canlib

Release 1.17.748

Kvaser AB <support@kvaser.com>

Jun 02, 2021

CONTENTS

The canlib module is a Python wrapper for Kvaser CANlib SDK.

"The CANlib Software Development Kit is your Application Programming Interface for working with all Kvaser hardware platforms."

Using the Python canlib package, you will be able to control most aspects of any Kvaser CAN interface via Python.

CHAPTER

ONE

CONTENTS

1.1 Welcome

The canlib package is a Python wrapper for Kvaser CANlib SDK.

"The CANlib Software Development Kit is your Application Programming Interface for working with all Kvaser hardware platforms."

Using the Python canlib package, you will be able to control most aspects of any Kvaser CAN interface via Python.

canlib - a Python wrapper for Kvaser CANlib

1.2 Supported Libraries and Installation

The Python canlib module wraps the Kvaser CANlib API in order to make it easy to control most aspects of any Kvaser CAN interface. For more information about Kvaser please go to https://www.kvaser.com/.

The latest version of this package is available on the Kvaser Download page (pycanlib.zip).

1.2.1 Supported platforms

Windows and Linux using Python v3.6+ (both 32 and 64 bit).

1.2.2 Supported libraries

The following libraries are currently supported:

Library	Module	Windows	Linux
CANlib	canlib.canlib	canlib32.dll	libcanlib.so
kvaMemoLibXML	canlib.kvamemolibxml	kvaMemoLibXML.dll	libkvamemolibxml.so
kvrlib	canlib.kvrlib	kvrlib.dll, irisdll.dll, irisflash.dll, libxml2.dll	not supported
kvmlib	canlib.kvmlib	kvaMemoLib0600.dll, kvaMemoLib0700.dll, kvaMemoLib.dll, kvmlib.dll	not supported,[2] libkvamemolib0700.so, libkvamemolib.so, libkvmlib.so
kvlclib	canlib.kvlclib	kvlclib.dll[1]	libkvlclib.so
kvaDbLib	canlib.kvadblib	kvaDbLib.dll	libkvadblib.so
LINlib	canlib.linlib	linlib.dll	liblinlib.so

1.2.3 What's new

For a complete set of release notes, see Release Notes.

1.2.4 Installation

Install the Python package using e.g. `pip`:

Unzip `pycanlib.zip`. Navigate to the unzipped pycanlib in the command line. It should contain the file `canlib-x.y.z-py2.py3-none-any.whl`, where x,y,z are version numbers. Run the following command:

```
$ pip install canlib-x.y.z-py2.py3-none-any.whl
```

The Kvaser CANlib DLLs or shared libraries also need to be installed:

Windows

On **Windows**, first install the `canlib32.dll` by downloading and installing "Kvaser Drivers for Windows" which can be found on the Kvaser Download page (kvaser_drivers_setup.exe) This will also install `kvrlib.dll`, `irisdll.dll`, `irisflash.dll` and `libxml2.dll` used by kvrlib.

The "Kvaser CANlib SDK" also needs to be downloaded from the same place (canlib.exe) and installed if more than just CANlib will be used. This will install the rest of the supported library dll's.

The two packages, "Kvaser Drivers for Windows" and "Kvaser CANlib SDK", contains both 32 and 64 bit versions of the included dll's.

[2] The kvaMemoLib0600.dll, which supports older devices, is not supported under Linux.
[1] The kvlclib.dll depends on dll files from matlab wich are installed alongside kvlclib.dll.

Linux

On **Linux**, first install the `libcanlib.so` by downloading and installing "Kvaser LINUX Driver and SDK" which can be found on the Kvaser Download page (linuxcan.tar.gz).

If more than just CANlib will be used, the rest of the supported libraries will be available by downloading and installing "Linux SDK library" (kvlibsdk.tar.gz).

1.2.5 Usage

Example of using canlib to list connected Kvaser devices:

```
from canlib import canlib

num_channels = canlib.getNumberOfChannels()
print("Found %d channels" % num_channels)
for ch in range(0, num_channels):
    chdata = canlib.ChannelData(ch)
    print("%d. %s (%s / %s)" % (ch, chdata.channel_name,
                                chdata.card_upc_no,
                                chdata.card_serial_no))
```

Which may result in:

```
Found 4 channels
0. Kvaser Memorator Professional HS/HS (channel 0) (73-30130-00351-4 / 12377)
1. Kvaser Memorator Professional HS/HS (channel 1) (73-30130-00351-4 / 12377)
2. Kvaser Virtual CAN Driver (channel 0) (00-00000-00000-0 / 0)
3. Kvaser Virtual CAN Driver (channel 1) (00-00000-00000-0 / 0)
```

1.2.6 Support

You are invited to visit the Kvaser web pages at https://www.kvaser.com/support/. If you don't find what you are looking for, you can obtain support on a time-available basis by sending an e-mail to support@kvaser.com.

Bug reports, contributions, suggestions for improvements, and similar things are much appreciated and can be sent by e-mail to support@kvaser.com.

1.2.7 References

- Kvaser CANlib SDK Page: https://www.kvaser.com/developer/canlib-sdk/
- Description of CANlib SDK library contents: https://www.kvaser.com/developer-blog/get-hardware-kvaser-sdk-libraries/

1.2.8 Wrapped libraries

- *canlib*
- *kvadblib*
- *kvamemolibxml*
- *kvlclib*
- *kvmlib*
- *kvrlib*
- *linlib*

1.3 Tutorials

1.3.1 canlib

Contents

The following sections contain sample code for inspiration on how to use Kvaser Python canlib.

List connected devices

This code print some basic information (device name, EAN number and serial number) from all connected devices.

```
from canlib import canlib

num_channels = canlib.getNumberOfChannels()
print("Found %d channels" % num_channels)
for ch in range(0, num_channels):
    chdata = canlib.ChannelData(ch)
    print("%d. %s (%s / %s)" % (ch, chdata.device_name,
                                chdata.card_upc_no,
                                chdata.card_serial_no))
```

Send and receive single frame

Here is some basic code to send and receive a single frame.

```
from canlib import canlib, Frame
from canlib.canlib import ChannelData

def setUpChannel(channel=0,
                 openFlags=canlib.Open.ACCEPT_VIRTUAL,
                 outputControl=canlib.Driver.NORMAL):
    ch = canlib.openChannel(channel, openFlags)
    print("Using channel: %s, EAN: %s" % (ChannelData(channel).channel_name,
                                          ChannelData(channel).card_upc_no))
    ch.setBusOutputControl(outputControl)
    # Specifying a bus speed of 500 kbit/s. See documentation
    # for more informationon how to set bus parameters.
    params = canlib.busparams.BusParamsTq(
        tq=8,
        phase1=2,
        phase2=2,
        sjw=1,
        prescaler=40,
        prop=3
    )
    ch.set_bus_params_tq(params)
    ch.busOn()
    return ch


def tearDownChannel(ch):
    ch.busOff()
    ch.close()


print("canlib version:", canlib.dllversion())

ch0 = setUpChannel(channel=0)
ch1 = setUpChannel(channel=1)

frame = Frame(
    id_=100,
    data=[1, 2, 3, 4],
    flags=canlib.MessageFlag.EXT
)
ch1.write(frame)

while True:
    try:
        frame = ch0.read()
        print(frame)
        break
    except (canlib.canNoMsg) as ex:
        pass
```

(continues on next page)

(continued from previous page)

```
    except (canlib.canError) as ex:
        print(ex)

tearDownChannel(ch0)
tearDownChannel(ch1)
```

Send and receive CAN FD frame

Here are some minimal code to send and receive a CAN FD frame.

```
from canlib import canlib, Frame

# Specifying an arbitration phase bus speed of 1 Mbit/s,
# and a data phase bus speed of 2 Mbit/s. See documentation
# for more information on how to set bus parameters.
params_arbitration = canlib.busparams.BusParamsTq(
    tq=40,
    phase1=8,
    phase2=8,
    sjw=8,
    prescaler=2,
    prop=23
)
params_data = canlib.busparams.BusParamsTq(
    tq=20,
    phase1=15,
    phase2=4,
    sjw=4,
    prescaler=2,
    prop=0
)

# open channel as CAN FD using the flag
ch0 = canlib.openChannel(channel=0, flags=canlib.Open.CAN_FD)
ch0.setBusOutputControl(drivertype=canlib.Driver.NORMAL)
ch0.set_bus_params_tq(params_arbitration, params_data)
ch0.busOn()

ch1 = canlib.openChannel(channel=1, flags=canlib.Open.CAN_FD)
ch1.setBusOutputControl(drivertype=canlib.Driver.NORMAL)
ch1.set_bus_params_tq(params_arbitration, params_data)
ch1.busOn()

# set FDF flag to send using CAN FD
# set BRS flag to send using higher bit rate in the data phase
frame = Frame(
    id_=100,
    data=range(32),
    flags=canlib.MessageFlag.FDF | canlib.MessageFlag.BRS
)
print('Sending', frame)
```

(continues on next page)

(continued from previous page)

```
ch0.write(frame)

frame = ch1.read(timeout=1000)
print('Receiving', frame)

ch0.busOff()
ch1.busOff()
```

1.3.2 kvrlib

Contents

The following sections contain sample code for inspiration on how to use Kvaser Python kvrlib.

Connect to your remote device

Use the discovery functions to scan and connect to a remote device. Our remote device has serial number 16 and is already connected to the same network as our computer:

```
from canlib import kvrlib

SERIAL_NO = 10545

print("kvrlib version: %s" % kvrlib.getVersion())
print("Connecting to device with serial number %s" % SERIAL_NO)

addressList = kvrlib.kvrDiscovery.getDefaultAddresses(kvrlib.kvrAddressTypeFlag_
↪BROADCAST)
print("Looking for device using addresses: %s" % addressList)
discovery = kvrlib.kvrDiscovery()
discovery.setAddresses(addressList)
deviceInfos = discovery.getResults()
print("Scanning result:\n%s" % deviceInfos)
# Connect to device with correct SERIAL_NO
for deviceInfo in deviceInfos:
    if deviceInfo.ser_no == SERIAL_NO:
        deviceInfo.connect()
        print('\nConnecting to the following device:')
        print('-----------------------------------------------')
        print(deviceInfo)
        discovery.storeDevices(deviceInfos)
        break;
discovery.close()
```

This results in the following:

```
kvrlib version: 2070
Connecting to device with serial number 10545
Looking for device using addresses: 10.0.255.255:0 (IPV4_PORT)
Scanning result:
[
name/hostname  : "MiMi-06348-000710" / "kv-06348-000710"
  ean/serial   : 73301-30006348 / 710
  fw           : 2.4.231
  addr/cli/AP  : 10.0.3.138 (IPV4) / 10.0.3.84 (IPV4) / - (UNKNOWN)
  availability : Availability.STORED|FOUND_BY_SCAN
  usage/access : DeviceUsage.FREE / Accessibility.PUBLIC
  pass/enc.key : yes / yes,
name/hostname  : "TestClient1-2-DUT-01" / "swtdut01"
  ean/serial   : 73301-30006713 / 10545
  fw           : 3.4.822
  addr/cli/AP  : 10.0.3.54 (IPV4) / 10.0.3.98 (IPV4) / - (UNKNOWN)
  availability : Availability.STORED|FOUND_BY_SCAN
  usage/access : DeviceUsage.REMOTE / Accessibility.PUBLIC
  pass/enc.key : yes / yes]

Connecting to the following device:
-------------------------------------------

name/hostname  : "TestClient1-2-DUT-01" / "swtdut01"
  ean/serial   : 73301-30006713 / 10545
  fw           : 3.4.822
  addr/cli/AP  : 10.0.3.54 (IPV4) / 10.0.3.98 (IPV4) / - (UNKNOWN)
  availability : Availability.STORED|FOUND_BY_SCAN
  usage/access : DeviceUsage.REMOTE / Accessibility.PUBLIC
  pass/enc.key : yes / yes
```

1.3.3 linlib

The following sections contain sample code for inspiration on how to use Kvaser Python linlib.

Basic master slave usage

This code opens up one master and one slave, sets bitrate and then the slave sends a wakeup message to the master.

```
# import the linlib wrapper from the canlib package
from canlib import linlib

# print information about device firmware version
print(linlib.getChannelData(channel_number=0,
                            item=linlib.ChannelData.CARD_FIRMWARE_REV))

# open the first channel as a Master
master = linlib.openChannel(channel_number=0,
                            channel_type=linlib.ChannelType.MASTER)

# open the next channel as a Slave
```

(continues on next page)

(continued from previous page)

```
slave = linlib.openChannel(channel_number=1,
                           channel_type=linlib.ChannelType.SLAVE)

# setup bitrate
master.setBitrate(10000)
slave.setBitrate(10000)

# activate the LIN interface by going bus on
master.busOn()
slave.busOn()

# send a wakeup frame from the slave
slave.writeWakeup()

# read the frame when it arrives at the master
frame = master.read(timeout=100)
print(frame)

# go bus off
master.busOff()
slave.busOff()
```

Sending message from master

Our next example uses two shorthand helper functions to open the channels. We then send some messages from the master and see that they arrive.

```
from canlib import linlib, Frame

# open the first channel as Master, using helper function
master = linlib.openMaster(0)

# open the next channel as a Slave, using helper function
slave = linlib.openSlave(1)

# go bus on
master.busOn()
slave.busOn()

# send some messages from master
NUM_MESSAGES = 2
for i in range(NUM_MESSAGES):
    master.writeMessage(Frame(id_=i, data=[1, 2, 3, 4, 5, 6, 7, 8]))
master.writeSync(100)

# print the received messages at the slave
for i in range(NUM_MESSAGES):
    frame = slave.read(timeout=100)
    print(frame)

# the master should also have recorded the messages
```

(continues on next page)

(continued from previous page)

```
for i in range(NUM_MESSAGES):
    frame = master.read(timeout=100)
    print(frame)

# go bus off
master.busOff()
slave.busOff()
```

Requesting LIN 2.0 message

As a last example, let's look at using LIN 2.0 and setting up a message, using the `Frame` object, on the slave which is then requested by the master.

```
from canlib import linlib, Frame


ID = 0x17
DATA = bytearray([1, 2, 3, 4])

# open the first channel as Master, using helper function
master = linlib.openMaster(0, bps=20000)

# open the next channel as a Slave, using helper function
slave = linlib.openSlave(1)

master.busOn()
slave.busOn()

# configure channels to use LIN 2.0
slave.setupLIN(flags=linlib.Setup.ENHANCED_CHECKSUM | linlib.Setup.VARIABLE_DLC)
master.setupLIN(flags=linlib.Setup.ENHANCED_CHECKSUM | linlib.Setup.VARIABLE_DLC)

# setup a message in the slave
slave.updateMessage(Frame(id_=ID, data=DATA))

# request the message and print it
master.requestMessage(ID)
frame = master.read(timeout=100)
print(frame)

# clear the message
slave.clearMessage(0x17)

# we should now get an empty message
master.requestMessage(0x17)
frame = master.read(timeout=100)
print(frame)

# go bus off
master.busOff()
slave.busOff()
```

1.4 Using canlib (CANlib)

The canlib module wraps the CAN bus API (CANlib), which is used to interact with Kvaser CAN devices connected to your computer and the CAN bus. At its core you have functions to set bus parameters (e.g. bit rate), go bus on/off and read/write CAN messages. You can also use CANlib to download and start t programs on supported devices.

1.4.1 Introduction

Hello, CAN!

Let's start with a simple example:

```
# The CANlib library is initialized when the canlib module is imported.
from canlib import canlib, Frame

# Open a channel to a CAN circuit. In this case we open channel 0 which
# should be the first channel on the CAN interface. EXCLUSIVE means we don't
# want to share this channel with any other currently executing program.
# We also set the CAN bus bit rate to 250 kBit/s, using a set of predefined
# bus parameters.
ch = canlib.openChannel(
    channel=0,
    flags=canlib.Open.EXCLUSIVE,
    bitrate=canlib.canBITRATE_250K
)

# Set the CAN bus driver type to NORMAL.
ch.setBusOutputControl(canlib.Driver.NORMAL)

# Activate the CAN chip.
ch.busOn()

# Transmit a message with (11-bit) CAN id = 123, length 6 and contents
# (decimal) 72, 69, 76, 76, 79, 33.
frame = Frame(id_=123, data=b'HELLO!', dlc=6)
ch.write(frame)

# Wait until the message is sent or at most 500 ms.
ch.writeSync(timeout=500)

# Inactivate the CAN chip.
ch.busOff()

# Close the channel.
ch.close()
```

canlib Core API Calls

The following calls can be considered the "core" of canlib as they are essential for almost any program that uses the CAN bus:

- `canlib.canlib.openChannel` and `canlib.canlib.Channel.close`
- `canlib.canlib.Channel.busOn` and `canlib.canlib.Channel.busOff`
- `canlib.canlib.Channel.read`
- `canlib.canlib.Channel.write` and `canlib.canlib.Channel.writeSync`

1.4.2 Initialization

Library Initialization

The underlaying CANlib library is initialized when the module `canlib.canlib` is imported. This will initialize the CANlib library and enumerate all currently available CAN channels.

Library Deinitialization and Cleanup

Strictly speaking it is not necessary to clean up anything before terminating the application. If the application quits unexpectedly, the device driver will ensure the CAN controller is deactivated and the driver will also ensure the firmware (if any) is left in a consistent state.

To reinitialize the library in an orderly fashion you may want to call `writeSync` with a short timeout for each open handle before closing them with `canlib.canlib.Channel.close`, to ensure the transmit queues are empty. You can then start afresh by calling `canlib.canlib.reinitializeLibrary`.

Note: When calling `canlib.canlib.reinitializeLibrary`, all previously opened CAN handles (`canlib.canlib.Channel`) will be closed and invalidated.

1.4.3 Devices and Channels

Identifying Devices and Channels

Once we have imported `canlib.canlib` which enumerates the connected Kvaser CAN devices, we can call `getNumberOfChannels` to get the number of enumerated channels in our system.

This code snippet reads the number of enumerated channels found in the PC:

```
>>> from canlib import canlib
>>> canlib.getNumberOfChannels()
8
```

Channel Information

Use `ChannelData` to obtain data for a specific channel, for example, the hardware type of the CAN interface.

We can use `ChannelData` for the CANlib channel numbers 0, 1, 2,..., n-1 (where n is the number returned by `getNumberOfChannels`) to get information about that specific channel.

To uniquely identify a device, we need to look at both the `ChannelData.card_upc_no` and `ChannelData.card_serial_no`.

The following code snippet loops through all known channels and prints the type of the CAN card they're on.

```
>>> from canlib import canlib
...
... num_channels = canlib.getNumberOfChannels()
... print("Found %d channels" % num_channels)
... for channel in range(0, num_channels):
...     chdata = canlib.ChannelData(channel)
...     print("%d. %s (%s / %s)" % (
...         channel,
...         chdata.channel_name,
...         chdata.card_upc_no,
...         chdata.card_serial_no)
...     )
Found 8 channels
0. Kvaser Leaf Light HS (channel 0) (73-30130-00241-8 / 1346)
1. Kvaser Memorator Pro 2xHS v2 (channel 0) (73-30130-00819-9 / 11573)
2. Kvaser Memorator Pro 2xHS v2 (channel 1) (73-30130-00819-9 / 11573)
3. Kvaser Leaf Pro HS v2 (channel 0) (73-30130-00843-4 / 10012)
4. Kvaser Hybrid 2xCAN/LIN (channel 0) (73-30130-00965-3 / 1100)
5. Kvaser Hybrid 2xCAN/LIN (channel 1) (73-30130-00965-3 / 1100)
6. Kvaser Virtual CAN Driver (channel 0) (00-00000-00000-0 / 0)
7. Kvaser Virtual CAN Driver (channel 1) (00-00000-00000-0 / 0)
```

Customized Channel Name

It is possible to set the customized name returned by `ChannelData.card_serial_no` on the device using Kvaser Device Guide by right clicking on the device channel and selecting "Edit Channel Name"

Now we can read the customized name:

```
>>> from canlib import canlib
>>> chdata = canlib.ChannelData(channel_number=0)
>>> chdata.custom_name
'Red Channel'
```

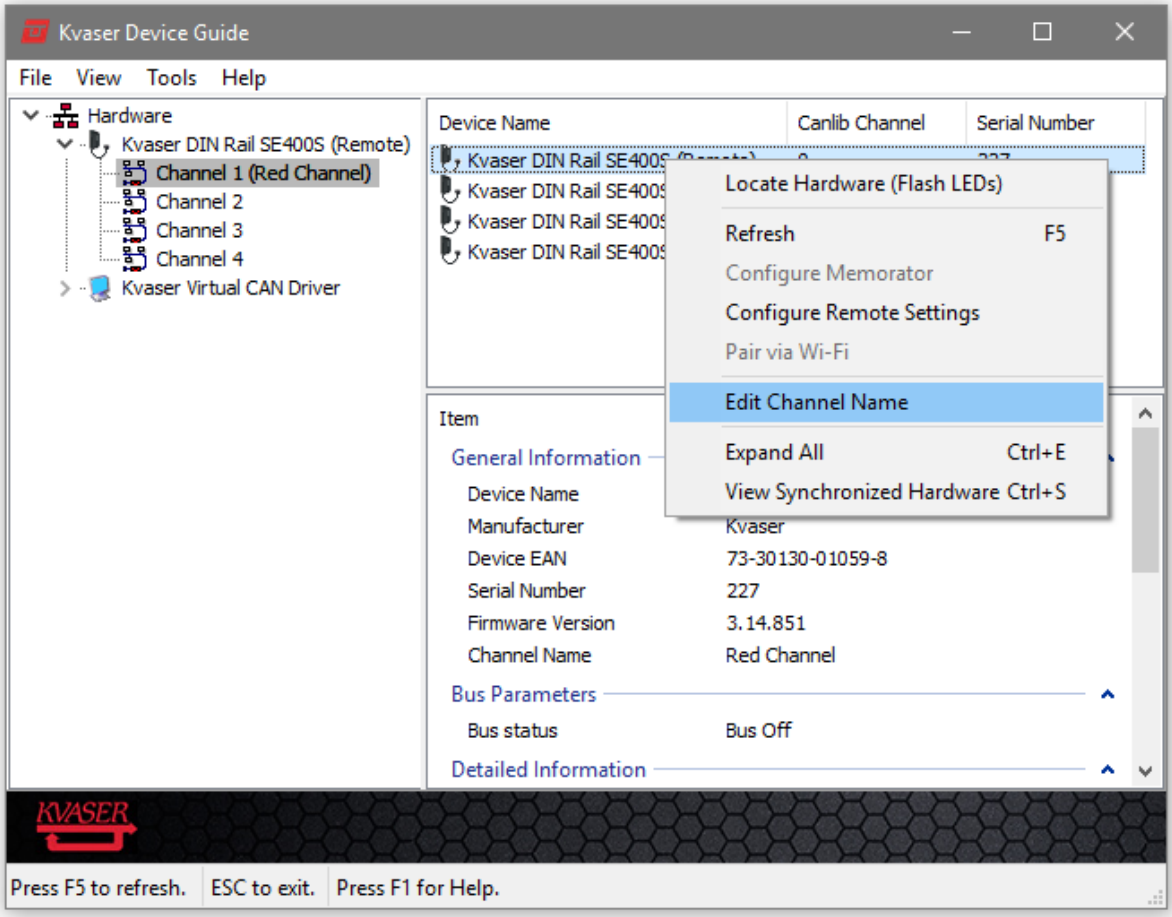

Fig. 1: Setting the device's Channel Name from inside Kvaser Device Guide

Virtual Channels

CANlib supports virtual channels that you can use for development, test or demonstration purposes when you don't have any hardware installed.

To open a virtual channel, call `canlib.canlib.openChannel` with the appropriate channel number, and specify `ACCEPT_VIRTUAL` in the flags argument to canOpenChannel().

1.4.4 Open Channel

Once we have imported `canlib.canlib` to enumerate the connected Kvaser CAN devices, the next call is likely to be a call to `openChannel`, which returns a `Channel` object for the specific CAN circuit. This object is then used for subsequent calls to the library. The `openChannel` function's first argument is the number of the desired channel, the second argument is modifier flags `canlib.canlib.Open`.

`openChannel` may raise several different exceptions, one of which is `CanNotFound`. This means that the channel specified in the first parameter was not found, or that the flags passed to `openChannel` is not applicable to the specified channel.

Open as CAN

No special `canlib.canlib.Open` modifier flag is needed in the flags argument to `openChannel` when opening a channel in CAN mode.

```
>>> from canlib import canlib
>>> canlib.openChannel(channel=0, flags=canlib.Open.EXCLUSIVE)
<canlib.canlib.channel.Channel object at 0x0000015B787EDA90>
```

Open as CAN FD

To open a channel in CAN FD mode, either `CAN_FD` or `CAN_FD_NONISO` needs to be given in the flags argument to `openChannel`.

This example opens channel 0 in CAN FD mode for exclusive usage by this application:

```
>>> from canlib import canlib
>>> ch = canlib.openChannel(
...     channel=0,
...     flags=canlib.Open.CAN_FD | canlib.Open.EXCLUSIVE,
... )
>>> ch.close()
```

Close Channel

Closing a channel is done using `close`. If no other handles are referencing the same CANlib channel, the channel is taken off bus.

The CAN channel can also be opened and closed using a context manager:

```
>>> from canlib import canlib
>>> with canlib.openChannel(channel=1) as ch:
...     ...
```

Check Channel Capabilities

Channel specific information and capabilities are made available by reading attributes of an instance of type `ChannelData`.

The device clock frequency can be obtained via `frequency()`:

```
>>> from canlib import canlib
>>> chd = canlib.ChannelData(channel_number=0)
>>> clock_info = chd.clock_info
>>> clock_info.frequency()
80000000
```

The capabilities of a channel can be obtained by reading attribute `channel_cap` and `channel_cap_ex`:

```
>>> from canlib import canlib
>>> chd = canlib.ChannelData(channel_number=0)
>>> chd.channel_cap
ChannelCap.IO_API|SCRIPT|LOGGER|SINGLE_SHOT|SILENT_MODE|CAN_FD_NONISO|CAN_FD|
TXACKNOWLEDGE|TXREQUEST|GENERATE_ERROR|ERROR_COUNTERS|BUS_STATISTICS|EXTENDED_CAN
>>> chd.channel_cap_ex[0]
ChannelCapEx.BUSPARAMS_TQ
```

A bitwise AND operator can be used to see if a channel has a specific capability.

```
>>> if (chd.channel_cap & canlib.ChannelCap.CAN_FD):
>>>   print("Channel has support for CAN FD!")
Channel has support for CAN FD!
```

The above printouts are just an example, and will differ for different devices and installed firmware.

Set CAN Bitrate

After opening the channel in classic CAN mode (see *Open as CAN*), use `set_bus_params_tq` to specify the bit timing parameters on the CAN bus. Bit timing parameters are packaged in an instance of type *BusParamsTq*. Note that the synchronization segment is excluded as it is always one time quantum long.

Example: Set the bus speed to 500 kbit/s on a CAN device with an 80 MHz oscillator:

```
>>> from canlib import canlib
>>> ch = canlib.openChannel(channel=0)
>>> params = canlib.busparams.BusParamsTq(
...     tq=8,
...     phase1=2,
...     phase2=2,
...     sjw=1,
...     prescaler=20,
...     prop=3
... )
>>> ch.set_bus_params_tq(params)
```

In the example a prescaler of 20 is used, resulting in each bit comprising of 160 time quanta (8 * 20). The nominal bus speed is given by 80 * 10^6 / (20 * 8) = 500 * 10^3.

If uncertain how to set a specific bus speed, one can use *calc_busparamstq*, which returns a *BusParamsTq* object:

```
>>> calc_busparamstq(
... target_bitrate=470_000,
... target_sample_point=82,
... target_sync_jump_width=15.3,
... clk_freq=clock_info.frequency(),
... target_prop_tq=50,
... prescaler=2)
BusParamsTq(tq=85, prop=25, phase1=44, phase2=15, sjw=13, prescaler=2)
```

For users that are not interested in specifying individual bit timing parameters, CANlib also provides a set of default parameter settings for the most common bus speeds through the `canlib.canlib.canBITRATE_xxx` constants. The predefined bitrate constants may be set directly in the call to `openChannel`:

```
>>> ch = canlib.openChannel(channel=0, bitrate=canlib.canBITRATE_500K)
```

Table 1: Bit timing parameters for some of the most common bus speeds on a CAN device with an 80 MHz oscillator[1]

	tq	phase1	phase2	sjw	prop	prescaler	Sample point	Bitrate
canBITRATE_10K	16	4	4	1	7	500	75%	10 kbit/s
canBITRATE_50K	16	4	4	1	7	100	75%	50 kbit/s
canBITRATE_62K	16	4	4	1	7	80	75%	62 kbit/s
canBITRATE_83K	8	2	2	2	3	120	75%	83 kbit/s
canBITRATE_100K	16	4	4	1	7	50	75%	100 kbit/s
canBITRATE_125K	16	4	4	1	7	40	75%	125 kbit/s
canBITRATE_250K	8	2	2	1	3	40	75%	250 kbit/s
canBITRATE_500K	8	2	2	1	3	20	75%	500 kbit/s
canBITRATE_1M	8	2	2	1	3	10	75%	1 Mbit/s

If uncertain how to calculate bit timing parameters, appropriate values can be acquired using the Bit Timing Calculator. Note that in classic CAN mode, only the nominal bus parameters are of concern when using the Bit Timing Calculator.

[1] See *Check Channel Capabilities* for information on clock frequency.

Set CAN FD Bitrate

After opening a channel in CAN FD mode (see *Open as CAN FD*), bit timing parameters for both the arbitration and data phases need to be set. This is done by a call to `set_bus_params_tq`, with two separate instances of type *BusParamsTq* as arguments.

Example: Set the arbitration phase bitrate to 500 kbit/s and the data phase bitrate to 1000 kbit/s, with sampling points at 80%.

```
>>> from canlib import canlib
>>> ch = canlib.openChannel(channel=0, flags=canlib.Open.CAN_FD)
>>> params_arbitration = canlib.busparams.BusParamsTq(
...     tq=80,
...     phase1=16,
...     phase2=16,
...     sjw=16,
...     prescaler=2,
...     prop=47
... )
>>> params_data = canlib.busparams.BusParamsTq(
...     tq=40,
...     phase1=31,
...     phase2=8,
...     sjw=8,
...     prescaler=2,
...     prop=0
... )
>>> ch.set_bus_params_tq(params_arbitration, params_data)
```

For users that are not interested in specifying individual bit timing parameters, CANlib also provides a set of default parameter settings for the most common bus speeds through the `canlib.canlib.canFD_BITRATE_xxx` constants. The predefined bitrates may be set directly in the call to `openChannel`:

```
>>> ch = canlib.openChannel(
...     channel=0,
...     flags=canlib.Open.CAN_FD,
...     bitrate=canlib.canlib.canFD_BITRATE_500K_80P,
...     data_bitrate=canlib.canFD_BITRATE_1M_80P,
... )
```

For bus speeds other than the predefined `canlib.canlib.canFD_BITRATE_xxx` constants, bit timing parameters have to be specified manually.

Table 2: Available predefined bitrate constants with corresponding bit timing parameters for a CAN FD device with an 80 MHz oscillator[Page 19, 1]

	tq	phase1	phase2	sjw	prop	prescaler	Sample point	Bitrate
canFD_BITRATE_500K_80P	40	8	8	8	23	4	80%	500 kbit/s
canFD_BITRATE_1M_80P	40	8	8	8	23	2	80%	1 Mbit/s
canFD_BITRATE_2M_80P	20	4	4	4	0	2	80%	2 Mbit/s
canFD_BITRATE_4M_80P	10	2	2	2	0	2	80%	4 Mbit/s
canFD_BITRATE_8M_60P	5	2	2	1	0	2	60%	8 Mbit/s

If uncertain how to calculate bit timing parameters, appropriate values can be acquired using the Bit Timing Calculator.

CAN Driver Modes

Use `setBusOutputControl` to set the bus driver mode. This is usually set to `canlib.canlib.Driver.NORMAL` to obtain the standard push-pull type of driver. Some controllers also support `canlib.canlib.Driver.SILENT` which makes the controller receive only, not transmit anything, not even ACK bits. This might be handy for e.g. when listening to a CAN bus without interfering.

```
>>> from canlib import canlib
>>> with canlib.openChannel(channel=1) as ch:
...     ch.setBusOutputControl(canlib.Driver.SILENT)
...     ...
```

`canlib.canlib.Driver.NORMAL` is set by default.

Legacy Functions

The following functions are still supported by canlib.

Set CAN Bitrate

`setBusParams` can be used to set the CAN bus parameters, including bitrate, the position of the sampling point etc, they are also described in most CAN controller data sheets. Depending on device and installed firmware, the requested parameters may be subject to scaling in order to accommodate device specific restrictions. As such, reading back bus parameters using `getBusParamsFd` can return bus parameter settings different than the ones supplied. Note however, that a successful call to `setBusParamsFd` will always result in the requested bit rate being set on the bus, along with bus parameters that for all intents and purposes are equivalent to the ones requested.

Set the speed to 125 kbit/s, each bit comprising 8 (= 1 + 4 + 3) quanta, the sampling point occurs at 5/8 of a bit; SJW = 1; one sampling point:

```
>>> ch.setBusParams(freq=125000, tseg1=4, tseg2=3, sjw=1, noSamp=1)
```

Set the speed to 111111 kbit/s, the sampling point to 75%, the SJW to 2 and the number of samples to 1:

```
>>> ch.setBusParams(freq=111111, tseg1=5, tseg2=2, sjw=2, noSamp=1)
```

For full bit timing control, use `set_bus_params_tq` instead.

Set CAN FD Bitrate

After a channel has been opened in CAN FD mode, `setBusParams`, and `setBusParamsFd` can be used to set the arbitration and data phase bitrates respectively. Depending on device and installed firmware, the requested parameters may be subject to scaling in order to accommodate device specific restrictions. As such, reading back bus parameters using `getBusParamsFd` can return bus parameter settings different than the ones supplied. Note however, that a successful call to `setBusParamsFd` will always result in the requested bit rate being set on the bus, along with bus parameters that for all intents and purposes are equivalent to the ones requested.

Set the nominal bitrate to 500 kbit/s and the data phase bitrate to 1000 kbit/s, with sampling points at 80%.

```
>>> ch.setBusParams(freq=500000, tseg1=63, tseg2=16, sjw=16, noSamp=1);
>>> ch.setBusParamsFd(freq_brs=1000000, tseg1_brs=31, tseg2_brs=8, sjw_brs=8);
```

For full bit timing control, use `set_bus_params_tq` instead.

1.4.5 CAN Frames

CAN Data Frames

The CAN Data Frame, represented by the *Frame* object, is the most common message type, which consists of the following major parts (a few details are omitted for the sake of brevity):

CAN identifier: *canlib.Frame.id* The CAN identifier, or Arbitration Field, determines the priority of the message when two or more nodes are contending for the bus. The CAN identifier contains for:

- CAN 2.0A, an 11-bit Identifier and one bit, the RTR bit, which is dominant for data frames.
- CAN 2.0B, a 29-bit Identifier, with the EXT bit set, (which also contains two recessive bits: SRR and IDE) and the RTR bit.

Data field: *canlib.Frame.data* The Data field contains zero to eight bytes of data.

Data Length Code: *canlib.Frame.dlc* The DLC field specifies the number of data bytes in the Data field.

CRC Field: The CRC Field contains a 15-bit checksum calculated on most parts of the message. This checksum is used for error detection.

Acknowledgement Slot: Any CAN controller that has been able to correctly receive the message sends an Acknowledgement bit at the end of each message. The transmitter checks for the presence of the Acknowledge bit and retransmits the message if no acknowledge was detected.

Note: It is worth noting that the presence of an Acknowledgement Bit on the bus does not mean that any of the intended addressees has received the message. The only thing we know is that one or more nodes on the bus has received it correctly. The Identifier in the Arbitration Field is not, despite of its name, necessarily identifying the contents of the message.

The *canlib.Frame.flags* attribute consists of message information flags, according to `canlib.canlib.MessageFlag`.

CAN FD Data Frames

A standard CAN network is limited to 1 MBit/s, with a maximum payload of 8 bytes per frame. CAN FD increases the effective data-rate by allowing longer data fields - up to 64 bytes per frame - without changing the CAN physical layer. CAN FD also retains normal CAN bus arbitration, increasing the bit-rate by switching to a shorter bit time only at the end of the arbitration process and returning to a longer bit time at the CRC Delimiter, before the receivers send their acknowledge bits. A realistic bandwidth gain of 3 to 8 times what's possible in CAN will particularly benefit flashing applications.

Error Frames

Nearly all hardware platforms support detection of Error Frames. If an Error Frame arrives, the flag `ERROR_FRAME` is set in the *Frame*. The identifier is garbage if an Error Frame is received, but for LAPcan it happens to be 2048 plus the error code from the SJA1000.

Many platforms support transmission of Error Frames as well. To send Error Frames, set the `ERROR_FRAME` flag in the *Frame* before sending using `write`.

Simply put, the Error Frame is a special message that violates the framing rules of a CAN message. It is transmitted when a node detects a fault and will cause all other nodes to detect a fault - so they will send Error Frames, too. The transmitter will then automatically try to retransmit the message. There is an elaborate scheme of error counters that ensures that a node can't destroy the bus traffic by repeatedly transmitting error frames.

The Error Frame consists of an Error Flag, which is 6 bits of the same value (thus violating the bit-stuffing rule) and an Error Delimiter, which is 8 recessive bits. The Error Delimiter provides some space in which the other nodes on the bus can send their Error Flags when they detect the first Error Flag.

Remote Requests

You can send remote requests by passing the `RTR` flag to `write`. Received remote frames are reported by `read` et.al. using the same flag.

The Remote Frame is just like the Data Frame, with two important differences:

- It is explicitly marked as a Remote Frame (the RTR bit in the Arbitration Field is recessive)
- There is no Data Field.

The intended purpose of the Remote Frame is to solicit the transmission of the corresponding Data Frame. If, say, node A transmits a Remote Frame with the Arbitration Field set to 234, then node B, if properly initialized, might respond with a Data Frame with the Arbitration Field also set to 234.

Remote Frames can be used to implement a type of request-response type of bus traffic management. In practice, however, the Remote Frame is little used. It is also worth noting that the CAN standard does not prescribe the behaviour outlined here. Most CAN controllers can be programmed either to automatically respond to a Remote Frame, or to notify the local CPU instead.

There's one catch with the Remote Frame: the Data Length Code must be set to the length of the expected response message even though no data is sent. Otherwise the arbitration will not work.

Sometimes it is claimed that the node responding to the Remote Frame is starting its transmission as soon as the identifier is recognized, thereby "filling up" the empty Remote Frame. This is not the case.

Overload Frames

Overload Frames aren't used nowadays. Certain old CAN controllers (Intel 82526) used them to delay frame processing in certain cases.

Other frame features of interest

There are some other frame features of interest:

- You can send wakeup frames (used for Single-Wire CAN) if your hardware supports it, for example, a LAPcan plus a DRVcan S. Just set the `WAKEUP` flag.
- For "low-speed CAN" (1053/1054 type transceivers), the `NERR` flag is set if a frame is received in "fault-tolerant" mode.

1.4.6 Send and Receive

Bus On / Bus Off

When the CAN controller is on bus, it is receiving messages and is sending acknowledge bits in response to all correctly received messages. A controller that is off bus is not taking part in the bus communication at all.

When you have a `canlib.canlib.Channel` object, use `busOn` to go on bus and `busOff` to go off bus.

If you have multiple Channel objects to the same controller, the controller will go off bus when the last of the Channel objects go off bus (i.e. all Channel objects must be off bus for the controller to be off bus). You can use readStatus and watch the flag BUS_OFF to see if the controller has gone off bus.

You can set a channel to silent mode by using the SILENT mode if you want it to be on-bus without interfering with the traffic in any way, see *CAN Driver Modes*.

This example opens a channel, takes it on-bus, then takes it off-bus and closes it:

```
>>> from canlib import canlib
... with canlib.openChannel(channel=1) as ch:
...     ch.busOn()
...     ...
...     ch.busOff()
```

Reading Messages

Incoming messages are placed in a queue in the driver. In most cases the hardware does message buffering as well. You can read the first message in the queue by calling read, which will raise the exception canlib.canlib.CanNoMsg if there was no message available.

The *flags* attribute of the *Frame* returned by read contains a combination of the MessageFlag flags, including FDF, BRS, and ESI if the CAN FD protocol is enabled, and error flags such as OVERRUN which provides you with more information about the message; for example, a frame with a 29-bit identifier will have the EXT bit set, and a remote frame will have the RTR bit set. Note that the flag argument is a combination of the MessageFlag, so more than one flag might be set.

See *CAN Frames* for more information.

Sometimes it is desirable to have a peek into the more remote parts of the queue. Is there, for example, any message waiting that has a certain identifier?

- If you want to read just a message with a specified identifier, and throw all others away, you can call readSpecificSkip. This routine will return the first message with the specified identifier, discarding any other message in front of the desired one.
- If you want to wait until a message arrives (or a timeout occurs) and then read it, call read with a timeout.
- If you want to wait until there is at least one message in the queue with a certain identifier, but you don't want to read it, call readSyncSpecific.

The following code fragment reads the next available CAN message, (using default bitrate 500 kbit/s):

```
>>> from canlib import canlib
... with canlib.openChannel(channel=0) as ch:
...     ch.busOn()
...     frame = ch.read(timeout=1000)
...     ch.busOff()
>>> frame
Frame(id=709, data=bytearray(b'\xb5R'), dlc=2, flags=<MessageFlag.STD: 2>, timestamp=3)
```

Acceptance Filters

You can set filters to reduce the number of received messages. CANlib supports setting of the hardware filters on the CAN interface board. This is done with the `canlib.canlib.Channel.canAccept` function.

You set an acceptance code and an acceptance mask which together determine which CAN identifiers are accepted or rejected.

If you want to remove an acceptance filter, call `canlib.canlib.Channel.canAccept` with the mask set to `NULL_MASK`.

To set the mask to 0xF0 and the code to 0x60:

```
>>> from canlib import canlib
>>> ch = canlib.openChannel(channel=0)
>>> ch.canAccept(0x0f0, canlib.AcceptFilterFlag.SET_MASK_STD)
>>> ch.canAccept(0x060, canlib.AcceptFilterFlag.SET_CODE_STD)
>>> ...
>>> ch.close()
```

This code snippet will cause all messages having a standard (11-bit) identifier with bit 7 - bit 4 in the identifier equal to 0110 (binary) will pass through. Other messages with standard identifiers will be rejected.

How acceptance filters can be used in a smaller project:

```
>>> from canlib import canlib
>>> ch = canlib.openChannel(channel=0)
>>> # The acceptance filter only have to be called once for each ch object
>>> ch.canAccept(0x0f0, canlib.AcceptFilterFlag.SET_MASK_STD)
>>> ch.canAccept(0x060, canlib.AcceptFilterFlag.SET_CODE_STD)
>>> ...
>>> # We can now run the rest of the program and the acceptance filter
>>> # will reject unwanted CAN messages.
>>> while(True):
>>>     frame = ch.read()
>>>     ...
>>> ...
```

Code and Mask Format

Explanation of the code and mask format used by the `canlib.canlib.Channel.canAccept` function:

> A binary 1 in a mask means "the corresponding bit in the code is relevant" A binary 0 in a mask means "the corresponding bit in the code is not relevant" A relevant binary 1 in a code means "the corresponding bit in the identifier must be 1" A relevant binary 0 in a code means "the corresponding bit in the identifier must be 0"

In other words, the message is accepted if ((code XOR id) AND mask) == 0.

Sending Messages

You transmit messages by calling `canlib.canlib.Channel.write`. Outgoing CAN messages are buffered in a transmit queue and sent on a First-In First-Out basis. You can use `canlib.canlib.Channel.writeSync` to wait until the messages in the queue have been sent.

Sending a CAN message:

```
>>> from canlib import canlib, Frame
... with canlib.openChannel(channel=0) as ch:
...     ch.busOn()
...     frame = Frame(id_=234, data=[1,2])
...     ch.write(frame)
...     ch.busOff()
```

Using Extended CAN (CAN 2.0B)

"Standard" CAN has 11-bit identifiers in the range 0 - 2047. "Extended" CAN, also called CAN 2.0B, has 29-bit identifiers. You specify which kind of identifiers you want to use in your call to canWrite(): if you set the `EXT` flag in the flag argument, the message will be transmitted with a 29-bit identifier. Conversely, received 29-bit-identifier messages have the `EXT` flag set.

The following code fragment sends a CAN message on an already open channel. The CAN message will have identifier 1234 (extended) and DLC = 8. The contents of the data bytes will be whatever the data array happens to contain:

```
>>> frame = Frame(id_=1234, data=[1,2,3,4,5,6,7,8], flags=canlib.MessageFlag.EXT)
>>> frame
Frame(id=1234, data=bytearray(b'\x01\x02\x03\x04\x05\x06\x07\x08'), dlc=8, flags=
↪<MessageFlag.EXT: 4>, timestamp=None)
>>> ch.write(frame)
```

Object Buffers

Object buffers are currently not supported in the Python wrapper.

1.4.7 Bus Errors

Obtaining Bus Status Information

Use `read_error_counters` to read the error counters of the CAN controller. There are two such counters in a CAN controller (they are required by the protocol definition). Not all CAN controllers allow access to the error counters, so CANlib may provide you with an "educated guess" instead.

Use `readStatus` to obtain the bus status (error active, error passive, bus off; as defined by the CAN standard).

Overruns

If the CAN interface or the driver runs out of buffer space, or if the bus load is so high that the CAN controller can't keep up with the traffic, an overload condition is flagged to the application.

The driver will set the `HW_OVERRUN` and/or `SW_OVERRUN` flags in the flag argument of `read` and its relatives. The flag(s) will be set in the first message read from the driver after the overrun or overload condition happened.

Not all hardware platforms can detect the difference between hardware overruns and software overruns, so your application should test for both conditions. You can use the symbol `OVERRUN` for this purpose.

Error Frames

When a CAN controller detects an error, it transmits an error frame. This is a special CAN message that causes all other CAN controllers on the bus to notice that an error has occurred.

CANlib will report error frames to the application just like it reports any other CAN message, but the `ERROR_FRAME` flag will be set in the flags parameter when e.g. `read` returns.

When an error frame is received, its identifier, DLC and data bytes will be undefined. You should test if a message is an error frame before checking its identifier, DLC or data bytes.

In an healthy CAN system, error frames should rarely, if ever, occur. Error frames usually mean there is some type of serious problem in the system, such as a bad connector, a bad cable, bus termination missing or faulty, or another node transmitting at wrong bit rate, and so on.

1.4.8 Time Measurement

CAN messages are time stamped as they arrive. This time stamping is, depending on your hardware platform, done either by the CAN interface hardware or by CANlib.

In the former case, the accuracy is pretty good, in the order of 1 - 10 microseconds; when CANlib does the job, the accuracy is more like 100 microseconds to 10 milliseconds and you may experience a rather large jitter. This is because Windows is not a real-time operating system.

Use `Channel.readTimer` to read the current time, the return value is the current time using the clock of that channel.

Accuracy

The accuracy of the time stamps depends on the hardware.

The members of the Kvaser Leaf family have an onboard CPU. The time stamp accuracy varies (check the hardware manual) but the high-end members have very precise time stamping. The accuracy can be as good as one microsecond depending on the hardware. If more than one Leaf is used, their clocks are automatically kept in sync by the Kvaser MagiSync™ technology.

Other CAN interfaces, like the Kvaser Leaf, LAPcan and USBcan II, have an on-board CPU and clock and provide very accurate time stamps for incoming CAN messages. The accuracy is typically 10-20 microseconds.

Certain interfaces, like the PCIcan (PCI) series of boards, don't have an on-board CPU so the driver relies on the clock in the PC to timestamp the incoming messages. As Windows is not a real-time operating system, this gives an accuracy which is in the order of one millisecond.

Resolution

The resolution of the time stamps is, by default, 1 ms. It can be changed to a better resolution if desired.

Use `IOControl` attribute `timer_scale` to change the resolution of the time stamps, if desired. This will not affect the accuracy of the time stamps.

1.4.9 Using Threads

Handles are thread-specific

CANlib supports programs with multiple threads as long as one important condition is met: A handle to a CAN circuit should be used in only one thread.

This means that you cannot share e.g. `canlib.Channel` objects between threads. Each thread has to open its own handle to the circuit.

Also note that you must call `busOn` and `busOff` once for each handle even if the handles are opened on the same physical channel.

Local echo feature

If you are using the same channel via multiple handles, note that the default behaviour is that the different handles will "hear" each other just as if each handle referred to a channel of its own. If you open, say, channel 0 from thread A and thread B and then send a message from thread A, it will be "received" by thread B. This behaviour can be changed using `IOControl` and `local_txecho`.

Init access

Init access means that the thread that owns the handle can set bit rate and CAN driver mode. Init access is the default. At most one thread can have init access to any given channel. If you try to set the bit rate or CAN driver mode for a handle to which you don't have init access, the call will silently fail, unless you enable access error reporting by using `IOControl` and `report_access_errors`. Access error reporting is by default off.

Using the same handle in different threads

In spite of what was said above, you can use a single handle in different threads, provided you create the appropriate mutual exclusion mechanisms yourself. Two threads should never call CANlib simultaneously unless they are using different handles.

1.4.10 I/O Pin Handling

Initialize

Some Kvaser products feature I/O pins that can be used in real-time applications using a part of the API dedicated to I/O Pin Handling. This API is initialized by confirming the I/O pin configuration, see kvIoConfirmConfig. Before the configuration is confirmed the user can only retrieve information about the pins.

```
>>> from canlib import canlib, Device
... device = Device.find(serial=66666)
... channel = device.channel_number()
... ch = canlib.openChannel(channel)
... config = canlib.iopin.Configuration(ch)
... ch.get_io_pin(86).pin_type
<PinType.ANALOG: 2>
>>> for pin in config:
...     print(pin)
Pin 0: <PinType.DIGITAL: 1> <Direction.OUT: 8> bits=1 range=0.0-24.0 (<ModuleType.
↪DIGITAL: 1>)
Pin 1: <PinType.DIGITAL: 1> <Direction.OUT: 8> bits=1 range=0.0-24.0 (<ModuleType.
↪DIGITAL: 1>)
 :
Pin 31: <PinType.DIGITAL: 1> <Direction.IN: 4> bits=1 range=0.0-24.0 HL_filter=5000 LH_
↪filter=5000 (<ModuleType.DIGITAL: 1>)
```

After the configuration has been confirmed the user may set or read any values of the I/O pins:

```
>>> config.confirm()

>>> ch.get_io_pin(0).value
0
>>> ch.get_io_pin(0).value = 1

>>> ch.get_io_pin(0).value
1
```

Pin Information

Pins are identified by their pin number, which is a number from zero up to, but not including, the value returned by `canlib.Channel.number_of_io_pins`. Using the pin number, the specific properties of any pin is retrieved and set using `canlib.iopin.IoPin`.

I/O pin types

There are currently three types of pins that is supported by the API dedicated to I/O Pin Handling. These include analog, digital and relay pins. To learn what pin type a given pin is, use `canlib.iopin.IoPin.pin_type`. See `canlib.iopin.PinType` to see all supported types.

Analog Pins

The analog pins are represented by multiple bits, the number of bits can be retrieved by calling `canlib.iopin.IoPin.number_of_bits`. The value of an analog pin is within in the interval given by `range_min` and `range_min`. The analog input pin has two configurable properties, namely the low pass filter order and the hysteresis. See `lp_filter_order` and `hysteresis`. Pins are read and set using `value`. When reading an output, the latest value set is retrieved.

Digital Pins

The digital pins have two configurable properties, namely the low-to-high and the high-to-low filter time. See `canlib.iopin.DigitalIn.high_low_filter` and `low_high_filter`. Pins are read and set using `value`. When reading an output, the latest value set is retrieved.

Relay Pins

The relay pins have no configurable properties. All of these pins are considered as outputs. Pins are set and read using `value`.

1.5 Examples

This section contains a number of examples or how-tos solving common problems. They are all scripts ready to be run, using python's `argparse` to accept arguments.

1.5.1 Convert a .kme50 file to plain ASCII

```
"""convert_kme_asc.py -- Convert a .kme50 logfile into plain ASCII

This example script uses canlib.kvlclib to convert a logfile from .kme50 format
into plain ASCII.

"""
import argparse
from pathlib import Path

from canlib import kvlclib


def try_set_propery(cnv, property, value=None):
    # Check if the format supports the given property
    if cnv.format.isPropertySupported(property):
        # If a value is specified, set the property to this value
        if value is not None:
            cnv.setProperty(property, value)

        # Get the property's default value
        default = cnv.format.getPropertyDefault(property)
        print('  %s is supported (Default: %s)' % (property, default))

        # Get the property's current value
        value = cnv.getProperty(property)
        print('    Current value: %s' % value)
    else:
        print('  %s is not supported' % property)


def convert_events(cnv):
```

(continues on next page)

(continued from previous page)

```
    # Get estimated number of remaining events in the input file. This
    # can be useful for displaying progress during conversion.
    total = cnv.eventCount()
    print("Converting about %d events..." % total)
    while True:
        try:
            # Convert events from input file one by one until EOF
            # is reached
            cnv.convertEvent()
            if cnv.isOutputFilenameNew():
                print("New output filename: '%s'" % cnv.getOutputFilename())
                print("About %d events left..." % cnv.eventCount())
        except kvlclib.KvlcEndOfFile:
            if cnv.isOverrunActive():
                print("NOTE! The extracted data contained overrun.")
                cnv.resetOverrunActive()
            if cnv.isDataTruncated():
                print("NOTE! The extracted data was truncated.")
                cnv.resetStatusTruncated()
            break


if __name__ == '__main__':
    parser = argparse.ArgumentParser(description="Convert a .kme50 logfile into plain␣
↪ASCII.")
    parser.add_argument(
        'filename', metavar='LOGFILE.KME50', help=("The filename of the .kme50 logfile.")
    )
    args = parser.parse_args()
    in_file = Path(args.filename)

    # set up formats
    out_fmt = kvlclib.WriterFormat(kvlclib.FileFormat.PLAIN_ASC)
    in_fmt = kvlclib.ReaderFormat(kvlclib.FileFormat.KME50)

    # set resulting output file name taking advantage of the extension
    # defined in the format.
    out_file = in_file.with_suffix('.' + out_fmt.extension)
    print("Output filename is '%s'" % out_file)

    # create converter
    cnv = kvlclib.Converter(out_file, out_fmt)

    # Set input file and format
    cnv.setInputFile(in_file, kvlclib.FileFormat.KME50)

    # split output files into max 100 MB files
    # The name of the resulting files will now end in '-partX.txt',
    # thus the first file will be named logfile-part0.txt, assuming we use
    # logfile.kme50 as input file name.
    try_set_propery(cnv, kvlclib.Property.SIZE_LIMIT, 100)
```

(continues on next page)

(continued from previous page)

```
    # allow output file to overwrite existing files
    try_set_propery(cnv, kvlclib.Property.OVERWRITE, 1)

    # we are only interested in the first channel
    cnv.setProperty(kvlclib.Property.CHANNEL_MASK, 1)

    # add nice header to the output file
    try_set_propery(cnv, kvlclib.Property.WRITE_HEADER, 1)

    # we are converting CAN traffic with max 8 bytes, so we can minimize
    # the width of the data output to 8 bytes
    try_set_propery(cnv, kvlclib.Property.LIMIT_DATA_BYTES, 8)

    convert_events(cnv)

    # force flush result to disk
    cnv.flush()
```

Description

We have created a wrapper function `try_set_propery` that will examine the property we are trying to set, and ignore the setting if the current format used does not support the property. While converting events in the `convert_events` function, we also inform the user if any overruns or data truncation was detected.

Sample Output

```
C:\example>python convert_kme_asc.py gensig.kme50
Output filename is 'C:\example\gensig.txt'
  Property.SIZE_LIMIT is supported (Default: 0)
    Current value: 100
  Property.OVERWRITE is supported (Default: 0)
    Current value: 1
  Property.WRITE_HEADER is supported (Default: 0)
    Current value: 1
  Property.LIMIT_DATA_BYTES is supported (Default: 64)
    Current value: 8
Converting about 310 events...
New output filename: 'C:\example\gensig-part0.txt
About 309 events left...
```

1.5.2 Create a Database

```
"""create_db.py -- Creating a .dbc database from scratch

This script will use canlib.kvadblib to create a new database file filled with
arbitrary data.

"""

import argparse
from collections import namedtuple

from canlib import kvadblib

Message = namedtuple('Message', 'name id dlc signals')
Signal = namedtuple('Signal', 'name size scaling limits unit')
EnumSignal = namedtuple('EnumSignal', 'name size scaling limits unit enums')

_messages = [
    Message(
        name='EngineData',
        id=100,
        dlc=8,
        signals=[
            Signal(
                name='PetrolLevel',
                size=(24, 8),
                scaling=(1, 0),
                limits=(0, 255),
                unit="l",
            ),
            Signal(
                name='EngPower',
                size=(48, 16),
                scaling=(0.01, 0),
                limits=(0, 150),
                unit="kW",
            ),
            Signal(
                name='EngForce',
                size=(32, 16),
                scaling=(1, 0),
                limits=(0, 0),
                unit="N",
            ),
            EnumSignal(
                name='IdleRunning',
                size=(23, 1),
                scaling=(1, 0),
                limits=(0, 0),
                unit="",
                enums={'Running': 0, 'Idle': 1},
            ),
```

(continues on next page)

(continued from previous page)

```
            Signal(
                name='EngTemp',
                size=(16, 7),
                scaling=(2, -50),
                limits=(-50, 150),
                unit="degC",
            ),
            Signal(
                name='EngSpeed',
                size=(0, 16),
                scaling=(1, 0),
                limits=(0, 8000),
                unit="rpm",
            ),
        ],
    ),
    Message(
        name='GearBoxInfo',
        id=1020,
        dlc=1,
        signals=[
            Signal(
                name='EcoMode',
                size=(6, 2),
                scaling=(1, 0),
                limits=(0, 1),
                unit="",
            ),
            EnumSignal(
                name='ShiftRequest',
                size=(3, 1),
                scaling=(1, 0),
                limits=(0, 0),
                unit="",
                enums={'Shift_Request_On': 1, 'Shift_Request_Off': 0},
            ),
            EnumSignal(
                name='Gear',
                size=(0, 3),
                scaling=(1, 0),
                limits=(0, 5),
                unit="",
                enums={
                    'Idle': 0,
                    'Gear_1': 1,
                    'Gear_2': 2,
                    'Gear_3': 3,
                    'Gear_4': 4,
                    'Gear_5': 5,
                },
            ),
        ],
```

(continues on next page)

(continued from previous page)

```
    ),
]


def create_database(name, filename):
    db = kvadblib.Dbc(name=name)

    for _msg in _messages:
        message = db.new_message(
            name=_msg.name,
            id=_msg.id,
            dlc=_msg.dlc,
        )

        for _sig in _msg.signals:
            if isinstance(_sig, EnumSignal):
                _type = kvadblib.SignalType.ENUM_UNSIGNED
                _enums = _sig.enums
            else:
                _type = kvadblib.SignalType.UNSIGNED
                _enums = {}
            message.new_signal(
                name=_sig.name,
                type=_type,
                byte_order=kvadblib.SignalByteOrder.INTEL,
                mode=kvadblib.SignalMultiplexMode.MUX_INDEPENDENT,
                size=kvadblib.ValueSize(*_sig.size),
                scaling=kvadblib.ValueScaling(*_sig.scaling),
                limits=kvadblib.ValueLimits(*_sig.limits),
                unit=_sig.unit,
                enums=_enums,
            )

    db.write_file(filename)
    db.close()


if __name__ == '__main__':
    parser = argparse.ArgumentParser(description="Create a database from scratch.")
    parser.add_argument('filename', help=("The filename to save the database to."))
    parser.add_argument(
        '-n',
        '--name',
        default='Engine example',
        help=("The name of the database (not the filename, the internal name."),
    )
    args = parser.parse_args()

    create_database(args.name, args.filename)
```

Description

While the name of the created database and the filename it is saved as is passed as arguments to `create_database`, the contents of the database is defined in the variable `_messages`. This is a list of `Message` namedtuples that describes all the messages to be put in the database:

- Their `name`, `id`, and `dlc` fields are passed to `canlib.kvadblib.Dbc.new_message`.
- Their `signals` attribute is a list of `Signal` or `EnumSignal` namedtuples. All their fields will be passed to `canlib.kvadblib.Dbc.new_signal`.

Sample Output

With the `_messages` variable as shown above, the following .dbc file is created:

```
VERSION "HIPBNYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY/4/%%%/4/'%**4YYY///"


NS_ :
     NS_DESC_
     CM_
     BA_DEF_
     BA_
     VAL_
     CAT_DEF_
     CAT_
     FILTER
     BA_DEF_DEF_
     EV_DATA_
     ENVVAR_DATA_
     SGTYPE_
     SGTYPE_VAL_
     BA_DEF_SGTYPE_
     BA_SGTYPE_
     SIG_TYPE_REF_
     VAL_TABLE_
     SIG_GROUP_
     SIG_VALTYPE_
     SIGTYPE_VALTYPE_
     BO_TX_BU_
     BA_DEF_REL_
     BA_REL_
     BA_DEF_DEF_REL_
     BU_SG_REL_
     BU_EV_REL_
     BU_BO_REL_
     SG_MUL_VAL_

BS_:

BU_:


BO_ 100 EngineData: 8 Vector__XXX
```

(continues on next page)

(continued from previous page)

```
 SG_ PetrolLevel : 24|8@1+ (1,0) [0|255] "l" Vector__XXX
 SG_ EngPower : 48|16@1+ (0.01,0) [0|150] "kW" Vector__XXX
 SG_ EngForce : 32|16@1+ (1,0) [0|0] "N" Vector__XXX
 SG_ IdleRunning : 23|1@1+ (1,0) [0|0] "" Vector__XXX
 SG_ EngTemp : 16|7@1+ (2,-50) [-50|150] "°C" Vector__XXX
 SG_ EngSpeed : 0|16@1+ (1,0) [0|8000] "rpm" Vector__XXX

BO_ 1020 GearBoxInfo: 1 Vector__XXX
 SG_ EcoMode : 6|2@1+ (1,0) [0|1] "" Vector__XXX
 SG_ ShiftRequest : 3|1@1+ (1,0) [0|0] "" Vector__XXX
 SG_ Gear : 0|3@1+ (1,0) [0|5] "" Vector__XXX


BA_DEF_  "BusType" STRING ;
BA_DEF_DEF_  "BusType" "";
BA_ "BusType" "CAN";
VAL_ 100 IdleRunning 0 "Running" 1 "Idle" ;
VAL_ 1020 ShiftRequest 1 "Shift_Request_On" 0 "Shift_Request_Off" ;
VAL_ 1020 Gear 0 "Idle" 2 "Gear_2" 1 "Gear_1" 5 "Gear_5" 3 "Gear_3" 4 "Gear_4" ;
```

1.5.3 Monitor a Channel Using a Database

```
"""dbmonitor.py -- Read CAN messages using a database

This script will use canlib.canlib and canlib.kvadblib to monitor a CAN
channel, and look up all messages received in a database before printing them.

It requires a CANlib channel with a connected device capable of receiving CAN
messages, some source of CAN messages, and the same database that the source is
using to generate the messages.

In this example the channel is opened with flag canOPEN_ACCEPT_LARGE_DLC (optional).
This enables a DLC larger than 8 bytes (up to 15 for classic CAN). If
canOPEN_ACCEPT_LARGE_DLC is excluded, generated frames with DLC > 8, will attain
a DLC of 8 on the receiving end, which may compromise the DLC equivalence
check.

The source of the messages may be e.g. the pinger.py example script.

"""
import argparse

from canlib import canlib, kvadblib

bitrates = {
    '1M': canlib.Bitrate.BITRATE_1M,
    '500K': canlib.Bitrate.BITRATE_500K,
    '250K': canlib.Bitrate.BITRATE_250K,
    '125K': canlib.Bitrate.BITRATE_125K,
    '100K': canlib.Bitrate.BITRATE_100K,
```

(continues on next page)

(continued from previous page)

```
    '62K': canlib.Bitrate.BITRATE_62K,
    '50K': canlib.Bitrate.BITRATE_50K,
    '83K': canlib.Bitrate.BITRATE_83K,
    '10K': canlib.Bitrate.BITRATE_10K,
}


def printframe(db, frame):
    try:
        bmsg = db.interpret(frame)
    except kvadblib.KvdNoMessage:
        print("<<< No message found for frame with id %s >>>" % frame.id)
        return

    if not bmsg._message.dlc == bmsg._frame.dlc:
        print(
            "<<< Could not interpret message because DLC does not match for frame with␣
↪id %s >>>"
            % frame.id
        )
        print("\t- DLC (database): %s" % bmsg._message.dlc)
        print("\t- DLC (received frame): %s" % bmsg._frame.dlc)
        return

    msg = bmsg._message

    print('', msg.name)

    if msg.comment:
        print('', '"%s"' % msg.comment)

    for bsig in bmsg:
        print('', bsig.name + ':', bsig.value, bsig.unit)

    print('')


def monitor_channel(channel_number, db_name, bitrate, ticktime):
    db = kvadblib.Dbc(filename=db_name)

    ch = canlib.openChannel(channel_number, canlib.canOPEN_ACCEPT_LARGE_DLC,␣
↪bitrate=bitrate)
    ch.setBusOutputControl(canlib.canDRIVER_NORMAL)
    ch.busOn()

    timeout = 0.5
    tick_countup = 0
    if ticktime <= 0:
        ticktime = None
    elif ticktime < timeout:
        timeout = ticktime
```

(continues on next page)

(continued from previous page)

```
    print("Listening...")
    while True:
        try:
            frame = ch.read(timeout=int(timeout * 1000))
            printframe(db, frame)
        except canlib.CanNoMsg:
            if ticktime is not None:
                tick_countup += timeout
                while tick_countup > ticktime:
                    print("tick")
                    tick_countup -= ticktime
        except KeyboardInterrupt:
            print("Stop.")
            break


if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description="Listen on a CAN channel and print all signals received, as␣
↪specified by a database."
    )
    parser.add_argument(
        'channel', type=int, default=1, nargs='?', help=("The channel to listen on.")
    )
    parser.add_argument(
        '--db',
        default="engine_example.dbc",
        help=("The database file to look up messages and signals in."),
    )
    parser.add_argument(
        '--bitrate', '-b', default='500k', help=("Bitrate, one of " + ', '.join(bitrates.
↪keys()))
    )
    parser.add_argument(
        '--ticktime',
        '-t',
        type=float,
        default=0,
        help=("If greater than zero, display 'tick' every this many seconds"),
    )
    args = parser.parse_args()

    monitor_channel(args.channel, args.db, bitrates[args.bitrate.upper()], args.ticktime)
```

Description

Any CAN messages received on the specified channel will be looked up in the database using `canlib.kvadblib.Dbc.interpret`, which allows the script to print the "phys" value of each signal instead of just printing the raw data (as *Monitor a Channel* does). The script also prints the message's name and comment (if available), as well as the signals name and unit.

Sample Output

```
EngineData
PetrolLevel: 0.0 l
EngPower: 12.0 kW
EngForce: 0.0 N
IdleRunning: 0.0
EngTemp: -30.0 °C
EngSpeed: 7735.0 rpm

GearBoxInfo
EcoMode: 0.0
ShiftRequest: 0.0
Gear: 0.0

EngineData
PetrolLevel: 0.0 l
EngPower: 28.0 kW
EngForce: 0.0 N
IdleRunning: 0.0
EngTemp: -30.0 °C
EngSpeed: 3467.0 rpm

GearBoxInfo
EcoMode: 1.0
ShiftRequest: 0.0
Gear: 0.0

EngineData
PetrolLevel: 0.0 l
EngPower: 0.0 kW
EngForce: 0.0 N
IdleRunning: 0.0
EngTemp: -50.0 °C
EngSpeed: 1639.0 rpm

GearBoxInfo
EcoMode: 1.0
ShiftRequest: 0.0
Gear: 1.0

EngineData
PetrolLevel: 60.0 l
EngPower: 0.0 kW
EngForce: 0.0 N
```

(continues on next page)

(continued from previous page)

```
 IdleRunning: 0.0
 EngTemp: 142.0 °C
 EngSpeed: 0.0 rpm

 GearBoxInfo
 EcoMode: 0.0
 ShiftRequest: 0.0
 Gear: 0.0

 EngineData
 PetrolLevel: 172.0 l
 EngPower: 51.0 kW
 EngForce: 0.0 N
 IdleRunning: 0.0
 EngTemp: -50.0 °C
 EngSpeed: 0.0 rpm

 GearBoxInfo
 EcoMode: 0.0
 ShiftRequest: 0.0
 Gear: 0.0
```

1.5.4 Examine the Contents of a Database

```
# examine_db.py
"""

This script uses canlib.kvadblib to parse a database and print its contents.

"""
import argparse
import sys

from canlib import kvadblib

INDENT = ' ' * 4


def print_db(db):
    print('DATABASE')
    print(db.name)
    for line in db_lines(db):
        print(INDENT + line)


def adef_lines(adef):
    yield 'type: ' + type(adef).__name__
    yield 'definition: ' + str(adef.definition)
    yield 'owner: ' + str(adef.owner)
```

(continues on next page)

(continued from previous page)

```
def attr_lines(attrib):
    yield str(attrib.name) + ' = ' + str(attrib.value)


def db_lines(db):
    yield 'flags: ' + str(db.flags)
    yield 'protocol: ' + str(db.protocol)
    yield ''

    yield 'ATTRIBUTE DEFINITIONS'
    for adef in db.attribute_definitions():
        yield str(adef.name)
        for line in adef_lines(adef):
            yield INDENT + line
    yield ''

    yield 'MESSAGES'
    for message in db:
        yield str(message.name)
        for line in msg_lines(message):
            yield INDENT + line
    yield ''


def enum_lines(enums):
    for name, val in enums.items():
        yield str(name) + ' = ' + str(val)


def msg_lines(message):
    yield 'id: ' + str(message.id)
    yield 'flags: ' + str(message.flags)
    yield 'dlc: ' + str(message.dlc)
    yield 'comment: ' + str(message.comment)
    yield ''

    yield 'ATTRIBUTES'
    for attr in message.attributes():
        for line in attr_lines(attr):
            yield line
    yield ''

    yield 'SIGNALS'
    for signal in message:
        yield str(signal.name)
        for line in sig_lines(signal):
            yield INDENT + line
    yield ''
```

(continues on next page)

(continued from previous page)

```
def sig_lines(signal):
    for name in ('type', 'byte_order', 'mode', 'size', 'scaling', 'limits', 'unit',
↪'comment'):
        yield name + ': ' + str(getattr(signal, name))
    yield ''

    try:
        enums = signal.enums
    except AttributeError:
        pass
    else:
        yield 'ENUMERATIONS'
        for line in enum_lines(enums):
            yield line
        yield ''

    yield 'ATTRIBUTES'
    for attr in signal.attributes():
        for line in attr_lines(attr):
            yield line
    yield ''


def examine_database(db_name):
    with kvadblib.Dbc(filename=db_name) as db:
        print_db(db)


if __name__ == '__main__':
    parser = argparse.ArgumentParser(description=sys.modules[__name__].__doc__)
    parser.add_argument(
        'db', type=str, metavar='<database.dbc>', help='The dbc database file to examine.
↪'
    )
    args = parser.parse_args()

    examine_database(args.db)
```

Description

The script is structured into several generator functions that take a `canlib.kvadblib` object and yield lines of information about it. This allows one function to add indentation to any other functions it uses.

Generally each function first yields information in the following order:

1. Any information about the object itself (e.g. `db.flags` and `db.protocol`)
2. An empty string
3. For each type of sub-object (e.g. attribute definitions):
    1. A heading (e.g. `'ATTRIBUTE_DEFINTIONS'`)
    2. For each object of that type (e.g. iterating through `canlib.kvadblib.Dbc.attribute_definitions`):

1. The objects name
2. All lines from the `*_lines` function for the object type (e.g. `adef_lines`), with added indentation

3. An empty string

Sample Output

Running this script on the database created by *Create a Database* gives the following:

```
DATABASE
engine_example
    flags: 0
    protocol: ProtocolType.CAN

    ATTRIBUTE DEFINITIONS
    BusType
        type: StringDefinition
        definition: DefaultDefinition(default='')
        owner: AttributeOwner.DB

    MESSAGES
    EngineData
        id: 100
        flags: MessageFlag.0
        dlc: 8
        comment:

        ATTRIBUTES

        SIGNALS
        PetrolLevel
            type: SignalType.UNSIGNED
            byte_order: SignalByteOrder.INTEL
            mode: -1
            size: ValueSize(startbit=24, length=8)
            scaling: ValueScaling(factor=1.0, offset=0.0)
            limits: ValueLimits(min=0.0, max=255.0)
            unit: l
            comment:

            ATTRIBUTES

        EngPower
            type: SignalType.UNSIGNED
            byte_order: SignalByteOrder.INTEL
            mode: -1
            size: ValueSize(startbit=48, length=16)
            scaling: ValueScaling(factor=0.01, offset=0.0)
            limits: ValueLimits(min=0.0, max=150.0)
            unit: kW
            comment:

            ATTRIBUTES
```

(continues on next page)

(continued from previous page)

```
        EngForce
            type: SignalType.UNSIGNED
            byte_order: SignalByteOrder.INTEL
            mode: -1
            size: ValueSize(startbit=32, length=16)
            scaling: ValueScaling(factor=1.0, offset=0.0)
            limits: ValueLimits(min=0.0, max=0.0)
            unit: N
            comment:

            ATTRIBUTES

        IdleRunning
            type: SignalType.UNSIGNED
            byte_order: SignalByteOrder.INTEL
            mode: 0
            size: ValueSize(startbit=23, length=1)
            scaling: ValueScaling(factor=1.0, offset=0.0)
            limits: ValueLimits(min=0.0, max=0.0)
            unit:
            comment:

            ENUMERATIONS
            Running = 0
            Idle = 1

            ATTRIBUTES

        EngTemp
            type: SignalType.UNSIGNED
            byte_order: SignalByteOrder.INTEL
            mode: -1
            size: ValueSize(startbit=16, length=7)
            scaling: ValueScaling(factor=2.0, offset=-50.0)
            limits: ValueLimits(min=-50.0, max=150.0)
            unit: °C
            comment:

            ATTRIBUTES

        EngSpeed
            type: SignalType.UNSIGNED
            byte_order: SignalByteOrder.INTEL
            mode: -1
            size: ValueSize(startbit=0, length=16)
            scaling: ValueScaling(factor=1.0, offset=0.0)
            limits: ValueLimits(min=0.0, max=8000.0)
            unit: rpm
            comment:

            ATTRIBUTES
```

(continues on next page)

(continued from previous page)

```
GearBoxInfo
    id: 1020
    flags: MessageFlag.0
    dlc: 1
    comment:

    ATTRIBUTES

    SIGNALS
    EcoMode
        type: SignalType.UNSIGNED
        byte_order: SignalByteOrder.INTEL
        mode: -1
        size: ValueSize(startbit=6, length=2)
        scaling: ValueScaling(factor=1.0, offset=0.0)
        limits: ValueLimits(min=0.0, max=1.0)
        unit:
        comment:

        ATTRIBUTES

    ShiftRequest
        type: SignalType.UNSIGNED
        byte_order: SignalByteOrder.INTEL
        mode: 0
        size: ValueSize(startbit=3, length=1)
        scaling: ValueScaling(factor=1.0, offset=0.0)
        limits: ValueLimits(min=0.0, max=0.0)
        unit:
        comment:

        ENUMERATIONS
        Shift_Request_On = 1
        Shift_Request_Off = 0

        ATTRIBUTES

    Gear
        type: SignalType.UNSIGNED
        byte_order: SignalByteOrder.INTEL
        mode: 0
        size: ValueSize(startbit=0, length=3)
        scaling: ValueScaling(factor=1.0, offset=0.0)
        limits: ValueLimits(min=0.0, max=5.0)
        unit:
        comment:

        ENUMERATIONS
        Gear_5 = 5
        Gear_1 = 1
```

(continues on next page)

(continued from previous page)

```
            Gear_3 = 3
            Idle = 0
            Gear_4 = 4
            Gear_2 = 2

            ATTRIBUTES
```

1.5.5 Print Information About All Channels

```
"""list_channels.py -- List all CANlib channel

This script uses canlib.canlib to list all CANlib channels and information
about the device that is using them.

"""
import argparse

from canlib import canlib


def print_channels():
    for ch in range(canlib.getNumberOfChannels()):
        chdata = canlib.ChannelData(ch)
        print(
            "{ch}. {name} ({ean} / {serial})".format(
                ch=ch,
                name=chdata.channel_name,
                ean=chdata.card_upc_no,
                serial=chdata.card_serial_no,
            )
        )


if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description="List all CANlib channels and information about them."
    )
    args = parser.parse_args()

    print_channels()
```

Sample Output

```
0. Kvaser Memorator Pro 2xHS v2 (channel 0) (73-30130-00819-9 / 10626)
1. Kvaser Memorator Pro 2xHS v2 (channel 1) (73-30130-00819-9 / 10626)
```

1.5.6 Monitor a Channel

```
"""monitor.py -- Print all data received on a CAN channel

This script uses canlib.canlib to listen on a channel and print all data
received.

It requires a CANlib channel with a connected device capable of receiving CAN
messages and some source of CAN messages.

The source of the messages may be e.g. the pinger.py example script.

Also see the dbmonitor.py example script for how to look up the messages
received in a database.

"""
import argparse
import shutil

from canlib import canlib

bitrates = {
    '1M': canlib.Bitrate.BITRATE_1M,
    '500K': canlib.Bitrate.BITRATE_500K,
    '250K': canlib.Bitrate.BITRATE_250K,
    '125K': canlib.Bitrate.BITRATE_125K,
    '100K': canlib.Bitrate.BITRATE_100K,
    '62K': canlib.Bitrate.BITRATE_62K,
    '50K': canlib.Bitrate.BITRATE_50K,
    '83K': canlib.Bitrate.BITRATE_83K,
    '10K': canlib.Bitrate.BITRATE_10K,
}


def printframe(frame, width):
    form = '^' + str(width - 1)
    print(format(" Frame received ", form))
    print("id:", frame.id)
    print("data:", bytes(frame.data))
    print("dlc:", frame.dlc)
    print("flags:", frame.flags)
    print("timestamp:", frame.timestamp)


def monitor_channel(channel_number, bitrate, ticktime):
    ch = canlib.openChannel(channel_number, bitrate=bitrate)
```

(continues on next page)

(continued from previous page)

```
    ch.setBusOutputControl(canlib.canDRIVER_NORMAL)
    ch.busOn()

    width, height = shutil.get_terminal_size((80, 20))

    timeout = 0.5
    tick_countup = 0
    if ticktime <= 0:
        ticktime = None
    elif ticktime < timeout:
        timeout = ticktime

    print("Listening...")
    while True:
        try:
            frame = ch.read(timeout=int(timeout * 1000))
            printframe(frame, width)
        except canlib.CanNoMsg:
            if ticktime is not None:
                tick_countup += timeout
                while tick_countup > ticktime:
                    print("tick")
                    tick_countup -= ticktime
        except KeyboardInterrupt:
            print("Stop.")
            break

    ch.busOff()
    ch.close()


if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description="Listen on a CAN channel and print all frames received."
    )
    parser.add_argument('channel', type=int, default=1, nargs='?')
    parser.add_argument(
        '--bitrate', '-b', default='500k', help=("Bitrate, one of " + ', '.join(bitrates.
↪keys()))
    )
    parser.add_argument(
        '--ticktime',
        '-t',
        type=float,
        default=0,
        help=("If greater than zero, display 'tick' every this many seconds"),
    )
    parser.add_argument
    args = parser.parse_args()

    monitor_channel(args.channel, bitrates[args.bitrate.upper()], args.ticktime)
```

Description

Any CAN frames received on the specified channel will be printed. Note that the signals contained in the frame is not be extracted, only the raw data is printed. To extract the signals, see *Monitor a Channel Using a Database*.

Sample Output

```
Listening...
 Frame received
id: 1020
data: b'\x00'
flags: MessageFlag.STD
timestamp: 939214
 Frame received
id: 100
data: b'\xd1\x06\x00\x19\x00\x00\x18\x15'
flags: MessageFlag.STD
timestamp: 939215
 Frame received
id: 1020
data: b'\x01'
flags: MessageFlag.STD
timestamp: 939417
 Frame received
id: 100
data: b'\x00\x00\x00\x19\x00\x00\xc82'
flags: MessageFlag.STD
timestamp: 939418
 Frame received
id: 1020
data: b'\x00'
flags: MessageFlag.STD
timestamp: 939620
 Frame received
id: 100
data: b'\x00\x00\x00\x00\x00\x00\x903'
flags: MessageFlag.STD
timestamp: 939621
 Frame received
id: 1020
data: b'@'
flags: MessageFlag.STD
timestamp: 939823
 Frame received
id: 100
data: b')\x03\x17\xf5\x00\x00\x00\x00'
flags: MessageFlag.STD
timestamp: 939824
 Frame received
id: 1020
data: b'\x02'
flags: MessageFlag.STD
```

(continues on next page)

(continued from previous page)

```
timestamp: 940026
 Frame received
id: 100
data: b'\x1b\x0eC\x00\x00\x00\x00\x00'
flags: MessageFlag.STD
timestamp: 940027
```

1.5.7 Send Random Messages on a Channel

```
"""pinger.py -- Send random CAN messages based on a database

This script uses canlib.canlib and canlib.kvadblib to send random messages from
a database with random data.

It requires a CANlib channel a connected device capable of sending CAN
messages, something that receives those messages, and a database to inspect for
the messages to send.

Messages can be received and printed by e.g. the dbmonitor.py example script.

"""
import argparse
import random
import time

from canlib import canlib, kvadblib

bitrates = {
    '1M': canlib.Bitrate.BITRATE_1M,
    '500K': canlib.Bitrate.BITRATE_500K,
    '250K': canlib.Bitrate.BITRATE_250K,
    '125K': canlib.Bitrate.BITRATE_125K,
    '100K': canlib.Bitrate.BITRATE_100K,
    '62K': canlib.Bitrate.BITRATE_62K,
    '50K': canlib.Bitrate.BITRATE_50K,
    '83K': canlib.Bitrate.BITRATE_83K,
    '10K': canlib.Bitrate.BITRATE_10K,
}


def set_random_framebox_signal(db, framebox, signals):
    sig = random.choice(signals)
    value = get_random_value(db, sig)
    framebox.signal(sig.name).phys = value


def get_random_value(db, sig):
    limits = sig.limits
    value = random.uniform(limits.min, limits.max)
```

(continues on next page)

(continued from previous page)

```
    # round value depending on type...
    if sig.type is kvadblib.SignalType.UNSIGNED or sig.type is kvadblib.SignalType.
↪SIGNED:
        # ...remove decimals if the signal was of type unsigned
        value = int(round(value))
    else:
        # ...otherwise, round to get only one decimal
        value = round(value, 1)

    return value


def ping_loop(channel_number, db_name, num_messages, quantity, interval, bitrate,␣
↪seed=0):
    db = kvadblib.Dbc(filename=db_name)

    ch = canlib.openChannel(channel_number, bitrate=bitrate)
    ch.setBusOutputControl(canlib.canDRIVER_NORMAL)
    ch.busOn()

    random.seed(seed)

    if num_messages == -1:
        used_messages = list(db)
    else:
        used_messages = random.sample(list(db), num_messages)

    print()
    print("Randomly selecting signals from the following messages:")
    print(used_messages)
    print("Seed used was " + repr(seed))
    print()

    while True:
        # Create an empty framebox each time, ignoring previously set signal
        # values.
        framebox = kvadblib.FrameBox(db)

        # Add all messages to the framebox, as we may use send any signal from
        # any of them.
        for msg in db:
            framebox.add_message(msg.name)

        # Make a list of all signals (which framebox has found in all messages
        # we gave it), so that set_random_framebox_signal() can pick a random
        # one.
        signals = [bsig.signal for bsig in framebox.signals()]

        # Set some random signals to random values
        for i in range(quantity):
            set_random_framebox_signal(db, framebox, signals)
```

(continues on next page)

(continued from previous page)

```
        # Send all messages/frames
        for frame in framebox.frames():
            print('Sending frame', frame)
            ch.writeWait(frame, timeout=5000)

        time.sleep(interval)


if __name__ == '__main__':
    parser = argparse.ArgumentParser(description="Send random CAN message based on a␣
↪database.")
    parser.add_argument(
        'channel', type=int, default=0, nargs='?', help=("The channel to send messages␣
↪on.")
    )
    parser.add_argument(
        '--bitrate', '-b', default='500k', help=("Bitrate, one of " + ', '.join(bitrates.
↪keys()))
    )
    parser.add_argument(
        '--db', default="engine_example.dbc", help=("The database file to base messages␣
↪on.")
    )
    parser.add_argument(
        '-Q', '--quantity', type=int, default=5, help=("The number of signals to send␣
↪each tick.")
    )
    parser.add_argument(
        '-I', '--interval', type=float, default=0.2, help=("The time, in seconds,␣
↪between ticks.")
    )
    parser.add_argument(
        '-n',
        '--num-messages',
        type=int,
        default=-1,
        help=("The number of message from the database to use, or -1 to use all."),
    )
    parser.add_argument(
        '-s',
        '--seed',
        nargs='?',
        default='0',
        help=(
            "The seed used for choosing messages. If possible, will be converted to an␣
↪int. If no argument is given, a random seed will be used."
        ),
    )
    args = parser.parse_args()

    if args.seed is None:
        seed = None
```

(continues on next page)

(continued from previous page)

```
    else:
        try:
            seed = int(args.seed)
        except ValueError:
            seed = args.seed

    ping_loop(
        channel_number=args.channel,
        db_name=args.db,
        num_messages=args.num_messages,
        quantity=args.quantity,
        interval=args.interval,
        bitrate=bitrates[args.bitrate.upper()],
        seed=args.seed,
    )
```

Description

Note: There must be some process reading the messages for `pinger.py` to work (see e.g. *Monitor a Channel Using a Database*).

`ping_loop` will first extract a random list of messages (see *Randomness*), and then enter a loop that creates a new `canlib.kvadblib.FrameBox` before adding some random signals with random values (the quantity specified by the `quantity`/`--quantity` argument).

Adding random signals is done with `set_random_framebox_signal`, which picks a random signal from the frame-box, and `get_random_value` which inspects the given signal and provides a random value based on the signal's definition.

Finally, the loop pauses for `interval`/`--interval` seconds between sending messages.

Randomness

The random selection of messages is done with the `seed`/`--seed` and `num_messages`/`num-messages` arguments. If `num_messages` is `-1`, all messages from the database will be used. Otherwise, `num_message` specifies the number of messages to be randomly picked from the database.

The `seed` argument will be sent to `random.seed` before the messages are selected (which is done with `random.sample`), which means as long as the seed remains the same, the same messages are selected. The `seed` can also be set to `None` for a pseudo-random seed.

Sample Output

```
Randomly selecting signals from the following messages:
[Message(name='EngineData', id=100, flags=<MessageFlag.0: 0>, dlc=8, comment=''),␣
↪Message(name='GearBoxInfo', id=1020, flags=<MessageFlag.0: 0>, dlc=1, comment='')]
Seed used was '0'

Sending frame Frame(id=1020, data=bytearray(b'\x00'), dlc=1, flags=<MessageFlag.0: 0>,␣
↪timestamp=None)
Sending frame Frame(id=100, data=bytearray(b'\x00\x00\x16]\x00\x00\x00\x00'), dlc=8,␣
↪flags=<MessageFlag.0: 0>, timestamp=None)
Sending frame Frame(id=1020, data=bytearray(b'\x00'), dlc=1, flags=<MessageFlag.0: 0>,␣
↪timestamp=None)
Sending frame Frame(id=100, data=bytearray(b'\x00\x00\x00\xdd\x00\x00\x00\x00'), dlc=8,␣
↪flags=<MessageFlag.0: 0>, timestamp=None)
Sending frame Frame(id=1020, data=bytearray(b'\x00'), dlc=1, flags=<MessageFlag.0: 0>,␣
↪timestamp=None)
Sending frame Frame(id=100, data=bytearray(b'\x00\x00\x00\xe0\x00\x00`\t'), dlc=8, flags=
↪<MessageFlag.0: 0>, timestamp=None)
Sending frame Frame(id=1020, data=bytearray(b'\x04'), dlc=1, flags=<MessageFlag.0: 0>,␣
↪timestamp=None)
Sending frame Frame(id=100, data=bytearray(b'f\x07\n\x00\x00\x00\x00\x00'), dlc=8, flags=
↪<MessageFlag.0: 0>, timestamp=None)
Sending frame Frame(id=1020, data=bytearray(b'\x00'), dlc=1, flags=<MessageFlag.0: 0>,␣
↪timestamp=None)
Sending frame Frame(id=100, data=bytearray(b'\x0c\x15-\x00\x00\x00\x00\x00'), dlc=8,␣
↪flags=<MessageFlag.0: 0>, timestamp=None)
```

1.5.8 Validate a Memorator Configuration

```
"""validate_memo_config.py -- validate a Memorator configuration

This script uses canlib.kvamemolibxml to load and validate a Memorator
configuration in an xml file, and then prints any errors and warnings.

It requires a Memorator configuration in xml format.

"""
import argparse

from canlib import kvamemolibxml


def validate(filename):
    # Read in the XML configuration file
    config = kvamemolibxml.load_xml_file(filename)

    # Validate the XML configuration
    errors, warnings = config.validate()

    # Print errors and warnings
```

(continues on next page)

(continued from previous page)

```
    for error in errors:
        print(error)
    for warning in warnings:
        print(warning)

    if errors or warnings:
        raise Exception("Please fix validation errors/warnings.")
    else:
        print("No errors found!")


if __name__ == '__main__':
    parser = argparse.ArgumentParser(description="Validate a Memorator configuration.")
    parser.add_argument(
        'filename', default='logall.xml', nargs='?', help=("The filename of the␣
↪configuration.")
    )
    args = parser.parse_args()

    validate(args.filename)
```

1.5.9 Write a Configuration to a Memorator

```
"""write_memo_config.py -- Write a configuration to a memorator

This example script uses canlib.kvamemolibxml and canlib.kvmlib to load a
configuration file in .xml format, validate it, and then write it to a
connected Memorator.

It requires a configuration xml file and a connected Memorator device.

"""
import argparse

from canlib import kvamemolibxml, kvmlib


def write_config(filename, channel_number):
    # Read in the XML configuration file
    config = kvamemolibxml.load_xml_file(filename)

    # Validate the XML configuration
    errors, warnings = config.validate()

    if errors or warnings:
        raise Exception("Errors or warnings found! Check validate_memo_config example.")

    # Open the device and write the configuration
    with kvmlib.openDevice(channel_number) as memo:
        memo.write_config(config.lif)
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description="Write a configuration to a Memorator.")
    parser.add_argument(
        'filename', default='logall.xml', nargs='?', help=("The filename of the␣
↪configuration.")
    )
    parser.add_argument(
        'channel',
        type=int,
        default=0,
        nargs='?',
        help=("The channel number of the device the configuration should be written to.
↪"),
    )
    args = parser.parse_args()

    write_config(args.filename, args.channel)
```

1.6 Reference

1.6.1 Base Exceptions

CanlibException

exception canlib.**CanlibException**
Base class for all exceptions in canlib

DllException

exception canlib.**DllException**
Bases: *canlib.exceptions.CanlibException*

Base class for exceptions from dll calls in canlib

All instances of this class must have a `status` attribute defined (this is enforced in the constructor). Its value is the status code that caused the exception.

property canERR
Deprecated name for `status`

1.6.2 EAN

class canlib.**EAN**(*source*)

Helper object for dealing with European Article Numbers

Depending on the format the ean is in, *EAN* objects are created in different ways;

For strings:

```
EAN('73-30130-01234-5')
```

For integers:

```
EAN(7330130012345)
```

For iterables of integers:

```
EAN([7, 3, 3, 0, 1, 3, 0, 0, 1, 2, 3, 4, 5])
```

For BCD-coded bytes or bytearrays (str in python 2):

```
EAN.from_bcd(b'\x45\x23\x01\x30\x01\x33\x07')
```

For "hi-lo" format, i.e. two 32-bit integers containing half the ean each, both BCD-coded:

```
EAN.from_hilo([eanHi, eanLo])
```

The various representations can then be produced from the resulting object:

```
>>> str(ean)
'73-30130-01234-5'
>>> int(ean)
7330130012345
>>> tuple(ean)  # or list(), or any other sequence type
(7, 3, 3, 0, 1, 3, 0, 0, 1, 2, 3, 4, 5)
>>> ean.bcd()
b'E#\x010\x013\x07'
>>> ean.hilo()
(805380933, 471809)
```

Sometimes it is easier to only use the last six digits of the ean, the product code and check digit. This is supported when working with string representations; the constructor supports six-digit (seven-character) input:

```
EAN('01234-5')
```

In that cases, the country and manufacturer code is assumed to be that of Kvaser AB (73-30130).

A string containing only the product code and check digit can also be retrieved:

```
ean.product()
```

Instances can also be indexed which yields specific digits as integers:

```
>>> ean[7]
0
>>> ean[7:]
(0, 1, 2, 3, 4, 5)
```

Note: The byteorder is currently always assumed to be 'little'.

bcd()
Return a binary-coded bytes object with this EAN

fmt = '##-#####-#####-#'

classmethod from_bcd(*bcd_bytes*)
Create an EAN object from a binary coded bytes-like object

The EAN is automatically shortened to the correct length.

classmethod from_hilo(*hilo*)
Create an EAN object from a pair of 32-bit integers, (eanHi, eanLo)

classmethod from_string(*ean_string*)
Create an EAN object from a specially formatted string

Deprecated since version 1.6: Use the constructor, `EAN(ean_string)`, instead.

hilo()
Return a pair of 32-bit integers, (eanHi, eanLo), with this EAN

num_digits = 13

product()
Return only the product code and check digit of the string representation

1.6.3 Device

class canlib.**Device**(*ean*, *serial*)
Class for keeping track of a physical device

This class represents a physical device regardless of whether it is currently connected or not, and on which channel it is connected.

If the device is currently connected, *Device.find* can be used to get a *Device* object:

```
dev = Device.find(ean=EAN.from_string('67890-1'))
```

Device.find supports searching for devices based on a variety of information, and will return a *Device* object corresponding to the first physical device that matched its arguments. In this case, it would be the first device found with an EAN of 73-30130-67890-1.

If the device wanted is not currently connected, *Device* objects can be created with their EAN and serial number (as this is the minimal information needed to uniquely identify a specific device):

```
dev = Device(ean=EAN.from_string('67890-1'), serial=42)
```

Two *Device* objects can be checked for equality (whether they refer to the same device) and be converted to a `str`. *Device.probe_info* can also be used for a more verbose string representation that queries the device (if connected) for various pieces of information.

This class also provides functions for creating the other objects of `canlib`:

- `canlib.Channel` – *Device.channel*
- `canlib.ChannelData` – *Device.channel_data*
- `canlib.IOControl` – *Device.iocontrol*

- kvmlib.Memorator– *Device.memorator*
- linlib.Channel – *Device.lin_master* and *Device.lin_slave*

Variables

- **Device.ean** (*canlib.EAN*) – The EAN of this device.
- **Device.serial** (int) – The serial number of this device.
- **last_channel_number** (int) – The channel number this device was last found on (used as an optimization; while the device stays on the same CANlib channel there is no need for a linear search of all channels).

New in version 1.6.

channel(**args*, ***kwargs*)
A canlib.Channel for this device's first channel

The experimental attribute _chan_no_on_card may be given, the int provided will be added (without any verifications) to the *channel_number* where this device was found on, and may thus be used to open a specific local channel on this device.

Note: When using the _chan_no_on_card attribute, you must make sure that the card actually have the assumed number of local channels. Using this argument with a too large int could return a channel belonging to a different device.

Arguments to canlib.openChannel other than the channel number can be passed to this function.

Changed in version 1.13: Added attribute _chan_no_on_card

Deprecated since version 1.16: Use *open_channel* instead

channel_data()
A canlib.ChannelData for this device's first channel

channel_number()
An int with this device's CANlib channel number

ean

classmethod find(*channel_number=None*, *ean=None*, *serial=None*, *channel_name=None*)
Searches for a specific device

Goes through all CANlib channels (from zero and up), until one of them matches the given arguments. If an argument is omitted or None, any device will match it. If no devices matches a canlib.CanNotFound exception will be raised.

Parameters

- **channel_number** (int) – Find a device on this CANlib channel (number).
- **ean** (*canlib.EAN*) – Find a device with this EAN.
- **serial** (int) – Find a device with this serial number.
- **channel_name** (str) – Find a device with this CANlib channel name.

iocontrol()
A canlib.IOControl for this device's first channel

isconnected()
A bool whether this device can currently be found

issubset(*other*)
Check if device is a subset of other Device.

This can be used to see if a found device fulfills criteria specified in other. Setting an attribute to None is regarded as a Any. This means that e.g. any serial number will be a subset of a serial number specified as None.

New in version 1.9.

last_channel_number

lin_master(**args*, ***kwargs*)
A `linlib.Channel` master for this device's first channel

Arguments to `linlib.openMaster` other than the channel number can be passed to this function.

lin_slave(**args*, ***kwargs*)
A `linlib.Channel` slave for this device's first channel

Arguments to `linlib.openSlave` other than the channel number can be passed to this function.

memorator(**args*, ***kwargs*)
A `canlib.kvmlib.Memorator` for this device's first channel

Arguments to `canlib.openChannel` other than the channel number can be passed to this function.

open_channel(*chan_no_on_card=0*, ***kwargs*)
A `canlib.Channel` for this device's first channel

The parameter `chan_no_on_card` will be added (without any verifications) to the *channel_number* where this device was found on, and may thus be used to open a specific local channel on this device.

Note: When using the `chan_no_on_card` parameter, you must make sure that the card actually have the assumed number of local channels. Using this parameter with a too large `int` could return a channel belonging to a different device.

Arguments to `canlib.open_channel`, other than the channel number, can be passed to this function, but must be passed as keyword arguments.

New in version 1.16.

probe_info()
A `str` with information about this device

This function is useful when the *Device*'s `str()` does not give enough information while debugging. When the device is connected various pieces of information such as the device name, firmware, and driver name is given. When the device is not connected only basic information can be given.

Note: Never inspect the return value of this function, only use it for displaying information. Exactly what is shown depends on whether the device is connected or not, and is not guaranteed to stay consistent between versions.

---

remote(**args*, ***kwargs*)
A `canlib.kvrlib.RemoteDevice` for this device

Arguments to `canlib.kvrlib.openDevice` other than the channel number can be passed to this function.

serial

connected_devices()

canlib.**connected_devices**()
Get all currently connected devices as *Device*s

Returns an iterator of *Device* object, one object for every physical device currently connected.

New in version 1.6.

Changed in version 1.7: *Device.last_channel_number* will now be set.

1.6.4 Frames

Frame

class canlib.**Frame**(*id_*, *data*, *dlc=None*, *flags=0*, *timestamp=None*)
Represents a CAN message

Parameters

- **id_** – Message id
- **data** – Message data, will pad zero to match dlc (if dlc is given)
- **dlc** – Message dlc, default is calculated from number of data
- **flags** (`canlib.MessageFlag`) – Message flags, default is 0
- **timestamp** – Optional timestamp

data

dlc

flags

id

timestamp

LINFrame

class canlib.**LINFrame**(**args*, ***kwargs*)
Bases: *canlib.frame.Frame*

Represents a LIN message

A *Frame* that also has an *info* attribute, which is a `linlib.MessageInfo` or `None`. This attribute is initialized via the *info* keyword-only argument to the constructor.

data

dlc

flags

id

info

timestamp

1.6.5 Version Numbers

VersionNumber

class canlib.**VersionNumber**(*major*, *minor=None*, *build=None*, *release=None*)

A tuple-subclass representing a version number

Version numbers can be created with from one to three positional arguments, representing the major, minor, and build number respectively:

```
v1 = VersionNumber(1)
v12 = VersionNumber(1, 2)
v123 = VersionNumber(1, 2, 3)
```

Keyword arguments can also be used:

```
v1 = VersionNumber(major=1)
v12 = VersionNumber(major=1, minor=2)
v123 = VersionNumber(major=1, minor=2, build=3)
```

A fourth number, the release number, can also be given as a keyword-only argument:

```
v1293 = VersionNumber(major=1, minor=2, release=9, build=3)
```

This release number is placed between the minor and build numbers, both for the string representation and in the tuple.

The major number is required and the other numbers are optional in the order minor, build, release.

All numbers can be accessed as attributes (*major*, *minor*, *release*, *build*). If the number is unavailable, accessing the attribute returns `None`.

property beta

property build

property major

property minor

property release

BetaVersionNumber

class canlib.**BetaVersionNumber**(*major*, *minor=None*, *build=None*, *release=None*)

A tuple-subclass representing a beta (preview) version number

A *VersionNumber* that also has the attribute *beta* set to true.

New in version 1.6.

property beta

1.6.6 canlib

Exceptions

CanError

CanNoMsg

CanNotFound

CanScriptFail

EnvvarException

EnvvarNameError

EnvvarValueError

IoNoValidConfiguration

IoPinConfigurationNotConfirmed

TxeFileIsEncrypted

Bus Parameters

calc_bitrate()

canlib.canlib.busparams.**calc_bitrate**(*target_bitrate*, *clk_freq*)

Calculate nearest available bitrate

Parameters

- **target_bitrate** (int) – Wanted bitrate (bit/s)
- **clk_freq** (int) – Device clock frequency (Hz)

Returns

The returned tuple is a `(bitrate, tq)` named tuple of –

1. `bitrate` (int): Available bitrate, could be a rounded value (bit/s)
2. `tq` (int): Number of time quanta in one bit

New in version 1.16.

calc_busparamstq()

canlib.canlib.busparams.**calc_busparamstq**(*target_bitrate*, *target_sample_point*, *target_sync_jump_width*, *clk_freq*, *target_prop_tq=None*, *prescaler=1*)

Calculate closest matching busparameters.

The device clock frequency, `clk_freq`, can be obtained via *ClockInfo.frequency()*:

```
>>> chd = canlib.ChannelData(channel_number=0)
>>> clock_info = chd.clock_info
>>> clock_info.frequency()
80000000.0
```

Now call *calc_busparamstq* with target values, and a *BusParamsTq* object will be returned:

```
>>> params = calc_busparamstq(
... target_bitrate=470_000,
... target_sample_point=82,
... target_sync_jump_width=15.3,
... clk_freq=clock_info.frequency())
>>> params
BusParamsTq(tq=170, prop=107, phase1=31, phase2=31, sjw=26, prescaler=1)
```

A target number of time quanta in the propagation segment can also be specified by the user.

The returned BusParamsTq may not be valid on all devices. If `Error.NOT_IMPLEMENTED` is encountered when trying to set the bitrate with the returned BusParamsTq, provide a prescaler argument higher than one and retry. This will lower the total number of time quanta in the bit and thus make the BusParamsTq valid.

Example

```
>>> params = calc_busparamstq(
... target_bitrate=470_000,
... target_sample_point=82,
... target_sync_jump_width=15.3,
... clk_freq=clock_info.frequency(),
... target_prop_tq=50,
... prescaler=2)
>>> params
BusParamsTq(tq=85, prop=25, phase1=44, phase2=15, sjw=13, prescaler=2)
```

Note:

- Minimum sjw returned is 1, maximum sjw is min(phase1, phase2).

Parameters

- **target_bitrate** (`float`) – Wanted bitrate (bit/s)
- **target_sample_point** (`float`) – Wanted sample point in percentage (0-100)
- **target_sync_jump_width** (`float`) – Wanted sync jump width in percentage (0-100)
- **clk_freq** (`float`) – Device clock frequency (Hz)

- **target_prop_tq** (int, Optional) – Wanted propagation segment (time quanta)
- **prescaler** (int, Optional) – Wanted prescaler (at most 2 for CAN FD)

Returns *BusParamsTq* – Calculated bus parameters

New in version 1.16.

Changed in version 1.17.

to_BusParamsTq()

canlib.canlib.busparams.**to_BusParamsTq**(*clk_freq*, *bus_param*, *prescaler=1*, *data=False*)

Convert *BitrateSetting* or tuple to *BusParamsTq*.

The device clock frequency, clk_freq, can be obtained via *ClockInfo.frequency()*:

```
>>> chd = canlib.ChannelData(channel_number=0)
>>> clock_info = chd.clock_info
>>> clock_info.frequency()
80000000.0
```

Parameters

- **clk_freq** (float) – Clock frequency of device.
- **bus_param** (*BitrateSetting* or tuple) – *BitrateSetting* object or
- **(freq, tseg1, tseg2, sjw) `tuple` to convert.**
- **prescaler** (int) – The prescaler to use in the created *BusParamsTq*
- **object.**
- **data** (bool) – Set to True if the resulting *BusParamsTq* should be
- **used for CAN FD data bitrate parameters.**

Returns *BusParamsTq* object with equivalent settings as the input argument.

New in version 1.17.

to_BitrateSetting()

canlib.canlib.busparams.**to_BitrateSetting**(*clk_freq*, *bus_param*)

Convert *BusParamsTq* to *BitrateSetting*.

The device clock frequency, clk_freq, can be obtained via *ClockInfo.frequency()*:

```
>>> chd = canlib.ChannelData(channel_number=0)
>>> clock_info = chd.clock_info
>>> clock_info.frequency()
80000000.0
```

Parameters

- **clk_freq** (float) – Clock frequency of device.
- **bus_param** (*BusParamsTq*) – *BusParamsTq* object to convert.

Returns *BitrateSetting* object with equivalent settings as the input argument.

New in version 1.17.

ClockInfo

class canlib.canlib.busparams.**ClockInfo**(*numerator*, *denominator*, *power_of_ten*, *accuracy*)

Information about clock a oscillator

The clock frequency is set in the form:

frequency = numerator / denominator * 10 ** power_of_ten +/- accuracy

New in version 1.16.

frequency()

Returns an approximation of the clock frequency as a float.

BusParamsTq

class canlib.canlib.busparams.**BusParamsTq**(*tq*, *phase1*, *phase2*, *sjw*, *prescaler=1*, *prop=None*)

Holds parameters for busparameters in number of time quanta.

If you don't want to specify the busparameters in time quanta directly, you may use *calc_busparamstq* which returns an object of this class.

```
>>> params = calc_busparamstq(
... target_bitrate=470_000,
... target_sample_point=82,
... target_sync_jump_width=33.5,
... clk_freq=clk_freq)
>>> params
BusParamsTq(tq=170, prop=107, phase1=31, phase2=31, sjw=57, prescaler=1)
```

You may now query for the actual Sample Point and Sync Jump Width expressed as percentages of total bit time quanta:

```
>>> params.sample_point()
81.76470588235294
```

```
>>> params.sync_jump_width()
33.52941176470588
```

If you supply the clock frequency, you may also calculate the corresponding bitrate:

```
>>> params.bitrate(clk_freq=80_000_000)
470588.23529411765
```

Parameters

- **tq** (int) – Number of time quanta in one bit.
- **phase1** (int) – Number of time quanta in Phase Segment 1.
- **phase2** (int) – Number of time quanta in Phase Segment 2.

- **sjw** (int) – Number of time quanta in Sync Jump Width.
- **prescaler** (int) – Prescaler value (1-2 to enable auto in CAN FD)
- **prop** (int, optional) – Number of time quanta in Propagation Segment.

New in version 1.16.

bitrate(*clk_freq*)
Return bitrate assuming the given clock frequency

Parameters **clk_freq** (int) – Clock frequency (in Hz)

sample_point()
Return sample point in percentage.

sample_point_ns(*clk_freq*)
Return sample point in ns.

New in version 1.17.

sync_jump_width()
Return sync jump width (SJW) in percentage.

BusParamHelper

BusParamHelperLegacy

BitrateSetting

class canlib.canlib.busparams.**BitrateSetting**(*freq*, *tseg1*, *tseg2*, *sjw*, *nosamp=1*, *syncMode=0*)
Class that holds bitrate setting.

Parameters

- **freq** – Bitrate in bit/s.
- **tseg1** – Number of quanta from (but not including) the Sync Segment to the sampling point.
- **tseg2** – Number of quanta from the sampling point to the end of the bit.
- **sjw** – The Synchronization Jump Width.
- **nosamp** – The number of sampling points, only 1 is supported.
- **syncMode** – Unsupported and ignored.

New in version 1.17.

classmethod from_predefined(*bitrate*)
Create a BitrateSetting object using one of the Bitrate or BitrateFD enumerations.

Channel

openChannel()

ErrorCounters

Channel

CanBusStatistics

class canlib.canlib.structures.**CanBusStatistics**

Result from reading bus statistics using `canlib.canlib.Channel.get_bus_statistics`.

Variables

- **busLoad** (`int`) – The bus load, expressed as an integer in the interval 0 - 10000 representing 0.00% - 100.00% bus load.
- **errFrame** (`int`) – Number of error frames.
- **extData** (`int`) – Number of received extended (29-bit identifiers) data frames.
- **extRemote** (`int`) – Number of received extended (29-bit identifiers) remote frames.
- **overruns** (`int`) – The number of overruns detected by the hardware, firmware or driver.
- **stdData** (`int`) – Number of received standard (11-bit identifiers) data frames.
- **stdRemote** (`int`) – Number of received standard (11-bit identifiers) remote frames.

ChannelData

EnvVar

Enumerations

AcceptFilterFlag

Bitrate

BitrateFD

BusTypeGroup

ChannelCap

ChannelDataItem

ChannelFlags

DeviceMode

Driver

DriverCap

EnvVarType

Error

HardwareType

IOControlItem

LEDAction

LoggerType

MessageFlag

Notify

Open

OperationalMode

RemoteType

ScriptRequest

ScriptStatus

ScriptStop

Stat

TransceiverType

TxeDataItem

IOControl

I/O pin

AddonModule

AnalogIn

unloadLibrary()

1.6.7 kvadblib

Exceptions

KvdError

KvdBufferTooSmall

KvdDbFileParse

KvdErrInParameter

KvdInUse

KvdNoAttribute

KvdNoMessage

KvdNoNode

KvdNotFound

KvdOnlyOneAllowed

KvdInUse

SignalNotFound

Attribute

Attribute Definitions

AttributeDefinition

DefaultDefinition

EnumDefaultDefinition

EnumDefinition

FloatDefinition

IntegerDefinition

ValueScaling

ValueSize

Miscellaneous

bytes_to_dlc()

dlc_to_bytes()

dllversion()

get_protocol_properties()

1.6.8 kvamemolibxml

Exceptions

KvaError

Confiurations

Configuration

load_lif()

load_lif_file()

load_xml()

load_xml_file()

ValidationMessage

ValidationErrorMessage

ValidationWarningMessage

Enumerations

Error

ValidationError

ValidationWarning

Miscellaneous

dllversion()

kvaBufferToXml()

kvaXmlToBuffer()

kvaXmlToFile()

kvaXmlValidate()

xmlGetLastError()

xmlGetValidationError()

xmlGetValidationStatusCount()

xmlGetValidationWarning()

1.6.9 kvlclib

Exceptions

KvlcError

KvlcEndOfFile

KvlcNotImplemented

Converter

Enumerations

ChannelMask

Error

FileFormat

Property

Reader Formats

reader_formats()

ReaderFormat

Writer Formats

writer_formats()

WriterFormat

Miscellaneous

dllversion

1.6.10 kvmlib

Exceptions

KvmError

KvmDiskError

KvmDiskNotFormated

KvmNoDisk

KvmNoLogMsg

LockedLogError

Enumerations

Device

Error

FileType

LoggerDataFormat

LogFileType

class canlib.kvmlib.enums.**LogFileType**(*value*)
kvmLogFileType_xxx

New in version 1.11.

ALL = 1

ERR = 0

dllversion()

1.6.11 kvrlib

Exceptions

KvrError

KvrBlank

DeviceNotInSetError

Discovery

Address

DeviceInfo

Discovery

get_default_discovery_addresses()

openDiscovery()

ServiceStatus

set_clear_stored_devices_on_exit()

start_discovery()

store_devices()

stored_devices()

Enumerations

Accessibility

AddressType

AddressTypeFlag

Availability

BasicServiceSet

verify_xml()

WEPKeys

1.6.12 linlib

Exceptions

LinError

LinNoMessageError

LinNotImplementedError

Channel

openChannel()

openMaster()

openSlave()

Channel

FirmwareVersion

Enumerations

ChannelData

ChannelType

Error

MessageDisturb

MessageFlag

MessageParity

Setup

MessageInfo

Miscellaneous

dllversion()

CHAPTER

TWO

RELEASE NOTES

2.1 Release Notes

This is the release notes for the pycanlib module.

Contents

2.1.1 New Features and Fixed Problems in V1.17.748 (16-FEB-2021)

- canlib.canlib:
  - Corrected set_bus_params_tq regarding type of flag attribute.
  - Added support for using setBusParams and getBusParams for channels that were opened using *BusParamsTq*.
  - Added Bitrate and BitrateFD enums for use with setBusParams and openChannel. canlib.canBITRATE_xxx and canlib.canFD_BITRATE_xxx constants are still supported but deprecated.
  - Added enum member BITRATE_8M_80P to BitrateFD and constant canlib.canFD_BITRATE_8M_80P.
- canlib.kvlclib
  - Added exception KvlcFileExists.

2.1.2 New Features and Fixed Problems in V1.16.588 (09-SEP-2020)

- canlib.canlib:
  - Added support for new bus parameter API in CANlib v.5.34. See section *Set CAN Bitrate* for more information.
  - Added attributes to canlib.IOControl.__dir__ and canlib.ChannelData.__dir__ in order to better support auto completion in IDEs.
  - Deprecated *canlib.Device.channel*, use *canlib.Device.open_channel* instead, which correctly handles keyword arguments
  - Added new Open flag canlib.canlib.Open.NOFLAG for parameter flags.
- canlib.kvadblib:
  - Corrected interpret when looking for CAN messages with extended id.
  - Updated get_message so that it requires EXT (bit 31) to be set on id if using extended id:s.
  - Added a new argument flags to get_message_by_id. If using messages with extended id:s, EXT should be set on flags.
- canlib.kvlclib:
  - The file_format parameter in canlib.kvlclib.Converter.setInputFile now accepts ReaderFormat as well.
  - Added a newer version of the BLF format, now also with CAN FD support

    'canlib.kvlclib.FileFormat.VECTOR_BLF_FD'. The format has both read and write capabilities.

2.1.3 New Features and Fixed Problems in V1.15.483 (27-MAY-2020)

- Dropped support for v2.7, v3.4 and v3.5, added v3.7 and v3.8.

2.1.4 New Features and Fixed Problems in V1.14.428 (02-APR-2020)

- Minor changes.

2.1.5 New Features and Fixed Problems in V1.13.390 (24-FEB-2020)

- `canlib.canlib`:
  - Added `HandleData` to wrap canGetHandleData. Also added `channel_data` as a helper function.
  - IOControl now returns utf-8 decoded strings instead of "bytes in string".
  - Fixed a bug where `isconnected` would return `False` if the `channel_number` attribute was larger than the total number of available CANlib channels, regardles of if the device was connected or not.
- `canlib`:
  - Corrected *`Frame`* comparison (!=) with other types, e.g. None

2.1.6 New Features and Fixed Problems in V1.12.251 (08-OCT-2019)

- Minor changes.

2.1.7 New Features and Fixed Problems in V1.11.226 (13-SEP-2019)

- `canlib.canlib`:
  - Added a slight delay in get_bus_statistics because the underlying functions in CANlib are asynchronous.
  - Added `read_error_counters` and `iocontrol clear_error_counters`.
  - Added `getBusOutputControl`.
  - Added `fileDiskFormat` that formats the disk in a remote device, i.e Kvaser DIN Rail.
- `canlib.BoundSignal.value`:
  - If the signal is an enum-signal, and the signal's value is not found in the enum definition, the raw value is now returned.
- `canlib.kvmlib`:
  - Marked using kvmlib class as deprecated (was deprecated in v1.6)
  - Replaced `estimate_events` with `Kme.event_count_estimation` in order to have same name as `LogFile.event_count_estimation`. Old function name is now deprecated.
  - When found, new 64 bit version of the dll call, `kvmLogFileMountEx`, `kvlcEventCountEx`, and `kvmKmeCountEventsEx` (added in CANlib v5.29), is now used.
  - Added `log_type` for supporting the different log types generated by Kvaser Memorator Light HS v2.
- `canlib.kvadblib`:
  - `Dbc` raises `KvdDbFileParse` if the dbc file loaded contains syntax errors.

2.1.8 New Features and Fixed Problems in V1.10.102 (12-MAY-2019)

- Reference documentation has been restructured.
- `Channel`:
  - Added support for slicing environment variables declared as char. Replaced low level function `scriptEnvvarSetData` with `script_envvar_set_data` and added `DataEnvVar` which is now returned when a char envvar is returned.
- `canlib.kvadblib`:
  - Error messages from the CAN database parser in `Dbc` can be retrieved using `get_last_parse_error()`.

2.1.9 New Features and Fixed Problems in V1.9.909 (03-MAR-2019)

- `canlib.kvadblib`:
  - Error texts are now fetched from the dll using kvaDbGetErrorText().
- `canlib.kvlclib`:
  - Added support for DLC mismatch handling included in CANlib v5.27
- `canlib.kvDevice`:
  - The `canlib.kvDevice.kvDevice` class is now deprecated, use *canlib.Device* instead
- *canlib.Device*:
  - Added method `Device.issubset` as a helper to find loosely specified devices.
- `canlib.canlib.iopin`:
  - Added attributes `fw_version` and `serial` to `IoPin`. To read these attributes, CANlib v5.27 is needed.
  - `AddonModule` is a new class, holding attributes of one add-on module.
  - `Config.modules` is now an attribute, calculated at creation time and containing an ordered list of `AddonModule` objects. The old functionality has been moved to `Config._modules`
  - `Config.issubset` is a new method to identify if a configuration contains the expected add-on modules.

2.1.10 New Features and Fixed Problems in V1.8.812 (26-NOV-2018)

- `canlib.canlib`:
  - Fixed issue were Channel.handle attribute would not be initialized when opening of the channel failed.
  - Added experimental support for accessing IO-pins on sub modules of the Kvaser DIN Rail SE 400S that was added to CANlib v5.26. This includes a new module `canlib.canlib.iopin`.
- `canlib.kvadblib`:
  - Fixed issue with signals were multiplexing mode, and scaling (factor and offset) returned wrong values from a loaded .dbc file.
  - Added show_all argument to Dbc.messages. Dbc.__iter__ now set show_all to False in order to skip VECTOR__INDEPENDENT_SIG_MSG messages.

2.1.11 New Features and Fixed Problems in V1.7.741 (16-SEP-2018)

- canlib.kvmlib:
  - Added canlib.kvmlib.event_count_estimation
  - Added canlib.kvmlib.kme Previous kvmlib.kmeXXX functions are now deprecated.
- canlib.canlib:
  - Added canlib.canlib.ScriptStatus
  - Added enums to canlib.canlib.ChannelCap
  - Fixed canlib.canlib.canWriteSync
- canlib.kvlclib:
  - Added API to access information about reader formats.
  - Added kvlclib.Property to replace old PROPERTY_XXX constants which are now deprecated.
  - Added kvlclib.reader_formats and kvlclib.writer_formats to replace now deprecated kvlclib.WriterFormat.getFirstWriterFormat and kvlclib.WriterFormat.getNextWriterFormat.

2.1.12 New Features and Fixed Problems in V1.6.615 (13-MAY-2018)

- Updated for CANlib SDK v5.23.
- Getting version numbers should now be done with dllversion(), which will return *canlib.BetaVersionNumber* if the dll is marked as Beta. Also added canlib.prodversion() to return the CANlib product version number.
- canlib.device:
  - New *canlib.device.Device* class (available as *canlib.Device*) that is a simpler version of kvDevice. *canlib.device.Device* objects can be defined using an EAN and serial number, or a connected device can be searched for using canlib.device.Device.find. These objects do not require the device to stay connected, and can be used to later create most other canlib objects, e.g. canlib.canlib.Channel, canlib.kvmlib.Memorator, etc.
  - New canlib.device.connected_devices which returns an iterator of *canlib.device.Device* objects, one for each device currently connected.
- canlib.ean:
  - *canlib.ean.EAN* objects can be tested for equality, both with other *canlib.ean.EAN* objects and with strings.
  - Added CanNotFound exception.
  - *canlib.ean.EAN* objects can now be directly instantiated from string, i.e. ean = canlib.EAN(ean_string) instead of ean = canlib.EAN.from_string(ean_string).
  - *canlib.ean.EAN* objects can be converted back into any of the representations that can be used to create them. See the documentation of *canlib.ean.EAN* for more info.
  - *canlib.ean.EAN* objects can be indexed and iterated upon, yielding the digits as ints.
- canlib.canlib:
  - canlib.canlib.EnvVar object raises EnvvarNameError when given an illegal name, instead of AssertionError.

  - canlib.canlib.openChannel can now set the bitrate of the channel opened.
  - canlib.canlib.Channel objects automatically close their handles when garbage collected
  - canlib.canlib.Channel has new methods canlib.canlib.Channel.scriptRequestText and canlib.canlib.Channel.scriptGetText to get text printed with printf() by a script. This text is returned as a canlib.canlib.ScriptText object.
- canlib.kvamemolibxml:
  - A new, object oriented way of dealing with kvamemolibxml using canlib.kvamemolibxml.Configuration objects.
- canlib.kvmlib:
  - Improved object model
    - New canlib.kvmlib.openDevice function that returns a canlib.kvmlib.Memorator object representing a connected Memorator device. See the documentation of canlib.kvmlib.Memorator for instructions on how to use this new class to more easily interface with your Memorators.
    - New canlib.kvmlib.openKmf function for opening .KMF files that returns a canlib.kvmlib.Kmf object that is similar to canlib.kvmlib.Memorator. See the docstring of canlib.kvmlib.Kmf for more information.
- canlib.linlib:
  - Getting version number with canlib.linlib.dllversion (requires CANlib SDK v5.23 or newer).
  - Explicit canlib.linlib.Channel.close function for forcing a linlib channel's internal handle to be closed.
- canlib.canlib:
  - Added support for accessing information within compiled t program (.txe) files.
    - Added wrapper function for kvScriptTxeGetData.
    - Added compiled t program (.txe) interface class canlib.canlib.Txe.
- canlib.kvadblib:
  - enums now returns non-empty dictionary in attribute definition returned from EnumDefinition.definition

2.1.13 New Features and Fixed Problems in V1.5.525 (12-FEB-2018)

- Updated for CANlib SDK v5.22.
- Added support for LIN bus API (LINlib)
- Added support for Database API (kvaDbLib) Needs version v5.22 of CANlib SDK to get supported dll.

Restructuring of code in order to make the API simpler and the code base more maintainable have resulted in the following changes (old style is deprecated, shown in details while running Python with the -Wd argument):

- canlib.kvMessage has been renamed *canlib.Frame*
  - *canlib.Frame* objects are now accepted and returned when writing and reading channels.
  - The new canlib.kvadblib module uses these *canlib.Frame* objects heavily.
- canlib.canlib:
  - Added wrapper functions for canReadStatus and canRequestChipStatus

  - Deprecated use of `canlib.canlib.canlib()` objects; all methods have been moved to the module.
    * See the docstring of `canlib.canlib.CANLib` for more information
  - Simplified the names of the channel-classes (old names are deprecated):
    * The channel class is now `canlib.canlib.Channel`, instead of `canlib.canChannel`.
    * `canlib.canlib.ChannelData_Channel_Flags` is now `canlib.canlib.ChannelFlags`
    * `canlib.canlib.ChannelData_Channel_Flags_bits` is now `canlib.canlib.ChannelFlagBits`
  - `canlib.canlib.Channel` now uses *canlib.Frame* objects for reading and writing.
    * `canlib.Channel.read` now returns a *canlib.Frame* object instead of a tuple. However, *canlib.Frame* objects are largely compatible with tuples.
    * `canlib.Channel.write` takes a single argument, a *canlib.Frame* object. The previous call signature has been taken over by `canlib.Channel.write_raw`.
    * Likewise for `canlib.Channel.writeWait` and its new friend `canlib.Channel.writeWait_raw`.
  - The class `canlib.canlib.canVersion` has been removed, and `canlib.canlib.getVersion` now returns a *canlib.VersionNumber*. The new class still supports conversion to string and accessing `.major` and `.minor` attributes.

- `canlib.kvmlib`:
  - Added wrapper functions for kvmKmeReadEvent.
  - Corrected encoding for Python 3 in kmeOpenFile().
  - Deprecated names for several classes to make them more logical and more pythonic:
    * `canlib.kvmlib.memoMsg` is now `canlib.kvmlib.LogEvent`
    * `canlib.kvmlib.logMsg` is now `canlib.kvmlib.MessageEvent`
    * `canlib.kvmlib.rtcMsg` is now `canlib.kvmlib.RTCEvent`
    * `canlib.kvmlib.trigMsg` is now `canlib.kvmlib.TriggerEvent`
    * `canlib.kvmlib.verMsg` is now `canlib.kvmlib.VersionEvent`
  - The class `canlib.kvmlib.kvmVersion` has been removed, and `canlib.kvmlib.KvmLib.getVersion` now returns a *canlib.VersionNumber*. The new class still supports conversion to string and accessing `.major`, `.minor`, and `.build` attributes.

- `canlib.kvlclib`:
  - Added method `canlib.kvlclib.addDatabaseFile` and helper object `canlib.kvlclib.ChannelMask`.
  - The `canlib.kvlclib.KvlcLib` object has been deprecated.
    * All functions that relate to converters have been moved to the more appropriately named `canlib.kvlclib.Converter`.
      · Some of these functions have been renamed:
      · `IsOutputFilenameNew`, `IsOverrunActive`, and `IsDataTruncated` have all had their initial "i" lower-cased, as the upper case "I" was an error.
      · `getPropertyDefault` and `isPropertySupported` are no longer available on the `Converter` object, they must be accessed via the `format` attribute:

```
converter.format.getPropertyDefault(...)
```

    - canlib.kvlclib.WriterFormat.getFirstWriterFormat and canlib.kvlclib.WriterFormat.getNextWriterFormat now returns a kvlclib.FileFormat object (which is based on the IntEnum class).
    - Other functions have been moved to the canlib.kvlclib module.
    - deleteConverter is no longer supported. Instead, converters are automatically deleted when garbage collected. If their contents must be flushed to file, see the new canlib.kvlclib.Converter.flush method.
  - The class canlib.kvlclib.KvlcVersion has been removed, and canlib.kvmlib.kvlclib.getVersion now returns a *canlib.VersionNumber*. The new class still supports conversion to string and accessing .major, .minor, and .build attributes.
- canlib.kvrlib:
  - The canlib.kvrlib.KvrLib object has been deprecated; all methods have been moved to the module.
  - canlib.kvrlib.getVersion no longer returns a canlib.kvrlib.kvrVersion but a *canlib.VersionNumber*. The return value still supports conversion to string and accessing .major and .minor attributes.
- canlib.kvamemolibxml:
  - Renamed from canlib.KvaMemoLibXml, however trying to import the old name will just import the new one instead.
  - Deprecated the use of canlib.kvamemolibxml.KvaMemoLib objects, all methods have been moved to the canlib.kvamemolibxml module itself.
  - Breaking change: Moved values that were incorrectly defined as constants into enums. In most cases this should not have any impact, as all the values are internal error codes and are turned into Python exceptions. But if you nonetheless use the kvamemolibxml status values directly, you'll need to change your code as follows:
    - KvaXmlStatusERR_XXX_XXX is now Error.XXX_XXX.
    - KvaXmlValidationStatusERR_XXX_XXX is now ValidationError.XXX_XXX
    - KvaXmlValidationStatusWARN is now ValidationWarning.XXX_XXX.
    - KvaXmlStatusFail is now Error.FAIL (Changed to be consistent with other KvaXmlStatus errors). The same is true for ValidationError.FAIL.
    - KvaXmlStatusOK and KvaXmlValidationStatusOK are still treated as if they are constants, as they are not error statuses.
  - canlib.kvamemolibxml.getVersion no longer returns a string but a *canlib.VersionNumber*. The return value still supports conversion to string.
- Exceptions:
  - Exceptions throughout the package have been standardised, and now all inherit from *canlib.exceptions.CanlibException*.
  - The canERR attribute that some exceptions had has been deprecated in favour of a status attribute. Furthermore, all canlib exceptions now have this attribute; the status code that was returned from a C call that triggered the specific exception.

2.1.14 New Features and Fixed Problems in V1.4.373 (13-SEP-2017)

- Minor changes.

2.1.15 New Features and Fixed Problems in V1.3.242 (05-MAY-2017)

- Added missing unicode conversions for Python3.
- Linux
  - Added support for new libraries (kvadblib, kvmlib, kvamemolibxml, kvlclib).
  - Added wrappers KvFileGetCount, kvFileGetName, kvFileCopyXxxx, kvDeviceSetMode, kvDeviceGetMode
- canlib:
  - Added wrapper for kvFileDelete
  - Enhanced printout from canScriptFail errors.
  - Second file argument in fileCopyFromDevice and fileCopyToDevice is now optional.
  - OS now loads all dependency dll (also adding KVDLLPATH to PATH in Windows).

2.1.16 New Features and Fixed Problems in V1.2.163 (15-FEB-2017)

- Added wrapper function canlib.getChannelData_Cust_Name()
- Added module canlib.kvlclib which is a wrapper for the Converter Library kvlclib in CANlib SDK.
- Added wrapper function canChannel.flashLeds().
- Added missing unicode conversions for Python3.
- Fixed bug where CANlib SDK install directory was not always correctly detected in Windows 10.

2.1.17 New Features and Fixed Problems in V1.1.23 (28-SEP-2016)

- canSetAcceptanceFilter and kvReadTimer was not implemented in Linux

2.1.18 New Features and Fixed Problems in V1.0.10 (15-SEP-2016)

- Initial module release.
- Added kvmlib.kmeSCanFileType()
- Added canChannel.canAccept() and canChannel.canSetAcceptanceFilter()

INDEX

O

P

R

S

T

V