.6 Creating a new Release

- Release from the `main` branch (except for pre-releases).
- Update the library version in `__init__.py` using semantic versioning.
- Check if any deprecations are pending.
- Run all tests and examples against available hardware.
- Update `CONTRIBUTORS.txt` with any new contributors.
- For larger changes update `doc/history.rst`.
- Sanity check that documentation has stayed inline with code.
- Create a temporary virtual environment. Run `python setup.py install` and `tox`.
- Create and upload the distribution: `python setup.py sdist bdist_wheel`.
- Sign the packages with gpg `gpg --detach-sign -a dist/python_can-X.Y.Z-py3-none-any.whl`.
- Upload with twine `twine upload dist/python-can-X.Y.Z*`.
- In a new virtual env check that the package can be installed with pip: `pip install python-can==X.Y.Z`.
- Create a new tag in the repository.
- Check the release on PyPi, Read the Docs and GitHub.

CHAPTER

TEN

HISTORY

10.1 Background

Originally written at Dynamic Controls for internal use testing and prototyping wheelchair components.

Maintenance was taken over and the project was open sourced by Brian Thorne in 2010.

10.2 Acknowledgements

Originally written by Ben Powell as a thin wrapper around the Kvaser SDK to support the leaf device.

Support for linux socketcan was added by Rose Lu as a summer coding project in 2011. The socketcan interface was helped immensely by Phil Dixon who wrote a leaf-socketcan driver for Linux.

The pcan interface was contributed by Albert Bloomfield in 2013. Support for pcan on Mac was added by Kristian Sloth Lauszus in 2018.

The usb2can interface was contributed by Joshua Villyard in 2015.

The IXXAT VCI interface was contributed by Giuseppe Corbelli and funded by Weightpack in 2016.

The NI-CAN and virtual interfaces plus the ASCII and BLF loggers were contributed by Christian Sandberg in 2016 and 2017. The BLF format is based on a C++ library by Toby Lorenz.

The slcan interface, ASCII listener and log logger and listener were contributed by Eduard Bröcker in 2017.

The NeoVi interface for ICS (Intrepid Control Systems) devices was contributed by Pierre-Luc Tessier Gagné in 2017.

Many improvements all over the library, cleanups, unifications as well as more comprehensive documentation and CI testing was contributed by Felix Divo in 2017 and 2018.

The CAN viewer terminal script was contributed by Kristian Sloth Lauszus in 2018.

The CANalyst-II interface was contributed by Shaoyu Meng in 2018.

@deonvdw added support for the Robotell interface in 2019.

Felix Divo and Karl Ding added type hints for the core library and many interfaces leading up to the 4.0 release.

Eric Evenchick added support for the CANtact devices in 2020.

Felix Divo added an interprocess virtual bus interface in 2020.

@jxltom added the gs_usb interface in 2020 supporting Geschwister Schneider USB/CAN devices and bytewerk.org candleLight USB CAN devices such as candlelight, canable, cantact, etc.

@jaesc added the nixnet interface in 2021 supporting NI-XNET devices from National Instruments.

Tuukka Pasanen @illuusio added the neousys interface in 2021.

Francisco Javier Burgos Maciá @fjburgos added ixxat FD support.

@domologic contributed a socketcand interface in 2021.

Felix N @felixn contributed the ETAS interface in 2021.

Felix Divo unified exception handling across every interface in the lead up to the 4.0 release.

Felix Divo prepared the python-can 4.0 release.

10.3 Support for CAN within Python

Python natively supports the CAN protocol from version 3.3 on, if running on Linux (with a sufficiently new kernel):

Python version	Feature	Link
3.3	Initial SocketCAN support	Docs
3.4	Broadcast Management (BCM) commands are natively supported	Docs
3.5	CAN FD support	Docs
3.7	Support for CAN ISO-TP	Docs
3.9	Native support for joining CAN filters	Docs

CHAPTER

ELEVEN

KNOWN BUGS

See the project bug tracker on github. Patches and pull requests very welcome!

Documentation generated

Dec 12, 2023

PYTHON MODULE INDEX

C

INDEX

Symbols

A

B

C

D

E

F

G

I

K

L

M

N

O

P

R

S

T

U

V

W

X