asammdf Documentation

Release "4.7.12.dev0"

Daniel Hrisca

Feb 25, 2019

Contents

asammdf is a fast parser/editor for ASAM (Associtation for Standardisation of Automation and Measuring Systems) MDF (Measurement Data Format) files.

asammdf supports MDF versions 2 (.dat), 3 (.mdf) and 4 (.mf4).

asammdf works on Python 2.7, and Python >= 3.4 (Travis CI tests done with Python 2.7 and Python >= 3.5)

CHAPTER 1

Introduction

1.1 Project goals

The main goals for this library are:

- to be faster than the other Python based mdf libraries
- to have clean and easy to understand code base

1.2 Features

- create new mdf files from scratch
- append new channels
- read unsorted MDF v3 and v4 files
- read CAN bus logging files
- filter a subset of channels from original mdf file
- cut measurement to specified time interval
- convert to different mdf version
- export to pandas, Excel, HDF5, Matlab (v4, v5 and v7.3), CSV and parquet
- merge multiple files sharing the same internal structure
- read and save mdf version 4.10 files containing zipped data blocks
- space optimizations for saved files (no duplicated blocks)
- split large data blocks (configurable size) for mdf version 4
- full support (read, append, save) for the following map types (multidimensional array channels):
  - mdf version 3 channels with CDBLOCK

  - mdf version 4 structure channel composition
  - mdf version 4 channel arrays with CNTemplate storage and one of the array types:
    * 0 - array
    * 1 - scaling axis
    * 2 - look-up
- add and extract attachments for mdf version 4
- handle large files (for example merging two fileas, each with 14000 channels and 5GB size, on a RaspberryPi) using *memory* = *minimum* argument
- extract channel data, master channel and extra channel information as *Signal* objects for unified operations with v3 and v4 files
- time domain operation using the *Signal* class
  - Pandas data frames are good if all the channels have the same time base
  - a measurement will usually have channels from different sources at different rates
  - the *Signal* class facilitates operations with such channels

1.3 Major features not implemented (yet)

- for version 3
  - functionality related to sample reduction block: the sample reduction blocks are simply ignored
- for version 4
  - functionality related to sample reduction block: the sample reduction blocks are simply ignored
  - handling of channel hierarchy: channel hierarchy is ignored
  - full handling of bus logging measurements: currently only CAN bus logging is implemented with the ability to *get* signals defined in the attached CAN database (.arxml or .dbc)
  - handling of unfinished measurements (mdf 4): warnings are logged based on the unfinished status flags but no further steps are taken to sanitize the measurement
  - full support for remaining mdf 4 channel arrays types
  - xml schema for MDBLOCK: most metadata stored in the comment blocks will not be available
  - full handling of event blocks: events are transferred to the new files (in case of calling methods that return new *MDF* objects) but no new events can be created
  - channels with default X axis: the default X axis is ignored and the channel group's master channel is used

1.4 Dependencies

asammdf uses the following libraries

- numpy : the heart that makes all tick
- numexpr : for algebraic and rational channel conversions
- wheel : for installation in virtual environments

- pandas : for DataFrame export
- canmatrix : to handle CAN bus logging measurements

optional dependencies needed for exports

- h5py : for HDF5 export
- xlsxwriter : for Excel export
- scipy : for Matlab v4 and v5 .mat export
- hdf5storage : for Matlab v7.3 .mat export
- fastparquet : for parquet export

other optional dependencies

- chardet : to detect non-standard unicode encodings
- PyQt5 : for GUI tool
- pyqtgraph : for GUI tool and Signal plotting
- matplotlib : as fallback for Signal plotting

1.5 Installation

asammdf is available on

- github: https://github.com/danielhrisca/asammdf/
- PyPI: https://pypi.org/project/asammdf/
- conda-forge: https://anaconda.org/conda-forge/asammdf

```
pip install asammdf
# or for anaconda
conda install -c conda-forge asammdf
```

1.6 Contributing & Support

Please have a look over the contributing guidelines

If you enjoy this library please consider making a donation to the numpy project

1.6.1 Contributors

Thanks to all who contributed with commits to *asammdf*

- Julien Grave JulienGrv.
- Jed Frey jed-frey.
- Mihai yahym.
- Jack Weinstein jackjweinstein.
- Isuru Fernando isuruf.
- Felix Kohlgrüber fkohlgrueber.

- Stanislav Frolov stanifrolov.
- Thomas Kastl kasuteru.
- venden venden.
- Marat K. *kopytjuk* <https://github.com/kopytjuk>‘_.
- freakatzz *freakatzz* <https://github.com/freakatzz>‘_.

CHAPTER 2

API

2.1 MDF

This class acts as a proxy for the *MDF2*, *MDF3* and *MDF4* classes. All attribute access is delegated to the underlying *_mdf* attribute (MDF2, MDF3 or MDF4 object). See MDF3 and MDF4 for available extra methods (MDF2 and MDF3 share the same implementation).

An empty MDF file is created if the *name* argument is not provided. If the *name* argument is provided then the file must exist in the filesystem, otherwise an exception is raised.

The best practice is to use the MDF as a context manager. This way all resources are released correctly in case of exceptions.

```
with MDF(r'test.mdf') as mdf_file:
    # do something
```

class asammdf.mdf.**MDF**(*name=None*, *version='4.10'*, ***kwargs*)

Unified access to MDF v3 and v4 files. Underlying _mdf's attributes and methods are linked to the *MDF* object via *setattr*. This is done to expose them to the user code and for performance considerations.

Parameters

name [string | BytesIO] mdf file name (if provided it must be a real file name) or file-like object

version [string] mdf file version from ('2.00', '2.10', '2.14', '3.00', '3.10', '3.20', '3.30', '4.00', '4.10', '4.11'); default '4.10'

callback [function] keyword only argument: function to call to update the progress; the function must accept two arguments (the current progress and maximum progress value)

use_display_names [bool] keyword only argument: for MDF4 files parse the XML channel comment to search for the display name; XML parsing is quite expensive so setting this to *False* can decrease the loading times very much; default *False*

static concatenate(*files*, *version='4.10'*, *sync=True*, ***kwargs*)

concatenates several files. The files must have the same internal structure (same number of groups, and same channels in each group)

Parameters

files [list | tuple] list of *MDF* file names or *MDF* instances

version [str] merged file version

sync [bool] sync the files based on the start of measurement, default *True*

Returns

concatenate [MDF] new *MDF* object with concatenated channels

Raises

MdfException [if there are inconsistencies between the files]

convert (*version*)

convert *MDF* to other version

Parameters

version [str] new mdf file version from ('2.00', '2.10', '2.14', '3.00', '3.10', '3.20', '3.30', '4.00', '4.10', '4.11'); default '4.10'

Returns

out [MDF] new *MDF* object

cut (*start=None*, *stop=None*, *whence=0*, *version=None*, *include_ends=True*)

cut *MDF* file. *start* and *stop* limits are absolute values or values relative to the first timestamp depending on the *whence* argument.

Parameters

start [float] start time, default *None*. If *None* then the start of measurement is used

stop [float] stop time, default *None*. If *None* then the end of measurement is used

whence [int] how to search for the start and stop values

- 0 : absolute
- 1 : relative to first timestamp

version [str] new mdf file version from ('2.00', '2.10', '2.14', '3.00', '3.10', '3.20', '3.30', '4.00', '4.10', '4.11'); default *None* and in this case the original file version is used

include_ends [bool] include the *start* and *stop* timestamps after cutting the signal. If *start* and *stop* are found in the original timestamps, then the new samples will be computed using interpolation. Default *True*

Returns

——-

out [MDF] new MDF object

export (*fmt*, *filename=None*, ***kargs*)

export *MDF* to other formats. The *MDF* file name is used is available, else the *filename* argument must be provided.

Parameters

fmt [string] can be one of the following:

- *csv* : CSV export that uses the ";" delimiter. This option will generate a new csv file for each data group (<MDFNAME>_DataGroup_<cntr>.csv)

- *hdf5* : HDF5 file output; each *MDF* data group is mapped to a *HDF5* group with the name 'DataGroup_<cntr>' (where <cntr> is the index)
- *excel* : Excel file output (very slow). This option will generate a new excel file for each data group (<MDFNAME>_DataGroup_<cntr>.xlsx)
- *mat* : Matlab .mat version 4, 5 or 7.3 export. If *single_time_base==False* the channels will be renamed in the mat file to 'D<cntr>_<channel name>'. The channel group master will be renamed to 'DM<cntr>_<channel name>' (<*cntr*> is the data group index starting from 0)
- *pandas* : export all channels as a single pandas DataFrame
- *parquet* : export to Apache parquet format

filename [string | pathlib.Path] export file name

****kwargs**

- *single_time_base*: resample all channels to common time base, default *False* (pandas export is by default single based)
- *raster*: float time raster for resampling. Valid if *single_time_base* is *True* and for *pandas* export
- *time_from_zero*: adjust time channel to start from 0
- *use_display_names*: use display name instead of standard channel name, if available.
- *empty_channels*: behaviour for channels without samples; the options are *skip* or *zeros*; default is *skip*
- *format*: only valid for *mat* export; can be '4', '5' or '7.3', default is '5'
- *oned_as*: only valid for *mat* export; can be 'row' or 'column'
- *keep_arrays* : keep arrays and structure channels as well as the component channels. If *True* this can be very slow. If *False* only the component channels are saved, and their names will be prefixed with the parent channel.

Returns

dataframe [pandas.DataFrame] only in case of *pandas* export

filter(*channels*, *version=None*)

return new *MDF* object that contains only the channels listed in *channels* argument

Parameters

channels [list] list of items to be filtered; each item can be :

- a channel name string
- (channel name, group index, channel index) list or tuple
- (channel name, group index) list or tuple
- (None, group index, channel index) list or tuple

version [str] new mdf file version from ('2.00', '2.10', '2.14', '3.00', '3.10', '3.20', '3.30', '4.00', '4.10', '4.11'); default *None* and in this case the original file version is used

Returns

mdf [MDF] new *MDF* file

Examples

```
>>> from asammdf import MDF, Signal
>>> import numpy as np
>>> t = np.arange(5)
>>> s = np.ones(5)
>>> mdf = MDF()
>>> for i in range(4):
...     sigs = [Signal(s*(i*10+j), t, name='SIG') for j in range(1,4)]
...     mdf.append(sigs)
...
>>> filtered = mdf.filter(['SIG', ('SIG', 3, 1), ['SIG', 2], (None, 1, 2)])
>>> for gp_nr, ch_nr in filtered.channels_db['SIG']:
...     print(filtered.get(group=gp_nr, index=ch_nr))
...
<Signal SIG:
        samples=[ 1.  1.  1.  1.  1.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
<Signal SIG:
        samples=[ 31.  31.  31.  31.  31.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
<Signal SIG:
        samples=[ 21.  21.  21.  21.  21.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
<Signal SIG:
        samples=[ 12.  12.  12.  12.  12.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
```

get_group (*index*, *raster=None*, *time_from_zero=False*, *use_display_names=False*)
get channel group as pandas DataFrames. If there are multiple occurences for the same channel name, then a counter will be used to make the names unique (<original_name>_<counter>)

Parameters

index [int] channel group index

Returns

df [pandas.DataFrame]

iter_channels (*skip_master=True*)
generator that yields a *Signal* for each non-master channel

Parameters

skip_master [bool] do not yield master channels; default *True*

iter_get (*name=None*, *group=None*, *index=None*, *raster=None*, *samples_only=False*, *raw=False*)
iterator over a channel

This is usefull in case of large files with a small number of channels.

If the *raster* keyword argument is not *None* the output is interpolated accordingly

Parameters

name [string] name of channel

group [int] 0-based group index

index [int] 0-based channel index

raster [float] time raster in seconds

samples_only [bool]

if *True* return only the channel samples as numpy array; if *False* return a *Signal* object

raw [bool] return channel samples without appling the conversion rule; default *False*

iter_groups()

generator that yields channel groups as pandas DataFrames. If there are multiple occurences for the same channel name inside a channel group, then a counter will be used to make the names unique (<original_name>_<counter>)

resample(*raster*, *version=None*)

resample all channels using the given raster. See *configure* to select the interpolation method for interger channels

Parameters

raster [float] time raster is seconds

version [str] new mdf file version from ('2.00', '2.10', '2.14', '3.00', '3.10', '3.20', '3.30', '4.00', '4.10', '4.11'); default *None* and in this case the original file version is used

Returns

mdf [MDF] new *MDF* with resampled channels

static scramble(*name*, ***kwargs*)

scramble text blocks and keep original file structure

Parameters

name [str | pathlib.Path] file name

Returns

name [str] scrambled file name

select(*channels*, *dataframe=False*, *record_offset=0*)

retreiv the channels listed in *channels* argument as *Signal* objects

Parameters

channels [list] list of items to be filtered; each item can be :

- a channel name string
- (channel name, group index, channel index) list or tuple
- (channel name, group index) list or tuple
- (None, group index, channel index) lsit or tuple

dataframe: bool return a pandas DataFrame instead of a list of *Signals*; in this case the signals will be interpolated using the union of all timestamps

Returns

signals [list] list of *Signal* objects based on the input channel list

Examples

```
>>> from asammdf import MDF, Signal
>>> import numpy as np
>>> t = np.arange(5)
>>> s = np.ones(5)
>>> mdf = MDF()
>>> for i in range(4):
...     sigs = [Signal(s*(i*10+j), t, name='SIG') for j in range(1,4)]
...     mdf.append(sigs)
...
>>> # select SIG group 0 default index 1 default, SIG group 3 index 1, SIG␣
↪group 2 index 1 default and channel index 2 from group 1
...
>>> mdf.select(['SIG', ('SIG', 3, 1), ['SIG', 2],  (None, 1, 2)])
[<Signal SIG:
        samples=[ 1.  1.  1.  1.  1.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
, <Signal SIG:
        samples=[ 31.  31.  31.  31.  31.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
, <Signal SIG:
        samples=[ 21.  21.  21.  21.  21.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
, <Signal SIG:
        samples=[ 12.  12.  12.  12.  12.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
]
```

static stack (*files*, *version='4.10'*, *sync=True*, ***kwargs*)

stack several files and return the stacked *MDF* object

Parameters

files [list | tuple] list of *MDF* file names or *MDF* instances

version [str] merged file version

sync [bool] sync the files based on the start of measurement, default *True*

Returns

stacked [MDF] new *MDF* object with stacked channels

to_dataframe(*channels=None*, *raster=None*, *time_from_zero=True*, *empty_channels='skip'*, *keep_arrays=False*, *use_display_names=False*)

generate pandas DataFrame

Parameters

* **channels** [list] filter a subset of channels; default *None*
* **raster** [float] time raster for resampling; default *None*
* **time_from_zero** [bool] adjust time channel to start from 0
* **empty_channels** [str] behaviour for channels without samples; the options are *skip* or *zeros*; default is *skip*
* **use_display_names** [bool] use display name instead of standard channel name, if available.
* **keep_arrays** [bool] keep arrays and structure channels as well as the component channels. If *True* this can be very slow. If *False* only the component channels are saved, and their names will be prefixed with the parent channel.

Returns

dataframe [pandas.DataFrame]

whereis(*channel*)

get ocurrences of channel name in the file

Parameters

channel [str] channel name string

Returns

ocurrences [tuple]

Examples

```
>>> mdf = MDF(file_name)
>>> mdf.whereis('VehicleSpeed') # "VehicleSpeed" exists in the file
((1, 2), (2, 4))
>>> mdf.whereis('VehicleSPD') # "VehicleSPD" doesn't exist in the file
()
```

2.2 MDF3

class asammdf.blocks.mdf_v3.**MDF3**(*name=None*, *version='3.30'*, ***kwargs*)

The *header* attibute is a *HeaderBlock*.

The *groups* attribute is a list of dicts, each one with the following keys

- `data_group` - DataGroup object
- `channel_group` - ChannelGroup object
- `channels` - list of Channel objects with the same order as found in the mdf file

- `channel_dependencies` - list of *ChannelArrayBlock* in case of channel arrays; list of Channel objects in case of structure channel composition
- `data_block` - address of data block
- `data_location`- integer code for data location (original file, temporary file or memory)
- `data_block_addr` - list of raw samples starting addresses
- `data_block_type` - list of codes for data block type
- `data_block_size` - list of raw samples block size
- `sorted` - sorted indicator flag
- `record_size` - dict that maps record ID's to record sizes in bytes
- `size` - total size of data block for the current group
- `trigger` - *Trigger* object for current group

Parameters

name [string | pathlib.Path] mdf file name (if provided it must be a real file name) or file-like object

version [string] mdf file version ('2.00', '2.10', '2.14', '3.00', '3.10', '3.20' or '3.30'); default '3.30'

callback [function] keyword only argument: function to call to update the progress; the function must accept two arguments (the current progress and maximum progress value)

Attributes

attachments [list] list of file attachments

channels_db [dict] used for fast channel access by name; for each name key the value is a list of (group index, channel index) tuples

groups [list] list of data group dicts

header [HeaderBlock] mdf file header

identification [FileIdentificationBlock] mdf file start block

masters_db [dict]

used for fast master channel access; for each group index key the value is the master channel index

memory [str] memory optimization option

name [string] mdf file name

version [str] mdf version

add_trigger(*group*, *timestamp*, *pre_time=0*, *post_time=0*, *comment=''*)
add trigger to data group

Parameters

group [int] group index

timestamp [float] trigger time

pre_time [float] trigger pre time; default 0

post_time [float] trigger post time; default 0

comment [str] trigger comment

append(*signals*, *acquisition_info='Python'*, *common_timebase=False*, *units=None*)
Appends a new data group.

For channel dependencies type Signals, the *samples* attribute must be a numpy.recarray

Parameters

signals [list | Signal | pandas.DataFrame] list of *Signal* objects, or a single *Signal* object, or a pandas *DataFrame* object. All bytes columns in the pandas *DataFrame* must be *latin-1* encoded

acquisition_info [str] acquisition information; default 'Python'

common_timebase [bool] flag to hint that the signals have the same timebase. Only set this if you know for sure that all appended channels share the same time base

units [dict] will contain the signal units mapped to the singal names when appending a pandas DataFrame

Examples

```
>>> # case 1 conversion type None
>>> s1 = np.array([1, 2, 3, 4, 5])
>>> s2 = np.array([-1, -2, -3, -4, -5])
>>> s3 = np.array([0.1, 0.04, 0.09, 0.16, 0.25])
>>> t = np.array([0.001, 0.002, 0.003, 0.004, 0.005])
>>> names = ['Positive', 'Negative', 'Float']
>>> units = ['+', '-', '.f']
>>> info = {}
>>> s1 = Signal(samples=s1, timstamps=t, unit='+', name='Positive')
>>> s2 = Signal(samples=s2, timstamps=t, unit='-', name='Negative')
>>> s3 = Signal(samples=s3, timstamps=t, unit='flts', name='Floats')
>>> mdf = MDF3('new.mdf')
>>> mdf.append([s1, s2, s3], 'created by asammdf v1.1.0')
>>> # case 2: VTAB conversions from channels inside another file
>>> mdf1 = MDF3('in.mdf')
>>> ch1 = mdf1.get("Channel1_VTAB")
>>> ch2 = mdf1.get("Channel2_VTABR")
>>> sigs = [ch1, ch2]
>>> mdf2 = MDF3('out.mdf')
>>> mdf2.append(sigs, 'created by asammdf v1.1.0')
>>> df = pd.DataFrame.from_dict({'s1': np.array([1, 2, 3, 4, 5]), 's2': np.
↪array([-1, -2, -3, -4, -5])})
>>> units = {'s1': 'V', 's2': 'A'}
>>> mdf2.append(df, units=units)
```

close()
if the MDF was created with memory='minimum' and new channels have been appended, then this must be called just before the object is not used anymore to clean-up the temporary file

configure(***, *read_fragment_size=None*, *write_fragment_size=None*, *use_display_names=None*, *single_bit_uint_as_bool=None*, *integer_interpolation=None*)
configure MDF parameters

Parameters

read_fragment_size [int] size hint of split data blocks, default 8MB; if the initial size is smaller, then no data list is used. The actual split size depends on the data groups' records

size

write_fragment_size [int] size hint of split data blocks, default 4MB; if the initial size is smaller, then no data list is used. The actual split size depends on the data groups' records size. Maximum size is 4MB to ensure compatibility with CANape

use_display_names [bool] search for display name in the Channel XML comment

single_bit_uint_as_bool [bool] return single bit channels are np.bool arrays

integer_interpolation [int] interpolation mode for integer channels:

- 0 - repeat previous sample
- 1 - use linear interpolation

extend(*index*, *signals*)

Extend a group with new samples. *signals* contains (values, invalidation_bits) pairs for each extended signal. Since MDF3 does not support invalidation bits, the second item of each pair must be None. The first pair is the master channel's pair, and the next pairs must respect the same order in which the signals were appended. The samples must have raw or physical values according to the *Signals* used for the initial append.

Parameters

index [int] group index

signals [list] list of (numpy.ndarray, None) objects

Examples

```
>>> # case 1 conversion type None
>>> s1 = np.array([1, 2, 3, 4, 5])
>>> s2 = np.array([-1, -2, -3, -4, -5])
>>> s3 = np.array([0.1, 0.04, 0.09, 0.16, 0.25])
>>> t = np.array([0.001, 0.002, 0.003, 0.004, 0.005])
>>> names = ['Positive', 'Negative', 'Float']
>>> units = ['+', '-', '.f']
>>> s1 = Signal(samples=s1, timstamps=t, unit='+', name='Positive')
>>> s2 = Signal(samples=s2, timstamps=t, unit='-', name='Negative')
>>> s3 = Signal(samples=s3, timstamps=t, unit='flts', name='Floats')
>>> mdf = MDF3('new.mdf')
>>> mdf.append([s1, s2, s3], 'created by asammdf v1.1.0')
>>> t = np.array([0.006, 0.007, 0.008, 0.009, 0.010])
>>> mdf2.extend(0, [(t, None), (s1.samples, None), (s2.samples, None), (s3.
↪samples, None)])
```

get(*name=None*, *group=None*, *index=None*, *raster=None*, *samples_only=False*, *data=None*, *raw=False*, *ignore_invalidation_bits=False*, *source=None*, *record_offset=0*, *record_count=None*, *copy_master=True*)

Gets channel samples. Channel can be specified in two ways:

- using the first positional argument *name*
  - if *source* is given this will be first used to validate the channel selection
  - if there are multiple occurances for this channel then the *group* and *index* arguments can be used to select a specific group.
  - if there are multiple occurances for this channel and either the *group* or *index* arguments is None then a warning is issued

- using the group number (keyword argument *group*) and the channel number (keyword argument *index*). Use *info* method for group and channel numbers

If the *raster* keyword argument is not *None* the output is interpolated accordingly.

Parameters

name [string] name of channel

group [int] 0-based group index

index [int] 0-based channel index

raster [float] time raster in seconds

samples_only [bool] if *True* return only the channel samples as numpy array; if *False* return a *Signal* object

data [bytes] prevent redundant data read by providing the raw data group samples

raw [bool] return channel samples without appling the conversion rule; default *False*

ignore_invalidation_bits [bool] only defined to have the same API with the MDF v4

source [str] source name used to select the channel

record_offset [int] if *data=None* use this to select the record offset from which the group data should be loaded

copy_master [bool] make a copy of the timebase for this channel

Returns

res [(numpy.array, None) | Signal] returns *Signal* if *samples_only*=*False* (default option), otherwise returns a (numpy.array, None) tuple (for compatibility with MDF v4 class.

The *Signal* samples are

- numpy recarray for channels that have CDBLOCK or BYTEARRAY type channels
- numpy array for all the rest

Raises

MdfException :

*** if the channel name is not found**

*** if the group index is out of range**

*** if the channel index is out of range**

Examples

```
>>> from asammdf import MDF, Signal
>>> import numpy as np
>>> t = np.arange(5)
>>> s = np.ones(5)
>>> mdf = MDF(version='3.30')
>>> for i in range(4):
...     sigs = [Signal(s*(i*10+j), t, name='Sig') for j in range(1, 4)]
...     mdf.append(sigs)
...
>>> # first group and channel index of the specified channel name
```

(continues on next page)

(continued from previous page)

```
...
>>> mdf.get('Sig')
UserWarning: Multiple occurances for channel "Sig". Using first occurance␣
↪from data group 4. Provide both "group" and "index" arguments to select␣
↪another data group
<Signal Sig:
        samples=[ 1.  1.  1.  1.  1.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
>>> # first channel index in the specified group
...
>>> mdf.get('Sig', 1)
<Signal Sig:
        samples=[ 11.  11.  11.  11.  11.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
>>> # channel named Sig from group 1 channel index 2
...
>>> mdf.get('Sig', 1, 2)
<Signal Sig:
        samples=[ 12.  12.  12.  12.  12.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
>>> # channel index 1 or group 2
...
>>> mdf.get(None, 2, 1)
<Signal Sig:
        samples=[ 21.  21.  21.  21.  21.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
>>> mdf.get(group=2, index=1)
<Signal Sig:
        samples=[ 21.  21.  21.  21.  21.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
>>> mdf.get('Sig', source='VN7060')
<Signal Sig:
        samples=[ 12.  12.  12.  12.  12.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
```

get_channel_comment(*name=None*, *group=None*, *index=None*)

Gets channel comment. Channel can be specified in two ways:

- using the first positional argument *name*

  - if there are multiple occurances for this channel then the *group* and *index* arguments can be used to select a specific group.
  - if there are multiple occurances for this channel and either the *group* or *index* arguments is None then a warning is issued

- using the group number (keyword argument *group*) and the channel number (keyword argument *index*). Use *info* method for group and channel numbers

If the *raster* keyword argument is not *None* the output is interpolated accordingly.

Parameters

name [string] name of channel

group [int] 0-based group index

index [int] 0-based channel index

Returns

comment [str] found channel comment

get_channel_name(*group*, *index*)
Gets channel name.

Parameters

group [int] 0-based group index

index [int] 0-based channel index

Returns

name [str] found channel name

get_channel_unit(*name=None*, *group=None*, *index=None*)
Gets channel unit.

Channel can be specified in two ways:

- using the first positional argument *name*
  - if there are multiple occurances for this channel then the *group* and *index* arguments can be used to select a specific group.
  - if there are multiple occurances for this channel and either the *group* or *index* arguments is None then a warning is issued
- using the group number (keyword argument *group*) and the channel number (keyword argument *index*). Use *info* method for group and channel numbers

If the *raster* keyword argument is not *None* the output is interpolated accordingly.

Parameters

name [string] name of channel

group [int] 0-based group index

index [int] 0-based channel index

Returns

unit [str] found channel unit

get_master(*index*, *data=None*, *raster=None*, *record_offset=0*, *record_count=None*, *copy_master=True*)
returns master channel samples for given group

Parameters

index [int] group index

data [(bytes, int)] (data block raw bytes, fragment offset); default None

raster [float] raster to be used for interpolation; default None

record_offset [int] if *data=None* use this to select the record offset from which the group data should be loaded

Returns

t [numpy.array] master channel samples

info()
get MDF information as a dict

Examples

```
>>> mdf = MDF3('test.mdf')
>>> mdf.info()
```

iter_get_triggers()
generator that yields triggers

Returns

trigger_info [dict] trigger information with the following keys:

- comment : trigger comment
- time : trigger time
- pre_time : trigger pre time
- post_time : trigger post time
- index : trigger index
- group : data group index of trigger

save(*dst*, *overwrite=False*, *compression=0*)
Save MDF to *dst*. If overwrite is *True* then the destination file is overwritten, otherwise the file name is appended with '.<cntr>', were '<cntr>' is the first counter that produces a new file name (that does not already exist in the filesystem).

Parameters

dst [str | pathlib.Path] destination file name

overwrite [bool] overwrite flag, default *False*

compression [int] does nothing for mdf version3; introduced here to share the same API as mdf version 4 files

Returns

output_file [str] output file name

2.2.1 MDF version 2 & 3 blocks

The following classes implement different MDF version3 blocks.

Channel Class

class asammdf.blocks.v2_v3_blocks.**Channel**(***kwargs*)

CNBLOCK class

If the *load_metadata* keyword argument is not provided or is False, then the conversion, source and display name information is not processed.

CNBLOCK fields

- `id` - bytes : block ID; always b'CN'
- `block_len` - int : block bytes size
- `next_ch_addr` - int : next CNBLOCK address
- `conversion_addr` - int : address of channel conversion block
- `source_addr` - int : address of channel source block
- `component_addr` - int : address of dependency block (CDBLOCK) of this channel
- `comment_addr` - int : address of TXBLOCK that contains the channel comment
- `channel_type` - int : integer code for channel type
- `short_name` - bytes : short signal name
- `description` - bytes : signal description
- `start_offset` - int : start offset in bits to determine the first bit of the signal in the data record
- `bit_count` - int : channel bit count
- `data_type` - int : integer code for channel data type
- `range_flag` - int : value range valid flag
- `min_raw_value` - float : min raw value of all samples
- `max_raw_value` - float : max raw value of all samples
- `sampling_rate` - float : sampling rate in *'s'* for a virtual time channel
- `long_name_addr` - int : address of TXBLOCK that contains the channel's name
- `display_name_addr` - int : address of TXBLOCK that contains the channel's display name
- `aditional_byte_offset` - int : additional Byte offset of the channel in the data recor

Other attributes

- `address` - int : block address inside mdf file
- `comment` - str : channel comment
- `conversion` - ChannelConversion : channel conversion; *None* if the channel has no conversion
- `display_name` - str : channel display name
- `name` - str : full channel name
- `source` - SourceInformation : channel source information; *None* if the channel has no source information

Parameters

address [int] block address; to be used for objects created from file

stream [handle] file handle; to be used for objects created from file

load_metadata [bool] option to load conversion, source and display_name; default *True*

for dynamically created objects : see the key-value pairs

Examples

```
>>> with open('test.mdf', 'rb') as mdf:
...     ch1 = Channel(stream=mdf, address=0xBA52)
>>> ch2 = Channel()
>>> ch1.name
'VehicleSpeed'
>>> ch1['id']
b'CN'
```

ChannelConversion Class

class asammdf.blocks.v2_v3_blocks.**ChannelConversion**(***kwargs*)

CCBLOCK class

ChannelConversion has the following common fields

- `id` - bytes : block ID; always b'CC'
- `block_len` - int : block bytes size
- `range_flag` - int : value range valid flag
- `min_phy_value` - float : min raw value of all samples
- `max_phy_value` - float : max raw value of all samples
- `unit` - bytes : physical unit
- `conversion_type` - int : integer code for conversion type
- `ref_param_nr` - int : number of referenced parameters

ChannelConversion has the following specific fields

- linear conversion
  - `a` - float : factor
  - `b` - float : offset
  - `CANapeHiddenExtra` - bytes : sometimes CANape appends extra information; not compliant with MDF specs
- algebraic conversion
  - `formula` - bytes : ecuation as string
- polynomial or rational conversion
  - `P1` to `P6` - float : parameters
- exponential or logarithmic conversion
  - `P1` to `P7` - float : parameters
- tabular with or without interpolation (grouped by index)
  - `raw_<N>` - int : N-th raw value (X axis)

  - phys_<N> - float : N-th physical value (Y axis)
- text table conversion
  - param_val_<N> - int : N-th raw value (X axis)
  - text_<N> - N-th text physical value (Y axis)
- text range table conversion
  - default_lower - float : default lower raw value
  - default_upper - float : default upper raw value
  - default_addr - int : address of default text physical value
  - lower_<N> - float : N-th lower raw value
  - upper_<N> - float : N-th upper raw value
  - text_<N> - int : address of N-th text physical value

Other attributes

- address - int : block address inside mdf file
- formula - str : formula string in case of algebraic conversion
- referenced_blocks - list : list of CCBLOCK/TXBLOCK referenced by the conversion
- unit - str : physical unit

Parameters

address [int] block address inside mdf file

raw_bytes [bytes] complete block read from disk

stream [file handle] mdf file handle

for dynamically created objects : see the key-value pairs

Examples

```
>>> with open('test.mdf', 'rb') as mdf:
...     cc1 = ChannelConversion(stream=mdf, address=0xBA52)
>>> cc2 = ChannelConversion(conversion_type=0)
>>> cc1['b'], cc1['a']
0, 100.0
```

ChannelDependency Class

class asammdf.blocks.v2_v3_blocks.**ChannelDependency**(***kwargs*)

CDBLOCK class

CDBLOCK fields

- id - bytes : block ID; always b'CD'
- block_len - int : block bytes size
- dependency_type - int : integer code for dependency type
- sd_nr - int : total number of signals dependencies

- dg_<N> - address of data group block (DGBLOCK) of N-th signal dependency
- dg_<N> - address of channel group block (CGBLOCK) of N-th signal dependency
- dg_<N> - address of channel block (CNBLOCK) of N-th signal dependency
- dim_<K> - int : Optional size of dimension *K* for N-dimensional dependency

Other attributes * address - int : block address inside mdf file * referenced_channels - list : list of (group index, channel index) pairs

Parameters

stream [file handle] mdf file handle

address [int] block address inside mdf file

for dynamically created objects : see the key-value pairs

ChannelExtension Class

class asammdf.blocks.v2_v3_blocks.**ChannelExtension**(***kwargs*)

CEBLOCK class

CEBLOCK has the following common fields

- id - bytes : block ID; always b'CE'
- block_len - int : block bytes size
- type - int : extension type identifier

CEBLOCK has the following specific fields

- for DIM block
  - module_nr - int: module number
  - module_address - int : module address
  - description - bytes : module description
  - ECU_identification - bytes : identification of ECU
  - reserved0 - bytes : reserved bytes
- for Vector CAN block
  - CAN_id - int : CAN message ID
  - CAN_ch_index - int : index of CAN channel
  - message_name - bytes : message name
  - sender_name - btyes : sender name
  - reserved0 - bytes : reserved bytes

Other attributes

- address - int : block address inside mdf file
- comment - str : extension comment
- name - str : extension name
- path - str : extension path

Parameters

stream [file handle] mdf file handle

address [int] block address inside mdf file

for dynamically created objects : see the key-value pairs

ChannelGroup Class

class asammdf.blocks.v2_v3_blocks.**ChannelGroup**(***kwargs*)

CGBLOCK class

CGBLOCK fields

- `id` - bytes : block ID; always b'CG'
- `block_len` - int : block bytes size
- `next_cg_addr` - int : next CGBLOCK address
- `first_ch_addr` - int : address of first channel block (CNBLOCK)
- `comment_addr` - int : address of TXBLOCK that contains the channel group comment
- `record_id` - int : record ID used as identifier for a record if the DGBLOCK defines a number of record IDs > 0 (unsorted group)
- `ch_nr` - int : number of channels
- `samples_byte_nr` - int : size of data record in bytes without record ID
- `cycles_nr` - int : number of cycles (records) of this type in the data block
- `sample_reduction_addr` - int : addresss to first sample reduction block

Other attributes

- `address` - int : block address inside mdf file
- `comment` - str : channel group comment

Parameters

stream [file handle] mdf file handle

address [int] block address inside mdf file

for dynamically created objects : see the key-value pairs

Examples

```
>>> with open('test.mdf', 'rb') as mdf:
...     cg1 = ChannelGroup(stream=mdf, address=0xBA52)
>>> cg2 = ChannelGroup(sample_bytes_nr=32)
>>> hex(cg1.address)
0xBA52
>>> cg1['id']
b'CG'
```

DataGroup Class

class asammdf.blocks.v2_v3_blocks.**DataGroup**(***kwargs*)

DGBLOCK class

DGBLOCK fields

- `id` - bytes : block ID; always b'DG'
- `block_len` - int : block bytes size
- `next_dg_addr` - int : next DGBLOCK address
- `first_cg_addr` - int : address of first channel group block (CGBLOCK)
- `trigger_addr` - int : address of trigger block (TRBLOCK)
- `data_block_addr` - addrfss of data block
- `cg_nr` - int : number of channel groups
- `record_id_len` - int : number of record IDs in the data block
- `reserved0` - bytes : reserved bytes

Other attributes

- `address` - int : block address inside mdf file

Parameters

stream [file handle] mdf file handle

address [int] block address inside mdf file

for dynamically created objects : see the key-value pairs

FileIdentificationBlock Class

class asammdf.blocks.v2_v3_blocks.**FileIdentificationBlock**(***kwargs*)

IDBLOCK class

IDBLOCK fields

- `file_identification` - bytes : file identifier
- `version_str` - bytes : format identifier
- `program_identification` - bytes : creator program identifier
- `byte_order` - int : integer code for byte order (endiannes)
- `float_format` - int : integer code for floating-point format
- `mdf_version` - int : version number of MDF format
- `code_page` - int : unicode code page number
- `reserved0` - bytes : reserved bytes
- `reserved1` - bytes : reserved bytes
- `unfinalized_standard_flags` - int : standard flags for unfinalized MDF
- `unfinalized_custom_flags` - int : custom flags for unfinalized MDF

Other attributes

- address - int : block address inside mdf file; should be 0 always

Parameters

stream [file handle] mdf file handle

version [int] mdf version in case of new file (dynamically created)

HeaderBlock Class

class asammdf.blocks.v2_v3_blocks.**HeaderBlock**(***kwargs*)

HDBLOCK class

HDBLOCK fields

- id - bytes : block ID; always b'HD'
- block_len - int : block bytes size
- first_dg_addr - int : address of first data group block (DGBLOCK)
- comment_addr - int : address of TXBLOCK taht contains the measurement file comment
- program_addr - int : address of program block (PRBLOCK)
- dg_nr - int : number of data groups
- date - bytes : date at which the recording was started in "DD:MM:YYYY" format
- time - btyes : time at which the recording was started in "HH:MM:SS" format
- author - btyes : author name
- organization - bytes : organization name
- project - bytes : project name
- subject - bytes : subject

Since version 3.2 the following extra keys were added

- abs_time - int : time stamp at which recording was started in nanoseconds.
- tz_offset - int : UTC time offset in hours (= GMT time zone)
- time_quality - int : time quality class
- timer_identification - bytes : timer identification (time source)

Other attributes

- address - int : block address inside mdf file; should be 64 always
- comment - int : file comment
- ‘‘program ‘‘ - ProgramBlock : program block
- author - str : measurement author
- department - str : author's department
- project - str : working project
- ‘‘subject ‘‘ - str : measurement subject

Parameters

stream [file handle] mdf file handle

version [int] mdf version in case of new file (dynamically created)

ProgramBlock Class

class asammdf.blocks.v2_v3_blocks.**ProgramBlock**(***kwargs*)

PRBLOCK class

PRBLOCK fields

- `id` - bytes : block ID; always b'PR'
- `block_len` - int : block bytes size
- `data` - btyes : creator program free format data

Other attributes * `address` - int : block address inside mdf file

Parameters

stream [file handle] mdf file handle

address [int] block address inside mdf file

TextBlock Class

class asammdf.blocks.v2_v3_blocks.**TextBlock**(***kwargs*)

TXBLOCK class

TXBLOCK fields

- `id` - bytes : block ID; always b'TX'
- `block_len` - int : block bytes size
- `text` - bytes : text content

Other attributes

- `address` - int : block address inside mdf file

Parameters

stream [file handle] mdf file handle

address [int] block address inside mdf file

text [bytes | str] bytes or str for creating a new TextBlock

Examples

```
>>> tx1 = TextBlock(text='VehicleSpeed')
>>> tx1.text_str
'VehicleSpeed'
>>> tx1['text']
b'VehicleSpeed'
```

TriggerBlock Class

class asammdf.blocks.v2_v3_blocks.**TriggerBlock**(***kwargs*)

TRBLOCK class

TRBLOCK fields

- `id` - bytes : block ID; always b'TR'
- `block_len` - int : block bytes size
- `text_addr` - int : address of TXBLOCK that contains the trigger comment text
- `trigger_events_nr` - int : number of trigger events
- `trigger_<N>_time` - float : trigger time [s] of trigger's N-th event
- `trigger_<N>_pretime` - float : pre trigger time [s] of trigger's N-th event
- `trigger_<N>_posttime` - float : post trigger time [s] of trigger's N-th event

Other attributes

- `address` - int : block address inside mdf file
- ‘‘comment ‘‘ - str : trigger comment

Parameters

stream [file handle] mdf file handle

address [int] block address inside mdf file

2.3 MDF4

class asammdf.blocks.mdf_v4.**MDF4**(*name=None*, *version='4.10'*, ***kwargs*)

The *header* attibute is a *HeaderBlock*.

The *groups* attribute is a list of dicts, each one with the following keys:

- `data_group` - DataGroup object
- `channel_group` - ChannelGroup object
- `channels` - list of Channel objects with the same order as found in the mdf file
- `channel_dependencies` - list of *ChannelArrayBlock* in case of channel arrays; list of Channel objects in case of structure channel composition
- `data_block` - address of data block
- `data_location`- integer code for data location (original file, temporary file or memory)
- `data_block_addr` - list of raw samples starting addresses
- `data_block_type` - list of codes for data block type
- `data_block_size` - list of raw samples block size
- `sorted` - sorted indicator flag
- `record_size` - dict that maps record ID's to record sizes in bytes (including invalidation bytes)
- `param` - row size used for tranposizition, in case of tranposed zipped blockss

Parameters

name [string] mdf file name (if provided it must be a real file name) or file-like object

- if *full* the data group binary data block will be memorised in RAM
- if *low* the channel data is read from disk on request, and the metadata is memorized into RAM
- if *minimum* only minimal data is memorized into RAM

version [string] mdf file version ('4.00', '4.10', '4.11'); default '4.10'

callback [function] keyword only argument: function to call to update the progress; the function must accept two arguments (the current progress and maximum progress value)

use_display_names [bool] keyword only argument: for MDF4 files parse the XML channel comment to search for the display name; XML parsing is quite expensive so setting this to *False* can decrease the loading times very much; default *False*

Attributes

attachments [list] list of file attachments

channels_db [dict] used for fast channel access by name; for each name key the value is a list of (group index, channel index) tuples

events [list] list event blocks

file_comment [TextBlock] file comment TextBlock

file_history [list] list of (FileHistory, TextBlock) pairs

groups [list] list of data group dicts

header [HeaderBlock] mdf file header

identification [FileIdentificationBlock] mdf file start block

masters_db [dict]

used for fast master channel access; for each group index key the value is the master channel index

name [string] mdf file name

version [str] mdf version

append(*signals*, *source_info='Python'*, *common_timebase=False*, *units=None*)

Appends a new data group.

For channel dependencies type Signals, the *samples* attribute must be a numpy.recarray

Parameters

signals [list | Signal | pandas.DataFrame] list of *Signal* objects, or a single *Signal* object, or a pandas *DataFrame* object. All bytes columns in the pandas *DataFrame* must be *utf-8* encoded

source_info [str] source information; default 'Python'

common_timebase [bool] flag to hint that the signals have the same timebase. Only set this if you know for sure that all appended channels share the same time base

units [dict] will contain the signal units mapped to the singal names when appending a pandas DataFrame

Examples

```
>>> # case 1 conversion type None
>>> s1 = np.array([1, 2, 3, 4, 5])
>>> s2 = np.array([-1, -2, -3, -4, -5])
>>> s3 = np.array([0.1, 0.04, 0.09, 0.16, 0.25])
>>> t = np.array([0.001, 0.002, 0.003, 0.004, 0.005])
>>> names = ['Positive', 'Negative', 'Float']
>>> units = ['+', '-', '.f']
>>> info = {}
>>> s1 = Signal(samples=s1, timstamps=t, unit='+', name='Positive')
>>> s2 = Signal(samples=s2, timstamps=t, unit='-', name='Negative')
>>> s3 = Signal(samples=s3, timstamps=t, unit='flts', name='Floats')
>>> mdf = MDF4('new.mdf')
>>> mdf.append([s1, s2, s3], 'created by asammdf v4.0.0')
>>> # case 2: VTAB conversions from channels inside another file
>>> mdf1 = MDF4('in.mf4')
>>> ch1 = mdf1.get("Channel1_VTAB")
>>> ch2 = mdf1.get("Channel2_VTABR")
>>> sigs = [ch1, ch2]
>>> mdf2 = MDF4('out.mf4')
>>> mdf2.append(sigs, 'created by asammdf v4.0.0')
>>> mdf2.append(ch1, 'just a single channel')
>>> df = pd.DataFrame.from_dict({'s1': np.array([1, 2, 3, 4, 5]), 's2': np.
↪array([-1, -2, -3, -4, -5])})
>>> units = {'s1': 'V', 's2': 'A'}
>>> mdf2.append(df, units=units)
```

attach(*data*, *file_name=None*, *comment=None*, *compression=True*, *mime='application/octet-stream'*, *embedded=True*)

attach embedded attachment as application/octet-stream

Parameters

data [bytes] data to be attached

file_name [str] string file name

comment [str] attachment comment

compression [bool] use compression for embedded attachment data

mime [str] mime type string

embedded [bool] attachment is embedded in the file

Returns

index [int] new attachment index

close()

if the MDF was created with memory=False and new channels have been appended, then this must be called just before the object is not used anymore to clean-up the temporary file

configure(*, *read_fragment_size=None*, *write_fragment_size=None*, *use_display_names=None*, *single_bit_uint_as_bool=None*, *integer_interpolation=None*)

configure MDF parameters

Parameters

read_fragment_size [int] size hint of split data blocks, default 8MB; if the initial size is smaller, then no data list is used. The actual split size depends on the data groups' records size

write_fragment_size [int] size hint of split data blocks, default 4MB; if the initial size is smaller, then no data list is used. The actual split size depends on the data groups' records size. Maximum size is 4MB to ensure compatibility with CANape

use_display_names [bool] search for display name in the Channel XML comment

single_bit_uint_as_bool [bool] return single bit channels are np.bool arrays

integer_interpolation [int] interpolation mode for integer channels:

- 0 - repeat previous sample
- 1 - use linear interpolation

extend(*index*, *signals*)

Extend a group with new samples. *signals* contains (values, invalidation_bits) pairs for each extended signal. The first pair is the master channel's pair, and the next pairs must respect the same order in which the signals were appended. The samples must have raw or physical values according to the *Signals* used for the initial append.

Parameters

index [int] group index

signals [list] list on (numpy.ndarray, numpy.ndarray) objects

Examples

```
>>> # case 1 conversion type None
>>> s1 = np.array([1, 2, 3, 4, 5])
>>> s2 = np.array([-1, -2, -3, -4, -5])
>>> s3 = np.array([0.1, 0.04, 0.09, 0.16, 0.25])
>>> t = np.array([0.001, 0.002, 0.003, 0.004, 0.005])
>>> names = ['Positive', 'Negative', 'Float']
>>> units = ['+', '-', '.f']
>>> s1 = Signal(samples=s1, timstamps=t, unit='+', name='Positive')
>>> s2 = Signal(samples=s2, timstamps=t, unit='-', name='Negative')
>>> s3 = Signal(samples=s3, timstamps=t, unit='flts', name='Floats')
>>> mdf = MDF3('new.mdf')
>>> mdf.append([s1, s2, s3], 'created by asammdf v1.1.0')
>>> t = np.array([0.006, 0.007, 0.008, 0.009, 0.010])
>>> # extend without invalidation bits
>>> mdf2.extend(0, [(t, None), (s1, None), (s2, None), (s3, None)])
>>> # some invaldiation btis
>>> s1_inv = np.array([0,0,0,1,1], dtype=np.bool)
>>> mdf2.extend(0, [(t, None), (s1.samples, None), (s2.samples, None), (s3.
↪samples, None)])
```

extract_attachment(*address=None*, *index=None*)

extract attachment data by original address or by index. If it is an embedded attachment, then this method creates the new file according to the attachment file name information

Parameters

address [int] attachment index; default *None*

index [int] attachment index; default *None*

Returns

data [(bytes, pathlib.Path)] tuple of attachment data and path

get(*name=None*, *group=None*, *index=None*, *raster=None*, *samples_only=False*, *data=None*, *raw=False*, *ignore_invalidation_bits=False*, *source=None*, *record_offset=0*, *record_count=None*, *copy_master=True*)

Gets channel samples. The raw data group samples are not loaded to memory so it is advised to use `filter` or `select` instead of performing several `get` calls.

Channel can be specified in two ways:

- using the first positional argument *name*
  - if *source* is given this will be first used to validate the channel selection
  - if there are multiple occurances for this channel then the *group* and *index* arguments can be used to select a specific group.
  - if there are multiple occurances for this channel and either the *group* or *index* arguments is None then a warning is issued
- using the group number (keyword argument *group*) and the channel number (keyword argument *index*). Use *info* method for group and channel numbers

If the *raster* keyword argument is not *None* the output is interpolated accordingly

Parameters

name [string] name of channel

group [int] 0-based group index

index [int] 0-based channel index

raster [float] time raster in seconds

samples_only [bool]

if *True* return only the channel samples as numpy array; if *False* return a *Signal* object

data [bytes] prevent redundant data read by providing the raw data group samples

raw [bool] return channel samples without appling the conversion rule; default *False*

ignore_invalidation_bits [bool] option to ignore invalidation bits

source [str] source name used to select the channel

record_offset [int] if *data=None* use this to select the record offset from which the group data should be loaded

record_count [int] number of records to read; default *None* and in this case all available records are used

copy_master [bool] make a copy of the timebase for this channel

Returns

res [(numpy.array, numpy.array) | Signal] returns *Signal* if *samples_only*=*False* (default option), otherwise returns a (numpy.array, numpy.array) tuple of samples and invalidation bits. If invalidation bits are not used or if *ignore_invalidation_bits* if False, then the second item will be None.

The *Signal* samples are:

- numpy recarray for channels that have composition/channel array address or for channel of type CANOPENDATE, CANOPENTIME
- numpy array for all the rest

Raises

MdfException :

*** if the channel name is not found**

*** if the group index is out of range**

*** if the channel index is out of range**

Examples

```
>>> from asammdf import MDF, Signal
>>> import numpy as np
>>> t = np.arange(5)
>>> s = np.ones(5)
>>> mdf = MDF(version='4.10')
>>> for i in range(4):
...     sigs = [Signal(s*(i*10+j), t, name='Sig') for j in range(1, 4)]
...     mdf.append(sigs)
...
>>> # first group and channel index of the specified channel name
...
>>> mdf.get('Sig')
UserWarning: Multiple occurances for channel "Sig". Using first occurance␣
↪from data group 4. Provide both "group" and "index" arguments to select␣
↪another data group
<Signal Sig:
        samples=[ 1.  1.  1.  1.  1.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
>>> # first channel index in the specified group
...
>>> mdf.get('Sig', 1)
<Signal Sig:
        samples=[ 11.  11.  11.  11.  11.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
>>> # channel named Sig from group 1 channel index 2
...
>>> mdf.get('Sig', 1, 2)
<Signal Sig:
        samples=[ 12.  12.  12.  12.  12.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
>>> # channel index 1 or group 2
...
>>> mdf.get(None, 2, 1)
<Signal Sig:
        samples=[ 21.  21.  21.  21.  21.]
        timestamps=[0 1 2 3 4]
        unit=""
```

(continues on next page)

(continued from previous page)

```
        info=None
        comment="">
>>> mdf.get(group=2, index=1)
<Signal Sig:
        samples=[ 21.  21.  21.  21.  21.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
>>> # validation using source name
...
>>> mdf.get('Sig', source='VN7060')
<Signal Sig:
        samples=[ 12.  12.  12.  12.  12.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
```

get_can_signal (*name*, *database=None*, *db=None*, *ignore_invalidation_bits=False*)

get CAN message signal. You can specify an external CAN database (*database* argument) or canmatrix databse object that has already been loaded from a file (*db* argument).

The signal name can be specified in the following ways

- `CAN<ID>.<MESSAGE_NAME>.<SIGNAL_NAME>` - the *ID* value starts from 1 and must match the ID found in the measurement (the source CAN bus ID) Example: CAN1.Wheels.FL_WheelSpeed
- `CAN<ID>.CAN_DataFrame_<MESSAGE_ID>.<SIGNAL_NAME>` - the *ID* value starts from 1 and the *MESSAGE_ID* is the decimal message ID as found in the database. Example: CAN1.CAN_DataFrame_218.FL_WheelSpeed
- `<MESSAGE_NAME>.SIGNAL_NAME` - in this case the first occurence of the message name and signal are returned (the same message could be found on muplit CAN buses; for example on CAN1 and CAN3) Example: Wheels.FL_WheelSpeed
- `CAN_DataFrame_<MESSAGE_ID>.<SIGNAL_NAME>` - in this case the first occurence of the message name and signal are returned (the same message could be found on muplit CAN buses; for example on CAN1 and CAN3). Example: CAN_DataFrame_218.FL_WheelSpeed
- `<SIGNAL_NAME>` - in this case the first occurence of the signal name is returned (the same signal anme coudl be found in multiple messages and on multiple CAN buses). Example: FL_WheelSpeed

Parameters

name [str] signal name

database [str] path of external CAN database file (.dbc or .arxml); default *None*

db [canmatrix.database] canmatrix CAN database object; default *None*

ignore_invalidation_bits [bool] option to ignore invalidation bits

Returns

sig [Signal] Signal object with the physical values

get_channel_comment (*name=None*, *group=None*, *index=None*)

Gets channel comment.

Channel can be specified in two ways:

- using the first positional argument *name*
  - if there are multiple occurrences for this channel then the *group* and *index* arguments can be used to select a specific group.
  - if there are multiple occurrences for this channel and either the *group* or *index* arguments is None then a warning is issued
- using the group number (keyword argument *group*) and the channel number (keyword argument *index*). Use *info* method for group and channel numbers

If the *raster* keyword argument is not *None* the output is interpolated accordingly.

Parameters

name [string] name of channel

group [int] 0-based group index

index [int] 0-based channel index

Returns

comment [str] found channel comment

get_channel_name(*group*, *index*)
Gets channel name.

Parameters

group [int] 0-based group index

index [int] 0-based channel index

Returns

name [str] found channel name

get_channel_unit(*name=None*, *group=None*, *index=None*)
Gets channel unit.

Channel can be specified in two ways:

- using the first positional argument *name*
  - if there are multiple occurrences for this channel then the *group* and *index* arguments can be used to select a specific group.
  - if there are multiple occurrences for this channel and either the *group* or *index* arguments is None then a warning is issued
- using the group number (keyword argument *group*) and the channel number (keyword argument *index*). Use *info* method for group and channel numbers

If the *raster* keyword argument is not *None* the output is interpolated accordingly.

Parameters

name [string] name of channel

group [int] 0-based group index

index [int] 0-based channel index

Returns

unit [str] found channel unit

get_invalidation_bits(*group_index*, *channel*, *fragment*)
get invalidation indexes for the channel

Parameters

group_index [int] group index

channel [Channel] channel object

fragment [(bytes, int)] (fragment bytes, fragment offset)

Returns

invalidation_bits [iterable] iterable of valid channel indexes; if all are valid *None* is returned

get_master(*index*, *data=None*, *raster=None*, *record_offset=0*, *record_count=None*, *copy_master=True*)
returns master channel samples for given group

Parameters

index [int] group index

data [(bytes, int)] (data block raw bytes, fragment offset); default None

raster [float] raster to be used for interpolation; default None

record_offset [int] if *data=None* use this to select the record offset from which the group data should be loaded

record_count [int] number of records to read; default *None* and in this case all available records are used

copy_master [bool] return a copy of the cached master

Returns

t [numpy.array] master channel samples

info()
get MDF information as a dict

Examples

```
>>> mdf = MDF4('test.mdf')
>>> mdf.info()
```

save(*dst*, *overwrite=False*, *compression=0*)
Save MDF to *dst*. If overwrite is *True* then the destination file is overwritten, otherwise the file name is appened with '.<cntr>', were '<cntr>' is the first conter that produces a new file name (that does not already exist in the filesystem)

Parameters

dst [str] destination file name, Default ''

overwrite [bool] overwrite flag, default *False*

compression [int] use compressed data blocks, default 0; valid since version 4.10

- 0 - no compression
- 1 - deflate (slower, but produces smaller files)
- 2 - transposition + deflate (slowest, but produces the smallest files)

Returns

output_file [pathlib.Path] path to saved file

2.3.1 MDF version 4 blocks

The following classes implement different MDF version4 blocks.

AttachmentBlock Class

class asammdf.blocks.v4_blocks.**AttachmentBlock**(***kwargs*)

When adding new attachments only embedded attachments are allowed, with keyword argument *data* of type bytes

AttachmentBlock has the following attributes, that are also available as dict like key-value pairs

ATBLOCK fields

- `id` - bytes : block ID; always b'##AT'
- `reserved0` - int : reserved bytes
- `block_len` - int : block bytes size
- `links_nr` - int : number of links
- `next_at_addr` - int : next ATBLOCK address
- `file_name_addr` - int : address of TXBLOCK that contains the attachment file name
- `mime_addr` - int : address of TXBLOCK that contains the attachment mime type description
- `comment_addr` - int : address of TXBLOCK/MDBLOCK that contains the attachment comment
- `flags` - int : ATBLOCK flags
- `creator_index` - int : index of file history block
- `reserved1` - int : reserved bytes
- `md5_sum` - bytes : attachment file md5 sum
- `original_size` - int : original uncompress file size in bytes
- `embedded_size` - int : embedded compressed file size in bytes
- `embedded_data` - bytes : embedded atatchment bytes

Other attributes

- `address` - int : attachment address
- `file_name` - str : attachment file name
- `mime` - str : mime type
- `comment` - str : attachment comment

Parameters

address [int] block address; to be used for objects created from file

stream [handle] file handle; to be used for objects created from file

for dynamically created objects : see the key-value pairs

extract()
extract attachment data

Returns

data [bytes]

Channel Class

class asammdf.blocks.v4_blocks.**Channel**(**kwargs*)
If the *load_metadata* keyword argument is not provided or is False, then the conversion, source and display name information is not processed. Further more if the *parse_xml_comment* is not provided or is False, then the display name information from the channel comment is not processed (this is done to avoid expensive XML operations)

Channel has the following attributes, that are also available as dict like key-value pairs

CNBLOCK fields

- `id` - bytes : block ID; always b'##CN'
- `reserved0` - int : reserved bytes
- `block_len` - int : block bytes size
- `links_nr` - int : number of links
- `next_ch_addr` - int : next ATBLOCK address
- `component_addr` - int : address of first channel in case of structure channel composition, or ChannelArrayBlock in case of arrays file name
- `name_addr` - int : address of TXBLOCK that contains the channel name
- `source_addr` - int : address of channel source block
- `conversion_addr` - int : address of channel conversion block
- `data_block_addr` - int : address of signal data block for VLSD channels
- `unit_addr` - int : address of TXBLOCK that contains the channel unit
- `comment_addr` - int : address of TXBLOCK/MDBLOCK that contains the channel comment
- `attachment_<N>_addr` - int : address of N-th ATBLOCK referenced by the current channel; if no ATBLOCK is referenced there will be no such key-value pair
- `default_X_dg_addr` - int : address of DGBLOCK where the default X axis channel for the current channel is found; this key-value pair will not exist for channels that don't have a default X axis
- `default_X_cg_addr` - int : address of CGBLOCK where the default X axis channel for the current channel is found; this key-value pair will not exist for channels that don't have a default X axis
- `default_X_ch_addr` - int : address of default X axis channel for the current channel; this key-value pair will not exist for channels that don't have a default X axis
- `channel_type` - int : integer code for the channel type
- `sync_type` - int : integer code for the channel's sync type
- `data_type` - int : integer code for the channel's data type
- `bit_offset` - int : bit offset
- `byte_offset` - int : byte offset within the data record

- `bit_count` - int : channel bit count
- `flags` - int : CNBLOCK flags
- `pos_invalidation_bit` - int : invalidation bit position for the current channel if there are invalidation bytes in the data record
- `precision` - int : integer code for teh precision
- `reserved1` - int : reserved bytes
- `min_raw_value` - int : min raw value of all samples
- `max_raw_value` - int : max raw value of all samples
- `lower_limit` - int : min physical value of all samples
- `upper_limit` - int : max physical value of all samples
- `lower_ext_limit` - int : min physical value of all samples
- `upper_ext_limit` - int : max physical value of all samples

Other attributes

- `address` - int : channel address
- `attachments` - list : list of referenced attachment blocks indexes; the index referece to the attachment block index
- `comment` - str : channel comment
- `conversion` - ChannelConversion : channel conversion; *None* if the channel has no conversion
- `display_name` - str : channel display name; this is extracted from the XML channel comment
- `name` - str : channel name
- `source` - SourceInformation : channel source information; *None* if the channel has no source information
- `unit` - str : channel unit

Parameters

address [int] block address; to be used for objects created from file

stream [handle] file handle; to be used for objects created from file

load_metadata [bool] option to load conversion, source and display_name; default *True*

parse_xml_comment [bool] option to parse XML channel comment to search for display name; default *True*

for dynamically created objects : see the key-value pairs

ChannelConversion Class

class asammdf.blocks.v4_blocks.**ChannelConversion**(***kwargs*)

ChannelConversion has the following attributes, that are also available as dict like key-value pairs

CCBLOCK common fields

- `id` - bytes : block ID; always b'##CG'
- `reserved0` - int : reserved bytes
- `block_len` - int : block bytes size

- `links_nr` - int : number of links
- `name_addr` - int : address of TXBLOCK that contains the conversion name
- `unit_addr` - int : address of TXBLOCK that contains the conversion unit
- `comment_addr` - int : address of TXBLOCK/MDBLOCK that contains the conversion comment
- `inv_conv_addr` int : address of invers conversion
- `conversion_type` int : integer code for conversion type
- `precision` - int : integer code for precision
- `flags` - int : conversion block flags
- `ref_param_nr` - int : number fo referenced parameters (linked parameters)
- `val_param_nr` - int : number of value parameters
- `min_phy_value` - float : minimum physical channel value
- `max_phy_value` - float : maximum physical channel value

CCBLOCK specific fields

- linear conversion
  - `a` - float : factor
  - `b` - float : offset
- rational conversion
  - `P1` to `P6` - float : parameters
- algebraic conversion
  - `formula_addr` - address of TXBLOCK that contains the the algebraic conversion formula
- tabluar conversion with or without interpolation
  - `raw_<N>` - float : N-th raw value
  - `phys_<N>` - float : N-th physical value
- tabular range conversion
  - `lower_<N>` - float : N-th lower value
  - `upper_<N>` - float : N-th upper value
  - `phys_<N>` - float : N-th physical value
- tabular value to text conversion
  - `val_<N>` - float : N-th raw value
  - `text_<N>` - int : address of N-th TXBLOCK that contains the physical value
  - `default` - int : address of TXBLOCK that contains the default physical value
- tabular range to text conversion
  - `lower_<N>` - float : N-th lower value
  - `upper_<N>` - float : N-th upper value
  - `text_<N>` - int : address of N-th TXBLOCK that contains the physical value
  - `default` - int : address of TXBLOCK that contains the default physical value

- text to value conversion
  - `val_<N>` - float : N-th physical value
  - `text_<N>` - int : address of N-th TXBLOCK that contains the raw value
  - `val_default` - float : default physical value
- text tranfosrmation (translation) conversion
  - `input_<N>_addr` - int : address of N-th TXBLOCK that contains the raw value
  - `output_<N>_addr` - int : address of N-th TXBLOCK that contains the physical value
  - `default_addr` - int : address of TXBLOCK that contains the default physical value

Other attributes

- `address` - int : channel conversion address
- `comment` - str : channel conversion comment
- `formula` - str : algebraic conversion formula; default ''
- `referenced_blocks` - list : list of refenced blocks; can be TextBlock objects for value to text, and text to text conversions; for partial conversions the referenced blocks can be ChannelConversion obejct as well
- `name` - str : channel conversion name
- `unit` - str : channel conversion unit

ChannelGroup Class

class asammdf.blocks.v4_blocks.**ChannelGroup**(***kwargs*)

ChannelGroup has the following attributes, that are also available as dict like key-value pairs

CGBLOCK fields

- `id` - bytes : block ID; always b'##CG'
- `reserved0` - int : reserved bytes
- `block_len` - int : block bytes size
- `links_nr` - int : number of links
- `next_cg_addr` - int : next channel group address
- `first_ch_addr` - int : address of first channel of this channel group
- `acq_name_addr` - int : address of TextBLock that contains the channel group acquisition name
- `acq_source_addr` - int : addres of SourceInformation that contains the channel group source
- `first_sample_reduction_addr` - int : address of first SRBLOCK; this is considered 0 since sample reduction is not yet supported
- `comment_addr` - int : address of TXBLOCK/MDBLOCK that contains the channel group comment
- `record_id` - int : record ID for thei channel group
- `cycles_nr` - int : number of cycles for this channel group
- `flags` - int : channel group flags
- `path_separator` - int : ordinal for character used as path separator

- `reserved1` - int : reserved bytes
- `samples_byte_nr` - int : number of bytes used for channels samples in the record for this channel group; this does not contain the invalidation bytes
- `invalidation_bytes_nr` - int : number of bytes used for invalidation bits by this channl group

Other attributes

- `acq_name` - str : acquisition name
- `acq_source` - SourceInformation : acquisition source information
- `address` - int : channel group address
- `comment` - str : channel group comment

DataGroup Class

class asammdf.blocks.v4_blocks.**DataGroup**(**kwargs*)

DataGroup has the following attributes, that are also available as dict like key-value pairs

DGBLOCK fields

- `id` - bytes : block ID; always b'##DG'
- `reserved0` - int : reserved bytes
- `block_len` - int : block bytes size
- `links_nr` - int : number of links
- `next_dg_addr` - int : address of next data group block
- `first_cg_addr` - int : address of first channel group for this data group
- `data_block_addr` - int : address of DTBLOCK, DZBLOCK, DLBLOCK or HLBLOCK that contains the raw samples for this data group
- `comment_addr` - int : address of TXBLOCK/MDBLOCK tha contains the data group comment
- `record_id_len` - int : size of record ID used in case of unsorted data groups; can be 1, 2, 4 or 8
- `reserved1` - int : reserved bytes

Other attributes

- `address` - int : dat group address
- `comment` - str : data group comment

DataList Class

class asammdf.blocks.v4_blocks.**DataList**(**kwargs*)

DataList has the following attributes, that are also available as dict like key-value pairs

DLBLOCK common fields

- `id` - bytes : block ID; always b'##DL'
- `reserved0` - int : reserved bytes
- `block_len` - int : block bytes size
- `links_nr` - int : number of links

- next_dl_addr - int : address of next DLBLOCK
- data_block_addr<N> - int : address of N-th data block
- flags - int : data list flags
- reserved1 - int : reserved bytes
- data_block_nr - int : number of data blocks referenced by thsi list

DLBLOCK specific fields

- for equall lenght blocks
  - data_block_len - int : equall uncompressed size in bytes for all referenced data blocks; last block can be smaller
- for variable lenght blocks
  - offset_<N> - int : byte offset of N-th data block

Other attributes

- address - int : data list address

DataBlock Class

class asammdf.blocks.v4_blocks.**DataBlock**(***kwargs*)

Common implementation for DTBLOCK/RDBLOCK/SDBLOCK

DataBlock has the following attributes, that are also available as dict like key-value pairs

DTBLOCK fields

- id - bytes : block ID; b'##DT' for DTBLOCK, b'##RD' for RDBLOCK or b'##SD' for SDBLOCK
- reserved0 - int : reserved bytes
- block_len - int : block bytes size
- links_nr - int : number of links
- data - bytes : raw samples

Other attributes

- address - int : data block address

Parameters

address [int] DTBLOCK/RDBLOCK/SDBLOCK address inside the file

stream [int] file handle

reduction [bool] sample reduction data block

FileIdentificationBlock Class

class asammdf.blocks.v4_blocks.**FileIdentificationBlock**(***kwargs*)

FileIdentificationBlock has the following attributes, that are also available as dict like key-value pairs

IDBLOCK fields

- file_identification - bytes : file identifier

- version_str - bytes : format identifier
- program_identification - bytes : creator program identifier
- reserved0 - bytes : reserved bytes
- mdf_version - int : version number of MDF format
- reserved1 - bytes : reserved bytes
- unfinalized_standard_flags - int : standard flags for unfinalized MDF
- unfinalized_custom_flags - int : custom flags for unfinalized MDF

Other attributes

- address - int : should always be 0

HeaderBlock Class

class asammdf.blocks.v4_blocks.**HeaderBlock**(***kwargs*)

HeaderBlock has the following attributes, that are also available as dict like key-value pairs

HDBLOCK fields

- id - bytes : block ID; always b'##HD'
- reserved0 - int : reserved bytes
- block_len - int : block bytes size
- links_nr - int : number of links
- first_dg_addr - int : address of first DGBLOCK
- file_history_addr - int : address of first FHBLOCK
- channel_tree_addr - int : address of first CHBLOCK
- first_attachment_addr - int : address of first ATBLOCK
- first_event_addr - int : address of first EVBLOCK
- comment_addr - int : address of TXBLOCK/MDBLOCK that contains the file comment
- abs_time - int : time stamp at which recording was started in nanoseconds.
- tz_offset - int : UTC time offset in hours (= GMT time zone)
- daylight_save_time - int : daylight saving time
- time_flags - int : time flags
- time_quality - int : time quality flags
- flags - int : file flags
- reserved1 - int : reserved bytes
- start_angle - int : angle value at measurement start
- start_distance - int : distance value at measurement start

Other attributes

- address - int : header address
- comment - str : file comment

- `author` - str : measurement author
- `department` - str : author's department
- `project` - str : working project
- `subject` - str : measurement subject

start_time
getter and setter the measurement start timestamp

Returns

timestamp [datetime.datetime] start timestamp

SourceInformation Class

class asammdf.blocks.v4_blocks.**SourceInformation**(***kwargs*)
SourceInformation has the following attributes, that are also available as dict like key-value pairs

SIBLOCK fields

- `id` - bytes : block ID; always b'##SI'
- `reserved0` - int : reserved bytes
- `block_len` - int : block bytes size
- `links_nr` - int : number of links
- `name_addr` - int : address of TXBLOCK that contains the source name
- `path_addr` - int : address of TXBLOCK that contains the source path
- `comment_addr` - int : address of TXBLOCK/MDBLOCK tha contains the source comment
- `source_type` - int : integer code for source type
- `bus_type` - int : integer code for source bus type
- `flags` - int : source flags
- `reserved1` - bytes : reserved bytes

Other attributes

- `address` - int : source information address
- `comment` - str : source comment
- `name` - str : source name
- `path` - str : source path

FileHistory Class

class asammdf.blocks.v4_blocks.**FileHistory**(***kwargs*)
FileHistory has the following attributes, that are also available as dict like key-value pairs

FHBLOCK fields

- `id` - bytes : block ID; always b'##FH'
- `reserved0` - int : reserved bytes
- `block_len` - int : block bytes size

- `links_nr` - int : number of links
- `next_fh_addr` - int : address of next FHBLOCK
- `comment_addr` - int : address of TXBLOCK/MDBLOCK that contains the file history comment
- `abs_time` - int : time stamp at which the file modification happened
- `tz_offset` - int : UTC time offset in hours (= GMT time zone)
- `daylight_save_time` - int : daylight saving time
- `time_flags` - int : time flags
- `reserved1` - bytes : reserved bytes

Other attributes

- `address` - int : file history address
- `comment` - str : history comment

TextBlock Class

class asammdf.blocks.v4_blocks.**TextBlock**(***kwargs*)

common TXBLOCK and MDBLOCK class

TextBlock has the following attributes, that are also available as dict like key-value pairs

TXBLOCK fields

- `id` - bytes : block ID; b'##TX' for TXBLOCK and b'##MD' for MDBLOCK
- `reserved0` - int : reserved bytes
- `block_len` - int : block bytes size
- `links_nr` - int : number of links
- `text` - bytes : actual text content

Other attributes

- `address` - int : text block address

Parameters

address [int] block address

stream [handle] file handle

meta [bool] flag to set the block type to MDBLOCK for dynamically created objects; default *False*

text [bytes/str] text content for dynamically created objects

EventBlock Class

class asammdf.blocks.v4_blocks.**EventBlock**(***kwargs*)

EventBlock has the following attributes, that are also available as dict like key-value pairs

EVBLOCK fields

- `id` - bytes : block ID; always b'##EV'

- `reserved0` - int : reserved bytes
- `block_len` - int : block bytes size
- `links_nr` - int : number of links
- `next_ev_addr` - int : address of next EVBLOCK
- `parent_ev_addr` - int : address of parent EVLBOCK
- `range_start_ev_addr` - int : address of EVBLOCK that is the start of the range for which this event is the end
- `name_addr` - int : address of TXBLOCK that contains the event name
- `comment_addr` - int : address of TXBLOCK/MDBLOCK that contains the event comment
- `scope_<N>_addr` - int : address of N-th block that represents a scope for this event (can be CGBLOCK, CHBLOCK, DGBLOCK)
- `attachemnt_<N>_addr` - int : address of N-th attachment referenced by this event
- `event_type` - int : integer code for event type
- `sync_type` - int : integer code for event sync type
- `range_type` - int : integer code for event range type
- `cause` - int : integer code for event cause
- `flags` - int : event flags
- `reserved1` - int : reserved bytes
- `scope_nr` - int : number of scopes referenced by this event
- `attachment_nr` - int : number of attachments referenced by this event
- `creator_index` - int : index of FHBLOCK
- `sync_base` - int : timestamp base value
- `sync_factor` - float : timestamp factor

Other attributes

- `address` - int : event block address
- `comment` - str : event comment
- `name` - str : event name
- `parent` - int : index of event block that is the parent for the current event
- `range_start` - int : index of event block that is the start of the range for which the current event is the end
- `scopes` - list : list of (group index, channel index) or channel group index that define the scope of the current event

2.4 Signal

class asammdf.signal.**Signal**(*samples=None*, *timestamps=None*, *unit=''*, *name=''*, *conversion=None*, *comment=''*, *raw=True*, *master_metadata=None*, *display_name=''*, *attachment=()*, *source=None*, *bit_count=None*, *stream_sync=False*, *invalidation_bits=None*, *encoding=None*)

The *Signal* represents a channel described by it's samples and timestamps. It can perform arithmetic operations against other *Signal* or numeric types. The operations are computed in respect to the timestamps (time correct). The non-float signals are not interpolated, instead the last value relative to the current timestamp is used. *samples*, *timstamps* and *name* are mandatory arguments.

Parameters

samples [numpy.array | list | tuple] signal samples

timestamps [numpy.array | list | tuple] signal timestamps

unit [str] signal unit

name [str] signal name

conversion [dict | channel conversion block] dict that contains extra conversion information about the signal , default *None*

comment [str] signal comment, default ''

raw [bool] signal samples are raw values, with no physical conversion applied

master_metadata [list] master name and sync type

display_name [str] display name used by mdf version 3

attachment [bytes, name] channel attachment and name from MDF version 4

source [SignalSource] source information named tuple

bit_count [int] bit count; useful for integer channels

stream_sync [bool] the channel is a synchronisation for the attachment stream (mdf v4 only)

invalidation_bits [numpy.array | None] channel invalidation bits, default *None*

encoding [str | None] encoding for string signals; default *None*

astype(*np_type*)

returns new *Signal* with samples of dtype *np_type*

Parameters

np_type [np.dtype] new numpy dtye

Returns

signal [Signal] new *Signal* with the samples of *np_type* dtype

cut(*start=None*, *stop=None*, *include_ends=True*, *interpolation_mode=0*)

Cuts the signal according to the *start* and *stop* values, by using the insertion indexes in the signal's *time* axis.

Parameters

start [float] start timestamp for cutting

stop [float] stop timestamp for cutting

include_ends [bool] include the *start* and *stop* timestamps after cutting the signal. If *start* and *stop* are found in the original timestamps, then the new samples will be computed using interpolation. Default *True*

interpolation_mode [int] interpolation mode for integer signals; default 0

- 0 - repeat previous samples
- 1 - linear interpolation

Returns

result [Signal] new *Signal* cut from the original

Examples

```
>>> new_sig = old_sig.cut(1.0, 10.5)
>>> new_sig.timestamps[0], new_sig.timestamps[-1]
0.98, 10.48
```

extend(*other*)
extend signal with samples from another signal

Parameters

other [Signal]

Returns

signal [Signal] new extended *Signal*

interp(*new_timestamps*, *interpolation_mode=0*)
returns a new *Signal* interpolated using the *new_timestamps*

Parameters

new_timestamps [np.array] timestamps used for interpolation

interpolation_mode [int] interpolation mode for integer signals; default 0

- 0 - repeat previous samples
- 1 - linear interpolation

Returns

signal [Signal] new interpolated *Signal*

physical()
get the physical samples values

Returns

phys [Signal] new *Signal* with physical values

plot()
plot Signal samples. Pyqtgraph is used if it is available; in this case see the GUI plot documentation to see the available commands

CHAPTER 3

Bus logging

Initial read only mode for mdf version 4.10 files containing CAN bus logging is now implemented.

To handle this, the **canmatrix** package was added to the dependencies; you will need to install the latest code from the **canmatrix** development branch.

Let's take for example the following situation: the .dbc contains the definition for the CAN message called "VehicleStatus" with *ID=123*. This message contains the signal "EngineStatus". Logging was made from the CAN bus with *ID=1* (CAN1).

There multiple ways to address this channel in this situation:

1. short signal name as found in the .dbc file

```
mdf.get('EngineStatus')
```

2. dbc message name and short signal name, delimited by dot

```
mdf.get('VehicleStatus.EngineStatus')
```

3. CAN bus ID, dbc message name and short signal name, delimited by dot

```
mdf.get('CAN1.VehicleStatus.EngineStatus')
```

4. ASAM conformant message ID and short signal name, delimited by dot

```
mdf.get('CAN_DataFrame_123.EngineStatus')
```

5. CAN bus ID, ASAM conformant message ID and short signal name, delimited by dot

```
mdf.get('CAN1.CAN_DataFrame_123.EngineStatus')
```

CHAPTER 4

Tips

4.1 Chunked data access

asammdf optimizes memory usage by processing samples in fragments. The read fragment size was tuned based on experimental measurements and should give a good compromise between execution time and memory usage.

You can further tune the read fragment size using the *configure* method, to favor execution speed (using larger fragment sizes) or memory usage (using lower fragment sizes).

4.2 Optimized methods

The *MDF* methods (*cut*, *filter*, *select*) are optimized and should be used instead of calling *get* for several channels.

4.3 Faster file loading

4.3.1 Skip XML parsing for MDF4 files

MDF4 uses the XML channel comment to define the channel's display name (this acts as an alias for the channel name). XML parsing is an expensive operation that can have a big impact on the loading performance of measurements with high channel count.

You can use the keyword only argument *use_display_names* when creating MDF objects to control the XML parsing (default is *False*). This means that the display names will not be available when calling the *get* method.

CHAPTER 5

Examples

5.1 Working with MDF

```
from asammdf import MDF, Signal
import numpy as np


# create 3 Signal objects

timestamps = np.array([0.1, 0.2, 0.3, 0.4, 0.5], dtype=np.float32)

# unit8
s_uint8 = Signal(samples=np.array([0, 1, 2, 3, 4], dtype=np.uint8),
                 timestamps=timestamps,
                 name='Uint8_Signal',
                 unit='u1')
# int32
s_int32 = Signal(samples=np.array([-20, -10, 0, 10, 20], dtype=np.int32),
                 timestamps=timestamps,
                 name='Int32_Signal',
                 unit='i4')

# float64
s_float64 = Signal(samples=np.array([-20, -10, 0, 10, 20], dtype=np.float64),
                   timestamps=timestamps,
                   name='Float64_Signal',
                   unit='f8')

# create empty MDf version 4.00 file
with MDF(version='4.10') as mdf4:

    # append the 3 signals to the new file
    signals = [s_uint8, s_int32, s_float64]
    mdf4.append(signals, 'Created by Python')
```

(continues on next page)

(continued from previous page)

```
    # save new file
    mdf4.save('my_new_file.mf4', overwrite=True)

    # convert new file to mdf version 3.10
    mdf3 = mdf4.convert(version='3.10')
    print(mdf3.version)

    # get the float signal
    sig = mdf3.get('Float64_Signal')
    print(sig)

    # cut measurement from 0.3s to end of measurement
    mdf4_cut = mdf4.cut(start=0.3)
    mdf4_cut.get('Float64_Signal').plot()

    # cut measurement from start of measurement to 0.4s
    mdf4_cut = mdf4.cut(stop=0.45)
    mdf4_cut.get('Float64_Signal').plot()

    # filter some signals from the file
    mdf4 = mdf4.filter(['Int32_Signal', 'Uint8_Signal'])

    # save using zipped transpose deflate blocks
    mdf4.save('out.mf4', compression=2, overwrite=True)
```

5.2 Working with Signal

```
from asammdf import Signal
import numpy as np


# create 3 Signal objects with different time stamps

# unit8 with 100ms time raster
timestamps = np.array([0.1 * t for t in range(5)], dtype=np.float32)
s_uint8 = Signal(samples=np.array([t for t in range(5)], dtype=np.uint8),
                 timestamps=timestamps,
                 name='Uint8_Signal',
                 unit='u1')

# int32 with 50ms time raster
timestamps = np.array([0.05 * t for t in range(10)], dtype=np.float32)
s_int32 = Signal(samples=np.array(list(range(-500, 500, 100)), dtype=np.int32),
                 timestamps=timestamps,
                 name='Int32_Signal',
                 unit='i4')

# float64 with 300ms time raster
timestamps = np.array([0.3 * t for t in range(3)], dtype=np.float32)
s_float64 = Signal(samples=np.array(list(range(2000, -1000, -1000)), dtype=np.int32),
                   timestamps=timestamps,
                   name='Float64_Signal',
                   unit='f8')
```

(continues on next page)

(continued from previous page)

```
# map signals
xs = np.linspace(-1, 1, 50)
ys = np.linspace(-1, 1, 50)
X, Y = np.meshgrid(xs, ys)
vals = np.linspace(0, 180. / np.pi, 100)
phi = np.ones((len(vals), 50, 50), dtype=np.float64)
for i, val in enumerate(vals):
    phi[i] *= val
R = 1 - np.sqrt(X**2 + Y**2)
samples = np.cos(2 * np.pi * X + phi) * R

timestamps = np.arange(0, 2, 0.02)

s_map = Signal(samples=samples,
               timestamps=timestamps,
               name='Variable Map Signal',
               unit='dB')
s_map.plot()


prod = s_float64 * s_uint8
prod.name = 'Uint8_Signal * Float64_Signal'
prod.unit = '*'
prod.plot()

pow2 = s_uint8 ** 2
pow2.name = 'Uint8_Signal ^ 2'
pow2.unit = 'u1^2'
pow2.plot()

allsum = s_uint8 + s_int32 + s_float64
allsum.name = 'Uint8_Signal + Int32_Signal + Float64_Signal'
allsum.unit = '+'
allsum.plot()

# inplace operations
pow2 *= -1
pow2.name = '- Uint8_Signal ^ 2'
pow2.plot()

# cut signal
s_int32.plot()
cut_signal = s_int32.cut(start=0.2, stop=0.35)
cut_signal.plot()
```

5.3 MF4 demo file generator

```
from asammdf import MDF, SUPPORTED_VERSIONS, Signal
import numpy as np

cycles = 100
sigs = []
```

(continues on next page)

(continued from previous page)

```
mdf = MDF()

t = np.arange(cycles, dtype=np.float64)

# no conversion
sig = Signal(
    np.ones(cycles, dtype=np.uint64),
    t,
    name='Channel_no_conversion',
    unit='s',
    conversion=None,
    comment='Unsigned 64 bit channel {}',
)
sigs.append(sig)

# linear
conversion = {
    'a': 2,
    'b': -0.5,
}
sig = Signal(
    np.ones(cycles, dtype=np.int64),
    t,
    name='Channel_linear_conversion',
    unit='Nm',
    conversion=conversion,
    comment='Signed 64bit channel with linear conversion',
)
sigs.append(sig)


# algebraic
conversion = {
    'formula': '2 * sin(X)',
}
sig = Signal(
    np.arange(cycles, dtype=np.int32) / 100.0,
    t,
    name='Channel_algebraic',
    unit='eV',
    conversion=conversion,
    comment='Sinus channel with algebraic conversion',
)
sigs.append(sig)

# rational
conversion = {
    'P1': 0,
    'P2': 4,
    'P3': -0.5,
    'P4': 0,
    'P5': 0,
    'P6': 1,
}
sig = Signal(
    np.ones(cycles, dtype=np.int64),
    t,
```

(continues on next page)

(continued from previous page)

```
    name='Channel_rational_conversion',
    unit='Nm',
    conversion=conversion,
    comment='Channel with rational conversion',
)
sigs.append(sig)

# string channel
sig = [
    'String channel sample {}'.format(j).encode('ascii')
    for j in range(cycles)
]
sig = Signal(
    np.array(sig),
    t,
    name='Channel_string',
    comment='String channel',
    encoding='latin-1',
)
sigs.append(sig)

# byte array
ones = np.ones(cycles, dtype=np.dtype('(8,)u1'))
sig = Signal(
    ones*111,
    t,
    name='Channel_bytearay',
    comment='Byte array channel',
)
sigs.append(sig)

# tabular
vals = 20
conversion = {
    'raw_{}'.format(i): i
    for i in range(vals)
}
conversion.update(
    {
        'phys_{}'.format(i): -i
        for i in range(vals)
    }
)
sig = Signal(
    np.arange(cycles, dtype=np.uint32) % 20,
    t,
    name='Channel_tabular',
    unit='-',
    conversion=conversion,
    comment='Tabular channel',
)
sigs.append(sig)

# value to text
vals = 20
conversion = {
    'val_{}'.format(i): i
```

(continues on next page)

(continued from previous page)

```
    for i in range(vals)
}
conversion.update(
    {
        'text_{}'.format(i): 'key_{}'.format(i).encode('ascii')
        for i in range(vals)
    }
)
conversion['default'] = b'default key'
sig = Signal(
    np.arange(cycles, dtype=np.uint32) % 30,
    t,
    name='Channel_value_to_text',
    conversion=conversion,
    comment='Value to text channel',
)
sigs.append(sig)

# tabular with range
vals = 20
conversion = {
    'lower_{}'.format(i): i * 10
    for i in range(vals)
}
conversion.update(
    {
        'upper_{}'.format(i): (i + 1) * 10
        for i in range(vals)
    }
)
conversion.update(
    {
        'phys_{}'.format(i): i
        for i in range(vals)
    }
)
conversion['default'] = -1
sig = Signal(
    2 * np.arange(cycles, dtype=np.float64),
    t,
    name='Channel_value_range_to_value',
    unit='order',
    conversion=conversion,
    comment='Value range to value channel',
)
sigs.append(sig)

# value range to text
vals = 20
conversion = {
    'lower_{}'.format(i): i * 10
    for i in range(vals)
}
conversion.update(
    {
        'upper_{}'.format(i): (i + 1) * 10
        for i in range(vals)
```

(continues on next page)

(continued from previous page)

```
    }
)
conversion.update(
    {
        'text_{}'.format(i): 'Level {}'.format(i)
        for i in range(vals)
    }
)
conversion['default'] = b'Unknown level'
sig = Signal(
    6 * np.arange(cycles, dtype=np.uint64) % 240,
    t,
    name='Channel_value_range_to_text',
    conversion=conversion,
    comment='Value range to text channel',
)
sigs.append(sig)


mdf.append(sigs, 'single dimensional channels', common_timebase=True)



sigs = []

# lookup tabel with axis
samples = [
    np.ones((cycles, 2, 3), dtype=np.uint64) * 1,
    np.ones((cycles, 2), dtype=np.uint64) * 2,
    np.ones((cycles, 3), dtype=np.uint64) * 3,
]

types = [
    ('Channel_lookup_with_axis', '(2, 3)<u8'),
    ('channel_axis_1', '(2, )<u8'),
    ('channel_axis_2', '(3, )<u8'),
]

sig = Signal(
    np.core.records.fromarrays(samples, dtype=np.dtype(types)),
    t,
    name='Channel_lookup_with_axis',
    unit='A',
    comment='Array channel with axis',
)
sigs.append(sig)

# lookup tabel with default axis
samples = [
    np.ones((cycles, 2, 3), dtype=np.uint64) * 4,
]

types = [
    ('Channel_lookup_with_default_axis', '(2, 3)<u8'),
]

sig = Signal(
```

(continues on next page)

(continued from previous page)

```
    np.core.records.fromarrays(samples, dtype=np.dtype(types)),
    t,
    name='Channel_lookup_with_default_axis',
    unit='mA',
    comment='Array channel with default axis',
)
sigs.append(sig)

# structure channel composition
samples = [
    np.ones(cycles, dtype=np.uint8) * 10,
    np.ones(cycles, dtype=np.uint16) * 20,
    np.ones(cycles, dtype=np.uint32) * 30,
    np.ones(cycles, dtype=np.uint64) * 40,
    np.ones(cycles, dtype=np.int8) * -10,
    np.ones(cycles, dtype=np.int16) * -20,
    np.ones(cycles, dtype=np.int32) * -30,
    np.ones(cycles, dtype=np.int64) * -40,
]

types = [
    ('struct_channel_0', np.uint8),
    ('struct_channel_1', np.uint16),
    ('struct_channel_2', np.uint32),
    ('struct_channel_3', np.uint64),
    ('struct_channel_4', np.int8),
    ('struct_channel_5', np.int16),
    ('struct_channel_6', np.int32),
    ('struct_channel_7', np.int64),
]

sig = Signal(
    np.core.records.fromarrays(samples, dtype=np.dtype(types)),
    t,
    name='Channel_structure_composition',
    comment='Structure channel composition',
)
sigs.append(sig)


# nested structures
l4_arr = [
    np.ones(cycles, dtype=np.float64) * 41,
    np.ones(cycles, dtype=np.float64) * 42,
    np.ones(cycles, dtype=np.float64) * 43,
    np.ones(cycles, dtype=np.float64) * 44,
]

types = [
    ('level41', np.float64),
    ('level42', np.float64),
    ('level43', np.float64),
    ('level44', np.float64),
]

l4_arr = np.core.records.fromarrays(l4_arr, dtype=types)
```

(continues on next page)

(continued from previous page)

```
l3_arr = [
    l4_arr,
    l4_arr,
    l4_arr,
]

types = [
    ('level31', l4_arr.dtype),
    ('level32', l4_arr.dtype),
    ('level33', l4_arr.dtype),
]

l3_arr = np.core.records.fromarrays(l3_arr, dtype=types)


l2_arr = [
    l3_arr,
    l3_arr,
]

types = [
    ('level21', l3_arr.dtype),
    ('level22', l3_arr.dtype),
]

l2_arr = np.core.records.fromarrays(l2_arr, dtype=types)


l1_arr = [
    l2_arr,
]

types = [
    ('level11', l2_arr.dtype),
]

l1_arr = np.core.records.fromarrays(l1_arr, dtype=types)


sigs.append(
    Signal(
        l1_arr,
        t,
        name='Nested_structures',
    )
)

mdf.append(sigs, 'arrays', common_timebase=True)

mdf.save('demo.mf4', overwrite=True)
```

CHAPTER 6

Benchmarks

6.1 Test setup

The benchmarks were done using two test files (available here) (for mdf version 3 and 4) of around 170MB. The files contain 183 data groups and a total of 36424 channels.

asamdf 5.0.0dev was compared against *mdfreader 3.0*.

For each category two aspect were noted: elapsed time and peak RAM usage.

6.1.1 Dependencies

You will need the following packages to be able to run the benchmark script

- psutil
- mdfreader

6.1.2 Usage

Extract the test files from the archive, or provide a folder that contains the files "test.mdf" and "test.mf4". Run the module *bench.py* (see –help option for available options)

6.2 x64 Python results

Benchmark environment

- 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:57:15) [MSC v.1915 64 bit (AMD64)]
- Windows-10-10.0.16299-SP0
- Intel64 Family 6 Model 94 Stepping 3, GenuineIntel

- numpy 1.16.1
- 16GB installed RAM

Notations used in the results

- compress = mdfreader mdf object created with compression=blosc
- nodata = mdfreader mdf object read with no_data_loading=True

Files used for benchmark:

- **mdf version 3.10**
  - 167 MB file size
  - 183 groups
  - 36424 channels
- **mdf version 4.00**
  - 183 MB file size
  - 183 groups
  - 36424 channels

Open file	Time [ms]	RAM [MB]
asammdf 5.0.0dev mdfv3	329	111
mdfreader 3.0 mdfv3	1942	425
mdfreader 3.0 compress mdfv3	1667	293
mdfreader 3.0 no_data_loading mdfv3	864	171
asammdf 5.0.0dev mdfv4	455	123
mdfreader 3.0 mdfv4	5217	448
mdfreader 3.0 compress mdfv4	4999	320
mdfreader 3.0 no_data_loading mdfv4	3644	178

Save file	Time [ms]	RAM [MB]
asammdf 5.0.0dev mdfv3	544	110
mdfreader 3.0 mdfv3	6017	453
mdfreader 3.0 no_data_loading mdfv3	6808	513
mdfreader 3.0 compress mdfv3	6124	452
asammdf 5.0.0dev mdfv4	449	124
mdfreader 3.0 mdfv4	3409	467
mdfreader 3.0 no_data_loading mdfv4	4841	485
mdfreader 3.0 compress mdfv4	3597	465

Get all channels (36424 calls)	Time [ms]	RAM [MB]
asammdf 5.0.0dev mdfv3	4783	111
mdfreader 3.0 mdfv3	59	425
mdfreader 3.0 nodata mdfv3	15259	207
mdfreader 3.0 compress mdfv3	200	292
asammdf 5.0.0dev mdfv4	8210	123
mdfreader 3.0 mdfv4	80	448
mdfreader 3.0 compress mdfv4	226	322
mdfreader 3.0 nodata mdfv4	22078	208

Convert file	Time [ms]	RAM [MB]
asammdf 5.0.0dev v3 to v4	3090	157
asammdf 5.0.0dev v4 to v3	2665	151

Merge 3 files	Time [ms]	RAM [MB]
asammdf 5.0.0dev v3	8409	173
mdfreader 3.0 v3	18133	1329
mdfreader 3.0 compress v3	18285	1277
mdfreader 3.0 nodata v3	17846	1432
asammdf 5.0.0dev v4	8902	174
mdfreader 3.0 v4	29391	1356
mdfreader 3.0 nodata v4	28493	1334
mdfreader 3.0 compress v4	29109	1298

6.2.1 Graphical results

CHAPTER 7

GUI

With the GUI tool you can

- visualize channels
- see channel, conversion and source metadata as stored in the MDF file
- access library functionality for single files (convert, export, cut, filter, resample) and multiple files (concatenate, stack)

After you pip install asammdf there will be a new script called *asammdf.exe* in the *python_installation_folder\Scripts* folder.

The following dependencies are required by the GUI

- PyQt5
- pyqtgraph

7.1 Menu

7.1.1 File

The only command here is *Open*. Depending on the page this is allow to open a single file, or multiple files.

7.1.2 Settings

The following settings are available

- **Plot lines**: controls the visual style of the channels plot lines:
  - `Simple`: a simple line is used to join the channel samples. It is the default option because it gives the best performance for high sample count

  - `With dots`: a simple line joins the channels sample, and the actual samples are marked by a dot. This allows for better visualization, but at the expense of lower performance

- **Search**: controls how the matching is done for the quick search field. Mathing is always done case insensitive.
  - `Match start`: the channel name must start with the input string
  - `Match contains`: the channel name must contain the input string

7.1.3 Plot

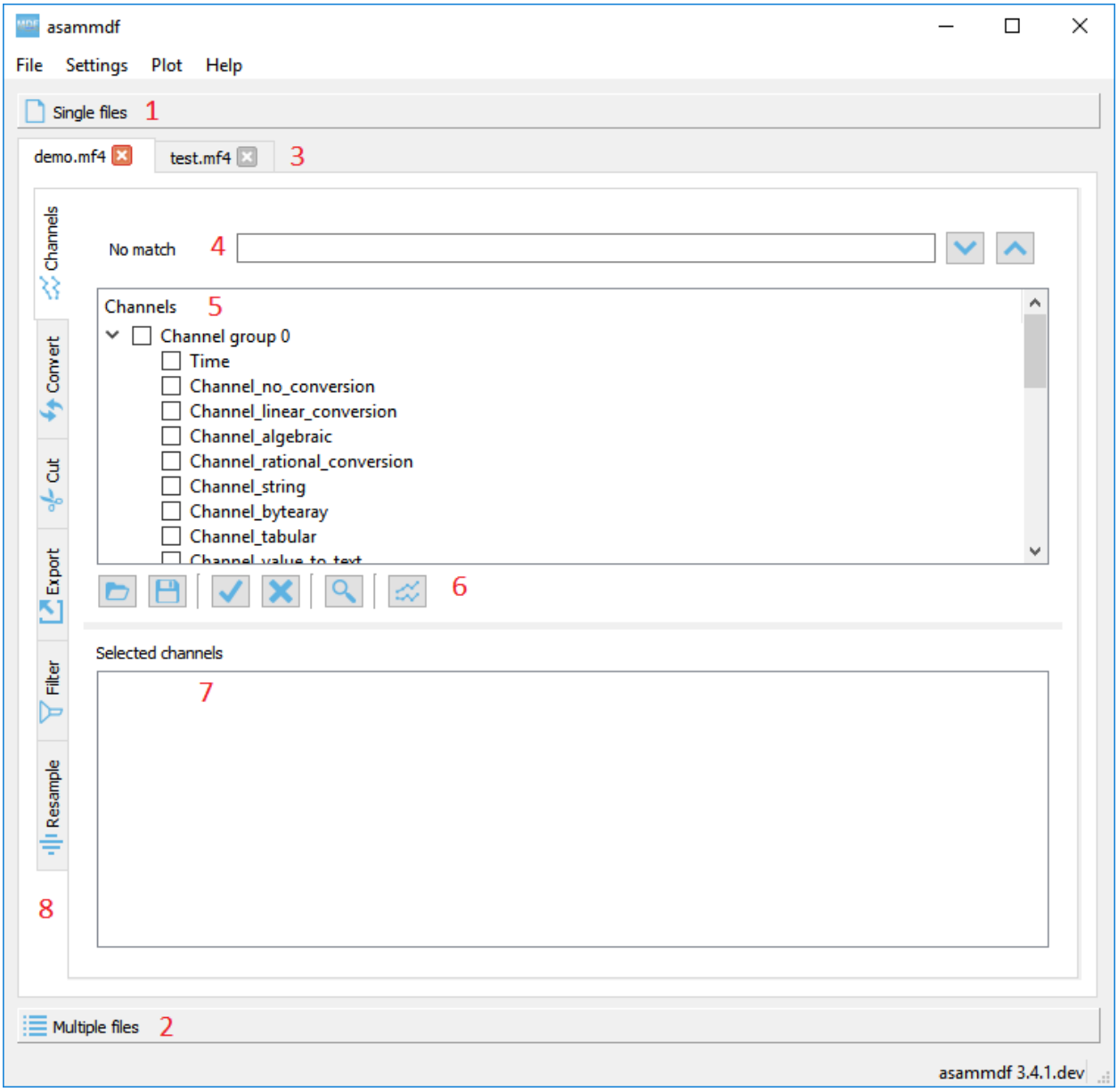
There are several keyboard shortcuts for handling the plots:

Short-cut	Action	Desctiption
C	Cursor	Displays a movable cursor that will trigger the display of the current value for all plot channels
F	Fit	Y-axis fit all active channels on the screen, keeping the current X-axis range
G	Grid	Toggle grid lines
H	Home	XY-axis fit all active channels
I	Zoom-in	X-axis zoom-in[1]
O	Zoom-out	X-axis zoom-out[1]
M	Statistics	Toggle the display of the statistic panel
R	Range	Display a movable range that will trigger the display of the delta values for all plot channels
S	Stack	Y Stack all active channels so that they don't overlap, keeping the X-axis range
←	Move cursor left	Moves the cursor to the next sample on the left
→	Move cursor right	Moves the cursor to the next sample on the right
Ins	Insert computation	Insert new channel in the plot using functions and operations
Ctrl+B	Bin	Toggle binary representation of integer channels
Ctrl+H	Hex	Toggle hex representation of integer channels
Ctrl+P	Physical	Toggle physical representation of integer channels
Ctrl+S	Save plot channels	Save all plotted channel in a new MF4 file

7.2 Single files

The *Single files* page is used to open several files individually for visualization and processing (for example exporting to csv or hdf5).

[1] If the cursor is present then zooming will center on it.



1. Single files page selector
2. Multiple files page selector
3. Opened files tabs
4. Quick channel search field for the current file
5. Complete channels tree
6. Command buttons
7. Selected channels list
8. Current file operations tabs

7.2.1 Opened files tabs

In the single files mode, you can open multiple files in parallel. The tab names have the title set to the short file name, and the complete file path can be seen as the tab tooltip.

There is no restriction, so the same file can be opened several times.

7.2.2 Quick channel search field for the current file

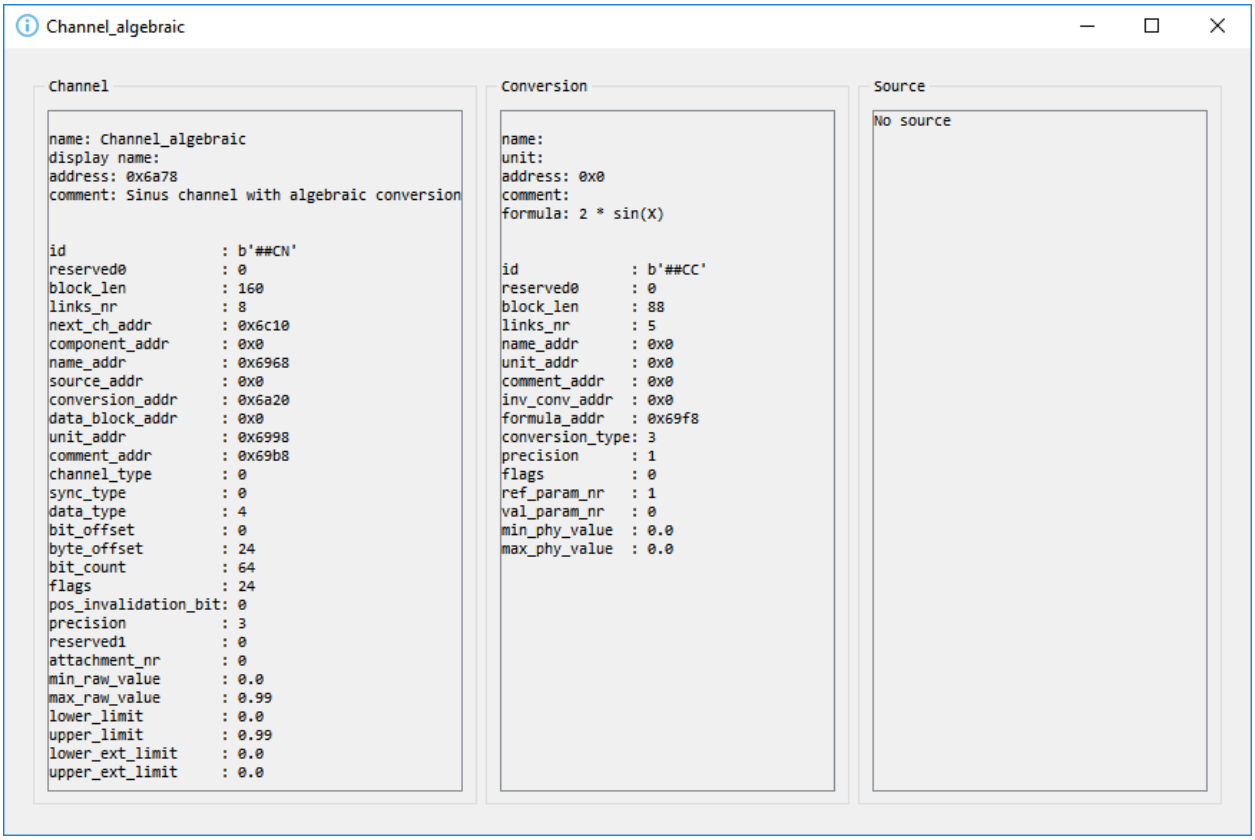
Using the *Settings->Search* menu option the user can choose how the search is performed. A positive search match will scroll the channels tree and highlight the channel entry.

When the same channel name exist several times in the file, you can switch between the occurrences using the arrow buttons.

7.2.3 Complete channels tree

This tree contains all the channels found in the measurement.

Double clicking a channel name will display a pop-up window with the channel information (CNBLOCK, CCBLOCK and SIBLOCK/CEBLOCK)

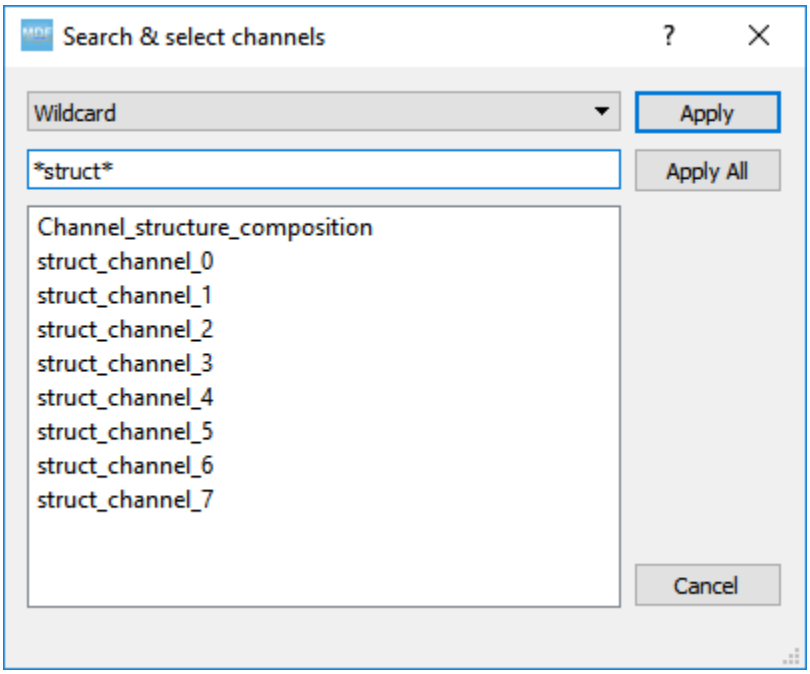


Only the channels that are checked in the channels tree will be selected for plotting when the *Plot* button is pressed. Checking or unchecking channels will not affect the current plot.

7.2.4 Command buttons

From left to right the buttons have the following functionality

- **Load channel selection list**: loads a channel selection list from a text file (one channel name per line) and checks them in the channels tree is they are found.
- **Save channel selection list**: saves the current checked channels names in a text file
- **Select all channels**: checks all channels in the channels tree
- **Reset selection**: unchecks all channels in the channels tree
- **Advanced search & select**: will open an advanced search dialog
  - the dialog can use wildcard and regex patterns
  - multiple channels can be selected, and thus checked in the channels tree



- **Plot**: generates the plot based on the current checked channels from the channels tree

7.2.5 Selected channels list

When the *Plot* button is pressed the checked channels will populate the *Selected channels list*.

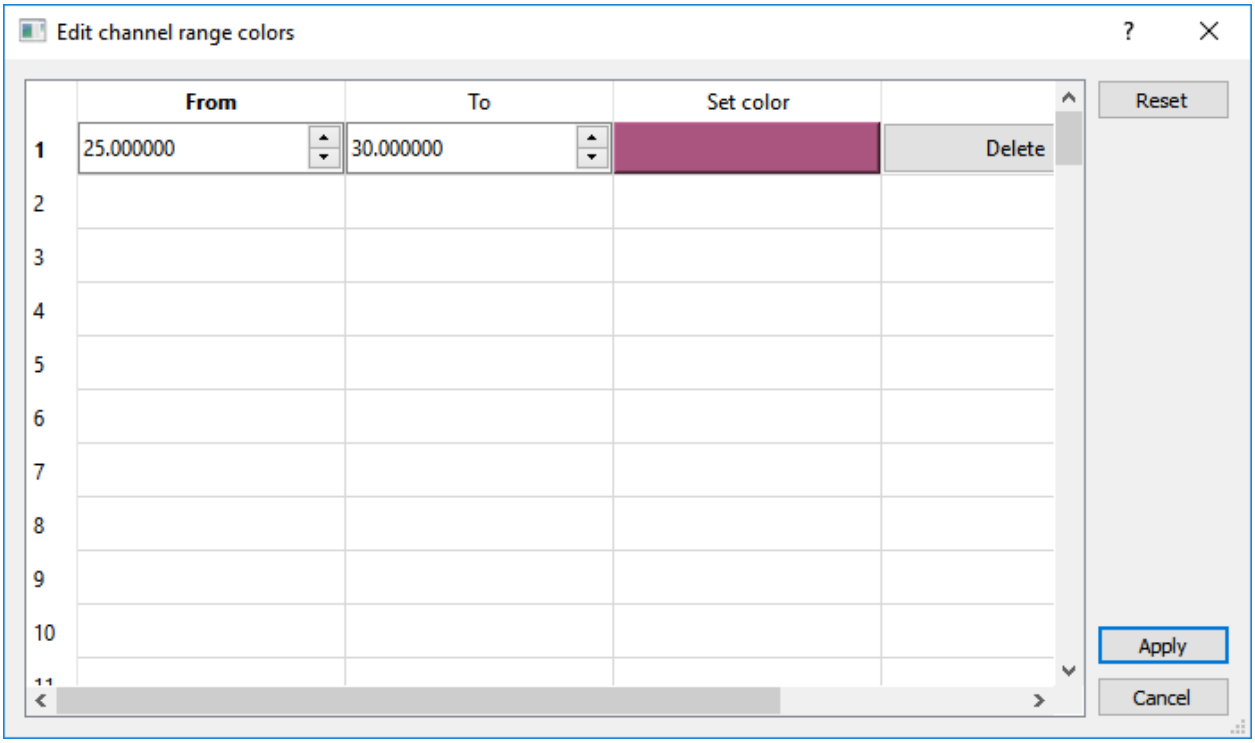
Selecting items from the *Selected channels list* will display their Y-axis on the right side of the plot, if the items are enabled for display.

It also necessary to select a single item when the *Statistics* panel is active to compute the statistics for the item's channel.

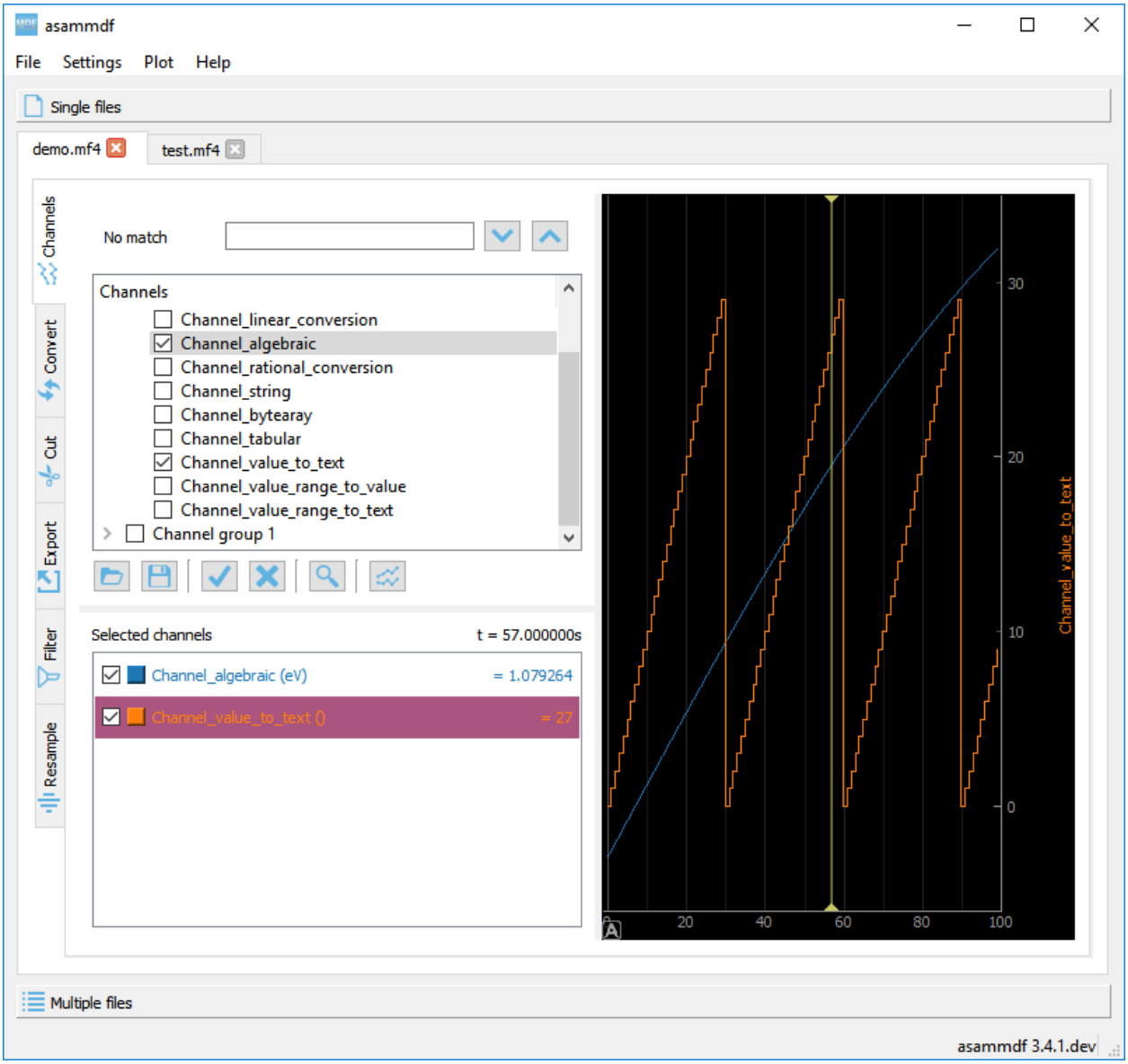
Each item has five elements

1. display enable checkbox
2. color select button
3. channel name and unit label
4. channel value label
5. common axis checkbox
    - the value is only displayed if the cursor or range are active. For the cursor is will show the current value, and for the range it will show the value delta between the range start and stop timestamps

Double clicking an item will open a range editor dialog

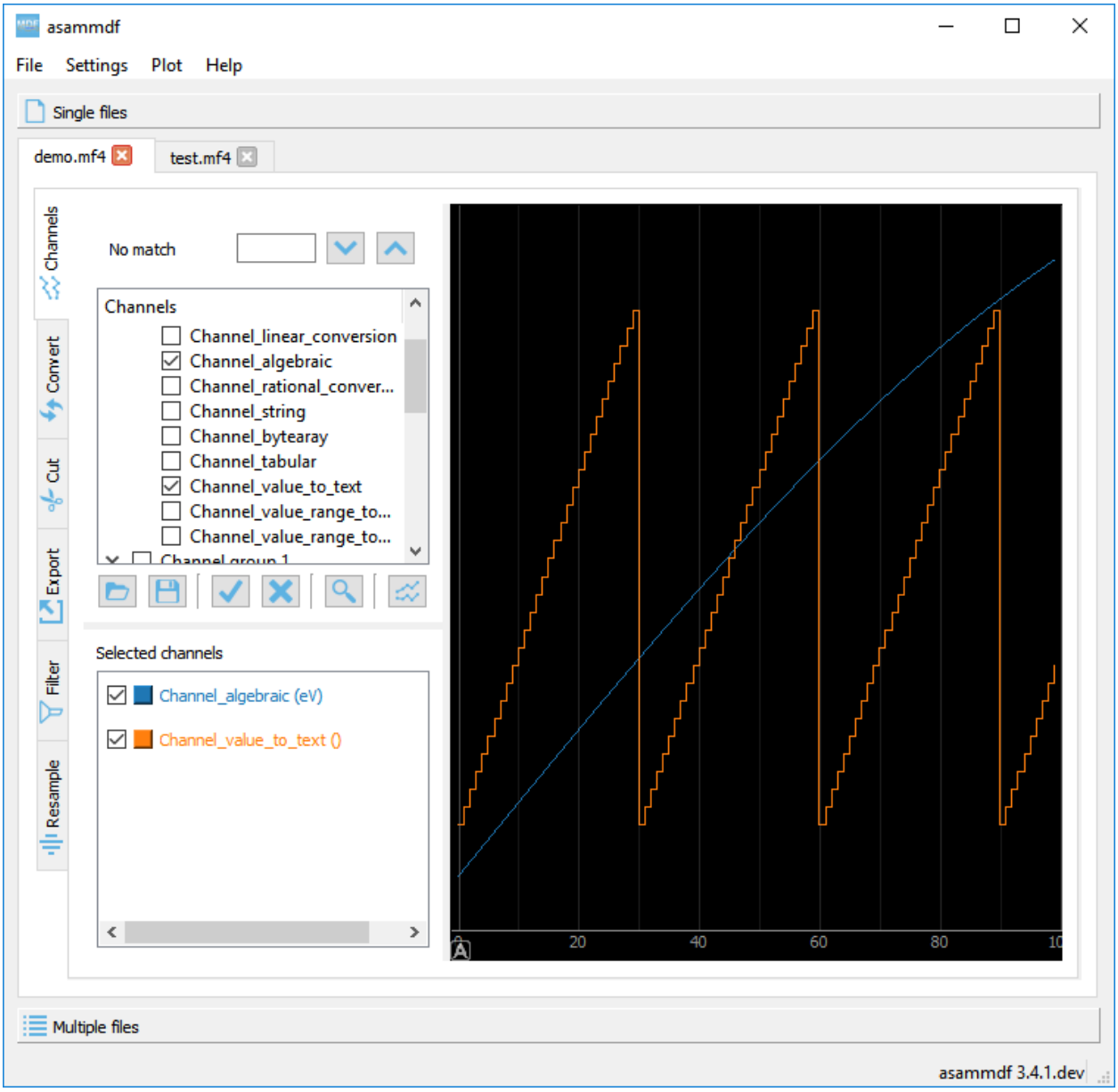


Here we can specify a range value visual alert. When the cursor is active and the current channel value is within the specified range, the item background will change to the selected color.

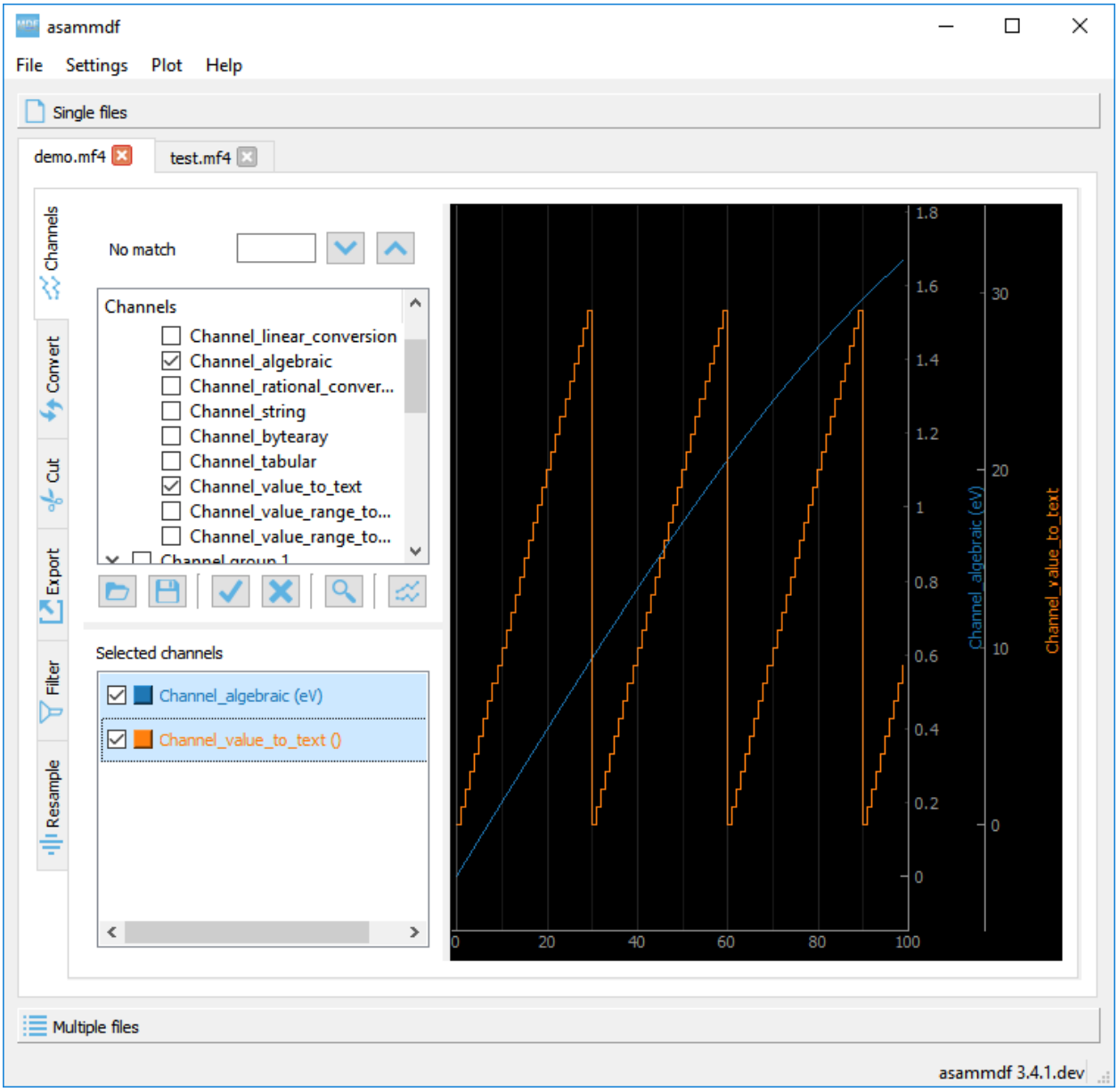


7.2.6 Plot

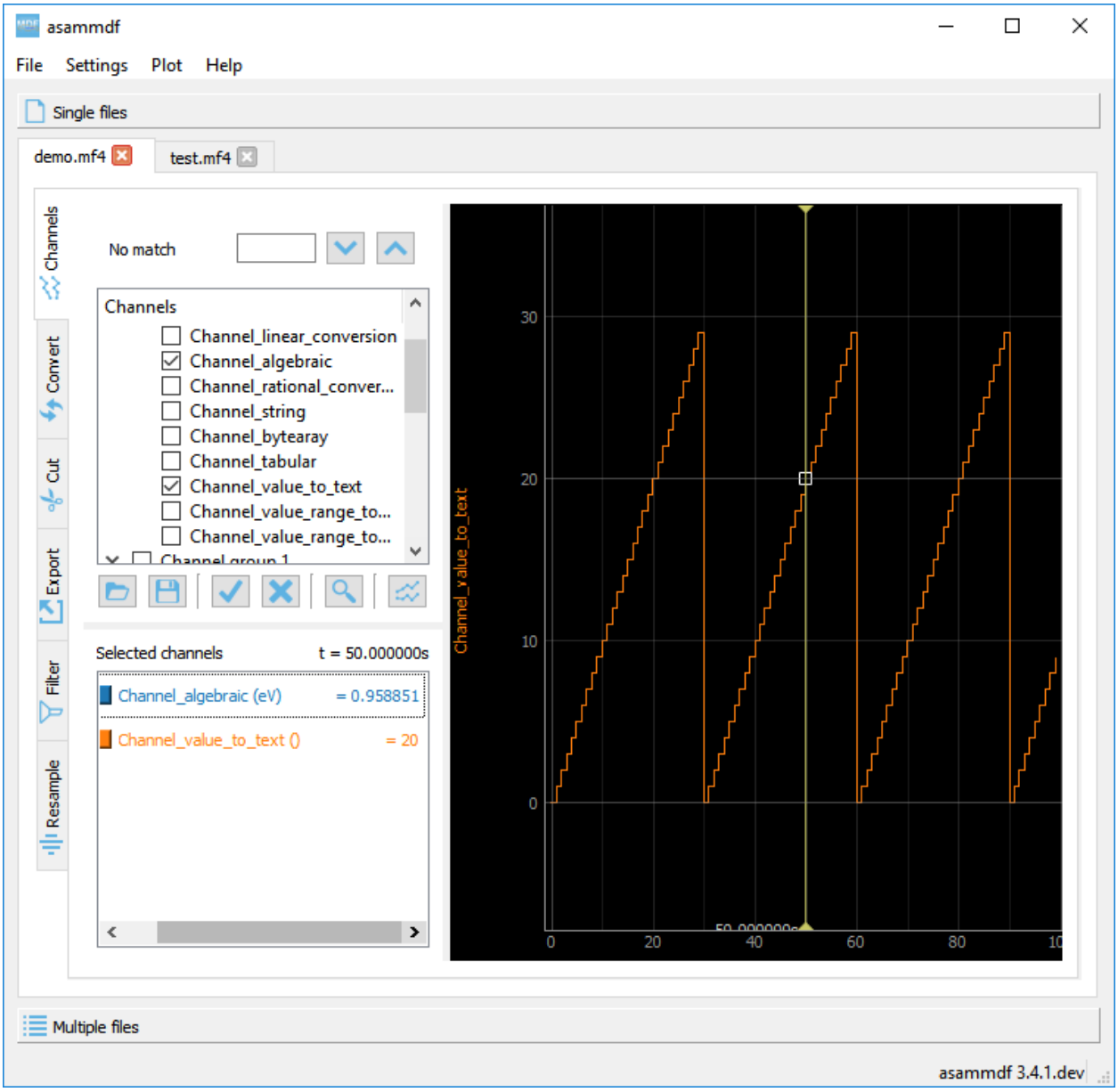
The initial plot will have all channels homed (see the *H* keyboard shortcut) and Y-axis disabled



Selecting items from the *Selected channels list* will enable the Y-axis

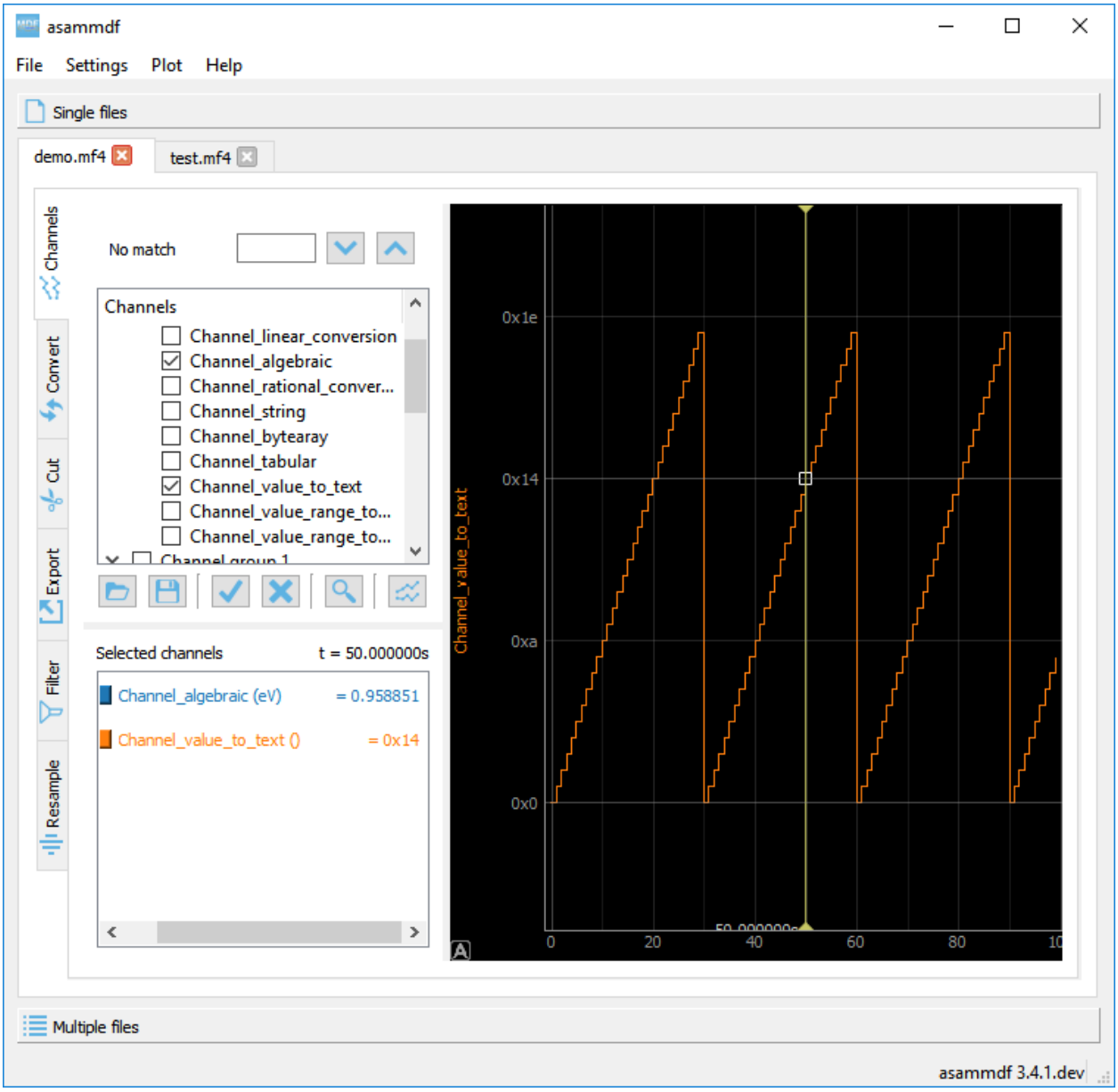


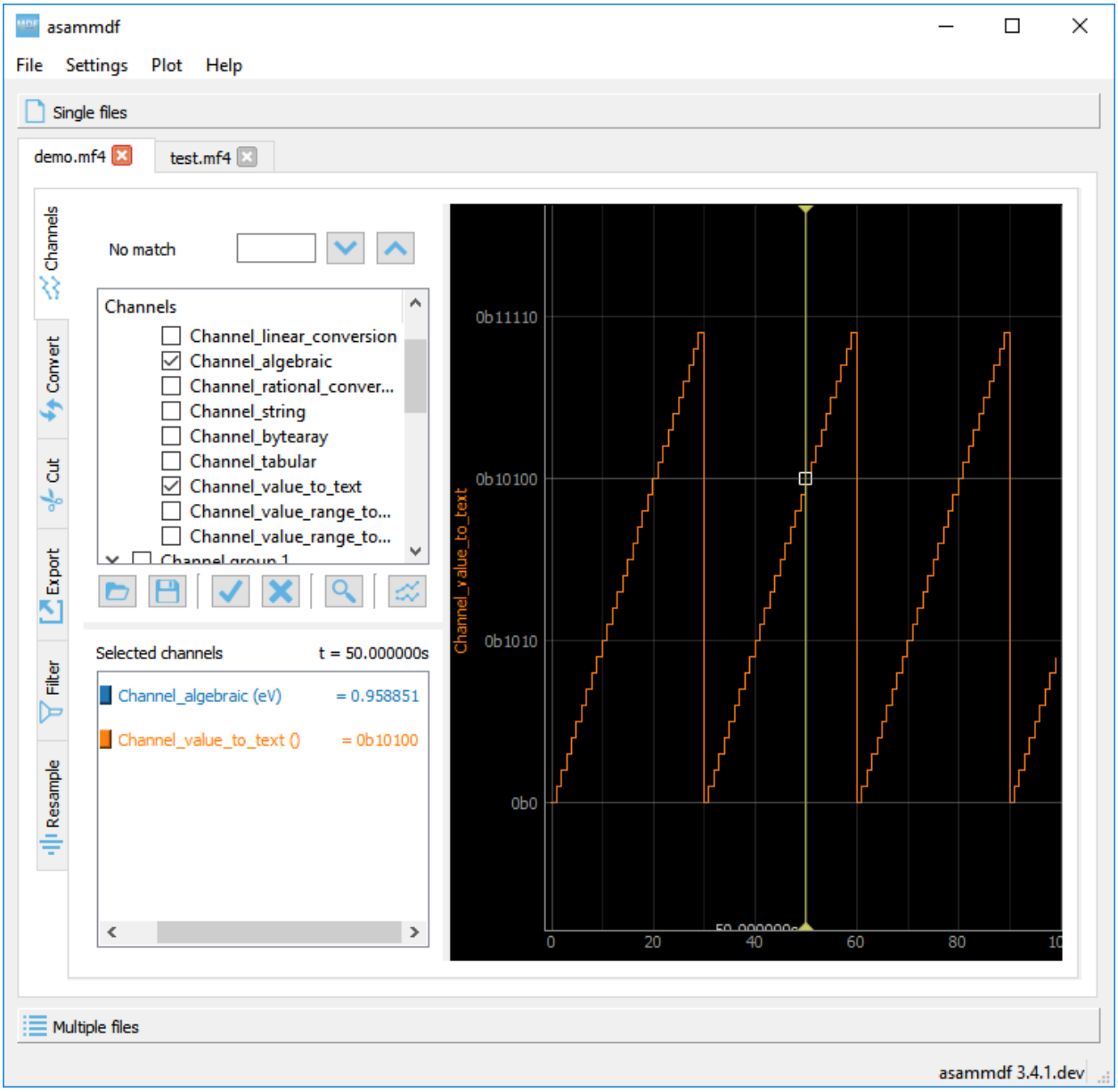
Using the *C* keyboard shortcut will toggle the cursor, and with it the channel values will be displayed for each item in the *Selected channels list*



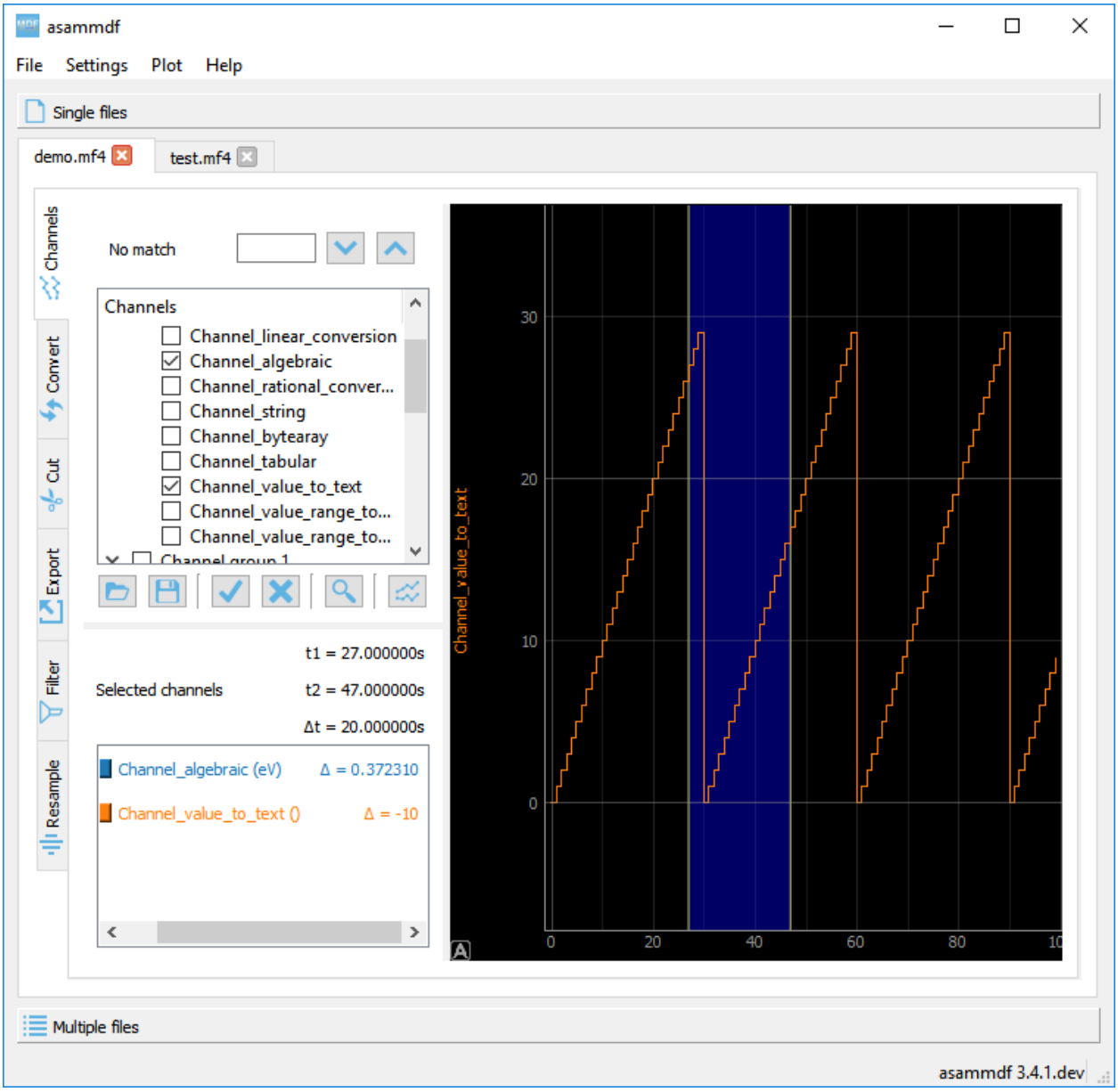
The *Ctrl+H* and *Ctrl+B* keyboard shortcuts will

- change the axis values for integer channels to hex and bin mode
- change the channel value display mode for each integer channel item in the *Selected channels list*

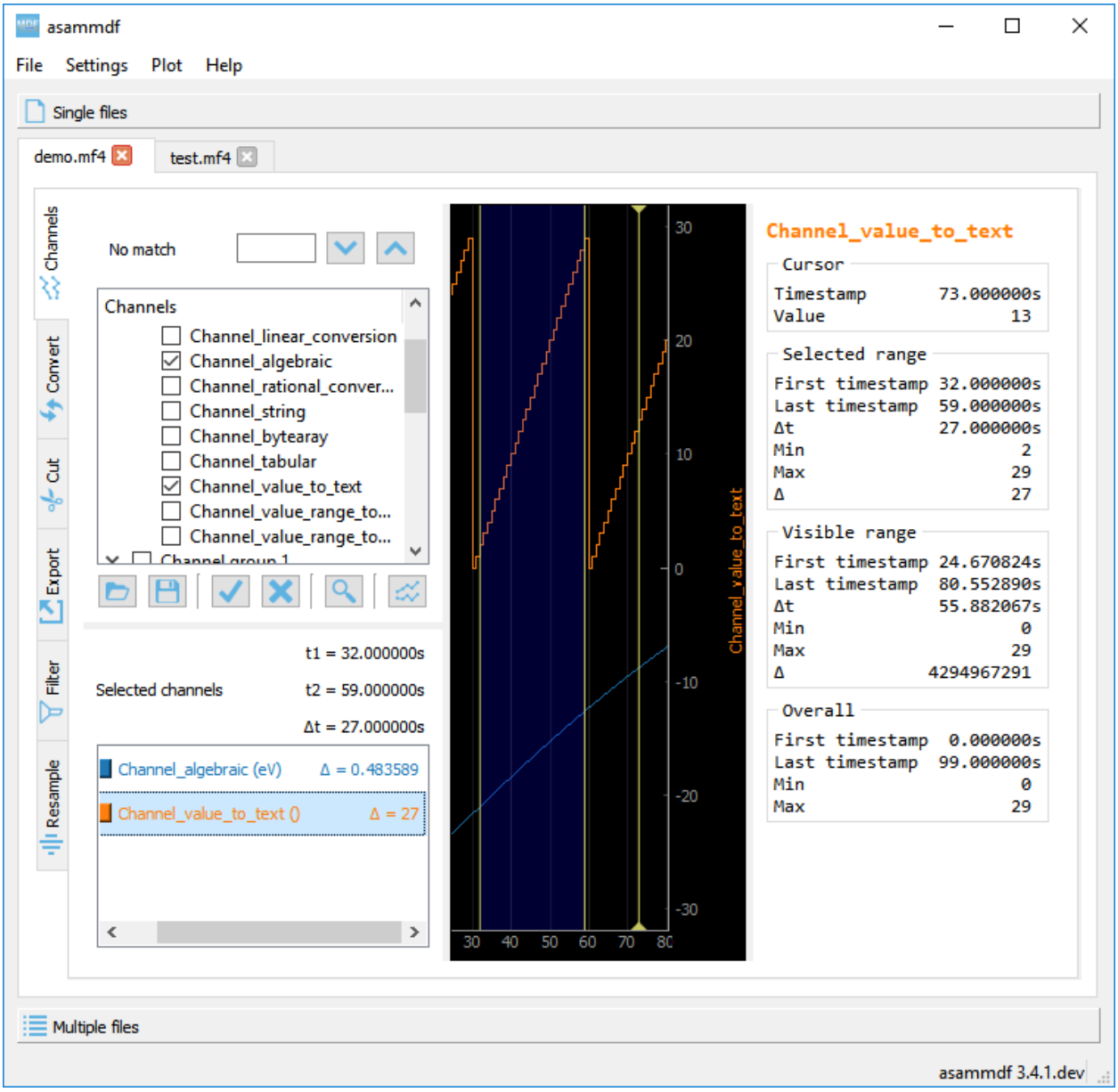

asammdf
File
Settings
Plot
Help
Single files
demo.mf4
test.mf4
Channels
Convert
Cut
Export
Filter
Resample
No match
Channels
Channel_linear_conversion
Channel_algebraic
Channel_rational_conver...
Channel_string
Channel_bytearay
Channel_tabular
Channel_value_to_text
Channel_value_range_to...
Channel_value_range_to...
Selected channels
t = 50.000000s
Channel_algebraic (eV)
= 0.958851
Channel_value_to_text ()
= 0x14
0x1e
0x14
0xa
0x0
Channel_value_to_text
0
20
40
60
80
Multiple files
asammdf 3.4.1.dev



Using the *R* keyboard shortcut will toggle the range, and with it the channel values will be displayed for each item in the *Selected channels list*. When the range is enabled, using the *H* keyboard shortcut will not home to the whole time range, but instead will use the range time interval.



The *Statistics* panel is toggle using the *M* keyboard shortcut

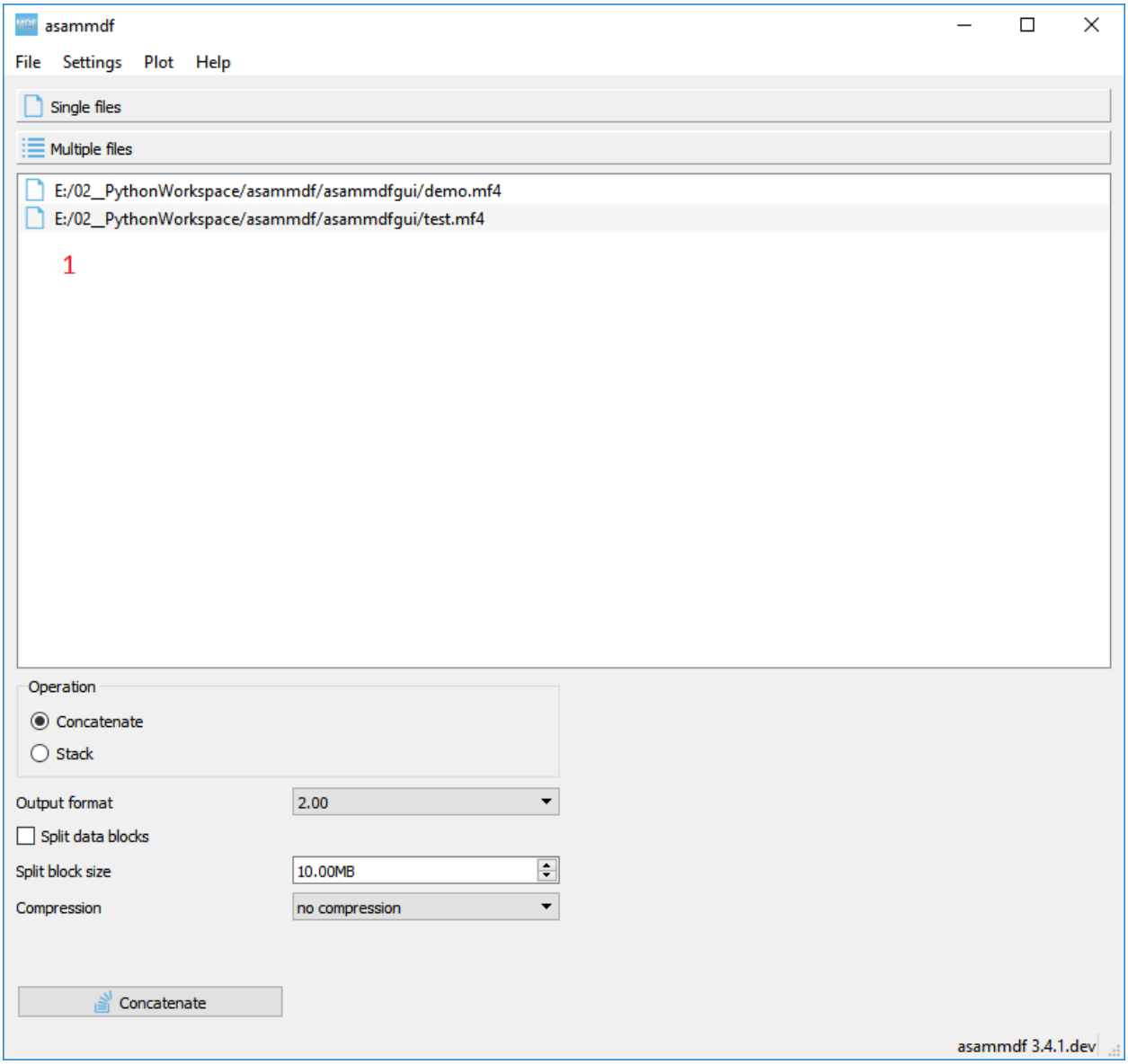


You can insert new computed channels by pressing the *insert* key. This will allow either to compute basic operations using the plot channels, or to apply a function on one of the plot channels.

The plot channels can be saved to a new file by pressing *Ctrl+S*.

7.3 Multiple files

The *Multiple files* toolbox page is used to concatenate or stack multiple files.



The files list can be rearranged in the list (1) by drag and dropping lines. Unwanted files can be deleted by selecting them and pressing the *DEL* key. The files order is considered from top to bottom.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Index

T

W