python-can

Release 3.3.4

Oct 07, 2020

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, motocycles, 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
# coding: utf-8

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

from __future__ import print_function

import can

def send_one():

    # this uses the default configuration (for example from the config file)
    # see https://python-can.readthedocs.io/en/stable/configuration.html
    bus = can.interface.Bus()

    # Using specific buses works similar:
    # bus = can.interface.Bus(bustype='socketcan', channel='vcan0', bitrate=250000)
    # bus = can.interface.Bus(bustype='pcan', channel='PCAN_USBBUS1', bitrate=250000)
    # bus = can.interface.Bus(bustype='ixxat', channel=0, bitrate=250000)
    # bus = can.interface.Bus(bustype='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("Message sent on {}".format(bus.channel_info))
    except can.CanError:
        print("Message NOT sent")

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

Contents:

CHAPTER 1

Installation

Install `can` with `pip`:

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

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 *CAN Interface Modules*.

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 the latest stable release of Python.
2. Install Kvaser's latest Windows CANLib drivers.
3. 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 from PEAK-System's download page.

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 uptime`.

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 SDK as the backend:

1. Install IXXAT's latest Windows VCI V3 SDK 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 *NEOVI Interface*.

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 2

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** before importing from `can.interfaces`.

```
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(bustype='socketcan', channel='vcan0', bitrate=500000)
```

2.2 Configuration File

On Linux 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. `~/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.interfaces.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

2.4 Interface Names

Lookup table of interface names:

Name	Documentation
`"socketcan"`	*SocketCAN*
`"kvaser"`	*Kvaser's CANLIB*
`"serial"`	*CAN over Serial*
`"slcan"`	*CAN over Serial / SLCAN*
`"ixxat"`	*IXXAT Virtual CAN Interface*
`"pcan"`	*PCAN Basic API*
`"usb2can"`	*USB2CAN Interface*
`"nican"`	*NI-CAN*
`"iscan"`	*isCAN*
`"neovi"`	*NEOVI Interface*
`"vector"`	*Vector*
`"virtual"`	*Virtual*
`"canalystii"`	*CANalyst-II*
`"systec"`	*SYSTEC interface*

CHAPTER 3

Library API

The main objects are the *BusABC* 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, 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, for example:

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

That bus is then able to handle the interface specific software/hardware interactions and implements the *BusABC* API.

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

3.1.1 Autoconfig Bus

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

Bases: can.bus.BusABC

Bus wrapper with configuration loading.

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

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** – The can interface identifier. Expected type is backend dependent.
- **can_filters** (*list*) – See *set_filters()* for details.
- **kwargs** (*dict*) – Any backend dependent configurations are passed in this dictionary

3.1.2 API

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

Bases: object

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.

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** – The can interface identifier. Expected type is backend dependent.
- **can_filters** (*list*) – See *set_filters()* for details.
- **kwargs** (*dict*) – Any backend dependent configurations are passed in this dictionary

__iter__()

Allow iteration on messages as they are received.

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

Yields *can.Message* msg objects.

RECV_LOGGING_LEVEL = 9

Log level for received messages

channel_info = 'unknown'

a string describing the underlying bus and/or channel

filters

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).

recv(*timeout=None*)

Block waiting for a message from the Bus.

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

Return type *can.Message* or None

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

Raises ***can.CanError*** – if an error occurred while reading

send(*msg*, *timeout=None*)
Transmit a message to the CAN bus.

Override this method to enable the transmit path.

Parameters

- **msg** (can.Message) – A message object.
- **timeout** (*float or 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 *can.CanError* – if the message could not be sent

send_periodic(*msg*, *period*, *duration=None*, *store_task=True*)
Start sending a message 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
- *BusABC.stop_all_periodic_tasks()* is called
- the task's CyclicTask.stop() method is called.

Parameters

- **msg** (can.Message) – Message to transmit
- **period** (*float*) – Period in seconds between each message
- **duration** (*float*) – The duration to keep sending this message at given rate. 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.

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

Return type *can.broadcastmanager.CyclicSendTaskABC*

Note: Note the duration before the message stops 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** – 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.

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

state
Return the current state of the hardware

Type can.BusState

stop_all_periodic_tasks(*remove_tasks=True*)
Stop sending any messages that were started using **bus.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.

3.1.3 Transmitting

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

3.1.4 Receiving

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

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

Alternatively the *Listener* api can be used, which is a list of *Listener* subclasses that receive notifications when new messages arrive.

3.1.5 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.

3.2 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*)

Bases: ObjectProxy

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.

3.3 Message

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

Bases: object

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*.

Raises **ValueError** – iff *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')
```

(continues on next page)

(continued from previous page)

```
>>> test.dlc
5
>>> print(test)
Timestamp:        0.000000    ID: 00000000    010    DLC: 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: 0064    S        DLC: 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        DLC: 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: 0000    S        DLC: 0
>>> print(Message(is_extended_id=True))
Timestamp:        0.000000    ID: 00000000    X        DLC: 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 E      DLC: 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   R    DLC: 0
```

is_fd

Type bool

Indicates that this message is a CAN FD message.

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

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

- timestamp,
- arbitration ID,
- flags,
- dlc,
- 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*)
Compares a given message with this one.

Parameters

- **other** (can.Message) – the message to compare with
- **timestamp_delta** (*float or int or None*) – the maximum difference at which two timestamps are still considered equal or None to not compare timestamps

Return type bool

Returns True iff the given message equals this one

3.4 Listeners

3.4.1 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.

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**
Bases: `object`

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

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

# now either call
listener(msg)
# or
listener.on_message_received(msg)

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

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

on_message_received(*msg*)
This method is called to handle the given message.

Parameters **msg** (`can.Message`) – the delivered message

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

Concrete implementations override.

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.

3.4.2 BufferedReader

class can.**BufferedReader**
Bases: `can.listener.Listener`

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 be called will raise an exception, see *on_message_received()*.

Attr bool is_stopped `True` iff the reader has been stopped

get_message(*timeout=0.5*)
Attempts to retrieve the latest message received by the instance. If no message is available it blocks for given timeout or until a message is received, or else returns None (whichever is shorter). 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.

Rytpe can.Message or None

Returns the message if there is one, or None if there is not.

on_message_received(*msg*)
Append a message to the buffer.

Raises BufferError if the reader has already been stopped

stop()
Prohibits any more additions to this reader.

class can.**AsyncBufferedReader**(*loop=None*)
Bases: can.listener.Listener

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)
```

get_message()
Retrieve the latest message when awaited for:

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

Return type *can.Message*

Returns The CAN message.

on_message_received(*msg*)
Append a message to the buffer.

Must only be called inside an event loop!

3.4.3 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**(*file*, *mode='rt'*)
Bases: *can.io.generic.BaseIOHandler*, can.listener.Listener

Logs CAN messages to a file.

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

- .asc: *can.ASCWriter*
- .blf *can.BLFWriter*
- .csv: *can.CSVWriter*
- .db: *can.SqliteWriter*
- .log *can.CanutilsLogWriter*
- other: *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 an 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** (*str*) – the mode that should be used to open the file, see `open()`, ignored if *file* is *None*

3.4.4 Printer

class can.**Printer**(*file=None*)

Bases: *can.io.generic.BaseIOHandler*, can.listener.Listener

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 bool write_to_file *True* iff this instance prints to a file instead of standard out

Parameters **file** – an optional path-like object or as file-like object to "print" to instead of writing to standard out (stdout) If this is a file-like object, is has to opened in text write mode, not binary write mode.

on_message_received(*msg*)

This method is called to handle the given message.

Parameters **msg** (can.Message) – the delivered message

3.4.5 CSVWriter

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

Bases: *can.io.generic.BaseIOHandler*, can.listener.Listener

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** – 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

on_message_received(*msg*)
This method is called to handle the given message.

Parameters **msg** (can.Message) – the delivered message

class can.**CSVReader**(*file*)
Bases: *can.io.generic.BaseIOHandler*

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** – 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.

3.4.6 SqliteWriter

class can.**SqliteWriter**(*file*, *table_name='messages'*)
Bases: *can.io.generic.BaseIOHandler*, can.listener.BufferedReader

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()

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** – a *str* or since Python 3.7 a path like object that points to the database file to use
- **table_name** (*str*) – the name of the table to store messages in

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'*)
Bases: *can.io.generic.BaseIOHandler*

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

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

Parameters

- **file** – a *str* or since Python 3.7 a path like object that points to the database file to use
- **table_name** (*str*) – the name of the table to look for the messages

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

3.4.7 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*)

Bases: *can.io.generic.BaseIOHandler*, can.listener.Listener

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** – 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** – a default channel to use when the message does not have a channel set

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

Add a message to the log file.

Parameters

- **message** (*str*) – an arbitrary message
- **timestamp** (*float*) – the absolute timestamp of the event

on_message_received(*msg*)

This method is called to handle the given message.

Parameters **msg** (can.Message) – the delivered message

stop()

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

Concrete implementations override.

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

class can.**ASCReader**(*file*)

Bases: *can.io.generic.BaseIOHandler*

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

TODO: turn relative timestamps back to absolute form

Parameters **file** – 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.

3.4.8 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*)
Bases: *can.io.generic.BaseIOHandler*, can.listener.Listener

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** – 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** – 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

on_message_received(*msg*)
This method is called to handle the given message.

Parameters **msg** (can.Message) – the delivered message

CanutilsLogReader reads CAN data from ASCII log files .log

class can.**CanutilsLogReader**(*file*)
Bases: *can.io.generic.BaseIOHandler*

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** – 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.

3.4.9 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*, *channel=1*)
Bases: *can.io.generic.BaseIOHandler*, can.listener.Listener

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

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

COMPRESSION_LEVEL = 9
ZLIB compression level

MAX_CACHE_SIZE = 131072
Max log container size of uncompressed data

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.

on_message_received(*msg*)
This method is called to handle the given message.

Parameters **msg** (can.Message) – 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*)
Bases: *can.io.generic.BaseIOHandler*

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** – 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.

3.5 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.

3.5.1 Example

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

```
import asyncio
import can

def print_message(msg):
    """Regular callback function. Can also be a coroutine."""
    print(msg)

async def main():
    can0 = can.Bus('vcan0', bustype='virtual', receive_own_messages=True)
    reader = can.AsyncBufferedReader()
    logger = can.Logger('logfile.asc')

    listeners = [
        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_event_loop()
    notifier = can.Notifier(can0, listeners, loop=loop)
    # Start sending first message
    can0.send(can.Message(arbitration_id=0))

    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
        can0.send(msg)
    # Wait for last message to arrive
    await reader.get_message()
    print('Done!')

    # Clean-up
    notifier.stop()
    can0.shutdown()

# Get the default event loop
loop = asyncio.get_event_loop()
# Run until main coroutine finishes
loop.run_until_complete(main())
loop.close()
```

3.6 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
# coding: utf-8
```

(continues on next page)

(continued from previous page)

```
"""
This example exercises the periodic sending capabilities.

Expects a vcan0 interface:

    python3 -m examples.cyclic

"""

from __future__ import print_function

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
    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):
    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 a ")
        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):
    print("Starting to send a message every 200ms. Initial data is ones")
    msg = can.Message(arbitration_id=0x0cf02200, data=[1, 1, 1, 1])
    task = bus.send_periodic(msg, 0.20)
```

(continues on next page)

(continued from previous page)

```
    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])
    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")


if __name__ == "__main__":

    reset_msg = can.Message(arbitration_id=0x00, data=[0, 0, 0, 0, 0, 0], is_extended_
↪id=False)

    for interface, channel in [
```

(continues on next page)

(continued from previous page)

```
        ('socketcan', 'vcan0'),
        #('ixxat', 0)
    ]:
        print("Carrying out cyclic tests with {} interface".format(interface))

        bus = can.Bus(interface=interface, channel=channel, bitrate=500000)
        bus.send(reset_msg)

        simple_periodic_send(bus)

        bus.send(reset_msg)

        limited_periodic_send(bus)

        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()

        bus.shutdown()

    time.sleep(2)
```

3.6.1 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**
Bases: object

Abstract Base for all cyclic tasks.

stop()
Cancel this periodic task.

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

class can.broadcastmanager.**CyclicSendTaskABC**(*message*, *period*)
Bases: *can.broadcastmanager.CyclicTask*

Message send task with defined period

Parameters

- **message** (can.Message) – The message to be sent periodically.
- **period** (*float*) – The rate in seconds at which to send the message.

class can.broadcastmanager.**LimitedDurationCyclicSendTaskABC**(*message*, *period*, *duration*)
Bases: *can.broadcastmanager.CyclicSendTaskABC*

Message send task with a defined duration and period.

Parameters

- **message** (can.Message) – The message to be sent periodically.

- **period** (*float*) – The rate in seconds at which to send the message.
- **duration** (*float*) – The duration to keep sending this message at given rate.

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

Bases: *can.broadcastmanager.CyclicSendTaskABC*

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 message at *subsequent_period*.

Parameters

- **channel** – See interface specific documentation.
- **message** (can.Message) –
- **count** (*int*) –
- **initial_period** (*float*) –
- **subsequent_period** (*float*) –

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

Bases: *can.broadcastmanager.CyclicSendTaskABC*

Adds support for modifying a periodic message

Parameters

- **message** (can.Message) – The message to be sent periodically.
- **period** (*float*) – The rate in seconds at which to send the message.

modify_data(*message*)

Update the contents of this periodically sent message without altering the timing.

Parameters **message** (can.Message) – The message with the new *can.Message.data*. Note: The arbitration ID cannot be changed.

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

Bases: *can.broadcastmanager.CyclicSendTaskABC*

Adds support for restarting a stopped cyclic task

Parameters

- **message** (can.Message) – The message to be sent periodically.
- **period** (*float*) – The rate in seconds at which to send the message.

start()

Restart a stopped periodic task.

Functional API

Warning: The functional API in *can.broadcastmanager.send_periodic()* is now deprecated and will be removed in version 4.0. Use the object oriented API via *can.BusABC.send_periodic()* instead.

can.broadcastmanager.**send_periodic**(*bus*, *message*, *period*, **args*, ***kwargs*)

Send a *Message* every *period* seconds on the given bus.

Parameters

- **bus** (can.BusABC) – A CAN bus which supports sending.
- **message** (can.Message) – Message to send periodically.
- **period** (*float*) – The minimum time between sending messages.

Returns A started task instance

3.7 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.

3.7.1 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. This is the entire ABC bus class with all internal methods:

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

Bases: object

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.

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** – The can interface identifier. Expected type is backend dependent.
- **can_filters** (*list*) – See *set_filters()* for details.
- **kwargs** (*dict*) – Any backend dependent configurations are passed in this dictionary

RECV_LOGGING_LEVEL = 9

Log level for received messages

__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** – The can interface identifier. Expected type is backend dependent.
- **can_filters** (*list*) – See *set_filters()* for details.
- **kwargs** (*dict*) – Any backend dependent configurations are passed in this dictionary

__iter__()

Allow iteration on messages as they are received.

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

Yields *can.Message* msg objects.

__metaclass__

alias of abc.ABCMeta

__str__()

Return str(self).

__weakref__

list of weak references to the object (if defined)

_apply_filters(*filters*)

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

Parameters **filters** (*Iterator[dict]*) – See *set_filters()* for details.

static _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.

Return type Iterator[dict]

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

_matches_filters(*msg*)
Checks whether the given message matches at least one of the current filters. See *set_filters()* for details on how the filters work.

This method should not be overridden.

Parameters **msg** (can.Message) – the message to check if matching

Return type bool

Returns whether the given message matches at least one filter

_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()*

Return type tuple[*can.Message*, bool] or tuple[None, bool]

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

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

_send_periodic_internal(*msg*, *period*, *duration=None*)
Default implementation of periodic message sending using threading.

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

Parameters

- **msg** (can.Message) – Message to transmit
- **period** (*float*) – Period in seconds between each message

- **duration** (*float*) – The duration to keep sending this message at given rate. If no duration is provided, the task will continue indefinitely.

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

Return type *can.broadcastmanager.CyclicSendTaskABC*

channel_info = 'unknown'
a string describing the underlying bus and/or channel

filters
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).

recv(*timeout=None*)
Block waiting for a message from the Bus.

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

Return type *can.Message* or None

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

Raises ***can.CanError*** – if an error occurred while reading

send(*msg*, *timeout=None*)
Transmit a message to the CAN bus.

Override this method to enable the transmit path.

Parameters

- **msg** (*can.Message*) – A message object.
- **timeout** (*float or 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 ***can.CanError*** – if the message could not be sent

send_periodic(*msg*, *period*, *duration=None*, *store_task=True*)
Start sending a message 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
- *BusABC.stop_all_periodic_tasks()* is called
- the task's `CyclicTask.stop()` method is called.

Parameters

- **msg** (*can.Message*) – Message to transmit
- **period** (*float*) – Period in seconds between each message

- **duration** (*float*) – The duration to keep sending this message at given rate. 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.

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

Return type *can.broadcastmanager.CyclicSendTaskABC*

Note: Note the duration before the message stops 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** – 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.

shutdown ()

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

state

Return the current state of the hardware

Type can.BusState

stop_all_periodic_tasks (*remove_tasks=True*)

Stop sending any messages that were started using **bus.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.

3.7.2 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 a generic class for file IO.

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

Bases: object

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

Can be used as a context manager.

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

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** (*str*) – the mode that should be used to open the file, see `open()`, ignored if *file* is *None*

3.7.3 Other Utilities

Utilities and configuration file parsing.

can.util.**channel2int**(*channel*)
Try to convert the channel to an integer.

Parameters **channel** – Channel string (e.g. can0, CAN1) or integer

Returns Channel integer or *None* if unsuccessful

Return type int

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** – Optional path to config file.
- **config** – A dict which may set the 'interface', and/or the 'channel', or neither. It may set other values that are passed through.
- **context** – Extra 'context' pass to config sources. This can be use 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 NotImplementedError if the `interface` isn't recognized

`can.util.`**`load_environment_config`**`()`
Loads config dict from environmental variables (if set):

- CAN_INTERFACE
- CAN_CHANNEL
- CAN_BITRATE

`can.util.`**`load_file_config`**`(`*path=None*, *section=None*`)`
Loads configuration from file with following content:

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

Parameters

- **`path`** – path to config file. If not specified, several sensible default locations are tried depending on platform.
- **`section`** – name of the section to read configuration from.

`can.util.`**`set_logging_level`**`(`*level_name=None*`)`
Set the logging level for the "can" logger. Expects one of: 'critical', 'error', 'warning', 'info', 'debug', 'subdebug'

3.8 Utilities

3.9 Notifier

The Notifier object is used as a message distributor for a bus.

`class` `can.`**`Notifier`**`(`*bus*, *listeners*, *timeout=1.0*, *loop=None*`)`
Bases: `object`

Manages the distribution of *`can.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`** (`can.BusABC`) – A *Bus* or a list of buses to listen to.
- **`listeners`** (*`list`*) – An iterable of *`Listener`*
- **`timeout`** (*`float`*) – An optional maximum number of seconds to wait for any message.

- **loop** (*asyncio.AbstractEventLoop*) – An asyncio event loop to schedule listeners in.

add_bus(*bus*)
Add a bus for notification.

Parameters **bus** (can.BusABC) – CAN bus instance.

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** (can.Listener) – Listener to be added to the list to be notified

exception = None
Exception raised in thread

remove_listener(*listener*)
Remove a listener from the notification list. This method trows an exception if the given listener is not part of the stored listeners.

Parameters **listener** (can.Listener) – Listener to be removed from the list to be notified

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

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.

3.10 Errors

class can.**CanError**
Bases: OSError

Indicates an error with the CAN network.

CHAPTER 4

CAN Interface Modules

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.

The available interfaces are:

4.1 SocketCAN

The full documentation for socketcan can be found in the kernel docs at networking/can.txt.

Note: Versions before 2.2 had two different implementations named `socketcan_ctypes` and `socketcan_native`. These are now deprecated and the aliases to `socketcan` will be removed in version 4.0. 3.x releases raise a DeprecationWarning.

4.1.1 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
```

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
```

Send Test Message

The can-utils library for linux includes a script *cansend* 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
```

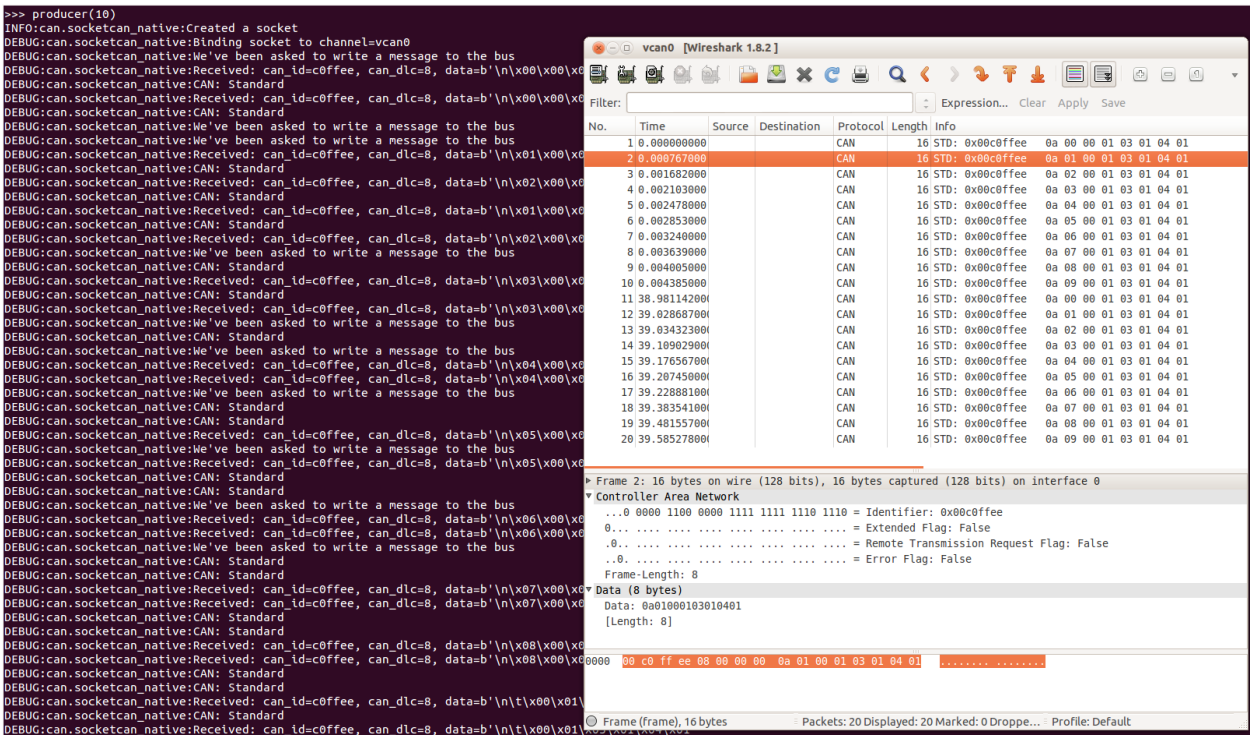
4.1.2 Wireshark

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

bustype = 'socketcan'
channel = 'vcan0'

def producer(id):
    """:param id: Spam the bus with messages including the data id."""
    bus = can.interface.Bus(channel=channel, bustype=bustype)
    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)
```

4.1.3 Reading and Timeouts

Reading a single CAN message off of the bus is simple with the `bus.recv()` function:

```
import can

can_interface = 'vcan0'
bus = can.interface.Bus(can_interface, bustype='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.

4.1.4 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.

4.1.5 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.

send_periodic()

An example that uses the send_periodic is included in `python-can/examples/cyclic.py`

The object returned can be used to halt, alter or cancel the periodic message task.

class can.interfaces.socketcan.**CyclicSendTask**(*bcm_socket*, *message*, *period*, *duration=None*)

Bases: *can.broadcastmanager.LimitedDurationCyclicSendTaskABC*, can.broadcastmanager.ModifiableCyclicTaskABC, can.broadcastmanager.RestartableCyclicTaskABC

A socketcan cyclic send task supports:

- setting of a task duration
- modifying the data
- stopping then subsequent restarting of the task

Parameters

- **bcm_socket** – An open bcm socket on the desired CAN channel.
- **message** (can.Message) – The message to be sent periodically.
- **period** (*float*) – The rate in seconds at which to send the message.
- **duration** (*float*) – Approximate duration in seconds to send the message.

modify_data(*message*)

Update the contents of this periodically sent message.

Note the Message must have the same *arbitration_id* like the first message.

start()

Restart a stopped periodic task.

stop()

Send a TX_DELETE message to cancel this task.

This will delete the entry for the transmission of the CAN-message with the specified can_id CAN identifier. The message length for the command TX_DELETE is {[bcm_msg_head]} (only the header).

4.1.6 Bus

class can.interfaces.socketcan.**SocketcanBus**(*channel=''*, *receive_own_messages=False*, *fd=False*, ***kwargs*)

Bases: can.bus.BusABC

Implements can.BusABC._detect_available_configs().

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.
- **fd** (*bool*) – If CAN-FD frames should be supported.
- **can_filters** (*list*) – See *can.BusABC.set_filters()*.

recv(*timeout=None*)
Block waiting for a message from the Bus.

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

Return type *can.Message* or None

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

Raises ***can.CanError*** – if an error occurred while reading

send(*msg*, *timeout=None*)
Transmit a message to the CAN bus.

Parameters

- **msg** (can.Message) – A message object.
- **timeout** (*float*) – Wait up to this many seconds for the transmit queue to be ready. If not given, the call may fail immediately.

Raises ***can.CanError*** – if the message could not be written.

shutdown()
Stops all active periodic tasks and closes the socket.

4.2 Kvaser's CANLIB

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

4.2.1 Bus

class can.interfaces.kvaser.canlib.**KvaserBus**(*channel*, *can_filters=None*, ***kwargs*)
Bases: can.bus.BusABC

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()*.

Backend Configuration

Parameters

- **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.
- **data_bitrate** (*int*) – Which bitrate to use for data phase in CAN FD. Defaults to arbitration bitrate.

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** (*can.Message*) – A message object.
- **timeout** (*float or 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 ***can.CanError*** – if the message could not be sent

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

4.2.2 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 are done 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 can.interfaces.kvaser.structures.BusStatistics

4.3 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.

4.3.1 Bus

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

Bases: `can.bus.BusABC`

Enable basic can communication over a serial device.

Note: See `can.interfaces.serial.SerialBus._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

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

Send a message over the serial device.

Parameters

- **msg** (`can.Message`) – Message to send.

  Note: Flags like `extended_id`, `is_remote_frame` and `is_error_frame` will be ignored.

  Note: If the timestamp is a float value it will be converted to an integer.

- **timeout** – This parameter will be ignored. The timeout value of the channel is used instead.

shutdown()

Close the serial interface.

4.3.2 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	0xBBS

4.4 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.

4.4.1 Supported devices

Todo: Document this.

4.4.2 Bus

class can.interfaces.slcan.**slcanBus**(*channel*, *ttyBaudrate=115200*, *bitrate=None*, *btr=None*, *sleep_after_open=2*, *rtscts=False*, ***kwargs*)

Bases: can.bus.BusABC

slcan interface

Raises **ValueError** – if both *bitrate* and *btr* are set

Parameters

- **channel** (*str*) – port of underlying serial or usb device (e.g. /dev/ttyUSB0, COM8, ...) Must not be empty.
- **ttyBaudrate** (*int*) – baudrate of underlying serial or usb device
- **bitrate** (*int*) – Bitrate in bit/s
- **btr** (*str*) – BTR register value to set custom can speed
- **poll_interval** (*float*) – 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

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

Transmit a message to the CAN bus.

Override this method to enable the transmit path.

Parameters

- **msg** (can.Message) – A message object.
- **timeout** (*float or 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 ***can.CanError*** – if the message could not be sent

shutdown()

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

4.4.3 Internals

Todo: Document the internals of slcan interface.

4.5 IXXAT Virtual CAN Interface

Interface to IXXAT Virtual CAN 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.

4.5.1 Bus

4.5.2 Configuration file

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.

- `bitrate` (default 500000) Channel bitrate
- `UniqueHardwareId` (default first device) Unique hardware ID of the IXXAT device
- `rxFifoSize` (default 16) Number of RX mailboxes
- `txFifoSize` (default 16) Number of TX mailboxes
- `extended` (default False) Allow usage of extended IDs

4.5.3 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 *do 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 `rxFifoSize` and `txFifoSize` options, defaulting at 16 for both.

The CAN filters act as a "whitelist" 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 PCAN Basic API

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

Windows driver: https://www.peak-system.com/Downloads.76.0.html?&L=1

Linux driver: https://www.peak-system.com/fileadmin/media/linux/index.htm#download and https://www.peak-system.com/Downloads.76.0.html?&L=1 (PCAN-Basic API (Linux))

Mac driver: http://www.mac-can.com

4.6.1 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

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

`bitrate`: (default 500000) Channel bitrate

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.

4.6.2 Linux installation

Kernels >= 3.4 supports the PCAN adapters natively via *SocketCAN*, refer to: *PCAN*.

4.6.3 Bus

class can.interfaces.pcan.**PcanBus**(*channel='PCAN_USBBUS1'*, *state=<BusState.ACTIVE: 1>*, *bitrate=500000*, **args*, ***kwargs*)

Bases: `can.bus.BusABC`

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' Default is 'PCAN_USBBUS1'
- **state** (*can.bus.BusState*) – BusState of the channel. Default is ACTIVE
- **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.

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

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** (*can.Message*) – A message object.
- **timeout** (*float or 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 *can.CanError* – if the message could not be sent

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

state
Return the current state of the hardware

Type can.BusState

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

4.7 USB2CAN Interface

4.7.1 OVERVIEW

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.

4.7.2 WINDOWS SUPPORT

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. This device can be used through one of three ways 1)Through python-can 2)CANAL API either using the DLL and C/C++ or through the python wrapper that has been added to this project 3)VSCP Using python-can is strongly suggested as with little extra work the same interface can be used on both Windows and Linux.

4.7.3 WINDOWS INSTALL

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. (Written against CANAL DLL version v1.0.6)

4.7.4 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.

4.7.5 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**

    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 length of information so 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. There is a Wireshark file posted in Issue #36 that demonstrates that the bus is only sending the data and not the extra zeros.
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!

4.7.6 Bus

4.7.7 Internals

4.8 NI-CAN

This interface adds support for 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.

4.8.1 Bus

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

Bases: can.bus.BusABC

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*) – Bitrate in bits/s
- **can_filters** (*list*) – 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 ***can.interfaces.nican.NicanError*** – If starting communication fails

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.

send(*msg*, *timeout=None*)
Send a message to NI-CAN.

Parameters **msg** (can.Message) – Message to send

Raises ***can.interfaces.nican.NicanError*** – If writing to transmit buffer fails. It does not wait for message to be ACKed currently.

set_filters(*can_filers=None*)
Unsupported. See note on *NicanBus*.

shutdown()
Close object.

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

Bases: *can.CanError*

Error from NI-CAN driver.

arguments = None
Arguments passed to function

error_code = None
Status code

function = None
Function that failed

4.9 isCAN

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

4.9.1 Bus

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

Bases: can.bus.BusABC

isCAN interface

Parameters

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

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

Transmit a message to the CAN bus.

Override this method to enable the transmit path.

Parameters

- **msg** (can.Message) – A message object.
- **timeout** (*float or 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 ***can.CanError*** – if the message could not be sent

shutdown()

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

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

Bases: *can.CanError*

4.10 NEOVI Interface

Warning: 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.

4.10.1 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-ics**

```
pip install python-ics
```

4.10.2 Configuration

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

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

4.10.3 Bus

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

Bases: can.bus.BusABC

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.

static get_serial_number(*device*)

Decode (if needed) and return the ICS device serial string

Parameters **device** – ics device

Returns ics device serial string

Return type str

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

Transmit a message to the CAN bus.

Override this method to enable the transmit path.

Parameters

- **msg** (can.Message) – A message object.

- **timeout** (*float or 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 *can.CanError* – if the message could not be sent

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

4.11 Vector

This interface adds support for CAN controllers by Vector.

By default this library uses the channel configuration for CANalyzer. To use a different application, open Vector Hardware Config 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
```

If you are using Python 2.7 it is recommended to install pywin32, otherwise a slow and CPU intensive polling will be used when waiting for new messages.

4.11.1 Bus

class can.interfaces.vector.**VectorBus**(*channel*, *can_filters=None*, *poll_interval=0.01*, *receive_own_messages=False*, *bitrate=None*, *rx_queue_size=16384*, *app_name='CANalyzer'*, *serial=None*, *fd=False*, *data_bitrate=None*, *sjwAbr=2*, *tseg1Abr=6*, *tseg2Abr=3*, *sjwDbr=2*, *tseg1Dbr=6*, *tseg2Dbr=3*, ***kwargs*)

Bases: `can.bus.BusABC`

The CAN Bus implemented for the Vector interface.

Parameters

- **channel** (*list*) – The channel indexes to create this bus with. Can also be a single integer or a comma separated string.
- **poll_interval** (*float*) – Poll interval in seconds.
- **bitrate** (*int*) – 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*) – Name of application in Hardware Config. If set to None, the channel should be a global channel index.
- **serial** (*int*) – 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 is unused.

- **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.

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

send(*msg*, *timeout=None*)
Transmit a message to the CAN bus.

Override this method to enable the transmit path.

Parameters

- **msg** (can.Message) – A message object.
- **timeout** (*float or 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 *can.CanError* – if the message could not be sent

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

exception can.interfaces.vector.**VectorError**(*error_code*, *error_string*, *function*)
Bases: *can.CanError*

4.12 Virtual

The virtual interface can be used as a way to write OS and driver independent tests.

A virtual CAN bus that can be used for automatic tests. Any Bus instances connecting to the same channel (in the same python program) will get each others messages.

```
import can

bus1 = can.interface.Bus('test', bustype='virtual')
bus2 = can.interface.Bus('test', bustype='virtual')

msg1 = can.Message(arbitration_id=0xabcde, data=[1,2,3])
bus1.send(msg1)
msg2 = bus2.recv()

assert msg1 == msg2
```

4.13 CANalyst-II

CANalyst-II(+) is a USB to CAN Analyzer. The controlcan library is originally developed by ZLG ZHIYUAN Electronics.

4.13.1 Bus

class can.interfaces.canalystii.**CANalystIIBus**(*channel*, *device=0*, *bitrate=None*, *baud=None*, *Timing0=None*, *Timing1=None*, *can_filters=None*, ***kwargs*)

Bases: can.bus.BusABC

Parameters

- **channel** – channel number
- **device** – device number
- **baud** – baud rate. Renamed to bitrate in next release.
- **Timing0** – customize the timing register if baudrate is not specified
- **Timing1** –
- **can_filters** – filters for packet

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

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

Parameters

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

Returns

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

4.14 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.

4.14.1 Installation

The interface requires installation of the **USBCAN32.dll** library. Download and install the driver for specific SYSTEC device.

4.14.2 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.

4.14.3 Bus

class can.interfaces.systec.ucanbus.**UcanBus**(*channel*, *can_filters=None*, ***kwargs*)
Bases: can.bus.BusABC

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.
- ***can.CanError*** – 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 *can.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 *can.CanError* – If the message could not be sent.

shutdown()
Shuts down all CAN interfaces and hardware interface.

state
Return the current state of the hardware

Type can.BusState

4.14.4 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

4.14.5 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`.

Additional interfaces can be added via a plugin interface. An external package can register a new interface by using the `can.interface` entry point in its setup.py.

The format of the entry point is `interface_name=module:classname` where `classname` is a concrete *can.BusABC* implementation.

```
entry_points={
    'can.interface': [
        "interface_name=module:classname",
    ]
},
```

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

CHAPTER 5

Scripts

The following modules are callable from python-can.

They can be called for example by `python -m can.logger` or `can_logger.py` (if installed using pip).

5.1 can.logger

Command line help, called with `--help`:

```
$ python -m can.logger -h
usage: python -m can.logger [-h] [-f LOG_FILE] [-v] [-c CHANNEL]
                            [-i {virtual,vector,usb2can,canalystii,kvaser,socketcan_
↪ctypes,socketcan,ixxat,slcan,neovi,iscan,nican,socketcan_native,pcan,serial,systec}]
                            [--filter ...] [-b BITRATE] [--active | --passive]

Log CAN traffic, printing messages to stdout or to a given file.

optional arguments:
  -h, --help            show this help message and exit
  -f LOG_FILE, --file_name LOG_FILE
                        Path and base log filename, for supported types see
                        can.Logger.
  -v                    How much information do you want to see at the command
                        line? You can add several of these e.g., -vv is DEBUG
  -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 interfaces valid channel examples include:
                        "can0", "vcan0"
  -i {virtual,vector,usb2can,canalystii,kvaser,socketcan_ctypes,socketcan,ixxat,slcan,
↪neovi,iscan,nican,socketcan_native,pcan,serial,systec}, --interface {virtual,vector,
↪usb2can,canalystii,kvaser,socketcan_ctypes,socketcan,ixxat,slcan,neovi,iscan,nican,
↪socketcan_native,pcan,serial,systec}
```

(continues on next page)

(continued from previous page)

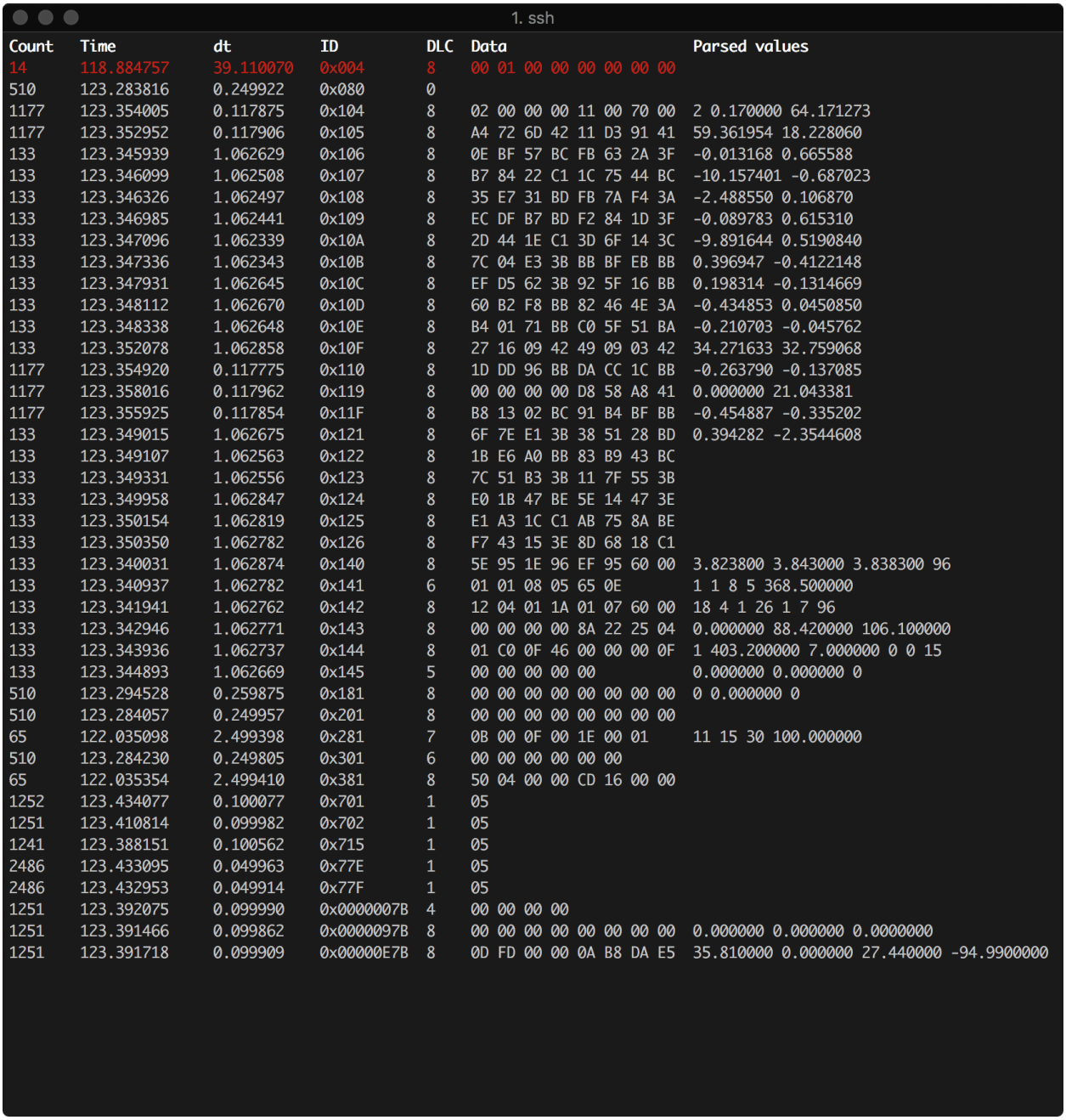
```
                        Specify the backend CAN interface to use. If left
                        blank, fall back to reading from configuration files.
  --filter ...          Comma separated filters can be specified for the given
                        CAN interface: <can_id>:<can_mask> (matches when
                        <received_can_id> & mask == can_id & mask)
                        <can_id>~<can_mask> (matches when <received_can_id> &
                        mask != can_id & mask)
  -b BITRATE, --bitrate BITRATE
                        Bitrate to use for the CAN bus.
  --active              Start the bus as active, this is applied by default.
  --passive             Start the bus as passive.
```

5.2 can.player

```
$ python -m can.player -h
usage: python -m can.player [-h] [-f LOG_FILE] [-v] [-c CHANNEL]
                            [-i {serial,kvaser,socketcan,socketcan_native,ixxat,iscan,
↪pcan,virtual,neovi,slcan,systec,usb2can,socketcan_ctypes,vector,canalystii,nican}]
                            [-b BITRATE] [--ignore-timestamps]
                            [--error-frames] [-g GAP] [-s SKIP]
                            input-file

Replay CAN traffic.

positional arguments:
  input-file            The file to replay. For supported types see
                        can.LogReader.

optional arguments:
  -h, --help            show this help message and exit
  -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
  -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 interfaces valid channel examples include:
                        "can0", "vcan0"
  -i {serial,kvaser,socketcan,socketcan_native,ixxat,iscan,pcan,virtual,neovi,slcan,
↪systec,usb2can,socketcan_ctypes,vector,canalystii,nican}, --interface {serial,
↪kvaser,socketcan,socketcan_native,ixxat,iscan,pcan,virtual,neovi,slcan,systec,
↪usb2can,socketcan_ctypes,vector,canalystii,nican}
                        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.
  --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
```

5.3 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.

5.3.1 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
Usage: python -m can.viewer [-h] [--version] [-b BITRATE] [-c CHANNEL]
                            [-d {<id>:<format>,<id>:<format>:<scaling1>:...:<scalingN>
↪,file.txt}]
                            [-f {<can_id>:<can_mask>,<can_id>~<can_mask>}]
                            [-i {canalystii,iscan,ixxat,kvaser,neovi,nican,pcan,
↪serial,slcan,socketcan,socketcan_ctypes,socketcan_native,systec,usb2can,vector,
↪virtual}]

A simple CAN viewer terminal application written in Python

Optional arguments:
  -h, --help            Show this help message and exit
  --version             Show program's version number and exit
  -b, --bitrate BITRATE
                        Bitrate to use for the given CAN interface
  -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 interfaces valid channel examples include:
                        "can0", "vcan0". (default: use default for the
                        specified interface)
  -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
```

(continues on next page)

(continued from previous page)

```
                        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.
                        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 alway interpreted as hex values
  -i, --interface {canalystii,iscan,ixxat,kvaser,neovi,nican,pcan,serial,slcan,
↪socketcan,socketcan_ctypes,socketcan_native,systec,usb2can,vector,virtual}
                        Specify the backend CAN interface to use.

Shortcuts:
        +---------+-------------------------+
        |   Key   |       Description       |
        +---------+-------------------------+
        | ESQ/q   | Exit the viewer         |
        | c       | Clear the stored frames |
        | s       | Sort the stored frames  |
        | SPACE   | Pause the viewer        |
        | UP/DOWN | Scroll the viewer       |
        +---------+-------------------------+
```

CHAPTER 6

Developer's Overview

6.1 Contributing

Contribute to source code, documentation, examples and report issues: https://github.com/hardbyte/python-can

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.

6.2 Building & Installing

The following assumes that the commands are executed from the root of the repository:

- The project can be built and installed with `python setup.py build` and `python setup.py install`.
- The unit tests can be run with `python setup.py test`. The tests can be run with `python2`, `python3`, `pypy` or `pypy3` to test with other python versions, if they are installed. Maybe, you need to execute `pip3 install python-can[test]` (or only `pip` for Python 2), if some dependencies are missing.
- The docs can be built with `sphinx-build doc/ doc/_build`. Appending `-n` to the command makes Sphinx complain about more subtle problems.

6.3 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 `can.interface.BACKENDS` and `can.interfaces.VALID_INTERFACES` in `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/*`.
- Update `doc/scripts.rst` accordingly.
- Add tests in `test/*` where appropriate.

6.4 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.
CAN	Legacy API. Deprecated.

6.5 Process for creating a new Release

Note many of these steps are carried out by the CI system on creating a tag in git.

- Release from the `master` branch.
- 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 `python setup.py test`.
- Ensure the `setuptools` and `wheel` tools are up to date: `pip install -U setuptools wheel`.
- Create and upload the distribution: `python setup.py sdist bdist_wheel`.
- [Optionally] 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 7

History and Roadmap

7.1 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.

7.2 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.

7.3 Support for CAN within Python

Python natively supports the CAN protocol from version 3.3 on, if running on Linux:

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

CHAPTER 8

Known Bugs

See the project bug tracker on github. Patches and pull requests very welcome!

Documentation generated

Oct 07, 2020

Python Module Index

C

Index

Symbols

A

B

C

D

E

F

T

U

V