cantools Documentation

Release 39.4.3

Erik Moqvist

Jan 15, 2024

CONTENTS

CHAPTER

ONE

ABOUT

CAN BUS tools in Python 3.

- DBC, KCD, SYM, ARXML 3&4 and CDD file parsing.
- CAN message encoding and decoding.
- Simple and extended signal multiplexing.
- Diagnostic DID encoding and decoding.
- `candump` output decoder.
- Node tester.
- *C* source code generator.
- CAN bus monitor.
- Graphical plots of signals.

Project homepage: https://github.com/cantools/cantools

Documentation: https://cantools.readthedocs.io

CHAPTER

TWO

INSTALLATION

```
python3 -m pip install cantools
```

CHAPTER

THREE

EXAMPLE USAGE

3.1 Scripting

The example starts by parsing a small DBC-file and printing its messages and signals.

```
>>> import cantools
>>> from pprint import pprint
>>> db = cantools.database.load_file('tests/files/dbc/motohawk.dbc')
>>> db.messages
[message('ExampleMessage', 0x1f0, False, 8, 'Example message used as template in␣
↪MotoHawk models.')]
>>> example_message = db.get_message_by_name('ExampleMessage')
>>> pprint(example_message.signals)
[signal('Enable', 7, 1, 'big_endian', False, 1.0, 0, 0.0, 0.0, '-', False, None, {0:
↪'Disabled', 1: 'Enabled'}, None),
 signal('AverageRadius', 6, 6, 'big_endian', False, 0.1, 0, 0.0, 5.0, 'm', False, None,␣
↪None, ''),
 signal('Temperature', 0, 12, 'big_endian', True, 0.01, 250, 229.53, 270.47, 'degK',␣
↪False, None, None, None)]
```

The example continues encoding a message and sending it on a CAN bus using the python-can package.

```
>>> import can
>>> can_bus = can.interface.Bus('vcan0', bustype='socketcan')
>>> data = example_message.encode({'Temperature': 250.1, 'AverageRadius': 3.2, 'Enable':␣
↪1})
>>> message = can.Message(arbitration_id=example_message.frame_id, data=data)
>>> can_bus.send(message)
```

Alternatively, a message can be encoded using the encode_message() method on the database object.

The last part of the example receives and decodes a CAN message.

```
>>> message = can_bus.recv()
>>> db.decode_message(message.arbitration_id, message.data)
{'AverageRadius': 3.2, 'Enable': 'Enabled', 'Temperature': 250.09}
```

See examples for additional examples.

3.2 Command line tool

3.2.1 The decode subcommand

Decode CAN frames captured with the Linux program `candump`.

```
$ candump vcan0 | python3 -m cantools decode tests/files/dbc/motohawk.dbc
  vcan0  1F0   [8]  80 4A 0F 00 00 00 00 00 ::
ExampleMessage(
    Enable: 'Enabled' -,
    AverageRadius: 0.0 m,
    Temperature: 255.92 degK
)
  vcan0  1F0   [8]  80 4A 0F 00 00 00 00 00 ::
ExampleMessage(
    Enable: 'Enabled' -,
    AverageRadius: 0.0 m,
    Temperature: 255.92 degK
)
  vcan0  1F0   [8]  80 4A 0F 00 00 00 00 00 ::
ExampleMessage(
    Enable: 'Enabled' -,
    AverageRadius: 0.0 m,
    Temperature: 255.92 degK
)
```

Alternatively, the decoded message can be printed on a single line:

```
$ candump vcan0 | python3 -m cantools decode --single-line tests/files/dbc/motohawk.dbc
  vcan0  1F0   [8]  80 4A 0F 00 00 00 00 00 :: ExampleMessage(Enable: 'Enabled' -,␣
↪AverageRadius: 0.0 m, Temperature: 255.92 degK)
  vcan0  1F0   [8]  80 4A 0F 00 00 00 00 00 :: ExampleMessage(Enable: 'Enabled' -,␣
↪AverageRadius: 0.0 m, Temperature: 255.92 degK)
  vcan0  1F0   [8]  80 4A 0F 00 00 00 00 00 :: ExampleMessage(Enable: 'Enabled' -,␣
↪AverageRadius: 0.0 m, Temperature: 255.92 degK)
```

3.2.2 The plot subcommand

The plot subcommand is similar to the decode subcommand but messages are visualized using matplotlib instead of being printed to stdout.

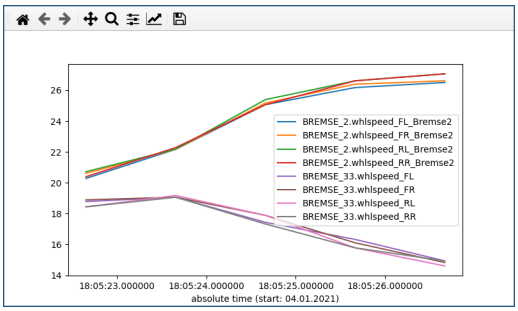
```
$ candump -l vcan0
$ cat candump-2021-01-04_180521.log
(1609779922.655421) vcan0 00000343#B204B9049C049C04
(1609779922.655735) vcan0 0000024A#120527052E051905
(1609779923.657524) vcan0 00000343#C404C404CB04C404
(1609779923.658086) vcan0 0000024A#8B058B058B059205
(1609779924.659912) vcan0 00000343#5C04790479045504
(1609779924.660471) vcan0 0000024A#44064B0659064406
(1609779925.662277) vcan0 00000343#15040704F203F203
(1609779925.662837) vcan0 0000024A#8B069906A706A706
```

(continues on next page)

(continued from previous page)

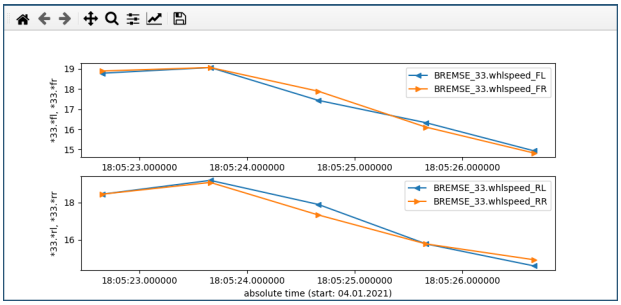
```
(1609779926.664191) vcan0 00000343#BC03B503A703BC03
(1609779926.664751) vcan0 0000024A#A006A706C406C406

$ cat candump-2021-01-04_180521.log | python3 -m cantools plot tests/files/dbc/abs.dbc
```



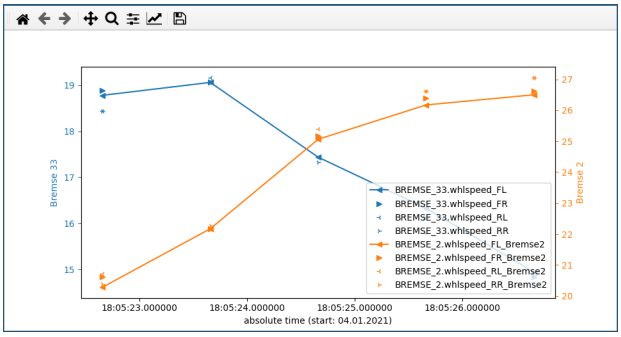
If you don't want to show all signals you can select the desired signals with command line arguments. A * can stand for any number of any character, a ? for exactly one arbitrary character. Signals separated by a - are displayed in separate subplots. Optionally a format can be specified after a signal, separated by a colon.

```
$ cat candump-2021-01-04_180521.log | python3 -m cantools plot tests/files/dbc/abs.dbc
↪'*33.*fl:-<' '*33.*fr:->' - '*33.*rl:-<' '*33.*rr:->'
```



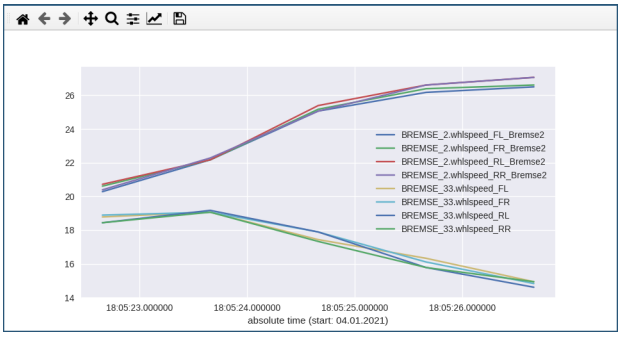
Signals with a different range of values can be displayed in the same subplot on different vertical axes by separating them with a comma.

```
$ cat candump-2021-01-04_180521.log | cantools plot --auto-color tests/files/dbc/abs.dbc␣
↪-- \
  --ylabel 'Bremse 33' '*_33.*fl*:-<' '*_33.*fr*:>' '*_33.*rl*:3' '*_33.*rr*:4' , \
  --ylabel 'Bremse 2' '*_2.*fl*:-<' '*_2.*fr*:>' '*_2.*rl*:3' '*_2.*rr*:4'
```



Matplotlib comes with different preinstalled styles that you can use:

```
$ cat candump-2021-01-04_180521.log | cantools plot tests/files/dbc/abs.dbc --style␣
↪seaborn
```



You can try all available styles with

```
$ cantools plot --list-styles . | sed -n '/^- /s/^- //p' | while IFS= read -r style; do
    cat candump-2021-01-04_180521.log | cantools plot tests/files/dbc/abs.dbc --style "
↪$style" --title "--style '$style'"
  done
```

For more information see

```
$ python3 -m cantools plot --help
```

Note that by default matplotlib is not installed with cantools. But it can be by specifying an extra at installation:

```
$ python3 -m pip install cantools[plot]
```

3.2.3 The dump subcommand

Dump given database in a human readable format:

```
$ python3 -m cantools dump tests/files/dbc/motohawk.dbc
================================= Messages =================================

  ------------------------------------------------------------------------

  Name:       ExampleMessage
  Id:         0x1f0
  Length:     8 bytes
  Cycle time: - ms
  Senders:    PCM1
  Layout:

                            Bit

               7   6   5   4   3   2   1   0
             +---+---+---+---+---+---+---+---+
           0 |<-x|<----------------------x|<--|
             +---+---+---+---+---+---+---+---+
               |                       +-- AverageRadius
```

(continues on next page)

(continued from previous page)

```
            +-- Enable
          +---+---+---+---+---+---+---+---+
        1 |-------------------------------|
          +---+---+---+---+---+---+---+---+
        2 |----------x|   |   |   |   |   |
    B     +---+---+---+---+---+---+---+---+
    y               +-- Temperature
    t     +---+---+---+---+---+---+---+---+
    e   3 |   |   |   |   |   |   |   |   |
          +---+---+---+---+---+---+---+---+
        4 |   |   |   |   |   |   |   |   |
          +---+---+---+---+---+---+---+---+
        5 |   |   |   |   |   |   |   |   |
          +---+---+---+---+---+---+---+---+
        6 |   |   |   |   |   |   |   |   |
          +---+---+---+---+---+---+---+---+
        7 |   |   |   |   |   |   |   |   |
          +---+---+---+---+---+---+---+---+

  Signal tree:

    -- {root}
       +-- Enable
       +-- AverageRadius
       +-- Temperature

  Signal choices:

    Enable
        0 Disabled
        1 Enabled

  ------------------------------------------------------------------------------
```

3.2.4 The list subcommand

Print all information of a given database in a human readable format. This is very similar to the "dump" subcommand, but the output is less pretty, slightly more comprehensive and easier to parse by shell scripts:

```
$ python3 -m cantools list -a tests/files/dbc/motohawk.dbc
ExampleMessage:
  Comment[None]: Example message used as template in MotoHawk models.
  Frame ID: 0x1f0 (496)
  Size: 8 bytes
  Is extended frame: False
  Signals:
    Enable:
      Type: Integer
      Start bit: 7
      Length: 1 bits
      Unit: -
```

(continues on next page)

(continued from previous page)

```
    Is signed: False
    Named values:
      0: Disabled
```

3.2.5 The generate C source subcommand

Generate *C* source code from given database.

The generated code contains:

- Message structs.
- Message pack and unpack functions.
- Signal encode and decode functions.
- Frame id, length, type, cycle time and signal choices defines.

Known limitations:

- The maximum signal size is 64 bits, which in practice is never exceeded.

Below is an example of how to generate C source code from a database. The database is `tests/files/dbc/motohawk.dbc`.

```
$ python3 -m cantools generate_c_source tests/files/dbc/motohawk.dbc
Successfully generated motohawk.h and motohawk.c.
```

See motohawk.h and motohawk.c for the contents of the generated files.

In this example we use `--use-float` so floating point numbers in the generated code are single precision (`float`) instead of double precision (`double`).

```
$ python3 -m cantools generate_c_source --use-float tests/files/dbc/motohawk.dbc
Successfully generated motohawk.h and motohawk.c.
```

In the next example we use `--database-name` to set a custom namespace for all generated types, defines and functions. The output file names are also changed by this option.

```
$ python3 -m cantools generate_c_source --database-name my_database_name tests/files/dbc/
↪motohawk.dbc
Successfully generated my_database_name.h and my_database_name.c.
```

See my_database_name.h and my_database_name.c for the contents of the generated files.

In the next example we use `--no-floating-point-numbers` to generate code without floating point types, i.e. `float` and `double`.

```
$ python3 -m cantools generate_c_source --no-floating-point-numbers tests/files/dbc/
↪motohawk.dbc
Successfully generated motohawk.h and motohawk.c.
```

See motohawk_no_floating_point_numbers.h and motohawk_no_floating_point_numbers.c for the contents of the generated files.

In the last example `--node` is used to generate message pack functions only for messages sent by the specified node and unpack functions only for messages with its signal receivers belonging to that node.

```
$ cantools generate_c_source tests/files/dbc/motohawk.dbc --node PCM1
Successfully generated motohawk.h and motohawk.c.
```

See motohawk_sender_node.h and motohawk_sender_node.c for the contents of the generated files.

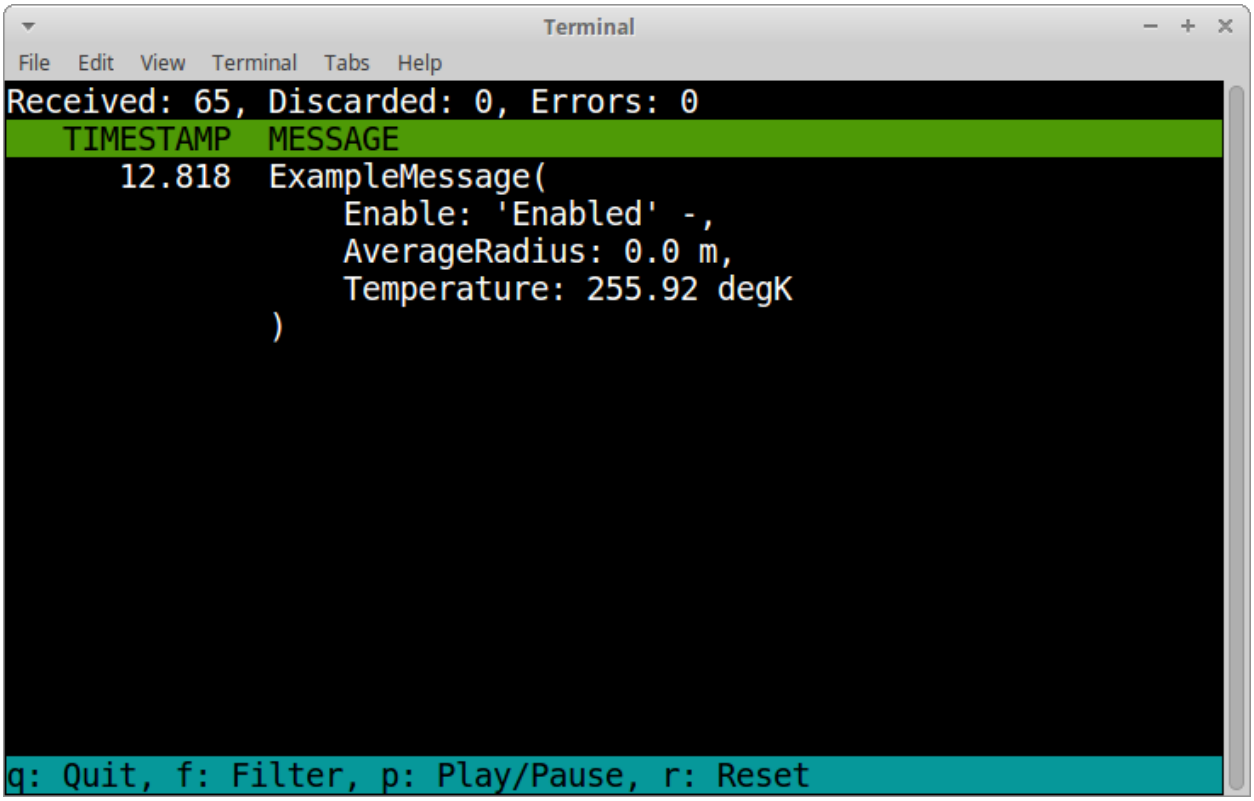
Other C code generators:

- http://www.coderdbc.com
- https://github.com/howerj/dbcc
- https://github.com/lonkamikaze/hsk-libs/blob/master/scripts/dbc2c.awk
- https://sourceforge.net/projects/comframe/

3.2.6 The monitor subcommand

Monitor CAN bus traffic in a text based user interface.

```
$ python3 -m cantools monitor tests/files/dbc/motohawk.dbc
```



The menu at the bottom of the monitor shows the available commands.

- Quit: Quit the monitor. Ctrl-C can be used as well.
- Filter: Only display messages matching given regular expression. Press <Enter> to return to the menu from the filter input line.
- Play/Pause: Toggle between playing and paused (or running and freezed).
- Reset: Reset the monitor to its initial state.

CHAPTER

FOUR

CONTRIBUTING

1. Fork the repository.
2. Install prerequisites.

```
python3 -m pip install -e .[dev]
```

3. Implement the new feature or bug fix.
4. Implement test case(s) to ensure that future changes do not break legacy.
5. Run the linters

```
ruff check src
mypy src
```

6. Run the tests.

```
tox -e py
```

7. Create a pull request.

CHAPTER

FIVE

FUNCTIONS AND CLASSES

cantools.database.**load_file**(*filename*, *database_format=None*, *encoding=None*, *frame_id_mask=None*, *prune_choices=False*, *strict=True*, *cache_dir=None*, *sort_signals=<function sort_signals_by_start_bit>*)

Open, read and parse given database file and return a *can.Database* or *diagnostics.Database* object with its contents.

database_format is one of `'arxml'`, `'dbc'`, `'kcd'`, `'sym'`, `cdd` and `None`. If `None`, the database format is selected based on the filename extension as in the table below. Filename extensions are case insensitive.

Extension	Database format
.arxml	`'arxml'`
.dbc	`'dbc'`
.kcd	`'kcd'`
.sym	`'sym'`
.cdd	`'cdd'`
<unknown>	`None`

encoding specifies the file encoding. If `None`, the encoding is selected based on the database format as in the table below. Use `open()` and *load()* if platform dependent encoding is desired.

Database format	Default encoding
`'arxml'`	`'utf-8'`
`'dbc'`	`'cp1252'`
`'kcd'`	`'utf-8'`
`'sym'`	`'cp1252'`
`'cdd'`	`'utf-8'`
`None`	`'utf-8'`

prune_choices abbreviates the names of choices by removing a common prefix ending on an underscore. If you want to have the original names you need to pass *prune_choices* = *False*.

cache_dir specifies the database cache location in the file system. Give as `None` to disable the cache. By default the cache is disabled. The cache key is the contents of given file. Using a cache will significantly reduce the load time when reloading the same file. The cache directory is automatically created if it does not exist. Remove the cache directory *cache_dir* to clear the cache.

See *load_string()* for descriptions of other arguments.

Raises an *UnsupportedDatabaseFormatError* exception if given file does not contain a supported database format.

```
>>> db = cantools.database.load_file('foo.dbc')
>>> db.version
'1.0'
```

cantools.database.**dump_file**(*database*, *filename*, *database_format=None*, *encoding=None*, *sort_signals='default'*)

Dump given database *database* to given file *filename*.

Depending on the output file format signals may be sorted by default. If you don't want signals to be sorted pass *sort_signals=None*. *sort_signals=None* is assumed by default if you have passed *sort_signals=None* to load_file. If you want the signals to be sorted in a special way pass something like *sort_signals = lambda signals: list(sorted(signals, key=lambda sig: sig.name))* For dbc files the default is to sort the signals by their start bit in descending order. For kcd files the default is to not sort the signals.

See *load_file()* for descriptions of other arguments.

The `'dbc'` database format will always have Windows-style line endings (`\r\n`). For other database formats the line ending depends on the operating system.

```
>>> db = cantools.database.load_file('foo.dbc')
>>> cantools.database.dump_file(db, 'bar.dbc')
```

Pass *sort_signals=None, prune_choices=False* to load_file in order to minimize the differences between foo.dbc and bar.dbc.

cantools.database.**load_string**(*string*, *database_format=None*, *frame_id_mask=None*, *prune_choices=False*, *strict=True*, *sort_signals=<function sort_signals_by_start_bit>*)

Parse given database string and return a *can.Database* or *diagnostics.Database* object with its contents.

database_format may be one of `'arxml'`, `'dbc'`, `'kcd'`, `'sym'`, `'cdd'` or `None`, where `None` means transparent format.

prune_choices is a bool indicating whether signal names are supposed to be abbreviated by stripping a common prefix ending on an underscore. This is enabled by default.

See *can.Database* for a description of *strict*.

sort_signals is a function taking a list of signals as argument and returning a list of signals. By default signals are sorted by their start bit when their Message object is created. If you don't want them to be sorted pass *sort_signals = None*. If you want the signals to be sorted in another way pass something like *sort_signals = lambda signals: list(sorted(signals, key=lambda sig: sig.name))*

Raises an *UnsupportedDatabaseFormatError* exception if given string does not contain a supported database format.

```
>>> with open('foo.dbc') as fin:
...     db = cantools.database.load_string(fin.read())
>>> db.version
'1.0'
```

cantools.database.**load**(*fp*, *database_format=None*, *frame_id_mask=None*, *prune_choices=False*, *strict=True*, *sort_signals=<function sort_signals_by_start_bit>*)

Read and parse given database file-like object and return a *can.Database* or *diagnostics.Database* object with its contents.

See *load_string()* for descriptions of other arguments.

Raises an *UnsupportedDatabaseFormatError* exception if given file-like object does not contain a supported database format.

```
>>> with open('foo.kcd') as fin:
...     db = cantools.database.load(fin)
>>> db.version
None
```

class cantools.database.can.**Database**(*messages=None*, *nodes=None*, *buses=None*, *version=None*, *dbc_specifics=None*, *autosar_specifics=None*, *frame_id_mask=None*, *strict=True*, *sort_signals=<function sort_signals_by_start_bit>*)

This class contains all messages, signals and definitions of a CAN network.

The factory functions *load()*, *load_file()* and *load_string()* returns instances of this class.

If *strict* is `True` an exception is raised if any signals are overlapping or if they don't fit in their message.

By default signals are sorted by their start bit when their Message object is created. If you don't want them to be sorted pass *sort_signals = None*. If you want the signals to be sorted in another way pass something like *sort_signals = lambda signals: list(sorted(signals, key=lambda sig: sig.name))*

property messages

A list of messages in the database.

Use *get_message_by_frame_id()* or *get_message_by_name()* to find a message by its frame id or name.

property nodes

A list of nodes in the database.

property buses

A list of CAN buses in the database.

property version

The database version, or `None` if unavailable.

property dbc

An object containing dbc specific properties like e.g. attributes.

property autosar

An object containing AUTOSAR specific properties like e.g. attributes.

is_similar(*other*, ***, *tolerance=1e-12*, *include_format_specifics=True*)

Compare two database objects inexactly

This means that small discrepanceies stemming from e.g. rounding errors are ignored.

add_arxml(*fp*)

Read and parse ARXML data from given file-like object and add the parsed data to the database.

add_arxml_file(*filename*, *encoding='utf-8'*)

Open, read and parse ARXML data from given file and add the parsed data to the database.

encoding specifies the file encoding.

add_arxml_string(*string*)

Parse given ARXML data string and add the parsed data to the database.

add_dbc(*fp*)

Read and parse DBC data from given file-like object and add the parsed data to the database.

```
>>> db = cantools.database.Database()
>>> with open ('foo.dbc', 'r') as fin:
...     db.add_dbc(fin)
```

add_dbc_file(*filename*, *encoding='cp1252'*)

Open, read and parse DBC data from given file and add the parsed data to the database.

encoding specifies the file encoding.

```
>>> db = cantools.database.Database()
>>> db.add_dbc_file('foo.dbc')
```

add_dbc_string(*string*)

Parse given DBC data string and add the parsed data to the database.

```
>>> db = cantools.database.Database()
>>> with open ('foo.dbc', 'r') as fin:
...     db.add_dbc_string(fin.read())
```

add_kcd(*fp*)

Read and parse KCD data from given file-like object and add the parsed data to the database.

add_kcd_file(*filename*, *encoding='utf-8'*)

Open, read and parse KCD data from given file and add the parsed data to the database.

encoding specifies the file encoding.

add_kcd_string(*string*)

Parse given KCD data string and add the parsed data to the database.

add_sym(*fp*)

Read and parse SYM data from given file-like object and add the parsed data to the database.

add_sym_file(*filename*, *encoding='utf-8'*)

Open, read and parse SYM data from given file and add the parsed data to the database.

encoding specifies the file encoding.

add_sym_string(*string*)

Parse given SYM data string and add the parsed data to the database.

as_dbc_string(***, *sort_signals='default'*, *sort_attribute_signals='default'*, *sort_attributes=None*, *sort_choices=None*, *shorten_long_names=True*)

Return the database as a string formatted as a DBC file. sort_signals defines how to sort signals in message definitions sort_attribute_signals defines how to sort signals in metadata -

comments, value table definitions and attributes

as_kcd_string(***, *sort_signals='default'*)

Return the database as a string formatted as a KCD file.

as_sym_string(***, *sort_signals='default'*)

Return the database as a string formatted as a SYM file.

get_message_by_name(*name*)

Find the message object for given name *name*.

get_message_by_frame_id(*frame_id*)

Find the message object for given frame id *frame_id*.

get_node_by_name(*name*)

Find the node object for given name *name*.

get_bus_by_name(*name*)

Find the bus object for given name *name*.

encode_message(*frame_id_or_name*, *data*, *scaling=True*, *padding=False*, *strict=True*)

Encode given signal data *data* as a message of given frame id or name *frame_id_or_name*. For regular Messages, *data* is a dictionary of signal name-value entries, for container messages it is a list of (ContainedMessageOrMessageName, ContainedMessageSignals) tuples.

If *scaling* is `False` no scaling of signals is performed.

If *padding* is `True` unused bits are encoded as 1.

If *strict* is `True` all signal values must be within their allowed ranges, or an exception is raised.

```
>>> db.encode_message(158, {'Bar': 1, 'Fum': 5.0})
b'\x01\x45\x23\x00\x11'
>>> db.encode_message('Foo', {'Bar': 1, 'Fum': 5.0})
b'\x01\x45\x23\x00\x11'
```

decode_message(*frame_id_or_name*, *data*, *decode_choices=True*, *scaling=True*, *decode_containers=False*, *allow_truncated=False*)

Decode given signal data *data* as a message of given frame id or name *frame_id_or_name*. Returns a dictionary of signal name-value entries.

If *decode_choices* is `False` scaled values are not converted to choice strings (if available).

If *scaling* is `False` no scaling of signals is performed.

```
>>> db.decode_message(158, b'\x01\x45\x23\x00\x11')
{'Bar': 1, 'Fum': 5.0}
>>> db.decode_message('Foo', b'\x01\x45\x23\x00\x11')
{'Bar': 1, 'Fum': 5.0}
```

If *decode_containers* is `True`, container frames are decoded. The reason why this needs to be explicitly enabled is that decoding container frames returns a list of `(Message, SignalsDict)` tuples which will cause code that does not expect this to misbehave. Trying to decode a container message with *decode_containers* set to `False` will raise a *DecodeError*.

refresh()

Refresh the internal database state.

This method must be called after modifying any message in the database to refresh the internal lookup tables used when encoding and decoding messages.

class cantools.database.can.**Message**(*frame_id*, *name*, *length*, *signals*, *contained_messages=None*, *header_id=None*, *header_byte_order='big_endian'*, *unused_bit_pattern=0*, *comment=None*, *senders=None*, *send_type=None*, *cycle_time=None*, *dbc_specifics=None*, *autosar_specifics=None*, *is_extended_frame=False*, *is_fd=False*, *bus_name=None*, *signal_groups=None*, *strict=True*, *protocol=None*, *sort_signals=<function sort_signals_by_start_bit>*)

A CAN message with frame id, comment, signals and other information.

If *strict* is `True` an exception is raised if any signals are overlapping or if they don't fit in the message.

By default signals are sorted by their start bit when their Message object is created. If you don't want them to be sorted pass *sort_signals = None*. If you want the signals to be sorted in another way pass something like *sort_signals = lambda signals: list(sorted(signals, key=lambda sig: sig.name))*

property header_id

The header ID of the message if it is part of a container message.

property header_byte_order

The byte order of the header ID of the message if it is part of a container message.

property frame_id

The message frame id.

property is_extended_frame

`True` if the message is an extended frame, `False` otherwise.

property is_fd

`True` if the message requires CAN-FD, `False` otherwise.

property name

The message name as a string.

property length

The message data length in bytes.

property signals

A list of all signals in the message.

property is_container

Returns if the message is a container message

property contained_messages

The list of messages potentially contained within this message

property unused_bit_pattern

The pattern used for unused bits of a message.

This prevents undefined behaviour and/or information leaks when encoding messages.

property signal_groups

A list of all signal groups in the message.

property comment

The message comment, or `None` if unavailable.

Note that we implicitly try to return the English comment if multiple languages were specified.

property comments

The dictionary with the descriptions of the message in multiple languages. `None` if unavailable.

property senders

A list of all sender nodes of this message.

property receivers

A set of all receiver nodes of this message.

This is equivalent to the set of nodes which receive at least one of the signals contained in the message.

property send_type

The message send type, or `None` if unavailable.

property cycle_time

The message cycle time, or `None` if unavailable.

property dbc

An object containing dbc specific properties like e.g. attributes.

property autosar

An object containing AUTOSAR specific properties

e.g. auxiliary data required to implement CRCs, secure on-board communication (secOC) or container messages.

property bus_name

The message bus name, or `None` if unavailable.

property protocol

The message protocol, or `None` if unavailable. Only one protocol is currently supported; `'j1939'`.

property signal_tree

All signal names and multiplexer ids as a tree. Multiplexer signals are dictionaries, while other signals are strings.

```
>>> foo = db.get_message_by_name('Foo')
>>> foo.signal_tree
['Bar', 'Fum']
>>> bar = db.get_message_by_name('Bar')
>>> bar.signal_tree
[{'A': {0: ['C', 'D'], 1: ['E']}}, 'B']
```

gather_signals(*input_data*, *node=None*)

Given a superset of all signals required to encode the message, return a dictionary containing exactly the ones required.

If a required signal is missing from the input dictionary, a `EncodeError` exception is raised.

gather_container(*contained_messages*, *signal_values*)

Given a superset of all messages required to encode all messages featured by a container message, return a list of (Message, SignalDict) tuples that can be passed to `encode()`.

If a required signal is missing from the input dictionary, a `EncodeError` exception is raised.

assert_signals_encodable(*input_data*, *scaling*, *assert_values_valid=True*, *assert_all_known=True*)

Given a dictionary of signal name to signal value mappings, ensure that all the signals required for encoding are present

As a minimum, all signals required to encode the message need to be specified. If they are not, a `KeyError` or an `EncodeError` exception is raised.

Depending on the parameters specified, the data of the dictionary must adhere to additonal requirements:

Parameters

- **scaling** – If `False` no scaling of signals is performed.
- **assert_values_valid** – If `True`, the values of all specified signals must be valid/encodable. If at least one is not, an `EncodeError` exception is raised. (Note that the values of multiplexer selector signals must always be valid!)

- **assert_all_known** – If `True`, all specified signals must be used by the encoding operation or an `EncodeError` exception is raised. This is useful to prevent typos.

assert_container_encodable(*input_data*, *scaling*, *assert_values_valid=True*, *assert_all_known=True*)

This method is identical to `assert_signals_encodable()` except that it is concerned with container messages.

encode(*data*, *scaling=True*, *padding=False*, *strict=True*)

Encode given data as a message of this type.

If the message is an "ordinary" frame, this method expects a key-to-value dictionary as *data* which maps the name of every required signal to a value that can be encoded by that signal. If the current message is a container message, it expects a list of *(contained_message, contained_data)* tuples where *contained_message* is either an integer with the header ID, the name or the message object of the contained message. Similarly, the *contained_data* can either be specified as raw binary data (*bytes*) or as a key-to-value dictionary of every signal needed to encode the featured message.

If *scaling* is `False` no scaling of signals is performed.

If *padding* is `True` unused bits are encoded as 1.

If *strict* is `True` the specified signals must exactly be the ones expected, and their values must be within their allowed ranges, or an *EncodeError* exception is raised.

```
>>> foo = db.get_message_by_name('Foo')
>>> foo.encode({'Bar': 1, 'Fum': 5.0})
b'\x01\x45\x23\x00\x11'
```

unpack_container(*data*, *allow_truncated=False*)

Unwrap the contents of a container message.

This returns a list of `(contained_message, contained_data)` tuples, i.e., the data for the contained message are `bytes` objects, not decoded signal dictionaries. This is required for verifying the correctness of the end-to-end protection or the authenticity of a contained message.

Note that `contained_message` is the header ID integer value if a contained message is unknown. Further, if something goes seriously wrong, a `DecodeError` is raised.

decode(*data*, *decode_choices=True*, *scaling=True*, *decode_containers=False*, *allow_truncated=False*, *allow_excess=True*)

Decode given data as a message of this type.

If *decode_choices* is `False` scaled values are not converted to choice strings (if available).

If *scaling* is `False` no scaling of signals is performed.

```
>>> foo = db.get_message_by_name('Foo')
>>> foo.decode(b'\x01\x45\x23\x00\x11')
{'Bar': 1, 'Fum': 5.0}
```

If *decode_containers* is `True`, the inner messages are decoded if the current message is a container frame. The reason why this needs to be explicitly enabled is that the result of *decode()* for container frames is a list of `(header_id, signals_dict)` tuples which might cause code that does not expect this to misbehave. Trying to decode a container message with *decode_containers* set to `False` will raise a *DecodeError*.

If *allow_truncated* is `True`, incomplete messages (i.e., ones where the received data is shorter than specified) will be partially decoded, i.e., all signals which are fully present in the received data will be decoded, and the remaining ones will be omitted. If 'allow_truncated` is set to `False`, *DecodeError* will be raised when trying to decode incomplete messages.

If *allow_excess* is `True`, data that is are longer than the expected message length is decoded, else a *ValueError* is raised if such data is encountered.

decode_simple(*data*, *decode_choices=True*, *scaling=True*, *allow_truncated=False*, *allow_excess=True*)

Decode given data as a container message.

This method is identical to `decode()` except that the message **must not** be a container. If the message is a container, an exception is raised.

decode_container(*data*, *decode_choices=True*, *scaling=True*, *allow_truncated=False*, *allow_excess=True*)

Decode given data as a container message.

This method is identical to `decode()` except that the message **must** be a container. If the message is not a container, an exception is raised.

is_multiplexed()

Returns `True` if the message is multiplexed, otherwise `False`.

```
>>> foo = db.get_message_by_name('Foo')
>>> foo.is_multiplexed()
False
>>> bar = db.get_message_by_name('Bar')
>>> bar.is_multiplexed()
True
```

refresh(*strict=None*)

Refresh the internal message state.

If *strict* is `True` an exception is raised if any signals are overlapping or if they don't fit in the message. This argument overrides the value of the same argument passed to the constructor.

class cantools.database.can.**Signal**(*name*, *start*, *length*, *byte_order='little_endian'*, *is_signed=False*, *raw_initial=None*, *raw_invalid=None*, *conversion=None*, *minimum=None*, *maximum=None*, *unit=None*, *dbc_specifics=None*, *comment=None*, *receivers=None*, *is_multiplexer=False*, *multiplexer_ids=None*, *multiplexer_signal=None*, *spn=None*)

A CAN signal with position, size, unit and other information. A signal is part of a message.

Signal bit numbering in a message:

```
Byte:       0        1        2        3        4        5        6        7
       +--------+--------+--------+--------+--------+--------+--------+--------+---␣
↪- -
       |        |        |        |        |        |        |        |        |
       +--------+--------+--------+--------+--------+--------+--------+--------+---␣
↪- -
Bit:    7      0 15     8 23    16 31    24 39    32 47    40 55    48 63    56
```

Big endian signal with start bit 2 and length 5 (0=LSB, 4=MSB):

```
Byte:       0        1        2        3
       +--------+--------+--------+--- - -
       |    |432|10|     |        |
       +--------+--------+--------+--- - -
Bit:    7      0 15     8 23    16 31
```

Little endian signal with start bit 2 and length 9 (0=LSB, 8=MSB):

```
Byte:       0        1        2        3
       +--------+--------+--------+--- - -
       |543210| |    |876|        |
       +--------+--------+--------+--- - -
Bit:    7      0 15     8 23    16 31
```

name

The signal name as a string.

conversion

The conversion instance, which is used to convert between raw and scaled/physical values.

minimum

The scaled minimum value of the signal, or `None` if unavailable.

maximum

The scaled maximum value of the signal, or `None` if unavailable.

start

The start bit position of the signal within its message.

length

The length of the signal in bits.

byte_order

Signal byte order as `'little_endian'` or `'big_endian'`.

is_signed

`True` if the signal is signed, `False` otherwise. Ignore this attribute if *is_float* is `True`.

raw_initial

The internal representation of the initial value of the signal, or `None` if unavailable.

initial

The initial value of the signal in units of the physical world, or `None` if unavailable.

raw_invalid

The raw value representing that the signal is invalid, or `None` if unavailable.

invalid

The scaled value representing that the signal is invalid, or `None` if unavailable.

unit

The unit of the signal as a string, or `None` if unavailable.

dbc

An object containing dbc specific properties like e.g. attributes.

receivers

A list of all receiver nodes of this signal.

is_multiplexer

`True` if this is the multiplexer signal in a message, `False` otherwise.

multiplexer_ids

The multiplexer ids list if the signal is part of a multiplexed message, `None` otherwise.

multiplexer_signal

The multiplexer signal if the signal is part of a multiplexed message, `None` otherwise.

spn

The J1939 Suspect Parameter Number (SPN) value if the signal has this attribute, `None` otherwise.

comments

The dictionary with the descriptions of the signal in multiple languages. `None` if unavailable.

raw_to_scaled(*raw_value*, *decode_choices=True*)

Convert an internal raw value according to the defined scaling or value table.

Parameters

- **raw_value** – The raw value
- **decode_choices** – If *decode_choices* is `False` scaled values are not converted to choice strings (if available).

Returns

The calculated scaled value

scaled_to_raw(*scaled_value*)

Convert a scaled value to the internal raw value.

Parameters

scaled_value – The scaled value.

Returns

The internal raw value.

property scale

The scale factor of the signal value.

property offset

The offset of the signal value.

property choices

A dictionary mapping signal values to enumerated choices, or `None` if unavailable.

property is_float

`True` if the raw signal value is a float, `False` otherwise.

property comment

The signal comment, or `None` if unavailable.

Note that we implicitly try to return the English comment if multiple languages were specified.

class cantools.database.namedsignalvalue.**NamedSignalValue**(*value*, *name*, *comments=None*)

Represents a named value of a signal.

Named values map an integer number to a human-readable string. Some file formats like ARXML support specifying descriptions for the named value.

name

The text intended for human consumption which the specified integer is mapped to.

value

The integer value that gets mapped.

property comments

The descriptions of the named value

This is a dictionary containing the descriptions in multiple languages. The dictionary is indexed by the language.

Example:

```
# retrieve the English comment of the named value or an empty
# string if none was specified.
named_value.comments.get("EN", "")
```

class cantools.database.conversion.**BaseConversion**

The BaseConversion class defines the interface for all signal conversion classes.

scale

the scaling factor of the conversion

offset

the offset value of the conversion

is_float

`True` if the raw/internal value is a floating datatype `False` if it is an integer datatype

choices

an optional mapping of raw values to their corresponding text value

static factory(*scale=1*, *offset=0*, *choices=None*, *is_float=False*)

Factory method that returns an instance of a conversion subclass based on the given parameters.

Parameters

- **scale** – The scale factor to use for the conversion.
- **offset** – The offset to use for the conversion.
- **choices** – A dictionary of named signal choices, mapping raw values to string labels.
- **is_float** – A boolean flag indicating whether the raw value is a float or an integer.

Returns

An instance of a conversion subclass, either an *IdentityConversion*, a *LinearIntegerConversion*, a *LinearConversion`or a `NamedSignalConversion*.

Raises

`TypeError` – If the given parameters are of the wrong type.

abstract raw_to_scaled(*raw_value*, *decode_choices=True*)

Convert an internal raw value according to the defined scaling or value table.

Parameters

- **raw_value** – The raw value
- **decode_choices** – If *decode_choices* is `False` scaled values are not converted to choice strings (if available).

Returns

The calculated scaled value

abstract scaled_to_raw(*scaled_value*)

Convert a scaled value to the internal raw value.

Parameters
scaled_value – The scaled value.

Returns
The internal raw value.

abstract numeric_scaled_to_raw(*scaled_value*)

Convert a numeric scaled value to the internal raw value.

Parameters
scaled_value – The numeric scaled value.

Returns
The internal raw value.

class cantools.database.diagnostics.**Database**(*dids=None*)

This class contains all DIDs.

The factory functions *load()*, *load_file()* and *load_string()* returns instances of this class.

property dids

A list of DIDs in the database.

add_cdd(*fp*)

Read and parse CDD data from given file-like object and add the parsed data to the database.

add_cdd_file(*filename*, *encoding='utf-8'*)

Open, read and parse CDD data from given file and add the parsed data to the database.

encoding specifies the file encoding.

add_cdd_string(*string*)

Parse given CDD data string and add the parsed data to the database.

get_did_by_name(*name*)

Find the DID object for given name *name*.

get_did_by_identifier(*identifier*)

Find the DID object for given identifier *identifier*.

refresh()

Refresh the internal database state.

This method must be called after modifying any DIDs in the database to refresh the internal lookup tables used when encoding and decoding DIDs.

class cantools.database.diagnostics.**Did**(*identifier*, *name*, *length*, *datas*)

A DID with identifier and other information.

property identifier

The did identifier as an integer.

property name

The did name as a string.

property length

The did name as a string.

property datas

The did datas as a string.

encode(*data*, *scaling=True*)

Encode given data as a DID of this type.

If *scaling* is `False` no scaling of datas is performed.

```
>>> foo = db.get_did_by_name('Foo')
>>> foo.encode({'Bar': 1, 'Fum': 5.0})
b'\x01\x45\x23\x00\x11'
```

decode(*data*, *decode_choices=True*, *scaling=True*, *allow_truncated=False*, *allow_excess=True*)

Decode given data as a DID of this type.

If *decode_choices* is `False` scaled values are not converted to choice strings (if available).

If *scaling* is `False` no scaling of datas is performed.

```
>>> foo = db.get_did_by_name('Foo')
>>> foo.decode(b'\x01\x45\x23\x00\x11')
{'Bar': 1, 'Fum': 5.0}
```

refresh()

Refresh the internal DID state.

class cantools.database.diagnostics.**Data**(*name*, *start*, *length*, *byte_order='little_endian'*, *conversion=None*, *minimum=None*, *maximum=None*, *unit=None*)

A data data with position, size, unit and other information. A data is part of a DID.

name

The data name as a string.

conversion

The conversion instance, which is used to convert between raw and scaled/physical values.

start

The start bit position of the data within its DID.

length

The length of the data in bits.

byte_order

Data byte order as `'little_endian'` or `'big_endian'`.

minimum

The minimum value of the data, or `None` if unavailable.

maximum

The maximum value of the data, or `None` if unavailable.

unit

The unit of the data as a string, or `None` if unavailable.

raw_to_scaled(*raw_value*, *decode_choices=True*)

Convert an internal raw value according to the defined scaling or value table.

Parameters

- **raw_value** – The raw value
- **decode_choices** – If *decode_choices* is `False` scaled values are not converted to choice strings (if available).

Returns
The calculated scaled value

scaled_to_raw(*scaled_value*)

Convert a scaled value to the internal raw value.

Parameters
scaled_value – The scaled value.

Returns
The internal raw value.

property scale

The scale factor of the signal value.

property offset

The offset of the signal value.

property choices

A dictionary mapping signal values to enumerated choices, or `None` if unavailable.

property is_float

`True` if the raw signal value is a float, `False` otherwise.

class cantools.database.**UnsupportedDatabaseFormatError**(*e_arxml*, *e_dbc*, *e_kcd*, *e_sym*, *e_cdd*)

This exception is raised when *load_file()*, *load()* and *load_string()* are unable to parse given database file or string.

class cantools.tester.**Tester**(*dut_name*, *database*, *can_bus*, *bus_name=None*, *on_message=None*, *decode_choices=True*, *scaling=True*, *padding=False*)

Test given node *dut_name* on given CAN bus *bus_name*.

database is a *Database* instance.

can_bus a CAN bus object, normally created using the python-can package.

The *on_message* callback is called for every successfully decoded received message. It is called with one argument, an *DecodedMessage* instance.

Here is an example of how to create a tester:

```
>>> import can
>>> import cantools
>>> can.rc['interface'] = 'socketcan'
>>> can.rc['channel'] = 'vcan0'
>>> can_bus = can.interface.Bus()
>>> database = cantools.database.load_file('tests/files/tester.kcd')
>>> tester = cantools.tester.Tester('PeriodicConsumer', database, can_bus,
↪'PeriodicBus')
```

start()

Start the tester. Starts sending enabled periodic messages.

```
>>> tester.start()
```

stop()

Stop the tester. Periodic messages will not be sent after this call. Call *start()* to resume a stopped tester.

```
>>> tester.stop()
```

property messages

Set and get signals in messages. Set signals takes effect immediately for started enabled periodic messages. Call *send()* for other messages.

```
>>> periodic_message = tester.messages['PeriodicMessage1']
>>> periodic_message
{'Signal1': 0, 'Signal2': 0}
>>> periodic_message['Signal1'] = 1
>>> periodic_message.update({'Signal1': 2, 'Signal2': 5})
>>> periodic_message
{'Signal1': 2, 'Signal2': 5}
```

enable(*message_name*)

Enable given message *message_name* and start sending it if its periodic and the tester is running.

```
>>> tester.enable('PeriodicMessage1')
```

disable(*message_name*)

Disable given message *message_name* and stop sending it if its periodic, enabled and the tester is running.

```
>>> tester.disable('PeriodicMessage1')
```

send(*message_name*, *signals=None*)

Send given message *message_name* and optional signals *signals*.

```
>>> tester.send('Message1', {'Signal2': 10})
>>> tester.send('Message1')
```

expect(*message_name*, *signals=None*, *timeout=None*, *discard_other_messages=True*)

Expect given message *message_name* and signal values *signals* within *timeout* seconds.

Give *signals* as `None` to expect any signal values.

Give *timeout* as `None` to wait forever.

Messages are read from the input queue, and those not matching given *message_name* and *signals* are discarded if *discard_other_messages* is `True`. *flush_input()* may be called to discard all old messages in the input queue before calling the expect function.

Returns the expected message, or `None` on timeout.

```
>>> tester.expect('Message2', {'Signal1': 13})
{'Signal1': 13, 'Signal2': 9}
```

flush_input()

Flush, or discard, all messages in the input queue.

class cantools.tester.**DecodedMessage**(*name*, *signals*)

A decoded message.

name

Message name.

signals

Message signals.

CHAPTER

SIX

CODING STYLE

The coding style for this package is defined as below. The rules are based on my personal preference.

- Blank lines before and after statements (if, while, return, ...) (1), unless at beginning or end of another statement or file (8).
- Two blank lines between file level definitions (2).
- Space before and after operators (3), except for keyword arguments where no space is allowed (4).
- One import per line (5).
- Comments and doc strings starts with capital letter and ends with a period, that is, just as sentences (6).
- Blank line after doc strings (7).
- Maximum line length of 90 characters, but aim for less than 80.
- All function arguments on one line, or one per line.
- Class names are CamelCase. Underscore is not allowed.
- Function and variable names are lower case with underscore separating words.

```
import sys
from os import path                     # (5)
from os import getcwd                   # (5)
                                        # (2)
                                        # (2)
def foo(bars, fum=None):                # (4)
    """This is a doc string.            # (6)

    """
                                        # (7)
    fies = []                           # (3)
    kam = path.join(getcwd(), '..')
                                        # (1)
    for bar in bars:
        if len(bar) == 1):              # (8)
            fies.append(ham + 2 * bar)  # (3)
                                        # (1)
    # This is a comment.                # (6)
    if fum in None:
        fum = 5                         # (3)
    else:
        fum += 1                        # (3)
```

(continues on next page)

(continued from previous page)

```
                                        # (1)
    fies *= fum                         # (3)
                                        # (1)
    return fies
                                        # (2)
                                        # (2)
def goo():
    return True
```

CHAPTER

SEVEN

TIPS AND TRICKS

Virtual CAN interface setup in Ubuntu:

```
sudo modprobe vcan
sudo ip link add dev vcan0 type vcan
sudo ip link set vcan0 mtu 72         # For CAN-FD
sudo ip link set up vcan0
```

INDEX

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V