can4python Documentation

Release 0.2.1

Jonas Berg

2016-10-17

Contents

Contents:

CHAPTER 1

Introduction to can4python

A package for handling CAN bus (Controller Area Network) signals on Linux SocketCAN, for Python 3.3 and later.

- Free software: BSD license

1.1 Web resources

- Source code on GitHub: https://github.com/caran/can4python
- Documentation: https://can4python.readthedocs.org.
- Python Package Index (PyPI): https://pypi.python.org/pypi/can4python

1.2 Features

- Sends and receives CAN frames.
- Handles parsing of CAN signals from CAN frames.
- Uses SocketCAN for Linux.
- For Python 3.3 or later. Python 3.4 is required for some functionality.
- Implemented as pure Python files, without any external dependencies.
- Suitable for use with BeagleBone and Raspberry Pi embedded Linux boards.
- Configuration using the open source KCD file format.
- Throttle incoming frames to reduce frame rate.
- Filtering of incoming frames on data changes. This is done via a bit mask in the Linux kernel.
- Periodic frame transmission executed by the Linux kernel (not by Python code).
- Useful for showing the contents of KCD files (also those converted from DBC files).

1.3 Configuration file format

This CAN library uses the KCD file format for defining CAN signals and CAN messages. It is an open-source file format for describing CAN bus relationships. See https://github.com/julietkilo/kcd for details on the format, and example files.

The licensing of the KCD file format is, according to the web page:

```
The files that are describing the format are published under the Lesser GPL license.
The KCD format is copyrighted by Jan-Niklas Meier (dschanoeh) and Jens Krueger (julietkilo).
According to the authors this does not imply any licensing restrictions on
software libraries implementing the KCD file format, or on software using those libraries.
```

Traditionally CAN bus relationships are described in DBC files, a file format owned by Vector Informatik GmbH. There are open source DBC-to-KCD file format converters available, for example the CANBabel tool: https://github.com/julietkilo/CANBabel

1.4 Known limitations

- Not all CAN functionality is implemented. 'Error frames' and 'remote request frames' are not handled, and CAN multiplex signals are not supported.
- Not all features of the KCD file format are implemented, for example 'Labels'.
- It is assumed that each CAN signal name only is available in a single CAN frame ID.

1.5 Dependencies

The can4python package itself has no dependencies, except for Python 3.3+ running on Linux.

For tests, a virtual CAN interface ('vcan') must be installed. It is part of the Linux kernel. See the Usage page of this documentation for details.

Dependencies for testing and documentation:

Dependency	Description	License	Debian/pip package
vcan0	Virtual CAN bus interface	Part of Linux kernel	
coverage	Test coverage measurement	Apache 2.0	P: coverage
texlive	Latex library (for PDF creation)	"Knuth"	D: texlive-full
Sphinx 1.3+	Documentation tool	BSD 2-cl	P: sphinx
Sphinx rtd theme	Theme for Sphinx	MIT	P: sphinx_rtd_theme

1.6 Installation and usage

See separate documentation pages.

1.7 Support

The preferred way is to open a question on Stack Overflow .

Found a bug? Open an issue on Github!

CHAPTER 2

Installation

At the command line:

```
pip3 install can4python
```

Installing dependencies for testing and documentation:

```
sudo pip3 install sphinx
sudo pip3 install sphinx_rtd_theme
sudo pip3 install coverage
```

For PDF, you also need to install (3 GByte):

```
sudo apt-get install texlive-full
```

CHAPTER 3

Usage

3.1 Technical background

CAN bus (Controller Area Network) is a bus frequently used in the automotive industry. Packets with up to eight bytes of data are sent. Each frame (packet) has a frame ID, which pretty much is a 'from' address. Typical speeds are 100 to 500 kbit/s.

In Linux the CAN protocol is implemented in SocketCan. It is modelled after network sockets, and in order to use a CAN interface a socket is opened to the Linux kernel. The CAN interface is often named something like 'can0'.

Each of the CAN frames contains a number of signals. In order to specify a signal, at least this must be known:

- signal type (signed/unsigned integer, etc)
- number of bits
- startbit
- bit numbering scheme
- endianness: little endian or big endian

For more details, see *CanSignalDefinition*.

3.2 Minimal examples

To use can4python in a project with the `'vcan0'` CAN interface, and reading the CAN signal definitions from a KCD file:

```
import can4python as can

bus = can.CanBus.from_kcd_file('documentation_example.kcd', 'vcan0', ego_node_ids=["1"])
bus.send_signals({'testsignal2': 3}) # Signal value = 3
```

The sent CAN frame is viewed using the 'candump' command line utility (described in a later section):

```
$ candump vcan0
vcan0  007   [8]  03 00 00 00 00 00 00 00
```

As our script will send out frames from the node "1", it will consider frame ID 7 (which holds testsignal2) as an outgoing frame. That is seen in the corresponding KCD file:

```
<?xml version="1.0" ?>
<NetworkDefinition xmlns="http://kayak.2codeornot2code.org/1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="Definition.xsd">
  <Document/>
  <Bus name="Mainbus">
    <Message id="0x007" length="8" name="testmessage">
      <Signal name="testsignal1" offset="56"/>
      <Signal name="testsignal2" offset="0"  length="16" endianess="little"/>
      <Signal name="testsignal3" offset="24" length="16" />
      <Signal name="testsignal4" offset="59" length="4" endianess="big">
        <Value type="signed"/>
      </Signal>
      <Producer>
        <NodeRef id="1"/>
      </Producer>
    </Message>
  </Bus>
</NetworkDefinition>
```

To receive CAN signals:

```
import can4python as can

bus = can.CanBus.from_kcd_file('documentation_example.kcd', 'vcan0', ego_node_ids=["2"])
received_signalvalues = bus.recv_next_signals()
print(received_signalvalues)
```

The `bus.recv_next_signals()` will recive one CAN frame, and unpack its signals. The `received_signalvalues` is a dictionary with the signal values (*numerical*), having the signal names (*str*) as keys. If a timeout is defined and no frame is received, a *`CanTimeoutException`* is raised.

Test it by sending a CAN frame using the 'cansend' command line utility:

```
$ cansend vcan0 007#0F0000FF000000F1
```

The Python script will print:

```
{'testsignal1': 1.0, 'testsignal3': 255.0, 'testsignal2': 15.0, 'testsignal4': -2.0}
```

Alternatively, you can also set the CAN frame definitions and CAN signal definitions in your source code (instead of in a KCD file):

```
import can4python as can

frame_def = can.CanFrameDefinition(7, name='testmessage')
frame_def.producer_ids = ["1"]
signal_def = can.CanSignalDefinition("testsignal2", 0, 16)

frame_def.signaldefinitions.append(signal_def)
config = can.Configuration({7: frame_def}, ego_node_ids=["1"])

bus = can.CanBus(config, 'vcan0')
bus.send_signals({'testsignal2': 3}) # Signal value = 3
```

3.3 Broadcast Manager (BCM) usage example

The Broadcast Manager (BCM) can automatically do periodic CAN frame transmission, and it can filter incoming CAN frame on data changes. Periodic transmission is done like this:

```
import time
import can4python as can

frame_def = can.CanFrameDefinition(7, name='testmessage')
frame_def.producer_ids = ["1"]
frame_def.cycletime = 250 # milliseconds
signal_def = can.CanSignalDefinition("testsignal2", 0, 16)

frame_def.signaldefinitions.append(signal_def)
config = can.Configuration({7: frame_def}, ego_node_ids=["1"])

bus = can.CanBus(config, 'vcan0', use_bcm=True)
bus.send_signals({'testsignal2': 5}) # Signal value = 5. Start periodic transmission.
time.sleep(10)
```

The output resulting CAN frames are:

```
$ candump vcan0
vcan0  007   [8]  05 00 00 00 00 00 00 00
vcan0  007   [8]  05 00 00 00 00 00 00 00
vcan0  007   [8]  05 00 00 00 00 00 00 00
vcan0  007   [8]  05 00 00 00 00 00 00 00
vcan0  007   [8]  05 00 00 00 00 00 00 00
vcan0  007   [8]  05 00 00 00 00 00 00 00
vcan0  007   [8]  05 00 00 00 00 00 00 00
(truncated)
```

3.4 Usage recommendations

This CAN library is designed for experiments on sending and receiving CAN messages, and extracting the signals within. It main use case is to read a limited number of CAN messages from a CAN bus, and send a few messages now and then.

When running on embedded Linux hardware (for example the BeagleBone), the speed is sufficient to unpack around 500 CAN frames per second. As the can4python library will instruct the Linux kernel to filter incoming messages according to the available message IDs in the KCD configuration file (or corresponding settings made from code), it is recommended to edit your KDC file to only include the messages and signals you are interested in.

3.5 Show an overview of settings

To have an overview of the messages and signals on the bus:

```
print(bus.get_descriptive_ascii_art())
```

It will print something like:

```
CAN bus 'Mainbus' on CAN interface: vcan0, having 1 frameIDs defined. Protocol RAW
    CAN configuration object. Busname 'Mainbus', having 1 frameIDs defined. Enacts these node IDs: 1
    Frame definitions:
```

```
CAN frame definition. ID=7 (0x007, standard) 'testmessage', DLC=8, cycletime None ms, producers:
    Signal details:
    ---------------


    Signal 'testsignal1' Startbit 56, bits 1 (min DLC 8) little endian, unsigned, scalingfactor 1
         valoffset 0.0 (range 0 to 1) min None, max None, default 0.0.

         Startbit normal bit numbering, least significant bit: 56
         Startbit normal bit numbering, most significant bit: 56
         Startbit backward bit numbering, least significant bit: 0

                  111111   22221111 33222222 33333333 44444444 55555544 66665555
         76543210 54321098 32109876 10987654 98765432 76543210 54321098 32109876
         Byte0    Byte1    Byte2    Byte3    Byte4    Byte5    Byte6    Byte7
                                                                          L
         66665555 55555544 44444444 33333333 33222222 22221111 111111
         32109876 54321098 76543210 98765432 10987654 32109876 54321098 76543210


    Signal 'testsignal2' Startbit 0, bits 16 (min DLC 2) little endian, unsigned, scalingfactor 1
         valoffset 0.0 (range 0 to 7e+04) min None, max None, default 0.0.

         Startbit normal bit numbering, least significant bit: 0
         Startbit normal bit numbering, most significant bit: 15
         Startbit backward bit numbering, least significant bit: 56

                  111111   22221111 33222222 33333333 44444444 55555544 66665555
         76543210 54321098 32109876 10987654 98765432 76543210 54321098 32109876
         Byte0    Byte1    Byte2    Byte3    Byte4    Byte5    Byte6    Byte7
         XXXXXXXL MXXXXXXX
         66665555 55555544 44444444 33333333 33222222 22221111 111111
         32109876 54321098 76543210 98765432 10987654 32109876 54321098 76543210


    Signal 'testsignal3' Startbit 24, bits 16 (min DLC 5) little endian, unsigned, scalingfactor
         valoffset 0.0 (range 0 to 7e+04) min None, max None, default 0.0.

         Startbit normal bit numbering, least significant bit: 24
         Startbit normal bit numbering, most significant bit: 39
         Startbit backward bit numbering, least significant bit: 32

                  111111   22221111 33222222 33333333 44444444 55555544 66665555
         76543210 54321098 32109876 10987654 98765432 76543210 54321098 32109876
         Byte0    Byte1    Byte2    Byte3    Byte4    Byte5    Byte6    Byte7
                                    XXXXXXXL MXXXXXXX
         66665555 55555544 44444444 33333333 33222222 22221111 111111
         32109876 54321098 76543210 98765432 10987654 32109876 54321098 76543210


    Signal 'testsignal4' Startbit 59, bits 4 (min DLC 8) big endian, signed, scalingfactor 1, uni
         valoffset 0.0 (range -8 to 7) min None, max None, default 0.0.

         Startbit normal bit numbering, least significant bit: 59
         Startbit normal bit numbering, most significant bit: 62
         Startbit backward bit numbering, least significant bit: 3

                  111111   22221111 33222222 33333333 44444444 55555544 66665555
```

```
             76543210 54321098 32109876 10987654 98765432 76543210 54321098 32109876
             Byte0    Byte1    Byte2    Byte3    Byte4    Byte5    Byte6    Byte7
                                                                              MXXL
             66665555 55555544 44444444 33333333 33222222 22221111 111111
             32109876 54321098 76543210 98765432 10987654 32109876 54321098 76543210
```

The numbers above "Byte0 Byte1 " etc are the bit numbers using the normal numbering scheme. The letters 'ML' indicate the most and least significant bits in the signal, respectively. The numbers at the bottom is the bit numbering in the backward numbering scheme.

3.6 Configuration file

This CAN library uses the KCD file format for defining CAN signals and CAN messages. It is an open-source file format for describing CAN bus relationships. See https://github.com/julietkilo/kcd for details on the format, and example files.

This can4python CAN library implements a subset of the KCD file format. For example 'multiplex' signals are not supported.

One common file format for CAN information is the proprietary DBC file format. The CAN Babel is a tool for converting DBC files to KCD files. See https://github.com/julietkilo/CANBabel

Configurations made in source code using can4python can be written to a KCD file:

```
mycanbus.write_configuration('outputfile.kcd')
```

3.7 Show the contents of a .KCD configuration file (possibly converted from a .DBC file)

It is easy to print an overview of a configuration file:

```
import can4python
config = can4python.FilehandlerKcd.read("tests/testfile_input.kcd")
print(config.get_descriptive_ascii_art())
```

It will print:

```
CAN configuration object. Busname 'Mainbus', having 2 frameIDs defined. Enacts these node IDs:
    Frame definitions:

    CAN frame definition. ID=1 (0x001, standard) 'testframedef1', DLC=8, cycletime None ms, producers
        Signal details:
        ---------------


        Signal 'testsignal11' Startbit 56, bits 1 (min DLC 8) little endian, unsigned, scalingfactor
             valoffset 0.0 (range 0 to 1) min None, max None, default 0.0.

             Startbit normal bit numbering, least significant bit: 56
             Startbit normal bit numbering, most significant bit: 56
             Startbit backward bit numbering, least significant bit: 0

                      111111   22221111 33222222 33333333 44444444 55555544 66665555
             76543210 54321098 32109876 10987654 98765432 76543210 54321098 32109876
```

```
             Byte0    Byte1    Byte2    Byte3    Byte4    Byte5    Byte6    Byte7
                                                                              L
             66665555 55555544 44444444 33333333 33222222 22221111 111111
             32109876 54321098 76543210 98765432 10987654 32109876 54321098 76543210


        Signal 'testsignal12' Startbit 8, bits 16 (min DLC 3) little endian, unsigned, scalingfactor
             valoffset 0.0 (range 0 to 7e+04) min 0.0, max 100.0, default 0.0.
             Test signal number 2
             Startbit normal bit numbering, least significant bit: 8

             (truncated)
```

3.8 Show filtering of incoming frames

To see the CAN frame receive filters (for RAW interface) that are applied (in Ubuntu):

```
cat /proc/net/can/rcv*
```

See also *SocketCanRawInterface.set_receive_filters()*

3.9 Running tests

In order ro run the tests:

```
sudo make test
```

The tests are possible to run on a desktop Linux PC, as well as embedded Linux hardware.

3.10 Virtual (simulated) CAN interfaces for testing

The can4python library uses socketCAN type of CAN interface, for use under Linux. The CAN interfaces are typically named 'can0', 'can1' etc. It is also possible to setup virtual (simulated) CAN interfaces for testing purposes, and they act as loopback interfaces.

To enable the 'vcan0' virtual CAN interface on your desktop Ubuntu Linux machine:

```
sudo modprobe vcan
sudo ip link add dev vcan0 type vcan
sudo ip link set up vcan0
```

To see what is sent on the virtual CAN interface, use the 'candump' tool:

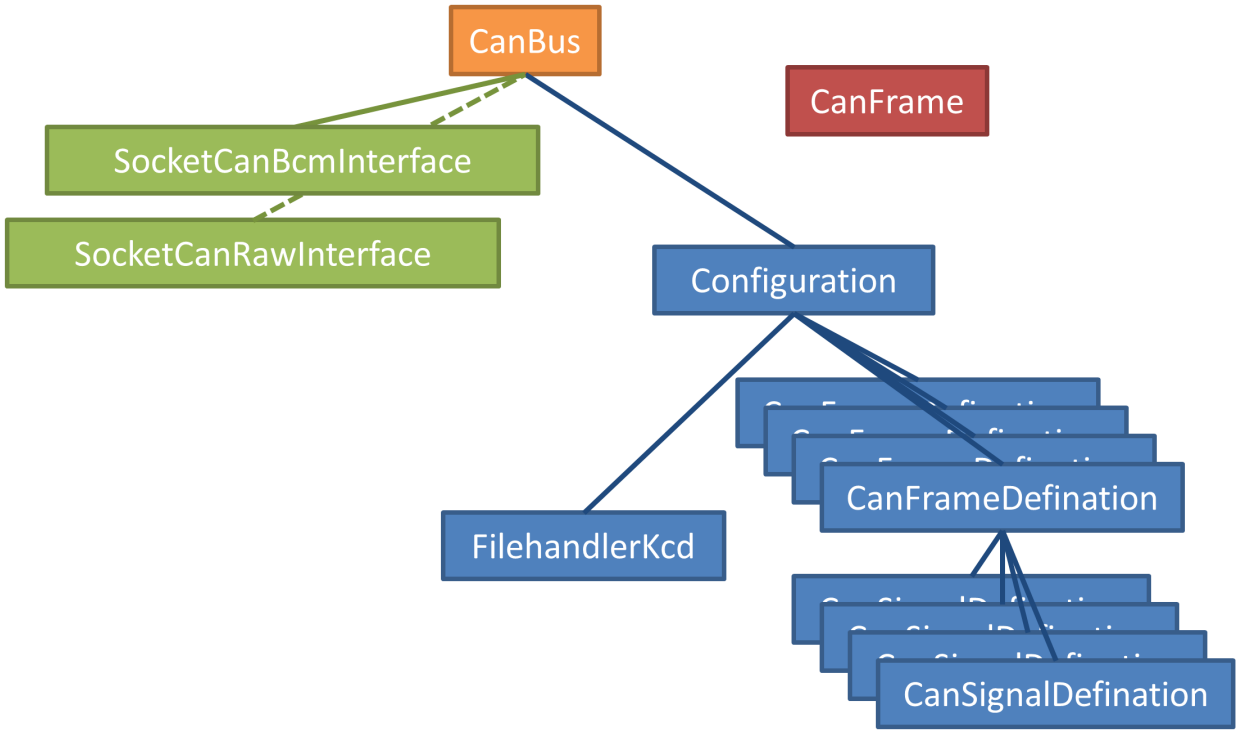
```
candump vcan0
```

3.11 Advanced usage

You can for example directly manipulate the CAN interface instance. If using the BCM CAN interface:

```
mycanbus.caninterface.stop_periodic_send(103)
```

3.12 Architectural overview



We define these object types:

CanBus See *CanBus*. Abstraction of the CAN bus, and uses a *SocketCanRawInterface* or a SocketCanBcmnterface. This is the main API object that developers will use.

SocketCanRawInterface See *SocketCanRawInterface*. Abstraction of the SocketCAN interface hardware (or simulated=virtual hardware), using the RAW protocol to communicate with the Linux kernel. Requires Python 3.3 or later.

SocketCanBcmInterface See *SocketCanBcmInterface*. Abstraction of the SocketCAN interface hardware (or simulated=virtual hardware), using the Broadcast Manager in the Linux kernel. Requires Python 3.4 or later.

CanFrame See *CanFrame*. A (physical) package with data sent on the CanBus.

Configuration See *Configuration*. An object holding configuration information about what is sent on the CAN bus. Has typically several *CanFrameDefinition* (each having a number of *CanSignalDefinition*).

CanFrameDefinition See *CanFrameDefinition*. Describes which signals that are sent in a frame with a specific ID. Has typically several *CanSignalDefinition* objects. Note that a *CanFrameDefinition* is a description of the different parts of the Can frame, but the *CanFrameDefinition* itself does not hold any data.

CanSignalDefinition See *CanSignalDefinition*. Defines where in a message this signal is located, how it is scaled etc.

FilehandlerKcd See *FilehandlerKcd*. Reads and writes configurations to file, in the KCD file format.

Either *SocketCanRawInterface* or *SocketCanBcmInterface* is used, not both simultaneously. You select which to use in the constructor of the *CanBus*.

It is possible to use only parts of the library. The architecture is such that it should be easy to write another CanInterface object.

CHAPTER 4

API for can4python

This page shows the public part of the API. For a more detailed documentation on all objects, see the can4python sub page: *CanBus*

If you are using KCD file based configuration, you should really only need to interact with the *CanBus* object.

class can4python.**CanBus**(*config*, *interfacename*, *timeout=None*, *use_bcm=False*)
CAN bus abstraction.

Uses Python 3.3 (or later) and the Linux SocketCAN interface.

The SocketCan Broadcast Manager (BCM) handles periodic sending of CAN frames, and can throttle incoming CAN frames to a slower rate, or to only receive frames when the data content has changed. Python 3.4 (or later) is required to use the BCM.

If you need to receive all frames, do not use the BCM.

Parameters

- **config** (*Configuration* object) – Configuration object describing what is happening on the bus.
- **interfacename** (*str*) – Name of the Linux SocketCan interface to use. For example 'vcan0' or 'can1'.
- **timeout** (*numerical*) – Timeout value in seconds for *recv_next_signals()*. Defaults to None (blocking read).
- **use_bcm** (*bool*) – True if the SocketCan Broadcast manager (BCM) should be used. Defaults to False.

classmethod from_kcd_file(*filename*, *interfacename*, *timeout=None*, *busname=None*, *use_bcm=False*, *ego_node_ids=None*)
Create a *CanBus*, using settings from a configuration file.

This is a convenience function, to avoid creating a separate configuration object.

Parameters

- **filename** (*str*) – Full path to existing configutation file, in the KCD file format.
- **interfacename** (*str*) – For example 'vcan0' or 'can1'.
- **timeout** (*numerical*) – Timeout value in seconds for *recv_next_signals()*. Defaults to None (the recv_next_signals call will be blocking).
- **busname** (*str or None*) – Which bus name in the messagedefinitions file to use. Defaults to None (using first alphabetically).

- **use_bcm** (*bool*) – True if the SocketCan Broadcast manager (BCM) should be used. Defaults to False.
- **ego_node_ids** (*set of strings*) – Set of nodes that this program will enact. You can also pass it a list.

config
Get the configuration (read-only). The configuration is set in the constructor.

use_bcm
Return True if BCM is used (read-only). Is set in the constructor.

init_reception()
Setup the CAN frame reception.

When using the RAW protocol, this enables filtering to reduce the input frame flow.

It works the opposite for the BCM protocol, where it explicitly subscribes to frame IDs.

recv_next_signals()
Receive one CAN frame, and unpack it to signal values.
Returns A dictionary of signal values, where the keys are the signalname (*str*) and the items are the values (*numerical*).
If the frame not is defined for this *CanBus* instance, an empty dictionary is returned.
Raises CanTimeoutException – If a timeout is defined and no frame is received. See *CanTimeoutException*.

recv_next_frame()
Receive one CAN frame. Returns a *CanFrame* object.
Raises CanTimeoutException – If a timeout is defined and no frame is received. See *CanTimeoutException*.

stop_reception()
Stop receiving, when using the BCM.

send_signals(**args*, ***kwargs*)
Send CAN signals in frames.
Parameters **signals_to_send** (*dict*) – The signal values to send_frame. The keys are the signalnames (*str*), and the items are the values (*numerical* or *None*). If the value is *None* the default value is used.
You can also use signal names as function arguments (keyword arguments). These are equal:

```
mycanbus.send_signals({"VehicleSpeed": 70.3, "EngineSpeed": 2821})
mycanbus.send_signals(VehicleSpeed=70.3, EngineSpeed=2821)
```

The signal names must be already defined for this *CanBus* instance.
Raises CanException – When failing to set signal value etc. See *CanException*.

start_sending_all_signals()
Start sending all configured frames, when using the BCM.

The default value for the signals are used, until updated via the *send_signals()* function.

If you do not use this *start_sending_all_signals()* method, the periodic transmission for each frame will start at first *send_signals()* call.

send_frame(*frame_to_send*)
Send a single CAN frame.
Parameters **frame_to_send** (*CanFrame*) – The frame to send.

stop_sending()
Stop periodic sending, when using the BCM.

stop()
Stop periodic sending and receiving, when using the BCM.

get_descriptive_ascii_art()
Display an overview of the *CanBus* object with frame definitions and signal definitions.
Returns A multi-line string.

write_configuration(*filename*)
Write configuration to file.
Parameters **filename** (*str*) – Full path to file with configuration.
Saves to an XML file in the KCD file format.

class can4python.**Configuration**(*framedefinitions=None*, *busname=None*, *ego_node_ids=None*)
Configuration object for the things that happen on the CAN bus. It holds frame definitions (including signal definitions), the busname etc. See below.

framedefinitions
dict

The keys are the frame_id (*int*) and the items are the corresponding *CanFrameDefinition* objects.

busname
str or None

Which bus name in the configuration file to use when reading. Defaults to `None` (using first alphabetically).

ego_node_ids
set of strings Set of nodes that this program will enact. You can pass it a list (it will convert to a set).

get_descriptive_ascii_art()
Display an overview of the *Configuration* object with frame definitions and signals.
Returns A multi-line string.

add_framedefinition(*framedef*)
Add a frame definition to the configutation.
Parameters **framedef** (*CanFrameDefinition* object) – The frame definition to add.
This is a convenience function. These two alternatives are equal:

```
myconfig.add_framedefinition(framedef1)
myconfig.framedefinitions[framedef1.frame_id] = framedef1
```

set_throttle_times(*inputdict*)
Set throttle_time for some of the framedefinitions in the configuration object.
Parameters **inputdict** (*dict*) – The keys are the frame IDs (int) and the values are the throttle times (numerical or None) in milliseconds.
This is a convenience function. You can instead do like this for each frame:

```
myconfig.framedefinitions[myframe_id].throttle_time = mythrottletime
```

set_throttle_times_from_signalnames(*inputdict*)
Set throttle_time for some of the framedefinitions in the configuration object (via signal names)

Parameters **inputdict** (*dict*) – The keys are the signalnames (str) and the values are the throttle times (numerical or None) in milliseconds.

Note that the throttle_time is set on the framedefinition holding the signalname. It will also affect other signals on the same frame. Setting different throttle_times to signals on the same frame will give an undefined result.

This is a convenience function. You can instead do like this for each signalname:

```
(first find myframe_id for a given signalname)
myconfig.framedefinitions[myframe_id].throttle_time = mythrottletime
```

set_receive_on_change_only(*inputlist*)

Set receive_on_change_only for some of the framedefinitions in the configuration object.

Parameters **inputlist** (*list of ints*) – The frame IDs that should be received only when the data has changed.

This is a convenience function. You can instead do like this for each frame ID:

```
myconfig.framedefinitions[myframe_id].receive_on_change_only = True
```

set_receive_on_change_only_from_signalnames(*inputlist*)

Set receive_on_change_only for some of the framedefinitions in the configuration object (via signal names).

Parameters **inputlist** (*list of str*) – The signal names that should be received only when the data has changed.

Note that the receive_on_change_only is set on the framedefinition holding the signalname. It will also affect other signals on the same frame.

This is a convenience function. You can instead do like this for each signalname:

```
(first find myframe_id for a given signalname)
myconfig.framedefinitions[myframe_id].receive_on_change_only = True
```

find_frameid_from_signalname(*input_signalname*)

Find which frame_id a specific signal name belongs.

Parameters **input_signalname** (*str*) – signal name to search for.

Returns: The frame_id (int) in which the signal is located.

Raises CanException when the given signal name not is found.

class can4python.**FilehandlerKcd**

File handler for the KCD file format.

Note that only a subset of the KCD file format is implemented. These tags are read:

```
* Network definition: xmlns
  * Bus: name
    * Message: id, name, length, interval, format,
      * Producer:
        * NodeRef: id
      * Signal: endianness, length, name, offset
        * Value: type, slope, intercept, unit, min, max
        * Notes:
```

Further, there are is some configuration information that not can be stored in a KCD file, for example message throttling and to only receive frames at data change.

static read(*filename*, *busname=None*)

Read configuration file in KCD format.

Parameters

- **filename** (*str*) – Full path to the KCD configuration file.
- **busname** (*str or None*) – Which bus name in the configuration file to use when reading. Defaults to `None` (using first alphabetically).

Returns a *Configuration* object.

Raises `CanException` – When failing to read and unpack the file. See *CanException*.

static write(*config*, *filename*)

Write configuration file in KCD frame_format (a type of XML file).

Parameters

- **config** (*Configuration* object) – Configuration details.
- **filename** (*str*) – Full path for output KCD file.

If the attribute 'config.busname' is None, then DEFAULT_BUSNAME will be used.

class can4python.**CanFrameDefinition**(*frame_id*, *name=''*, *dlc=8*, *cycletime=None*, *frame_format='standard'*)

A class for describing a CAN frame definition.

This object defines how the signals are laid out etc, but it does not hold the value of the frame or the values of the signals.

To add a *CanSignalDefinition* object to this *CanFrameDefinition* object:

```
myframedef1.signaldefinitions.append(mysignal1)
```

name

str

Frame name

signaldefinitions

list of CanSignalDefinition objects

Defaults to an empty list. See *CanSignalDefinition*.

receive_on_change_only

bool

Receive this frame only for updated data value (a data bitmask will be calculated). Defaults to False.

frame_id

int Frame ID. Should be in the range `0` to `0x7FF` for standard frame format, or in the range `0` to `0x1FFFFFFF` for extended frames.

dlc

int Number of bytes that should appear in the frame. Should be in the range `0` to `8`. Default: `8` bytes.

cycletime

numerical or None Shortest cycle time (in milliseconds) when sending. Defaults to `None`.

throttle_time

numerical or None Shortest update time (in milliseconds) for this frame when receiving. Defaults to `None` (no throttling).

frame_format

str Frame format. Should be `'standard'` or `'extended'`. Defaults to standard frame format.

producer_ids

set of strings Set of nodes (ECUs) that produce this frame. You can pass it a list (it will convert to a set).

get_descriptive_ascii_art()

Display an overview of the frame definition with its signals.

Returns A multi-line string.

get_signal_mask()
Calculate signal mask.

Returns a bytes object (length 8 bytes). A 1 in a position indicates that there is an interesting signal.

is_outbound(*ego_node_ids*)

Parameters **ego_node_ids** (*list/set of strings*) – List of nodes that this program will enact.

The frames with producer IDs matching some in the ego_node_ids list are considered outgoing/outbound frames.

Defaults to inbound, for example if no producer_ids or ego_node_ids are given.

Returns True if this frame is outbound (ie will be sent). Otherwise it is inbound (will be received).

class can4python.**CanSignalDefinition**(*signalname*, *startbit*, *numberofbits*, *scalingfactor=1*, *valueoffset=0*, *defaultvalue=None*, *unit=''*, *comment=''*, *minvalue=None*, *maxvalue=None*, *endianness='little'*, *signaltype='unsigned'*)

A class for describing a CAN signal definition (not the value of the signal).

signalname
str

Signal name

unit
str

Unit for the value. Defaults to `''`.

comment
str

A human-readable comment. Defaults to `''`.

Raises `CanException` – For wrong startbit, endianness etc. See *CanException*.

Warning: When setting the `numberofbits` attribute, then the attributes `endianness` and `startbit` must already be correct. Otherwise the error-checking mechanism might raise an error.
Also, the `minvalue`, `maxvalue` and `defaultvalue` should be within the limits defined by `numberofbits`, `scalingfactor`, `signaltype` etc.

Note: The byte order in a CAN frame is `0 1 2 3 4 5 6 7` (left to right)

The byte `0` in the CAN frame is sent first.

Bit order (significance) is decreasing from left to right. So in a byte, the rightmost bit is least significant.

Bit numbering in the CAN frame (standard bit numbering):

- In the first byte the least significant bit (rightmost, value 1) is named `0`, and the most significant bit (leftmost, value 128) is named `7`.
- In next byte, the least significant bit is named `8` etc.

This results in this bit numbering for the CAN frame:

```
7,6,5,4,3,2,1,0  15,14,13,12,11,10,9,8  23,22,21,20,19,18,17,16  31,30,29,28,27,26,25,24 etc.
Byte0            Byte1                  Byte2                   Byte3
```

Note: The start bit is given for the least significant bit in the signal, in standard bit numbering.

When a signal spans several bytes in the frame, the CAN frame can be constructed in two ways:

- In big-endian (Motorola, Network) byte order, the most significant byte is sent first.
- In little-endian (Intel) byte order, the least significant byte is sent first.

For example, an integer `0x0102030405060708` can be transmitted as big-endian or little-endian:

- Big-endian (most significant byte first): `01 02 03 04 05 06 07 08`
- Little-endian (least significant byte first): `08 07 06 05 04 03 02 01`

Note: If the signal is fitting into a single byte (not crossing any byte borders), there is no difference between big and little endian.

There is an alternate overall bit numbering scheme, known as "backwards" bit numbering.

Other variants (not used in this software):

- Startbit is sometimes given as the most significant bit.

endianness
str `'big'` or `'little'`. Defaults to using little endian (as the KCD file format defaults to little endian).

signaltype
str Should be `'unsigned'`, `'signed'`, `'single'` or `'double'`. (The last two are floats). Defaults to using unsigned signal type.

scalingfactor
numerical Scaling factor. Multiply with this value when extracting the signal from the CAN frame. Defaults to `1`. Should be positive.

valueoffset
numerical Offset. Add this value when extracting the signal from the CAN frame. Defaults to `0`.

startbit
int Position of least significant bit (in the standard bit numbering). Should be in the range `0` to `63` (inclusive).

defaultvalue
numerical or None Default value to send in frames if the signal value not is known. Defaults to `None` (Use the 'valueoffset' value).

minvalue
numerical or None Minimum allowed physical value. Defaults to `None` (no checking is done).

maxvalue
numerical or None Maximum allowed physical value. Defaults to `None` (no checking is done).

numberofbits
int Number of bits in the signal. Should be in the range `1` to `64` (inclusive).

get_descriptive_ascii_art()
Create a visual indication how the signal is located in the frame_definition.

Returns A multi-line string.

get_maximum_possible_value()
Get the largest value that technically could be sent with this signal.

The largest integer we can store is `2**numberofbits - 1`. Also the `scalingfactor`, `valueoffset` and the `signaltype` affect the result.

This method is used to calculate the allowed ranges for the attributes `minvalue`, ‘`maxvalue` and `defaultvalue`. When using the signal, you should respect the `minvalue` and `maxvalue`.

Returns The largest possible value (*numerical*).

See the twos_complement functions for discussion of value ranges for signed integers.

get_minimum_possible_value()
Get the smallest value that technically could be sent with this signal.

This method is used to calculate the allowed ranges for the attributes `minvalue`, ‘`maxvalue` and `defaultvalue`. When using the signal, you should respect the `minvalue` and `maxvalue`.

Returns The smallest possible value (*numerical*).

get_minimum_dlc()
Calculate the smallest number of bytes (DLC) that a frame must have, to be able to send this signal.

Returns Minimum DLC (int)

exception can4python.**CanException**
Base exception for CAN package

exception can4python.**CanTimeoutException**
Timeout for CAN package

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

5.1 Types of Contributions

5.1.1 Helping other users

If you successfully have used can4python, it highly appreciated if you help other users. This could for example be answering questions on Stack Overflow:

can4python on Stack Overflow

5.1.2 Report Bugs

Report bugs at https://github.com/caran/can4python/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.1.3 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” is open to whoever wants to implement it.

5.1.4 Implement Features

Look through the GitHub issues for features. Anything tagged with “feature” is open to whoever wants to implement it.

5.1.5 Write Documentation

can4python could always use more documentation, whether as part of the official can4python docs, in docstrings, or even on the web in blog posts, articles, and such.

5.1.6 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/caran/can4python/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.2 Get Started!

Ready to contribute? Here's how to set up *can4python* for local development.

1. Fork the *can4python* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/can4python.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv can4python
$ cd can4python/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 can4python tests
$ python setup.py test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for the Python versions mentioned in the setup.py file. Check https://travis-ci.org/caran/can4python/pull_requests and make sure that the tests pass for all supported Python versions.

5.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_cansignal
```

CHAPTER 6

Developer information

6.1 KCD file validation

The KCD file format is described here: https://github.com/julietkilo/kcd

There is an example file as well as a XML schema definition file (.XSD format).

Use some online XML schema validator service to make sure the imported and exported KCD files to/from can4python are valid.

6.2 Header for BCM communication

The BCM header has this format:

- opcode, u32 (4 bytes)
- flags, u32 (4 bytes)
- ival1_count, u32 (4 bytes)
- (possible paddding, 4 bytes)
- ival1_seconds, long (platform dependent, 4 or 8 bytes)
- ival1_useconds, long (platform dependent, 4 or 8 bytes)
- ival2_seconds, long (platform dependent, 4 or 8 bytes)
- ival2_useconds, long (platform dependent, 4 or 8 bytes)
- frame_id_std_ext, 32 bits (4 bytes)
- number_of_bcm_frames, u32 (4 bytes)
- (possible paddding, 4 bytes)

Use the 'native' byte alignment character to have automatic alignment between the different struct members. It is necessary to align the header end to 8 bytes, as there are CAN frames afterwards. Use zero occurances of an 8-byte struct member.

6.3 TODO

- Handle Labels (Enums, name constants) in KCD files. For example: PowerMode='EngineRunning'

- More usage examples, also with BCM.
- Abstract BCM more from CanBus.

6.4 Release procedure

Development is done in the 'dev' git branch.

To do a release:

- Change version number in the version.py file
- Update HISTORY.rst
- Run tests
- Verify that documentation builds for HTML and PDF works

Commit to dev, and push to master:

```
git add HISTORY.rst
git add can4python/version.py
git commit -m "Version 0.2.0"
git pull origin dev
git push origin dev
git checkout master
git pull origin master
git checkout dev
git merge master
git checkout master
git merge dev
git push origin master
```

Make a tag:

```
git tag -a 0.2.0 -m "Version 0.2.0"
git push origin --tags
```

Upload to PyPI:

```
python3 setup.py register
python3 setup.py sdist bdist_wheel upload
```

Update Readthedocs.io by clicking the "Build" button on the "Project Home" page. You need to build within a virtualenv on Readthedocs to have API documnentation working (adjust the project settings). Restrict Readthedocs.io to publish the "latest" branch of the documentation.

CHAPTER 7

Credits

7.1 Development Lead

- Jonas Berg <caranopensource@semcon.com>

7.2 Contributors

None yet. Why not be the first?

7.3 Acknowledgements

The Python file structure is set up using the Cookiecutter tool: https://github.com/audreyr/cookiecutter

The KCD file format is copyrighted by Jan-Niklas Meier and Jens Krueger. See https://github.com/julietkilo/kcd

Documentation is generated using the Sphinx tool: http://sphinx-doc.org/

CHAPTER 8

History

8.1 0.2.1 (2016-09-30)

- Adjust duocumentation build configuration to better fit readthedocs.io

8.2 0.2.0 (2016-09-30)

- Better support for other architectures. The broadcast manager (BCM) is now functional also on 32 bit ARM processors.
- Implemented the read-only properties config and read_bcm on the canbus object.
- Implemented the read-only property interfacename on the caninterface objects.
- Better checks for invalid settings.
- Improved repr() for canframe definition.
- Improved documentation.

8.3 0.1.1 (2015-11-16)

- Improved documentation

8.4 0.1.0 (2015-09-22)

- First release on GitHub.

CHAPTER 9

can4python

9.1 can4python package

9.1.1 Submodules

9.1.2 can4python.canbus module

class can4python.canbus.**CanBus**(*config*, *interfacename*, *timeout=None*, *use_bcm=False*)

Bases: object

CAN bus abstraction.

Uses Python 3.3 (or later) and the Linux SocketCAN interface.

The SocketCan Broadcast Manager (BCM) handles periodic sending of CAN frames, and can throttle incoming CAN frames to a slower rate, or to only receive frames when the data content has changed. Python 3.4 (or later) is required to use the BCM.

If you need to receive all frames, do not use the BCM.

Parameters

- **config** (*Configuration* object) – Configuration object describing what is happening on the bus.
- **interfacename** (*str*) – Name of the Linux SocketCan interface to use. For example 'vcan0' or 'can1'.
- **timeout** (*numerical*) – Timeout value in seconds for *recv_next_signals()*. Defaults to None (blocking read).
- **use_bcm** (*bool*) – True if the SocketCan Broadcast manager (BCM) should be used. Defaults to False.

classmethod from_kcd_file(*filename*, *interfacename*, *timeout=None*, *busname=None*, *use_bcm=False*, *ego_node_ids=None*)

Create a *CanBus*, using settings from a configuration file.

This is a convenience function, to avoid creating a separate configuration object.

Parameters

- **filename** (*str*) – Full path to existing configutation file, in the KCD file format.
- **interfacename** (*str*) – For example 'vcan0' or 'can1'.

- **timeout** (*numerical*) – Timeout value in seconds for *recv_next_signals()*. Defaults to None (the recv_next_signals call will be blocking).
- **busname** (*str or None*) – Which bus name in the messagedefinitions file to use. Defaults to None (using first alphabetically).
- **use_bcm** (*bool*) – True if the SocketCan Broadcast manager (BCM) should be used. Defaults to False.
- **ego_node_ids** (*set of strings*) – Set of nodes that this program will enact. You can also pass it a list.

config
Get the configuration (read-only). The configuration is set in the constructor.

use_bcm
Return True if BCM is used (read-only). Is set in the constructor.

init_reception()
Setup the CAN frame reception.

When using the RAW protocol, this enables filtering to reduce the input frame flow.

It works the opposite for the BCM protocol, where it explicitly subscribes to frame IDs.

recv_next_signals()
Receive one CAN frame, and unpack it to signal values.

Returns A dictionary of signal values, where the keys are the signalname (*str*) and the items are the values (*numerical*).

If the frame not is defined for this *CanBus* instance, an empty dictionary is returned.

Raises CanTimeoutException – If a timeout is defined and no frame is received. See *CanTimeoutException*.

recv_next_frame()
Receive one CAN frame. Returns a *CanFrame* object.

Raises CanTimeoutException – If a timeout is defined and no frame is received. See *CanTimeoutException*.

stop_reception()
Stop receiving, when using the BCM.

send_signals(**args*, ***kwargs*)
Send CAN signals in frames.

Parameters **signals_to_send** (*dict*) – The signal values to send_frame. The keys are the signalnames (*str*), and the items are the values (*numerical* or *None*). If the value is *None* the default value is used.

You can also use signal names as function arguments (keyword arguments). These are equal:

```
mycanbus.send_signals({"VehicleSpeed": 70.3, "EngineSpeed": 2821})
mycanbus.send_signals(VehicleSpeed=70.3, EngineSpeed=2821)
```

The signal names must be already defined for this *CanBus* instance.

Raises CanException – When failing to set signal value etc. See *CanException*.

start_sending_all_signals()
Start sending all configured frames, when using the BCM.

The default value for the signals are used, until updated via the *send_signals()* function.

If you do not use this *start_sending_all_signals()* method, the periodic transmission for each frame will start at first *send_signals()* call.

send_frame(*frame_to_send*)
Send a single CAN frame.

Parameters **frame_to_send** (*CanFrame*) – The frame to send.

stop_sending()
Stop periodic sending, when using the BCM.

stop()
Stop periodic sending and receiving, when using the BCM.

get_descriptive_ascii_art()
Display an overview of the *CanBus* object with frame definitions and signal definitions.

Returns A multi-line string.

write_configuration(*filename*)
Write configuration to file.

Parameters **filename** (*str*) – Full path to file with configuration.

Saves to an XML file in the KCD file format.

9.1.3 can4python.canframe module

class can4python.canframe.**CanFrame**(*frame_id*, *frame_data*, *frame_format='standard'*)
Bases: object

CAN frame with data. Does not know how the signals are laid out etc.

Raises CanException – For wrong frame ID. See *CanException*.

To find the DLC, use one of:

```
len(myframe)
len(myframe.frame_data)
```

classmethod **from_empty_bytes**(*frame_id*, *number_of_bytes*, *frame_format='standard'*)
Create a *CanFrame* with empty bytes.

Parameters

- **frame_id** (*int*) – CAN frame ID number
- **number_of_bytes** (*int*) – number of empty data bytes to initialize the frame with.
- **frame_format** (*str*) – Frame format. Should be 'standard' or 'extended'. Defaults to standard frame format.

classmethod **from_rawframe**(*rawframe*)
Create a *CanFrame* from a raw frame from the SocketCAN interface.

Parameters **rawframe** (*bytes*) – 16 bytes long, includes frame ID, frame format etc

frame_id
int CAN frame ID number

frame_data
bytes object 0-8 bytes of CAN data

frame_format
str Frame format. Should be `'standard'` or `'extended'`. Defaults to standard frame format.

get_signalvalue(*signaldefinition*)
Extract a signal value from the frame.

Parameters **signaldefinition** (*CanSignalDefinition* object) – The definition of the signal

Returns The extracted signal physical value (numerical).

set_signalvalue(*signaldefinition*, *physical_value=None*)
Set a signal physical_value in the frame.

Parameters

- **signaldefinition** (*CanSignalDefinition* object) – The definition of the signal
- **physical_value** – The physical_value (numerical) of the signal.

Raises `CanException` – For wrong startbit or values. See *CanException*.

unpack(*frame_definitions*)
Unpack the CAN frame, and return all signal values.

Parameters **frame_definitions** (*dict*) – The keys are frame_id (int) and the items are *CanFrameDefinition* objects.

Raises `CanException` – For wrong DLC. See *CanException*.

Returns A dictionary of signal values. The keys are the signalname (str) and the items are the values (numerical).

If the frame not is described in the 'frame_definitions', an empty dictionary is returned.

get_rawframe()
Returns a 16 bytes long 'bytes' object.

get_descriptive_ascii_art()
Create a visual indication of the frame data

Returns A multi-line string.

9.1.4 can4python.canframe_definition module

class `can4python.canframe_definition.`**CanFrameDefinition**(*frame_id*, *name=''*, *dlc=8*, *cycletime=None*, *frame_format='standard'*)

Bases: `object`

A class for describing a CAN frame definition.

This object defines how the signals are laid out etc, but it does not hold the value of the frame or the values of the signals.

To add a *CanSignalDefinition* object to this *CanFrameDefinition* object:

```
myframedef1.signaldefinitions.append(mysignal1)
```

name
str

Frame name

signaldefinitions
list of CanSignalDefinition objects

Defaults to an empty list. See `CanSignalDefinition`.

receive_on_change_only
bool

Receive this frame only for updated data value (a data bitmask will be calculated). Defaults to False.

frame_id
int Frame ID. Should be in the range `0` to `0x7FF` for standard frame format, or in the range `0` to `0x1FFFFFFF` for extended frames.

dlc
int Number of bytes that should appear in the frame. Should be in the range `0` to `8`. Default: `8` bytes.

cycletime
numerical or None Shortest cycle time (in milliseconds) when sending. Defaults to `None`.

throttle_time
numerical or None Shortest update time (in milliseconds) for this frame when receiving. Defaults to `None` (no throttling).

frame_format
str Frame format. Should be `'standard'` or `'extended'`. Defaults to standard frame format.

producer_ids
set of strings Set of nodes (ECUs) that produce this frame. You can pass it a list (it will convert to a set).

get_descriptive_ascii_art()
Display an overview of the frame definition with its signals.

Returns A multi-line string.

get_signal_mask()
Calculate signal mask.

Returns a bytes object (length 8 bytes). A 1 in a position indicates that there is an interesting signal.

is_outbound(*ego_node_ids*)

Parameters **ego_node_ids** (*list/set of strings*) – List of nodes that this program will enact.

The frames with producer IDs matching some in the ego_node_ids list are considered outgoing/outbound frames.

Defaults to inbound, for example if no producer_ids or ego_node_ids are given.

Returns True if this frame is outbound (ie will be sent). Otherwise it is inbound (will be received).

9.1.5 can4python.caninterface_bcm module

class can4python.caninterface_bcm.**SocketCanBcmInterface**(*interfacename*, *timeout=None*)

Bases: `object`

A Linux SocketCAN interface, using the Broadcast Manager (BCM) in the Linux kernel.

Parameters

- **interfacename** (*str*) – For example ‘vcan0’ or ‘can1’

- **timeout** (*numerical or None*) – Timeout value in seconds receiving BCM messages from the kernel. Defaults to None (blocking).

Raises

- CanException – For interface problems. See *CanException*.
- CanTimeoutException – At timeout. See *CanTimeoutException*.

interfacename
Get the interface name (read-only). The interface name is set in the constructor.

close()
Close the socket

recv_next_frame()
Receive one CAN frame.

Returns a *CanFrame* object.

send_frame(*input_frame*)
Send a single CAN frame (a *CanFrame* object)

setup_periodic_send(*input_frame*, *interval=None*, *restart_timer=True*)
Setup periodic transmission for a frame ID.

Parameters

- **input_frame** (*CanFrame* object) – The frame (including data and frame ID) to send periodically.
- **interval** (*float or None*) – Interval between consecutive transmissions (in milliseconds). Defaults to None (do not update the timing information).
- **restart_timer** (*bool*) – Start or restart the transmission timer. Defaults to True. Set this to false if you just would like to update the data to be sent, but not force reset of the transmission timer.

stop_periodic_send(*frame_id*, *frame_format='standard'*)
Stop the periodic transmission for this frame_id.

Parameters

- **frame_id** (*int*) – Frame ID
- **frame_format** (*str*) – Frame format. Should be 'standard' or 'extended'. Defaults to standard frame format.

setup_reception(*frame_id*, *frame_format='standard'*, *min_interval=None*, *data_mask=None*)
Setup reception for this frame_id (pretty much subscribe).

Parameters

- **frame_id** (*int*) – Frame ID
- **frame_format** (*str*) – Frame format. Should be 'standard' or 'extended'. Defaults to standard frame format.
- **min_interval** (*float or None*) – Minimum interval between received frames (in milliseconds). Useful for throttling rapid data streams. Defaults to None (no throttling). A min_interval of 0 corresponds to no throttling.
- **data_mask** (*bytes or None*) – Enable filtering on data changes. The mask is bytes object (length 8 bytes). Set the corresponding bits to 1 to detect data change in that loca-

tion. Defaults to None (data is not studied for changes, all incoming frames are given to the user).

stop_reception(*frame_id*, *frame_format='standard'*)
Disable the reception for this frame_id.

Parameters

- **frame_id** (*int*) – Frame ID
- **frame_format** (*str*) – Frame format. Should be `'standard'` or `'extended'`. Defaults to standard frame format.

_send_via_socket(*input_bytes*)
Send data on the object's socket. Handles OSError.

Parameters **input_bytes** (*byte*) – Data to send

can4python.caninterface_bcm.**_build_bcm_header**(*opcode*, *flags*, *interval*, *frame_id*, *frame_format*, *number_of_bcm_frames*)
Build a BCM message header.

Parameters

- **opcode** (*int*) – Command to the BCM
- **flags** (*int*) – Flags to the BCM
- **interval** (*float*) – Timing interval in milliseconds
- **frame_id** (*int*) – Frame ID.
- **frame_format** (*str*) – Frame format. Should be `'standard'` or `'extended'`
- **number_of_bcm_frames** (*int*) – Number of attached raw frames to the header.

Returns the header as bytes (length 56 bytes)

Note that 'interval' is the ival2 in Linux kernel documentation.

can4python.caninterface_bcm.**_parse_bcm_header**(*header*)
Parse a BCM message header.

Parameters **header** (*bytes*) – BCM header. Should have a length of 56 bytes.

Returns the tuple (opcode, flags, ival1_count, ival1, ival2, frame_id, frame_format, number_of_bcm_frames)

9.1.6 can4python.caninterface_raw module

class can4python.caninterface_raw.**SocketCanRawInterface**(*interfacename*, *timeout=None*)
Bases: `object`

A Linux Socket-CAN interface, using the RAW protocol to the Linux kernel.

Parameters

- **interfacename** (*str*) – For example 'vcan0' or 'can1'
- **timeout** (*numerical*) – Timeout value in seconds for *recv_next_signals()*. Defaults to None (blocking recv_next_signals).

Raises

- `CanException` – For interface problems. See *CanException*.

- CanTimeoutException – At timeout. See *CanTimeoutException*.

interfacename
Get the interface name (read-only). The interface name is set in the constructor.

close()
Close the socket

recv_next_frame()
Receive one CAN frame. Returns a *CanFrame* object.

send_frame(*input_frame*)
Send a can frame (a *CanFrame* object)

set_receive_filters(*framenumbers*)
Set the receive filters of the CAN interface (in the Linux kernel).

Parameters **framenumbers** (*list of int*) – The CAN IDs to listen for.

Uses one CAN receive filter per CAN ID. It is used only if listening to fewer than MAX_NUMBER_OF_RAW_RECEIVE_FILTERS CAN IDs, otherwise it is silently ignoring kernel CAN ID filering.

To see the filters that are applied (in Ubuntu):

```
cat /proc/net/can/rcv*
```

9.1.7 can4python.cansignal module

class can4python.cansignal.**CanSignalDefinition**(*signalname*, *startbit*, *numberofbits*, *scalingfactor=1*, *valueoffset=0*, *defaultvalue=None*, *unit=''*, *comment=''*, *minvalue=None*, *maxvalue=None*, *endianness='little'*, *signaltype='unsigned'*)

Bases: object

A class for describing a CAN signal definition (not the value of the signal).

signalname
str

Signal name

unit
str

Unit for the value. Defaults to ''.

comment
str

A human-readable comment. Defaults to ''.

Raises CanException – For wrong startbit, endianness etc. See *CanException*.

Warning: When setting the *numberofbits* attribute, then the attributes *endianness* and *startbit* must already be correct. Otherwise the error-checking mechanism might raise an error.
Also, the *minvalue*, *maxvalue* and *defaultvalue* should be within the limits defined by *numberofbits*, *scalingfactor*, *signaltype* etc.

Note: The byte order in a CAN frame is `0 1 2 3 4 5 6 7` (left to right)

The byte `0` in the CAN frame is sent first.

Bit order (significance) is decreasing from left to right. So in a byte, the rightmost bit is least significant.

Bit numbering in the CAN frame (standard bit numbering):

- In the first byte the least significant bit (rightmost, value 1) is named `0`, and the most significant bit (leftmost, value 128) is named `7`.
- In next byte, the least significant bit is named `8` etc.

This results in this bit numbering for the CAN frame:

```
7,6,5,4,3,2,1,0  15,14,13,12,11,10,9,8  23,22,21,20,19,18,17,16  31,30,29,28,27,26,25,24 etc.
Byte0            Byte1                  Byte2                    Byte3
```

Note: The start bit is given for the least significant bit in the signal, in standard bit numbering.

When a signal spans several bytes in the frame, the CAN frame can be constructed in two ways:

- In big-endian (Motorola, Network) byte order, the most significant byte is sent first.
- In little-endian (Intel) byte order, the least significant byte is sent first.

For example, an integer `0x0102030405060708` can be transmitted as big-endian or little-endian:

- Big-endian (most significant byte first): `01 02 03 04 05 06 07 08`
- Little-endian (least significant byte first): `08 07 06 05 04 03 02 01`

Note: If the signal is fitting into a single byte (not crossing any byte borders), there is no difference between big and little endian.

There is an alternate overall bit numbering scheme, known as "backwards" bit numbering.

Other variants (not used in this software):

- Startbit is sometimes given as the most significant bit.

endianness
str `'big'` or `'little'`. Defaults to using little endian (as the KCD file format defaults to little endian).

signaltype
str Should be `'unsigned'`, `'signed'`, `'single'` or `'double'`. (The last two are floats). Defaults to using unsigned signal type.

scalingfactor
numerical Scaling factor. Multiply with this value when extracting the signal from the CAN frame. Defaults to `1`. Should be positive.

valueoffset
numerical Offset. Add this value when extracting the signal from the CAN frame. Defaults to `0`.

startbit
int Position of least significant bit (in the standard bit numbering). Should be in the range `0` to `63` (inclusive).

defaultvalue
numerical or None Default value to send in frames if the signal value not is known. Defaults to `None` (Use the 'valueoffset' value).

minvalue
numerical or None Minimum allowed physical value. Defaults to `None` (no checking is done).

maxvalue
numerical or None Maximum allowed physical value. Defaults to `None` (no checking is done).

numberofbits
int Number of bits in the signal. Should be in the range `1` to `64` (inclusive).

get_descriptive_ascii_art()
Create a visual indication how the signal is located in the frame_definition.

Returns A multi-line string.

get_maximum_possible_value()
Get the largest value that technically could be sent with this signal.

The largest integer we can store is `2**numberofbits - 1`. Also the *scalingfactor*, *valueoffset* and the *signaltype* affect the result.

This method is used to calculate the allowed ranges for the attributes *minvalue*, '*maxvalue* and *defaultvalue*. When using the signal, you should respect the *minvalue* and *maxvalue*.

Returns The largest possible value (*numerical*).

See the twos_complement functions for discussion of value ranges for signed integers.

get_minimum_possible_value()
Get the smallest value that technically could be sent with this signal.

This method is used to calculate the allowed ranges for the attributes *minvalue*, '*maxvalue* and *defaultvalue*. When using the signal, you should respect the *minvalue* and *maxvalue*.

Returns The smallest possible value (*numerical*).

get_minimum_dlc()
Calculate the smallest number of bytes (DLC) that a frame must have, to be able to send this signal.

Returns Minimum DLC (int)

_check_signal_value_range(*attributename*, *value*)

_get_overview_string()
Generate an overview string, of length 64 bits.

Returns the tuple (outputstring, stopbit).

9.1.8 can4python.configuration module

class can4python.configuration.**Configuration**(*framedefinitions=None*, *busname=None*, *ego_node_ids=None*)

Bases: `object`

Configuration object for the things that happen on the CAN bus. It holds frame definitions (including signal definitions), the busname etc. See below.

framedefinitions
dict

The keys are the frame_id (*int*) and the items are the corresponding *CanFrameDefinition* objects.

busname
str or None

Which bus name in the configuration file to use when reading. Defaults to `None` (using first alphabetically).

ego_node_ids
set of strings Set of nodes that this program will enact. You can pass it a list (it will convert to a set).

get_descriptive_ascii_art()
Display an overview of the *Configuration* object with frame definitions and signals.

Returns A multi-line string.

add_framedefinition(*framedef*)
Add a frame definition to the configutation.

Parameters **framedef** (*CanFrameDefinition* object) – The frame definition to add.

This is a convenience function. These two alternatives are equal:

```
myconfig.add_framedefinition(framedef1)
myconfig.framedefinitions[framedef1.frame_id] = framedef1
```

set_throttle_times(*inputdict*)
Set throttle_time for some of the framedefinitions in the configuration object.

Parameters **inputdict** (*dict*) – The keys are the frame IDs (int) and the values are the throttle times (numerical or None) in milliseconds.

This is a convenience function. You can instead do like this for each frame:

```
myconfig.framedefinitions[myframe_id].throttle_time = mythrottletime
```

set_throttle_times_from_signalnames(*inputdict*)
Set throttle_time for some of the framedefinitions in the configuration object (via signal names)

Parameters **inputdict** (*dict*) – The keys are the signalnames (str) and the values are the throttle times (numerical or None) in milliseconds.

Note that the throttle_time is set on the framedefinition holding the signalname. It will also affect other signals on the same frame. Setting different throttle_times to signals on the same frame will give an undefined result.

This is a convenience function. You can instead do like this for each signalname:

```
(first find myframe_id for a given signalname)
myconfig.framedefinitions[myframe_id].throttle_time = mythrottletime
```

set_receive_on_change_only(*inputlist*)
Set receive_on_change_only for some of the framedefinitions in the configuration object.

Parameters **inputlist** (*list of ints*) – The frame IDs that should be received only when the data has changed.

This is a convenience function. You can instead do like this for each frame ID:

```
myconfig.framedefinitions[myframe_id].receive_on_change_only = True
```

set_receive_on_change_only_from_signalnames(*inputlist*)
Set receive_on_change_only for some of the framedefinitions in the configuration object (via signal names).

Parameters **inputlist** (*list of str*) – The signal names that should be received only when the data has changed.

Note that the receive_on_change_only is set on the framedefinition holding the signalname. It will also affect other signals on the same frame.

This is a convenience function. You can instead do like this for each signalname:

```
(first find myframe_id for a given signalname)
myconfig.framedefinitions[myframe_id].receive_on_change_only = True
```

find_frameid_from_signalname(*input_signalname*)
Find which frame_id a specific signal name belongs.

Parameters **input_signalname** (*str*) – signal name to search for.

Returns: The frame_id (int) in which the signal is located.

Raises CanException when the given signal name not is found.

9.1.9 can4python.constants module

9.1.10 can4python.exceptions module

exception can4python.exceptions.**CanException**
Bases: Exception

Base exception for CAN package

exception can4python.exceptions.**CanTimeoutException**
Bases: *can4python.exceptions.CanException*

Timeout for CAN package

exception can4python.exceptions.**CanNotFoundByKernelException**
Bases: *can4python.exceptions.CanException*

SocketCan in Linux kernel could probably not find the specified frame.

9.1.11 can4python.filehandler_kcd module

class can4python.filehandler_kcd.**FilehandlerKcd**
Bases: object

File handler for the KCD file format.

Note that only a subset of the KCD file format is implemented. These tags are read:

```
* Network definition: xmlns
  * Bus: name
    * Message: id, name, length, interval, format,
      * Producer:
        * NodeRef: id
      * Signal: endianness, length, name, offset
        * Value: type, slope, intercept, unit, min, max
        * Notes:
```

Further, there are is some configuration information that not can be stored in a KCD file, for example message throttling and to only receive frames at data change.

static read(*filename*, *busname=None*)
Read configuration file in KCD format.

Parameters

- **filename** (*str*) – Full path to the KCD configuration file.
- **busname** (*str or None*) – Which bus name in the configuration file to use when reading. Defaults to None (using first alphabetically).

Returns a *Configuration* object.

Raises CanException – When failing to read and unpack the file. See *CanException*.

static write(*config*, *filename*)
Write configuration file in KCD frame_format (a type of XML file).

Parameters

- **config** (*Configuration* object) – Configuration details.
- **filename** (*str*) – Full path for output KCD file.

If the attribute 'config.busname' is None, then DEFAULT_BUSNAME will be used.

9.1.12 can4python.utilities module

can4python.utilities.**calculate_backward_bitnumber**(*normal_bitnumber*)
Calculate the bit position in the "backward" numbering format.

Parameters **normal_bitnumber** (*int*) – bit position in the standard format.

Raises CanException – For wrong bitnumber. See *CanException*.

Returns The bit position (int) in the "backward" numbering format.

The "backward" is a numbering scheme where the bits are numbered:

```
63,62 61,60,59,58,57,56  55,54,53,52,51,50,49,48  47,46 etc
Byte0                    Byte1                    Byte2
```

In full detail:

```
66665555 55555544 44444444 33333333 33222222 22221111 111111
32109876 54321098 76543210 98765432 10987654 32109876 54321098 76543210
Byte0    Byte1    Byte2    Byte3    Byte4    Byte5    Byte6    Byte7
```

For reference, the standard bit numbering is:

```
7,6,5,4,3,2,1,0  15,14,13,12,11,10,9,8  23,22,21,20,19,18,17,16  31,30,29,28,27,26,25,24 etc.
Byte0            Byte1                  Byte2                    Byte3
```

In full detail:

```
         111111   22221111 33222222 33333333 44444444 55555544 66665555
76543210 54321098 32109876 10987654 98765432 76543210 54321098 32109876
Byte0    Byte1    Byte2    Byte3    Byte4    Byte5    Byte6    Byte7
```

can4python.utilities.**calculate_normal_bitnumber**(*backward_bitnumber*)
Calculate the bitnumber, from the bitnumber in backwards numbering scheme.

This is the inverse of *calculate_backward_bitnumber()*.

Parameters **backward_bitnumber** (*int*) – The bitnumber (in backwards numbering scheme)

Raises CanException – For wrong bitnumber. See *CanException*.

Returns The bit position (int) in the standard format.

can4python.utilities.**generate_bit_byte_overview**(*inputstring*, *number_of_indent_spaces=4*, *show_reverse_bitnumbering=False*)

Generate a nice overview of a CAN frame.

Parameters

- **inputstring** (*str*) – String that should be printed. Should be 64 characters long.
- **number_of_indent_spaces** (*int*) – Size of indentation

Raises ValueError when *inputstring* has wrong length.

Returns A multi-line string.

can4python.utilities.**generate_can_integer_overview**(*value*)

Generate a nice overview of an integer, interpreted as a CAN frame/fr_def.

Parameters **value** (*int*) – Integer representing the data of a CAN frame

Returns A multi-line string.

can4python.utilities.**can_bytes_to_int**(*input_bytes*)

Convert bytes to an integer (after padding the bytes).

Parameters **input_bytes** (*bytes object*) – holds 0-8 bytes of data. Will be padded with empty bytes on right side.

Returns An integer corresponding to 8 bytes of data.

Note: An input of b”“” will be padded to b”“”. This corresponds to an integer of 72057594037927936.

can4python.utilities.**int_to_can_bytes**(*dlc*, *dataint*)

Convert an integer to 8 bytes, and cut it from the left according to the dlc.

Parameters

- **dlc** (*int*) – how many bytes it should be encoded into
- **dataint** (*int*) – holds 8 bytes of data

Returns a bytes object: 0-8 bytes of CAN data

For example a dataint value of 1 is converted to b’“’. If the dlc is given as 1, then the return value will be b’‘.

can4python.utilities.**twos_complement**(*value*, *bits*)

Calculate two’s complement for a value.

Parameters

- **value** (*int*) – input value (positive or negative)
- **bits** (*int*) – field size

Returns a positive integer. If in the upper part of the range, it should be interpreted as negative.

The allowed input value range is -2**(bits-1) to +2**(bits-1)-1. For example, 8 bits gives a range of -128 to +127.

can4python.utilities.**from_twos_complement**(*value*, *bits*)

Calculate the inverse (?) of two’s complement for a value.

Parameters

- **value** (*int*) – input value (positive)

- **bits** (*int*) – field size

Returns a positive or negative integer, in the range range is -2**(bits-1) to +2**(bits-1)-1. For example, 8 bits gives an output range of -128 to +127.

can4python.utilities.**split_seconds_to_full_and_part**(*seconds_float*)
Split a time value into full and fractional parts.

Parameters **seconds_float** (*float*) – Number of seconds

Returns (seconds_full, useconds) which both are integers. They represent the time in full seconds and microseconds respectively.

can4python.utilities.**check_frame_id_and_format**(*frame_id*, *frame_format*)
Check the validity of frame_id.

Parameters

- **frame_id** (*int*) – frame_id to be checked
- **frame_format** (*str*) – Frame format. Should be 'standard' or 'extended'.

can4python.utilities.**get_busvalue_from_bytes**(*input_bytes*, *endianness*, *numberofbits*, *startbit*)
Get the busvalue from bytes.

Parameters

- **input_bytes** (*bytes object*) – up to 8 bytes of data
- **endianness** (*str*) – 'big' or 'little'
- **numberofbits** (*int*) – Number of bits in the signal
- **startbit** (*int*) – LSB in normal bit numbering

Returns the bus value, which is the bits interpreted as an unsigned integer. For example '0110' is interpreted as the unsigned integer 6. Later, it will then be converted to whatever (maybe signed or unsigned integer).

can4python.utilities.**get_shiftedvalue_from_busvalue**(*input_value*, *endianness*, *numberofbits*, *startbit*)
Get the shifted value from the bus value.

Parameters

- **input_value** (*int*) – Integer corresponding to the bus value.
- **endianness** (*str*) – 'big' or 'little'
- **numberofbits** (*int*) – Number of bits in the signal
- **startbit** (*int*) – LSB in normal bit numbering

Returns the shifted value, which later will be put into the frame using AND/OR operations together with a mask.

Earlier the physical value has been converted to a shifted value and then to a bus value. For example, a bus value input_value '0110' is interpreted as the unsigned integer 6.

9.1.13 can4python.version module

9.1.14 Module contents

CHAPTER 10

Unittests

10.1 test_cansignal

Tests for *cansignal* module.

class tests.test_cansignal.**TestCanSignal**(*methodName='runTest'*)

setUp()

testConstructor()

testConstructorWrongValues()

testProperties()

testPropertiesWrongValues()

testRepr()

testGetDescriptiveAsciiArt()

testMaximumPossibleValueGet()

testMinimumPossibleValueGet()

testGetMinimumDlc()

10.2 test_canframedefinition

Tests for *canframe_definition* module.

class tests.test_canframedefinition.**TestCanFrameDefinition**(*methodName='runTest'*)

setUp()

testConstructor()

testConstructorCycletime()

testConstructorCycletimeNone()

testConstructorNamedArguments()

testConstructorWrongValues()

testProperties()

testPropertiesWrongValues()

testIsOutbound()

testIsOutboundWrongType()

testGetSignalMask()

testRepr()

testReprThrottling()

testReprNoSignals()

testGetDescriptiveAsciiArt()

10.3 test_configuration

Tests for *configuration* module.

class tests.test_configuration.**TestConfiguration**(*methodName='runTest'*)

setUp()

testConstructor()

testRepr()

testProperties()

testPropertiesWrongValues()

testSetThrottleTimes()

testSetThrottleTimesWrongValues()

testSetThrottleTimesFromSignalnames()

testSetThrottleTimesFromSignalnamesWrongValues()

testSetReceiveOnChangeOnly()

testSetReceiveOnChangeOnlyWrongValue()

testSetReceiveOnChangeOnlyFromSignalnames()

testSetReceiveOnChangeOnlyFromSignalnamesWrongValues()

testGetDescriptiveAsciiArt()

testAddFramedefinition()

10.4 test_filehandler_kcd

Tests for *filehandler_kcd* module.

class tests.test_filehandler_kcd.**TestConfiguration**(*methodName='runTest'*)

OUTPUT_FILENAME_1 = 'test_out_1_TEMPORARY.kcd'

OUTPUT_FILENAME_2 = 'test_out_2_TEMPORARY.kcd'

OUTPUT_FILENAME_3 = 'test_out_3_TEMPORARY.kcd'

OUTPUT_FILENAME_10 = 'test_out_10_TEMPORARY.kcd'

setUp()

tearDown()

testReadKcdFile()

testReadKcdFileFaulty()

testReadKcdFileWrongBusname()

testReadKcdFileNoBusnameGiven()

testWriteKcdFile()

testSaveLoadedConfigurationToFile()

testWriteKcdFileNoBusnameGiven()

testWriteKcdFileNoProducerGiven()

10.5 test_utilities

Tests for *utilities* module.

class tests.test_utilities.**TestCalculateBackwardBitnumber**(*methodName='runTest'*)

knownValues = ((0, 56), (1, 57), (7, 63), (56, 0), (63, 7))

test_known_values()

testWrongInputValue()

class tests.test_utilities.**TestCalculateNormalBitnumber**(*methodName='runTest'*)

knownValues = ((0, 56), (1, 57), (7, 63), (56, 0), (63, 7))

test_known_values()

testWrongInputValue()

class tests.test_utilities.**TestSanityBitnumber**(*methodName='runTest'*)

test_known_values()

class tests.test_utilities.**TestGenerateBitByteOverview**(*methodName='runTest'*)

test_known_values()

testWrongInputValue()

class tests.test_utilities.**TestGenerateCanIntegerOverview**(*methodName='runTest'*)

test_known_values()

class tests.test_utilities.**TestBytesToInt**(*methodName='runTest'*)

knownValues = ((b'\x00', 0), (b'\x01', 72057594037927936), (b'\x01\x00\x00\x00\x00\x00\x00\x00', 72057594037927936),

testKnownValues()

class tests.test_utilities.**TestIntToBytes**(*methodName='runTest'*)

knownValues = ((1, 0, b''), (1, 1, b'\x00'), (1, 2, b'\x00\x00'))

testKnownValues()

class tests.test_utilities.**TestTwosComplement**(*methodName='runTest'*)

knownValues = ((-4, 3, 4), (-3, 3, 5), (-2, 3, 6), (-1, 3, 7), (0, 3, 0), (1, 3, 1), (2, 3, 2), (3, 3, 3), (-128, 8, 128), (-127, 8, 129), (-1

testKnownValues()

testWrongInputValue()

class tests.test_utilities.**TestFromTwosComplement**(*methodName='runTest'*)

knownValues = ((-4, 3, 4), (-3, 3, 5), (-2, 3, 6), (-1, 3, 7), (0, 3, 0), (1, 3, 1), (2, 3, 2), (3, 3, 3), (-128, 8, 128), (-127, 8, 129), (-1

testKnownValues()

testWrongInputValue()

class tests.test_utilities.**TestTwosComplementSanity**(*methodName='runTest'*)

testSanity()

class tests.test_utilities.**TestSplitSeconds**(*methodName='runTest'*)

knownValues = ((0, 0, 0), (1e-06, 0, 1), (0.001, 0, 1000), (0.02, 0, 20000), (0.25, 0, 250000), (0.33, 0, 330000), (0.9, 0, 900000

testKnownValues()

testWrongInputValue()

class tests.test_utilities.**TestCheckFrameId**(*methodName='runTest'*)

testKnownValues()

testWrongInputValue()

testWrongInputType()

class tests.test_utilities.**TestGetBusvalueFromBytes**(*methodName='runTest'*)

knownValues = ((b'\x00\x00\x00\x00\x00\x00\x00\x01', 'big', 8, 56, 1), (b'\x00\x00\x00\x00\x00\x00\x00\x06', 'big', 4, 56,

testKnownValues()

class tests.test_utilities.**TestGetShiftedvalueFromBusvalue**(*methodName='runTest'*)

knownValues = ((b'\x00\x00\x00\x00\x00\x00\x00\x01', 'big', 8, 56, 1), (b'\x00\x00\x00\x00\x00\x00\x00\x06', 'big', 4, 56,

testKnownValues()

10.6 test_canframe

Tests for *canframe* module.

class tests.test_canframe.**TestCanFrame**(*methodName='runTest'*)

setUp()

testConstructor()

testConstructorNamedArguments()

testConstructorFromEmptyBytes()

testConstructorFromRawframes()

testWrongConstructor()

testWrongConstructorFromEmptyBytes()

testWrongConstructorFromRawframe()

testFrameidGet()

testFrameidSet()

testFrameidSetWrongValue()

testFrameFormatGet()

testFrameFormatSet()

testFrameFormatSetWrongValue()

testFramedataGet()

testFramedataSet()

testFramedataSetWrongValue()

testSignalvalueSet()

testSignalvalueSetSigned()

testSignalvalueSetSingle()

testSignalvalueSetSingleLittle()

testSignalvalueGetSingle()

testSignalvalueGetSingleLittle()

testSignalvalueSetDouble()

testSignalvalueSetDoubleLittle()

testSignalvalueGetDouble()

testSignalvalueGetDoubleLittle()

testSignalvalueSetTooShortFrame()

testSignalvalueGetSetMin()

testSignalvalueGetSetMax()

testSignalvalueSetWrongValue()

testSignalvalueGet()

testSignalvalueGetSigned()

testGetRawFrameStandard()

testGetRawFrameExtended()

testUnpack()

testUnpackWrongFrameId()

testUnpackWrongFramelength()

testRepr()

testLen()

testGetDescriptiveAsciiArt()

10.7 test_caninterface_raw

Tests for *caninterface_raw* module.

Notes

A virtual CAN interface 'vcan' must be enabled for this test. See enable_virtual_can_bus(). Must be run as sudo.

tests.test_caninterface_raw.**enable_virtual_can_bus**()

tests.test_caninterface_raw.**disable_virtual_can_bus**()

class tests.test_caninterface_raw.**TestSocketCanRawInterface**(*methodName='runTest'*)

NUMBER_OF_LOOPS = 10000

FRAME_SENDER_SPACING_MILLISECONDS = 0.1

FRAME_ID_RECEIVE = 4

FRAME_ID_SEND = 1

FRAME_NUMBER_OF_DATABYTES = 8

NONEXISTING_CAN_BUS_NAME = 'vcan8'

setUp()

tearDown()

start_can_frame_sender()
Send CAN frames using the cangen command.

testConstructor()

testConstructorWrongValue()

testConstructorWrongType()

testConstructorSeveralInterfaces()

testCreateNonExistingBus()

testWriteToInterfacenameAttribute()

testRepr()

testReceiveData()

testReceiveSpeed()

testReceiveNoData()

testReceiveClosedBus()

testReceiveClosedInterface()

testSend()

testSendClosedBus()

testSendClosedInterface()

testTooFewTooManyFiltersDefined()

10.8 test_caninterface_bcm

Tests for *caninterface_bcm* module.

Notes

A virtual CAN interface 'vcan' must be enabled for this test. See enable_virtual_can_bus(). Must be run as sudo.

class tests.test_caninterface_bcm.**TestSocketCanBcmInterface**(*methodName='runTest'*)

NUMBER_OF_LOOPS = 10000

FRAME_SENDER_SPACING_MILLISECONDS = 0.1

FRAME_ID_RECEIVE = 4

FRAME_ID_SEND = 1

FRAME_NUMBER_OF_DATABYTES = 8

NONEXISTING_CAN_BUS_NAME = 'vcan8'

NONEXISTING_FRAME_ID = 22

setUp()

tearDown()

start_can_frame_sender(*interval_milliseconds=0.1*)
Send CAN frames using the cangen command. Unlimited number of frames.

testConstructor()

testConstructorWrongValue()

testConstructorWrongType()

testConstructorSeveralInterfaces()

testCreateNonExistingBus()

testWriteToInterfacenameAttribute()

testRepr()

testSetupReception()

testSetupReceptionWrongValue()

testSetupReceptionWrongType()

testReceiveData()

testReceiveStoppedReception()

testReceiveWrongId()

testReceiveSpeed()

testReceiveSpeedThrottled()

testReceiveDataChanged()
Verify that data change filtering works, by measuring time to receive a small number of frames from a larger data flow.

testReceiveDataChangedShorterDlcThanMask()
Verify that filtering works also when the input frame is shorter (3 bytes) than the data mask (8 bytes).

testReceiveDlcChanged()
Verify that we are receiving frames where each frame is longer (or no data) than the previous.

testReceiveNoData()

testReceiveClosedBus()

testReceiveClosedInterface()

testSendSingleFrameWrongType()

testSendSingleFrame()

testSetupPeriodicSendWrongValue()

testSetupPeriodicSendWrongType()

testSendPeriodicAndChangeFrameAndStop()
Start periodic CAN transmission, update frame data (not interval), and finally stop transmission.

testSendPeriodicAndChangeFrameAndStopExtended()
Start periodic CAN transmission, update frame data (not interval), and finally stop transmission.

testStopNonexistingPeriodicTask()

testSendClosedBus()

testSendClosedInterface()

10.9 test_canbus

Tests for *canbus* module.

Notes

A virtual CAN interface 'vcan' must be enabled for this test. Must be run as sudo.

class tests.test_canbus.**TestCanBus**(*methodName='runTest'*)

NUMBER_OF_LOOPS = 1000

FRAME_SENDER_SPACING_MILLISECONDS = 1

FRAME_NUMBER_OF_DATABYTES = 8

OUTPUT_FILENAME_4 = 'test_out_4_TEMPORARY.kcd'

OUTPUT_FILENAME_5 = 'test_out_5_TEMPORARY.kcd'

OUTPUT_FILENAME_6 = 'test_out_6_TEMPORARY.kcd'

OUTPUT_FILENAME_7 = 'test_out_7_TEMPORARY.kcd'

setUp()

tearDown()

testUseBcmAttribute()

testWriteToBcmAttribute()

testWriteToConfigAttribute()

testConstructor()

testReadConfigFromFile()
See test_configuration.py and others for more complete configuration read testing

testRepr()

testGetDescriptiveAsciiArt()

testSendRaw()

testSendRawKeywordArguments()

testSendBcm()

testSendBcmFrame()

testSendSpeedAndDetectAllRaw()

testSendSpeedAndDetectAllBcm()

testPeriodicSendingUpdateSignalsAndStop()

testStartSendingAllSignals()

testSendWrongSignal()

testSendWrongSignalValue()

testUsingBcmCommandsForRawInterface()

testInitReception()

testReceiveRaw()

testReceiveBcmAndStop()

testReceiveBcmFrameAndStop()

testReceiveSpeedRaw()

testReceiveSpeedBcm()

testReceiveAllSentFramesRaw()

testReceiveAllSentFramesBcm()

testReceiveNoData()

testSaveDefinitionToFileRaw()

testSaveLoadedDefinitionToFileRaw()

testSaveDefinitionToFileBcm()

testSaveLoadedDefinitionToFileBcm()

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

C

Symbols

A

B

C

D

E

F

G

I

M

N

P

R

S

T

U

V

W