python-can

Release 2.0.0-alpha.2

February 28, 2017

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:

```
from __future__ import print_function
import can


def send_one():
    bus = can.interface.Bus()
    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.

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.

Windows dependencies

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.

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.

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.

NI-CAN

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

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

neoVI

See neoVI Interface.

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.

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'
from can.interfaces.interface import Bus

bus = Bus()
```

Configuration File

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

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

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

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>
```

Environment Variables

Configuration can be pulled from these environmental variables:

- CAN_INTERFACE
- CAN_CHANNEL

Interface Names

Lookup table of interface names:

Name	Documentation
`"socketcan"`	Socketcan
`"kvaser"`	Kvaser's CANLIB
`"serial"`	CAN over Serial
`"ixxat"`	IXXAT Virtual CAN Interface
`"pcan"`	PCAN Basic API
`"usb2can"`	USB2CAN Interface
`"nican"`	NI-CAN
`"neovi"`	neoVI Interface
`"remote"`	Remote
`"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.

Bus

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

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.

API

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

Bases: `object`

CAN Bus Abstract Base Class

Concrete implementations must implement the following methods:

- send
- recv

As well as setting the *channel_info* attribute to a string describing the interface.

Parameters

- **`channel`** – The can interface identifier. Expected type is backend dependent.
- **`can_filters`** (*list*) – A list of dictionaries each containing a "can_id" and a "can_mask".

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

A filter matches, when `<received_can_id> & can_mask == can_id & can_mask`

- **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 channel

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.

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

send(*msg*, *timeout=None*)
Transmit a message to CAN bus. Override this method to enable the transmit path.

Parameters

- **msg** – A *can.Message* object.
- **timeout** (*float*) – If > 0, wait up to this many seconds for message to be ACK:ed. If timeout is exceeded, an exception will be raised. Might not be supported by all interfaces.

Raise `can.CanError` if the message could not be written.

set_filters(*can_filters=None*)
Apply filtering to all messages received by this Bus.

Calling without passing any filters will reset the applied filters.

Parameters **can_filters** (*list*) – A list of dictionaries each containing a "can_id" and a "can_mask".

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

A filter matches, when `<received_can_id> & can_mask == can_id & can_mask`

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

class `can.interface.`**Bus**
Bases: `object`

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

Transmitting

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

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.

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*)

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, and contain data.

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    000    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))
0.000000    00000000    0002    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 = can.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.

```
>>> 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    000    DLC: 0
>>> print(Message(extended_id=True))
Timestamp:        0.000000    ID: 00000000    010    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.

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.

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    010    DLC: 0
>>> test2 = Message(data=[1, 2, 3, 4, 5])
>>> print(test2)
Timestamp:        0.000000    ID: 00000000    010    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 a four-digit hexadecimal number. The arbitration ID field 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.

Listeners

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.

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.

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*
- other: *can.Printer*

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

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.

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

CSVWriter

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

Writes a comma separated text file of timestamp, arbitration id, flags, dlc, data for each messages received.

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

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.

class can.**ASCWriter**(*filename*, *channel=1*)
Bases: can.listener.Listener

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

log_event(*message*, *timestamp=None*)
Add an arbitrary message to the log file.

stop()
Stops logging and closes the file.

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.

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 = 7
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.

Broadcast Manager

The broadcast manager isn't yet supported by all interfaces. It allows the user to setup periodic message jobs.

This example shows the ctypes socketcan using the broadcast manager:

```
#!/usr/bin/env python3
"""
This example exercises the periodic sending capabilities.

Expects a vcan0 interface:

    python3 -m examples.cyclic

"""

import logging
import time

import can
logging.basicConfig(level=logging.INFO)

channel = 'vcan0'


def test_simple_periodic_send():
    print("Starting to send a message every 200ms. Initial data is zeros")
    msg = can.Message(arbitration_id=0x123, data=[0, 0, 0, 0, 0, 0], extended_id=False)
    task = can.send_periodic('vcan0', msg, 0.20)
    time.sleep(2)
    task.stop()
    print("stopped cyclic send")


def test_extended_periodic_send():
    print("Starting to send a message every 200ms. Initial data is zeros")
    msg = can.Message(arbitration_id=0x12345678, data=[0, 0, 0, 0, 0, 0], extended_id=True)
    task = can.send_periodic('vcan0', msg, 0.20)
    time.sleep(2)
    task.stop()
    print("stopped cyclic send")


def test_periodic_send_with_modifying_data():
    print("Starting to send a message every 200ms. Initial data is ones")
    msg = can.Message(arbitration_id=0x0cf02200, data=[1, 1, 1, 1])
    task = can.send_periodic('vcan0', msg, 0.20)
    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])
    task.modify_data(msg)
    time.sleep(1)
    print("starting again")
    task.start()
    time.sleep(1)
    task.stop()
    print("done")


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__":

    for interface in {'socketcan_ctypes', 'socketcan_native'}:
        print("Carrying out cyclic tests with {} interface".format(interface))
        can.rc['interface'] = interface

        test_simple_periodic_send()

        test_extended_periodic_send()

        test_periodic_send_with_modifying_data()

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

Functional API

can.**send_periodic**(*channel*, *message*, *period*)
Send a message every *period* seconds on the given channel.

Class based API

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

Parameters

- **channel** (*str*) – The name of the CAN channel to connect to.
- **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 Message.data. Note it must have the same *arbitration_id*.

stop()
Send a TX_DELETE message to the broadcast manager to cancel this task.

This will delete the entry for the transmission of the CAN message specified.

class can.**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*.

Utilities

Utilities and configuration file parsing.

can.util.**choose_socketcan_implementation**()
Set the best version of SocketCAN for this system.

Parameters **config** – The can.rc configuration dictionary

Raises **Exception** – If the system doesn't support SocketCAN

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
- 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, remote, virtual.

Note: If you pass "socketcan" this automatically selects between the native and ctypes version.

Parameters

- **path** – Optional path to config file.
- **config** – A dict which may set the 'interface', and/or the 'channel', or neither.

Returns

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

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

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

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

•CAN_INTERFACE

•CAN_CHANNEL

`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'

Notifier

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

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

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

Parameters

- **bus** – The *Bus* to listen too.
- **listeners** – An iterable of `Listeners`
- **timeout** – An optional maximum number of seconds to wait for any message.

exception = None
Exception raised in thread

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

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:

Socketcan

There are two implementations of socketcan backends. One written with `ctypes` to be compatible with Python 2 and 3, and one written for future versions of Python3 which feature native support.

SocketCAN (ctypes)

socketcan_ctypes.py is a ctypes wrapper class around libc. It contains replications of constants and structures found in various linux header files. With Python 3.3, much of the functionality of this library is likely to be available natively in the Python socket module.

Bus

class `can.interfaces.socketcan.`**`SocketcanCtypes_Bus`**(*channel=0*, *receive_own_messages=False*, **args*, ***kwargs*)

Bases: `can.bus.BusABC`

An implementation of the `can.bus.BusABC` for SocketCAN using `ctypes`.

Parameters **`channel`** (*`str`*) – The can interface name with which to create this bus. An example channel would be 'vcan0'.

`set_filters`(*can_filters=None*)

Apply filtering to all messages received by this Bus.

Calling without passing any filters will reset the applied filters.

Parameters **`can_filters`** (*`list`*) – A list of dictionaries each containing a "can_id" and a "can_mask".

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

A filter matches, when `<received_can_id> & can_mask == can_id & can_mask`

Broadcast-Manager

The `socketcan_ctypes` 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.socketcan_ctypes.`**`CyclicSendTask`**(*channel*, *message*, *period*)

Bases: `can.interfaces.socketcan.socketcan_ctypes.SocketCanCtypesBCMBase`, `can.broadcastmanager.CyclicSendTaskABC`

Parameters

- **`channel`** – The name of the CAN channel to connect to.
- **`message`** – The message to be sent periodically.
- **`period`** – The rate in seconds at which to send the message.

`modify_data`(*message*)

Update the contents of this periodically sent message.

`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).

Internals

createSocket

`can.interfaces.socketcan.socketcan_ctypes.`**`createSocket`**(*protocol=1*)

This function creates a RAW CAN socket.

The socket returned needs to be bound to an interface by calling `bindSocket()`.

Parameters **`protocol`** (*int*) – The type of the socket to be bound. Valid values include CAN_RAW and CAN_BCM

Returns

0	protocol invalid
-1	socket creation unsuccessful
socketID	successful creation

bindSocket

can.interfaces.socketcan.socketcan_ctypes.**bindSocket**(*socketID*, *channel_name*)
Binds the given socket to the given interface.

Parameters

- **socketID** (*int*) – The ID of the socket to be bound
- **channel_name** (*str*) – The interface name to find and bind.

Returns

The error code from the bind call.

0	protocol invalid
-1	socket creation unsuccessful

connectSocket

can.interfaces.socketcan.socketcan_ctypes.**connectSocket**(*socketID*, *channel_name*)
Connects the given socket to the given interface.

Parameters

- **socketID** (*int*) – The ID of the socket to be bound
- **channel_name** (*str*) – The interface name to find and bind.

Returns The error code from the bind call.

capturePacket

can.interfaces.socketcan.socketcan_ctypes.**capturePacket**(*socketID*)
Captures a packet of data from the given socket.

Parameters **socketID** (*int*) – The socket to read from

Returns

A dictionary with the following keys:

- *"CAN ID"* (int)
- *"DLC"* (int)
- *"Data"* (list)
- *"Timestamp"* (float)

SocketCAN (python)

Python 3.3 added support for socketcan for linux systems.

The socketcan_native interface directly uses Python's socket module to access SocketCAN on linux. This is the most direct route to the kernel and should provide the most responsive.

The implementation features efficient filtering of can_id's, this filtering occurs in the kernel and is much much more efficient than filtering messages in Python.

Python 3.4 added support for the Broadcast Connection Manager (BCM) protocol, which if enabled should be used for queueing periodic tasks.

Documentation for the socket can backend file can be found:

https://www.kernel.org/doc/Documentation/networking/can.txt

Bus

class can.interfaces.socketcan.**SocketcanNative_Bus**(*channel*, ***kwargs*)
Bases: can.bus.BusABC

Parameters

- **channel** (*str*) – The can interface name with which to create this bus. An example channel would be 'vcan0'.
- **can_filters** (*list*) – A list of dictionaries, each containing a "can_id" and a "can_mask".

Internals

createSocket

can.interfaces.socketcan.socketcan_native.**createSocket**(*can_protocol=None*)
Creates a CAN socket. The socket can be BCM or RAW. The socket will be returned unbound to any interface.

Parameters **can_protocol** (*int*) –

The protocol to use for the CAN socket, either:

- socket.CAN_RAW
- socket.CAN_BCM.

Returns

- -1 if socket creation unsuccessful
- socketID - successful creation

bindSocket

can.interfaces.socketcan.socketcan_native.**bindSocket**(*sock*, *channel='can0'*)
Binds the given socket to the given interface.

Parameters **socketID** (*Socket*) – The ID of the socket to be bound

Raise OSError if the specified interface isn't found.

capturePacket

can.interfaces.socketcan.socketcan_native.**capturePacket**(*sock*)
Captures a packet of data from the given socket.

Parameters **sock** (socket) – The socket to read a packet from.

Returns A namedtuple with the following fields: * timestamp * arbitration_id * is_extended_frame_format * is_remote_transmission_request * is_error_frame * dlc * data

Unless you're running Python3.3 or lower the recommended backend is socketcan_native.

Socketcan Quickstart

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

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:

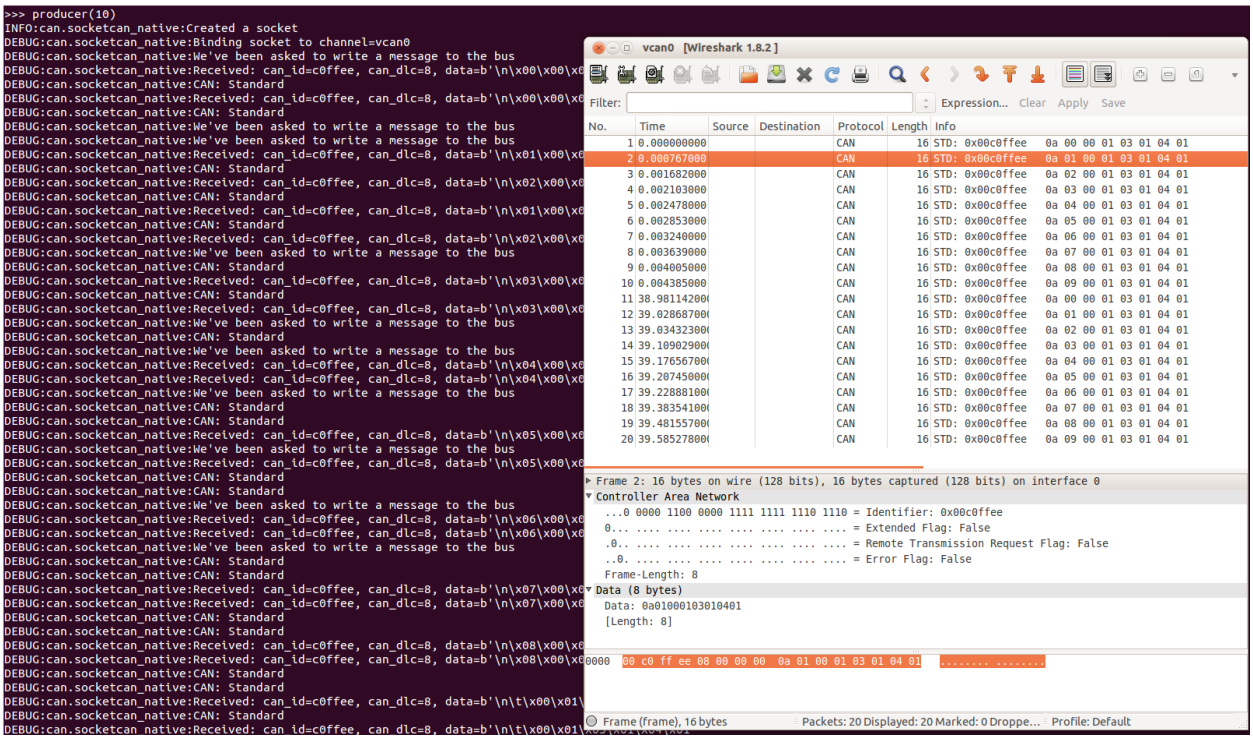
```
import time
import can

bustype = 'socketcan_native'
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:



The process to follow bus traffic is even easier:

```
for message in Bus(can_interface):
    print(message)
```

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_native')
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.

Kvaser's CANLIB

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

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*) – A list of dictionaries each containing a "can_id" and a "can_mask".

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

Backend Configuration

Parameters

- **bitrate** (*int*) – Bitrate of channel in bit/s
- **tseg1** (*int*) – Time segment 1, that is, the number of quanta from (but not including) the Sync Segment to the sampling point. 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 Synchronisation Jump Width. Decides the maximum number of time quanta that the controller can resynchronise 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.

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.

recv(*timeout=None*)
Read a message from kvaser device.

set_filters(*can_filters=None*)
Apply filtering to all messages received by this Bus.

Calling without passing any filters will reset the applied filters.

Since Kvaser only supports setting one filter per handle, the filtering will be done in the recv() if more than one filter is requested.

Parameters **can_filters** (*list*) – A list of dictionaries each containing a "can_id" and a "can_mask".

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

A filter matches, when `<received_can_id> & can_mask == can_id & can_mask`

timer_offset = None
Approximate offset between time.time() and CAN timestamps (~2ms accuracy) There will always be some lag between when the message is on the bus to when it reaches Python. Allow messages to be on the bus for a while before reading this value so it has a chance to correct itself

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.

CAN over Serial

A text based interface. For example use over bluetooth with `/dev/rfcomm0`

Bus

class `can.interfaces.serial.serial_can.`**SerialBus**(*channel*, **args*, ***kwargs*)
Bases: `can.bus.BusABC`

A serial interface to CAN.

Parameters **channel** (*str*) – The serial device to open.

Internals

Todo

Implement and document serial interface.

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.

Bus

class can.interfaces.ixxat.**IXXATBus**(*channel*, *can_filters=None*, ***config*)
Bases: can.bus.BusABC

The CAN Bus implemented for the IXXAT interface.

Parameters

- **channel** (*int*) – The Channel id to create this bus with.
- **can_filters** (*list*) – A list of dictionaries each containing a "can_id" and a "can_mask".

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

- **UniqueHardwareId** (*int*) – UniqueHardwareId to connect (optional, will use the first found if not supplied)
- **bitrate** (*int*) – Channel bitrate in bit/s

flush_tx_buffer()
Flushes the transmit buffer on the IXXAT

recv(*timeout=None*)
Read a message from IXXAT device.

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

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.

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.

Configuration

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

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

Bus

class `can.interfaces.pcan.`**`PcanBus`**(*channel*, **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
- **`bitrate`** (*`int`*) – Bitrate of channel in bit/s. Default is 500 Kbs

`flash`(*flash*)

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

`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

USB2CAN Interface

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.

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.

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. **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)

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.

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!

Bus

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

Bases: `can.bus.BusABC`

Interface to a USB2CAN Bus.

Note the USB2CAN interface doesn't implement set_filters, or flush_tx_buffer methods.

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.

`shutdown`()

Shut down the device safely

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*)

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.

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.

Parameters

- **channel** (*str*) – Name of the object to open (e.g. 'CAN0')
- **bitrate** (*int*) – Bitrate in bits/s
- **can_filters** (*list*) – A list of dictionaries each containing a "can_id" and a "can_mask".

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

- **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.

recv(*timeout=None*)
Read a message from NI-CAN.

Parameters **timeout** (*float*) – Max time to wait in seconds or None if infinite

Returns The CAN message or None if timeout

Return type *can.Message*

Raises *can.interfaces.nican.NicanError* – If reception fails

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.

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

neoVI Interface

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

Interface to Intrepid Control Systems neoVI API range of devices via pyneovi wrapper on Windows.

Note: This interface is not supported on Linux, however on Linux neoVI devices are supported via Socketcan with ICS Kernel-mode SocketCAN module for Intrepid devices and icsscand

Installation

This neoVI interface requires the installation of the ICS neoVI DLL and pyneovi package.

- **Download and install the Intrepid Product Drivers** Intrepid Product Drivers
- **Install pyneovi using pip and the pyneovi bitbucket repo:**

```
pip install https://bitbucket.org/Kemp_J/pyneovi/get/default.zip
```

Configuration

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

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

Bus

class can.interfaces.neovi_api.**NeoVIBus**(*channel=None*, *can_filters=None*, ***config*)
Bases: can.bus.BusABC

The CAN Bus implemented for the pyneovi interface.

Parameters

- **channel** (*int*) – The Channel id to create this bus with.
- **can_filters** (*list*) – A list of dictionaries each containing a "can_id" and a "can_mask".

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

set_filters(*can_filters=None*)
Apply filtering to all messages received by this Bus.

Calling without passing any filters will reset the applied filters.

Parameters **can_filters** (*list*) – A list of dictionaries each containing a "can_id" and a "can_mask".

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

A filter matches, when `<received_can_id> & can_mask == can_id & can_mask`

Remote

The remote interface works as a networked bridge between the computer running the application and the computer owning the physical CAN interface.

Multiple clients may connect to the same server simultaneously. Each client will create its own bus instance on the server, so this must be supported by the real interface.

Server

The computer which owns the CAN interface must start a server which accepts incoming connections. If more than one channel is to be shared, multiple servers must be started on different ports.

Start a server using default interface and channel:

```
$ canserver
```

Specify interface, channel and port number explicitly:

```
$ canserver --interface kvaser --channel 0 --port 54702
```

It can also be started as a module:

```
$ python -m can.interfaces.remote
```

Client

The application must specify `remote` as interface and `host:port` as channel. The port number can be omitted if default port is used. The bitrate to use on the CAN bus can also be specified.

```
bus = can.interface.Bus('192.168.0.10:54701',
                        bustype='remote',
                        bitrate=500000,
                        can_filters=[
                            {'can_id': 0x11},
                            {'can_mask': 0xff}
                        ])
```

Alternatively in a .canrc file:

```
[default]
interface = remote
channel = myhostname:54701
```

The can_logger.py script could be started like this:

```
$ can_logger.py -i remote -c myhostname:54701
```

Internals

The client uses a standard Bus class to connect to the server.

class `can.interfaces.remote.`**`RemoteBus`**(*channel*, *can_filters=None*, ***config*)

Bases: `can.bus.BusABC`

CAN bus over a network connection bridge.

Parameters

- **`channel`** (*`str`*) – Address of server as host:port (port may be omitted).
- **`can_filters`** (*`list`*) – A list of dictionaries each containing a "can_id" and a "can_mask".

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

The filters are handed to the actual CAN interface on the server.

- **`bitrate`** (*`int`*) – Bitrate in bits/s to use on CAN bus. May be ignored by the interface.

Any other backend specific configuration will be silently ignored.

`recv`(*timeout=None*)

Block waiting for a message from the Bus.

Parameters **`timeout`** (*`float`*) – Seconds to wait for a message.

Returns None on timeout or a Message object.

Return type *can.Message*

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

Transmit a message to CAN bus.

Parameters **`msg`** (`can.Message`) – A Message object.

Raises *can.interfaces.remote.CanRemoteError* – On failed transmission to socket.

shutdown()
Close socket connection.

socket = None
Socket connection to the server

exception can.interfaces.remote.**CanRemoteError**
Bases: can.CanError

An error occurred on socket connection or on the remote end.

The server uses the following classes to implement the connections.

class can.interfaces.remote.**RemoteServer**(*host='0.0.0.0'*, *port=None*, ***config*)
Bases: SocketServer.ThreadingMixIn, SocketServer.TCPServer

Server for CAN communication.

Parameters

- **host** (*str*) – Address to listen to.
- **port** (*int*) – Network port to listen to.
- **channel** – The can interface identifier. Expected type is backend dependent.
- **bustype** (*str*) – CAN interface to use.
- **bitrate** (*int*) – Forced bitrate in bits/s.

serve_forever(*poll_interval=0.5*)
Start listening for incoming connections.

shutdown()
Stops the serve_forever loop.

Blocks until the loop has finished. This must be called while serve_forever() is running in another thread, or it will deadlock.

server_close()
Clean-up the server.

clients = None
List of *can.interfaces.remote.server.ClientBusConnection* instances

class can.interfaces.remote.server.**ClientBusConnection**(*request*, *client_address*, *server*)
Bases: SocketServer.BaseRequestHandler

A client connection on the server.

send_msg(*msg*)
Send a CAN message to the bus.

Protocol

The protocol is a stream of events over a TCP socket. Each event starts with one byte that represents the event id, followed by event specific data of arbitrary length in big-endian byte order.

The client start with sending a *BusRequest* followed by a *FilterConfig*. The server will reply with a *BusResponse*.

Each event class inherits from the base event class:

class can.interfaces.remote.events.**BaseEvent**
Bases: object

Events should inherit this class.

encode()
Convert event data to bytes.

Returns Bytestring representing the event data.

Return type bytes

classmethod from_buffer(*buf*)
Parse the data and return a new event.

Parameters **buf** (*bytes*) – Bytestring representing the event data.

Returns Event decoded from buffer.

Raises **can.interfaces.remote.events.NeedMoreDataError** – If not enough data exists.

The available events that can occurr and their specification is listed below:

class can.interfaces.remote.events.**BusRequest**(*version*, *bitrate*)
Bases: *can.interfaces.remote.events.BaseEvent*

Request for connecting to CAN bus.

Byte	Type	Contents
0	U8	Protocol version used by client
1 - 4	S32	Bitrate in bits/s requested

Parameters

- **version** (*int*) – Network protocol version
- **bitrate** (*int*) – Bitrate to use on CAN

EVENT_ID = 1
Event ID

bitrate = None
Bitrate in bits/s

version = None
Network protocol version

class can.interfaces.remote.events.**BusResponse**(*channel_info*)
Bases: *can.interfaces.remote.events.BaseEvent*

Response after connected to CAN bus.

Byte	Type	Contents
0	U8	Length of channel info string
1 - x	STR	Channel info (UTF-8)

Parameters **channel_info** (*str*) – Text describing the channel

EVENT_ID = 2
Event ID

channel_info = None
Text describing the channel

class can.interfaces.remote.events.**CanMessage**(*msg*)
Bases: *can.interfaces.remote.events.BaseEvent*

CAN message being received or transmitted.

Byte	Type	Contents
0 - 7	F64	Timestamp
8 - 11	U32	Arbitration ID
12	U8	DLC
13	U8	**Flags:** • Bit 0: Extended ID • Bit 1: Remote frame • Bit 2: Error frame
14 - 21	U8	Data padded to an 8 byte array

Parameters **msg** (can.Message) – A Message object.

EVENT_ID = 3
Event ID

msg = None
A *can.Message* instance.

class can.interfaces.remote.events.**TransmitSuccess**
Bases: *can.interfaces.remote.events.BaseEvent*

A message has been successfully transmitted to CAN.

EVENT_ID = 4
Event ID

class can.interfaces.remote.events.**RemoteException**(*exc*)
Bases: *can.interfaces.remote.events.BaseEvent*

An exception has occurred on the server.

Byte	Type	Contents
0	U8	Length of exception string
1 - x	STR	Exception description (UTF-8)

Parameters **exc** (*Exception*) – The exception to send.

EVENT_ID = 6
Event ID

exc = None
The exception

class can.interfaces.remote.events.**FilterConfig**(*can_filters=None*)
Bases: *can.interfaces.remote.events.BaseEvent*

CAN filter configuration.

Byte	Type	Contents
0	U8	Number of filters
1 - 4	U32	CAN ID for filter 1
5 - 8	U32	CAN mask for filter 1
9 - 12	U32	CAN ID for filter 2
13 - 16	U32	CAN mask for filter 2
...	...	...

Parameters **can_filters** (*list*) – List of CAN filters

EVENT_ID = 10
Event ID

can_filters = None
A list of CAN filter dictionaries as: >> {'can_id': 0x03, 'can_mask': 0xff}

class can.interfaces.remote.events.**ConnectionClosed**
Bases: *can.interfaces.remote.events.BaseEvent*

Connection closed by peer.

Will be automatically emitted if the socket is closed.

EVENT_ID = 255
Event ID

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
```

The *Interface Names* are listed in Configuration.

CHAPTER 5

Scripts

The following scripts are installed along with python-can.

canlogger

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

```
usage: canlogger [-h] [-f LOG_FILE] [-v] [-c CHANNEL]
                 [-i {pcan,remote,ixxat,socketcan_ctypes,virtual,usb2can,nican,serial,kvaser,socketcan
                 [--filter ...]

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 {pcan,remote,ixxat,socketcan_ctypes,virtual,usb2can,nican,serial,kvaser,socketcan,socketcan_nati
                        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)
```

canplayer

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

```
usage: canplayer [-h] [-f LOG_FILE] [-v] [-c CHANNEL]
                 [-i {pcan,remote,ixxat,socketcan_ctypes,virtual,usb2can,nican,serial,kvaser,socketcan
                 [--ignore-timestamps] [-g GAP] [-s SKIP]
                 input-file

Replay CAN traffic

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

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 {pcan,remote,ixxat,socketcan_ctypes,virtual,usb2can,nican,serial,kvaser,socketcan,socketcan_nati
                        Specify the backend CAN interface to use. If left
                        blank, fall back to reading from configuration files.
  --ignore-timestamps   Ignore timestamps (send all frames immediately with
                        minimum gap between frames)
  -g GAP, --gap GAP     <ms> minimum time between replayed frames
  -s SKIP, --skip SKIP  <s> skip gaps greater than 's' seconds
```

canserver

Command line help (`canserver --help` or `python -m can.interfaces.remote --help`):

```
usage: canserver [-h] [-v] [-c CHANNEL]
                 [-i {pcan,remote,ixxat,socketcan_ctypes,virtual,usb2can,nican,serial,kvaser,socketcan
                 [-b BITRATE] [-H HOST] [-p PORT]

Remote CAN server

optional arguments:
  -h, --help            show this help message and exit
  -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". The server will only serve this
                        channel. Start additional servers at different ports
                        to share more channels.
  -i {pcan,remote,ixxat,socketcan_ctypes,virtual,usb2can,nican,serial,kvaser,socketcan,socketcan_nati
                        Specify the backend CAN interface to use. If left
                        blank, fall back to reading from configuration files.
```

```
  -b BITRATE, --bitrate BITRATE
                        Force to use a specific bitrate. This will override
                        any requested bitrate by the clients.
  -H HOST, --host HOST  Host to listen to (default 0.0.0.0).
  -p PORT, --port PORT  TCP port to listen on (default 54701).
```

CHAPTER 6

Developer's Overview

Contributing

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

Creating a Release

- Release from the `master` branch.
- Update the library version in `setup.py` and in `doc/conf.py` using semantic versioning.
- Run all tests and examples against available hardware.
- Update *CONTRIBUTORS.txt* with any new contributors.
- Sanity check that documentation has stayed inline with code. For large changes update `doc/history.rst`
- 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 upload --sign`
- In a new virtual env check that the package can be installed with pip: `pip install python-can`
- Create a new tag in the repository.
- Check the release on PyPi and github.

Code Structure

The modules in `python-can` are:

Module	Description
interfaces	Contains interface dependent code.
bus	Contains the interface independent Bus object.
CAN	Contains modules to emulate a CAN system, such as a time stamps, read/write streams and listeners.
message	Contains the interface independent Message object.
notifier	An object which can be used to notify listeners.
broadcastmanager	Contains interface independent broadcast manager code.

CHAPTER 7

History and Roadmap

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.

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, remote 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.

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 8

Known Bugs

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

Documentation generated

February 28, 2017

C

Symbols

A

B

C

D

E

F

G

I

K

L

M

N

O

P

R

S

T

U

V