python-can

Release 2.2.1

Sep 24, 2018

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:

```python
#!/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 http://python-can.readthedocs.io/en/latest/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],
                      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 Mercurial 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.interfaces.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` 1. `can.ini` (current working directory) 2. `$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>
```

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*

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

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

3.1.1 API

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

Bases: `object`

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

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.
- **config** (`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.

`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`) – 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`) – 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.

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

`send_periodic`(*msg*, *period*, *duration=None*)
Start sending a message at a given period on this bus.

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

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.

`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 :return: ACTIVE, PASSIVE or ERROR :rtype: NamedTuple

3.1.2 Transmitting

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

3.1.3 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.

3.1.4 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 exemple 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 Autoconfig Bus

class `can.interface.`**`Bus`**(*channel*, *can_filters=None*, ***config*)

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.
- **`config`** (`dict`) – Any backend dependent configurations are passed in this dictionary

3.4 Message

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

Bases: `object`

The `Message` object is used to represent CAN messages for both sending and receiving.

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

When testing for equality of the messages, the timestamp and the channel is not used for comparing.

Note: This class does not strictly check the input. Thus, the caller must prevent the creation of invalid messages. 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.

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

```
>>> from can import Message
>>> test = Message(data=[1, 2, 3, 4, 5])
>>> test.data
bytearray(b'\x01\x02\x03\x04\x05')
>>> test.dlc
5
>>> print(test)
Timestamp:        0.000000    ID: 00000000    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.

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(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 Link Count) 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.

---

is_extended_id

Type bool

This flag controls the size of the *arbitration_id* field.

```
>>> print(Message(extended_id=False))
Timestamp:        0.000000        ID: 0000    S        DLC: 0
>>> print(Message(extended_id=True))
Timestamp:        0.000000    ID: 00000000    X        DLC: 0
```

Previously this was exposed as *id_type*.

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 boolean

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.

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.

__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.

3.5 Listeners

3.5.1 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`

 stop()
  Override to cleanup any open resources.

3.5.2 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.

`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 (whichever is shorter),

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

Returns the `Message` if there is one, or None if there is not.

3.5.3 Logger

The `can.Logger` uses the following `can.Listener` types to create *.asc*, *.csv* and *.db* files with the messages received.

class can.**Logger**
Bases: `object`

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`

Note this class itself is just a dispatcher, an object that inherits from Listener will be created when instantiating this class.

3.5.4 Printer

class can.**Printer**(*output_file=None*)
Bases: `can.listener.Listener`

The Printer class is a subclass of `Listener` which simply prints any messages it receives to the terminal (stdout).

Parameters **`output_file`** – An optional file to "print" to.

`stop`()
Override to cleanup any open resources.

3.5.5 CSVWriter

class can.**CSVWriter**(*filename*)
Bases: `can.listener.Listener`

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

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 seperator.

stop()

Override to cleanup any open resources.

3.5.6 SqliteWriter

class can.**SqliteWriter**(*filename*)

Bases: `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.

Note: When the listener's *stop()* method is called the thread writing to the sql file 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 sql file.

However if the bus is still saturated with messages, the Listener will continue receiving until the *MAX_TIME_BETWEEN_WRITES* timeout is reached.

GET_MESSAGE_TIMEOUT = 0.25

Number of seconds to wait for messages from internal queue

MAX_TIME_BETWEEN_WRITES = 5

Maximum number of seconds to wait between writes to the database

stop()

Override to cleanup any open resources.

Database table format

The messages are written to the table `messages` in the sqlite database. 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

3.5.7 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**(*filename*, *channel=1*)

Bases: `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 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.

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

stop()

Stops logging and closes the file.

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

class can.**ASCReader**(*filename*)

Bases: `object`

Iterator of CAN messages from a ASC logging file.

TODO: turn realtive timestamps back to absolute form

3.5.8 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.io.**CanutilsLogWriter**(*filename*, *channel='vcan0'*)

Bases: `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.

stop()
Stops logging and closes the file.

CanutilsLogReader reads CAN data from ASCII log files .log

class can.io.**CanutilsLogReader**(*filename*)
Bases: `object`

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

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

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

3.5.9 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**(*filename*, *channel=1*)
Bases: `can.listener.Listener`

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

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.

stop()
Stops logging and closes the file.

class can.**BLFReader**(*filename*)
Bases: `object`

Iterator of CAN messages from a Binary Logging File.

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

3.6 Broadcast Manager

The broadcast manager isn't yet supported by all interfaces. Currently SockerCAN and IXXAT are supported at least partially. It allows the user to setup periodic message jobs.

If periodic transmission is not supported natively, a software thread based scheduler is used as a fallback.

This example shows the socketcan_ctypes backend using the broadcast manager:

```python
#!/usr/bin/env python
# coding: utf-8

"""
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], 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], extended_
↪id=True)
    task = bus.send_periodic(msg, 0.20, 1)
    if not isinstance(task, can.LimitedDurationCyclicSendTaskABC):
        print("This interface doesn't seem to support a ")
        task.stop()
        return

    time.sleep(1.5)
    print("stopped cyclic send")

```

(continues on next page)

(continued from previous page)

```

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)
    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__":
```

(continues on next page)

(continued from previous page)

```python

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


    for interface, channel in [
        ('socketcan_ctypes', 'can0'),
        ('socketcan_native', 'can0')
        #('ixxat', 0)
    ]:
        print("Carrying out cyclic tests with {} interface".format(interface))

        bus = can.interface.Bus(bustype=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 Class based API

class can.broadcastmanager.**CyclicTask**

Bases: `object`

Abstract Base for all Cyclic Tasks

stop()

Cancel this periodic task.

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

Bases: *can.broadcastmanager.CyclicTask*

Message send task with defined period

Parameters

- **message** – The *can.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** – The `can.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.`**RestartableCyclicTaskABC**(*message*, *period*)

Bases: `can.broadcastmanager.CyclicSendTaskABC`

Adds support for restarting a stopped cyclic task

Parameters

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

start()

Restart a stopped periodic task.

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

Bases: `can.broadcastmanager.CyclicSendTaskABC`

Adds support for modifying a periodic message

Parameters

- **message** – The `can.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** – The `Message` with new `can.Message.data`.

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

Bases: `can.broadcastmanager.CyclicSendTaskABC`

Exposes more of the full power of the TX_SETUP opcode.

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

class `can.broadcastmanager.`**ThreadBasedCyclicSendTask**(*bus*, *lock*, *message*, *period*, *duration=None*)

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

Fallback cyclic send task using thread.

start()

Restart a stopped periodic task.

stop()

Cancel this periodic task.

3.6.2 Functional API

Note: The functional API in `can.broadcastmanager.send_periodic()` is now deprecated. 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 channel.

Parameters

- **`bus`** – The `can.BusABC` to transmit to.
- **`message`** – The `can.Message` instance to periodically send

Returns A started task instance

3.7 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*`)`

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.

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*`)`

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.

`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'

`can.`**`detect_available_configs`**`()`

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

This might be quite time consuming.

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

Parameters **`interfaces`** – either - the name of an interface to be searched in as a string, - an iterable of interface names to search in, or - *None* to search in all known interfaces.

Return type list[dict]

Returns an iterable of dicts, each suitable for usage in *can.interface.Bus*'s constructor.

3.8 Notifier

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

class can.**Notifier**(*bus*, *listeners*, *timeout=1*)

Bases: object

Manages the distribution of **Messages** from a given bus/buses to a list of listeners.

Parameters

- **bus** (can.BusABC) – The ***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.

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.

3.9 Errors

class can.**CanError**

Bases: exceptions.IOError

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 3.0. Future 2.x release may 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
```

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:

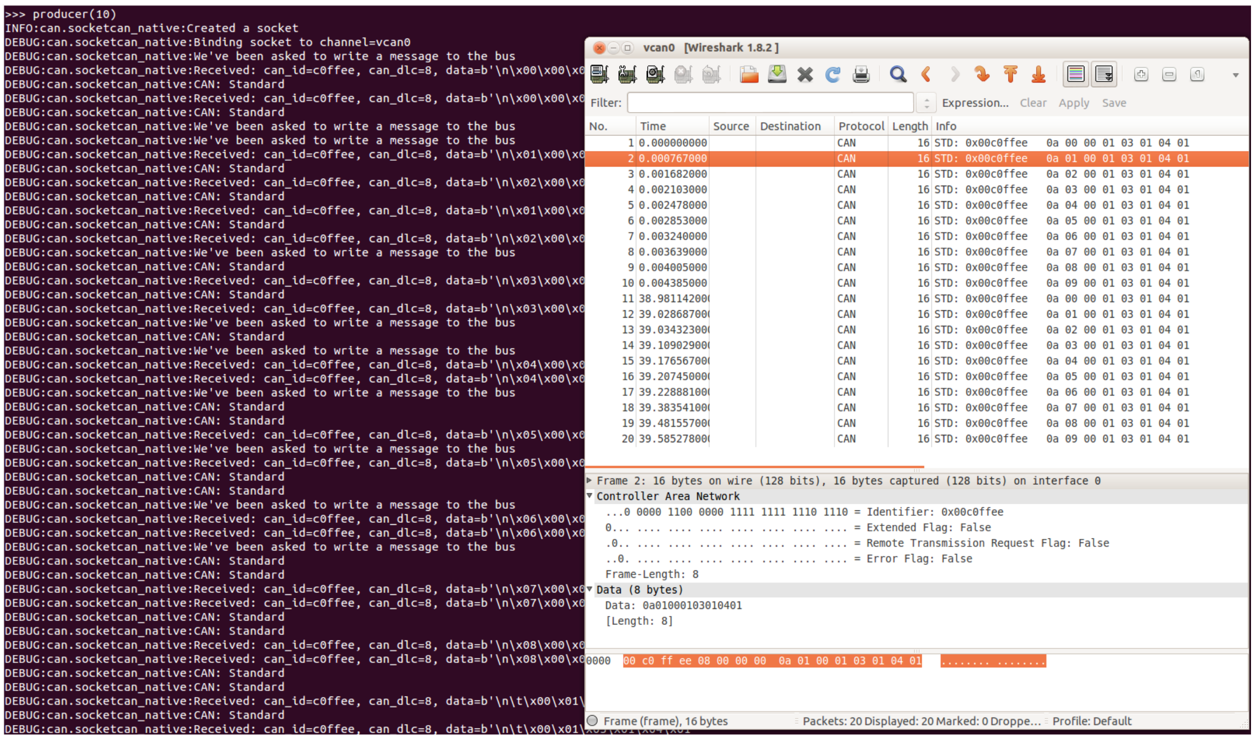
```python
import time
import can

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],␣
↪extended_id=False)
        bus.send(msg)
    # Issue #3: Need to keep running to ensure the writing threads stay alive. ?
    time.sleep(1)

producer(10)
```

With debugging turned right up this looks something like this:



```
>>> producer(10)
INFO:can.socketcan_native:Created a socket
DEBUG:can.socketcan_native:Binding socket to channel=vcan0
DEBUG:can.socketcan_native:We've been asked to write a message to the bus
DEBUG:can.socketcan_native:Received: can_id=c0ffee, can_dlc=8, data=b'\n\x00\x00\x0
DEBUG:can.socketcan_native:CAN: Standard
DEBUG:can.socketcan_native:Received: can_id=c0ffee, can_dlc=8, data=b'\n\x00\x00\x0
DEBUG:can.socketcan_native:CAN: Standard
DEBUG:can.socketcan_native:We've been asked to write a message to the bus
DEBUG:can.socketcan_native:We've been asked to write a message to the bus
DEBUG:can.socketcan_native:Received: can_id=c0ffee, can_dlc=8, data=b'\n\x01\x00\x0
DEBUG:can.socketcan_native:CAN: Standard
DEBUG:can.socketcan_native:Received: can_id=c0ffee, can_dlc=8, data=b'\n\x02\x00\x0
DEBUG:can.socketcan_native:CAN: Standard
DEBUG:can.socketcan_native:Received: can_id=c0ffee, can_dlc=8, data=b'\n\x01\x00\x0
DEBUG:can.socketcan_native:CAN: Standard
DEBUG:can.socketcan_native:Received: can_id=c0ffee, can_dlc=8, data=b'\n\x02\x00\x0
DEBUG:can.socketcan_native:We've been asked to write a message to the bus
DEBUG:can.socketcan_native:CAN: Standard
DEBUG:can.socketcan_native:Received: can_id=c0ffee, can_dlc=8, data=b'\n\x03\x00\x0
DEBUG:can.socketcan_native:CAN: Standard
DEBUG:can.socketcan_native:Received: can_id=c0ffee, can_dlc=8, data=b'\n\x03\x00\x0
DEBUG:can.socketcan_native:We've been asked to write a message to the bus
DEBUG:can.socketcan_native:CAN: Standard
DEBUG:can.socketcan_native:We've been asked to write a message to the bus
DEBUG:can.socketcan_native:Received: can_id=c0ffee, can_dlc=8, data=b'\n\x04\x00\x0
DEBUG:can.socketcan_native:Received: can_id=c0ffee, can_dlc=8, data=b'\n\x04\x00\x0
DEBUG:can.socketcan_native:We've been asked to write a message to the bus
DEBUG:can.socketcan_native:CAN: Standard
DEBUG:can.socketcan_native:CAN: Standard
DEBUG:can.socketcan_native:Received: can_id=c0ffee, can_dlc=8, data=b'\n\x05\x00\x0
DEBUG:can.socketcan_native:We've been asked to write a message to the bus
DEBUG:can.socketcan_native:Received: can_id=c0ffee, can_dlc=8, data=b'\n\x05\x00\x0
DEBUG:can.socketcan_native:CAN: Standard
DEBUG:can.socketcan_native:CAN: Standard
DEBUG:can.socketcan_native:We've been asked to write a message to the bus
DEBUG:can.socketcan_native:Received: can_id=c0ffee, can_dlc=8, data=b'\n\x06\x00\x0
DEBUG:can.socketcan_native:Received: can_id=c0ffee, can_dlc=8, data=b'\n\x06\x00\x0
DEBUG:can.socketcan_native:We've been asked to write a message to the bus
DEBUG:can.socketcan_native:CAN: Standard
DEBUG:can.socketcan_native:CAN: Standard
DEBUG:can.socketcan_native:Received: can_id=c0ffee, can_dlc=8, data=b'\n\x07\x00\x0
DEBUG:can.socketcan_native:Received: can_id=c0ffee, can_dlc=8, data=b'\n\x07\x00\x0
DEBUG:can.socketcan_native:CAN: Standard
DEBUG:can.socketcan_native:CAN: Standard
DEBUG:can.socketcan_native:Received: can_id=c0ffee, can_dlc=8, data=b'\n\x08\x00\x0
DEBUG:can.socketcan_native:Received: can_id=c0ffee, can_dlc=8, data=b'\n\x08\x00\x0
DEBUG:can.socketcan_native:CAN: Standard
DEBUG:can.socketcan_native:CAN: Standard
DEBUG:can.socketcan_native:Received: can_id=c0ffee, can_dlc=8, data=b'\n\t\x00\x01\
DEBUG:can.socketcan_native:CAN: Standard
DEBUG:can.socketcan_native:Received: can_id=c0ffee, can_dlc=8, data=b'\n\t\x00\x01\
```

```
No.  Time         Source  Destination  Protocol  Length  Info
  1  0.000000000                       CAN           16  STD: 0x00c0ffee   0a 00 00 01 03 01 04 01
  2  0.000767000                       CAN           16  STD: 0x00c0ffee   0a 01 00 01 03 01 04 01
  3  0.001682000                       CAN           16  STD: 0x00c0ffee   0a 02 00 01 03 01 04 01
  4  0.002103000                       CAN           16  STD: 0x00c0ffee   0a 03 00 01 03 01 04 01
  5  0.002478000                       CAN           16  STD: 0x00c0ffee   0a 04 00 01 03 01 04 01
  6  0.002853000                       CAN           16  STD: 0x00c0ffee   0a 05 00 01 03 01 04 01
  7  0.003240000                       CAN           16  STD: 0x00c0ffee   0a 06 00 01 03 01 04 01
  8  0.003639000                       CAN           16  STD: 0x00c0ffee   0a 07 00 01 03 01 04 01
  9  0.004005000                       CAN           16  STD: 0x00c0ffee   0a 08 00 01 03 01 04 01
 10  0.004385000                       CAN           16  STD: 0x00c0ffee   0a 09 00 01 03 01 04 01
 11  38.98114200                       CAN           16  STD: 0x00c0ffee   0a 00 00 01 03 01 04 01
 12  39.02868700                       CAN           16  STD: 0x00c0ffee   0a 01 00 01 03 01 04 01
 13  39.03432300                       CAN           16  STD: 0x00c0ffee   0a 02 00 01 03 01 04 01
 14  39.10902900                       CAN           16  STD: 0x00c0ffee   0a 03 00 01 03 01 04 01
 15  39.17656700                       CAN           16  STD: 0x00c0ffee   0a 04 00 01 03 01 04 01
 16  39.20745000                       CAN           16  STD: 0x00c0ffee   0a 05 00 01 03 01 04 01
 17  39.22888100                       CAN           16  STD: 0x00c0ffee   0a 06 00 01 03 01 04 01
 18  39.38354100                       CAN           16  STD: 0x00c0ffee   0a 07 00 01 03 01 04 01
 19  39.48155700                       CAN           16  STD: 0x00c0ffee   0a 08 00 01 03 01 04 01
 20  39.58527800                       CAN           16  STD: 0x00c0ffee   0a 09 00 01 03 01 04 01

▶ Frame 2: 16 bytes on wire (128 bits), 16 bytes captured (128 bits) on interface 0
▼ Controller Area Network
    ...0 0000 1100 0000 1111 1111 1110 1110 = Identifier: 0x00c0ffee
    0... .... .... .... .... .... .... .... = Extended Flag: False
    .0.. .... .... .... .... .... .... .... = Remote Transmission Request Flag: False
    ..0. .... .... .... .... .... .... .... = Error Flag: False
    Frame-Length: 8
▼ Data (8 bytes)
    Data: 0a01000103010401
    [Length: 8]

0000  00 c0 ff ee 08 00 00 00  0a 01 00 01 03 01 04 01   ........ ........

Frame (frame), 16 bytes    Packets: 20 Displayed: 20 Marked: 0 Droppe...    Profile: Default
```

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`**(*channel*, *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

- **`channel`** (*`str`*) – The name of the CAN channel to connect to.
- **`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.

send_periodic(*msg*, *period*, *duration=None*)
Start sending a message at a given period on this bus.

The kernel's broadcast manager will be used.

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

Return type *can.interfaces.socketcan.CyclicSendTask*

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.

shutdown()
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*, ***config*)
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*) – 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.

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

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.

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. 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. These 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*, **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).

`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	Pay-load	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 the serial port, it makes slcan-compatible interfaces usable with Windows also. Hint: Arduino-Interface could easyly be build https://github.com/latonita/arduino-canbus-monitor

Usage: use `port[@baurate]` to open the device. For example use `/dev/ttyUSB0@115200` or `COM4@9600`

4.4.1 Bus

4.4.2 Internals

Todo: Document internals of slcan interface.

4.5 IXXAT Virtual CAN Interface

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

Note: 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, 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 farly 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.

Hint: Module uses `can.ixxat` logger and at DEBUG level logs every frame sent or received. It may be too verbose for your purposes.

4.6 PCAN Basic API

Warning: This `PCAN` documentation is a work in progress. Feedback and revisions are most welcome!

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

4.6.1 Configuration

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

```
[default]
interface = pcan
channel = PCAN_USBBUS1
```

4.6.2 Bus

class `can.interfaces.pcan.`**`PcanBus`**(*channel*, *state=<property object>*, *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'
- **state** (*BusState*) – BusState of the channel. Default is ACTIVE
- **bitrate** (*int*) – Bitrate of channel in bit/s. Default is 500 kbit/s.

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*) – 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.

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

shutdown()

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

status()

Query the PCAN bus status.

Returns The status code. See values in pcan_constants.py

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 runni**

   (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**

   (a) Use usb2canWin.py to find the serial number

   (b) Look on the device and enter it either through a prompt/barcode scanner/hardcode it.(Not recommended)

   (c) 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

class `can.interfaces.usb2can.`**`Usb2canBus`**(*channel*, **args*, ***kwargs*)

Bases: `can.bus.BusABC`

Interface to a USB2CAN Bus.

Parameters

- **`channel`** (`str`) – The device's serial number. If not provided, Windows Management Instrumentation will be used to identify the first such device. The *kwarg serial* may also be used.
- **`bitrate`** (`int`) – Bitrate of channel in bit/s. Values will be limited to a maximum of 1000 Kb/s. Default is 500 Kbs
- **`flags`** (`int`) – Flags to directly pass to open function of the usb2can abstraction layer.

`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`) – 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.

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

`shutdown`()

Shut down the device safely

4.7.7 Internals

class `can.interfaces.usb2can.`**`Usb2CanAbstractionLayer`**

A low level wrapper around the usb2can library.

Documentation: http://www.8devices.com/media/products/usb2can/downloads/CANAL_API.pdf

`blocking_receive`(*handle*, *msg*, *timeout*)

`blocking_send`(*handle*, *msg*, *timeout*)

close (*handle*)

get_library_version ()

get_statistics (*handle, CanalStatistics*)

get_status (*handle, CanalStatus*)

get_vendor_string ()

get_version ()

open (*pConfigureStr, flags*)

receive (*handle, msg*)

send (*handle, msg*)

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 throughly 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

flush_tx_buffer ()

Resets the CAN chip which includes clearing receive and transmit 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`) – 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.

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

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*, ***config*)
Bases: `can.bus.BusABC`

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

Parameters

- **channel** (*int*) – The Channel id to create this bus with.
- **can_filters** (*list*) – See `can.BusABC.set_filters()` for details.
- **use_system_timestamp** – 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)

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*) – 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.

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

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'*, *fd=False*, *data_bitrate=None*, *sjwAbr=2*, *tseg1Abr=6*, *tseg2Abr=3*, *sjwDbr=2*, *tseg1Dbr=6*, *tseg2Dbr=3*, ***config*)

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.
- **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*) – 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.

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

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.

```python
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
```

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

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

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

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

CHAPTER 5

Scripts

The following modules are callable from python-can.

5.1 can.logger

Command line help (`python -m can.logger --help`):

```
usage: python -m can.logger [-h] [-f LOG_FILE] [-v] [-c CHANNEL]
                            [-i {iscan,slcan,virtual,socketcan_ctypes,usb2can,ixxat,
↪socketcan_native,kvaser,neovi,vector,nican,pcan,serial,remote,socketcan}]
                            [--filter ...] [-b BITRATE]

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, extension can be .txt,
                        .asc, .csv, .db, .npz
  -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 {iscan,slcan,virtual,socketcan_ctypes,usb2can,ixxat,socketcan_native,kvaser,
↪neovi,vector,nican,pcan,serial,remote,socketcan}, --interface {iscan,slcan,virtual,
↪socketcan_ctypes,usb2can,ixxat,socketcan_native,kvaser,neovi,vector,nican,pcan,
↪serial,remote,socketcan}
                        Specify the backend CAN interface to use. If left
                        blank, fall back to reading from configuration files.
```

(continues on next page)

(continued from previous page)

```
  --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.
```

5.2 can.player

Command line help (`python -m can.player --help`):

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

Replay CAN traffic

positional arguments:
  input-file            The file to replay. Supported types: .db, .blf

optional arguments:
  -h, --help            show this help message and exit
  -f LOG_FILE, --file_name LOG_FILE
                        Path and base log filename, extension can be .txt,
                        .asc, .csv, .db, .npz
  -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 {kvaser,virtual,slcan,nican,neovi,ixxat,serial,usb2can,socketcan_ctypes,remote,
↪socketcan_native,iscan,vector,pcan,socketcan}, --interface {kvaser,virtual,slcan,
↪nican,neovi,ixxat,serial,usb2can,socketcan_ctypes,remote,socketcan_native,iscan,
↪vector,pcan,socketcan}
                        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)
  -g GAP, --gap GAP     <s> minimum time between replayed frames
  -s SKIP, --skip SKIP  <s> skip gaps greater than 's' seconds
```

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.

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 subdirctory depending on the complexity) inside `can.interfaces`
- Implement the central part of the backend: the bus class that extends *`can.BusABC`*. See below for more info on this one!

- Register your backend bus class in `can.interface.BACKENDS` and `can.interfaces.VALID_INTERFACES`.
- Add docs where appropiate, like in `doc/interfaces.rst` and add an entry in `doc/interface/*`.
- Add tests in `test/*` where appropiate.

CHAPTER 7

About the `BusABC` class

Concrete implementations *have to* 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 discrading any messages yet to be sent
- `shutdown()` to override how the bus should shut down
- `send_periodic()` 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:

7.1 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.

7.2 Creating a new Release

- Release from the `master` branch.
- Update the library version in `__init__.py` using semantic versioning.
- 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`
- Create and upload the distribution: `python setup.py sdist bdist_wheel`
- Sign the packages with gpg `gpg --detach-sign -a dist/python_can-X.Y.Z-py3-none-any.whl`
- Upload with twine `twine upload dist/python-can-X.Y.Z*`
- In a new virtual env check that the package can be installed with pip: `pip install python-can==X.Y.Z`
- Create a new tag in the repository.
- Check the release on PyPi, Read the Docs and GitHub.

CHAPTER 8

History and Roadmap

8.1 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.

8.2 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.

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.

8.3 Support for CAN within Python

The 'socket' module contains support for SocketCAN from Python 3.3.

From Python 3.4 broadcast management commands are natively supported.

CHAPTER 9

Known Bugs

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

Documentation generated

Sep 24, 2018

Python Module Index

Index