asammdf Documentation

Release 3.4.3

Daniel Hrisca

May 15, 2018

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) and CSV
- 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 based
  - 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
- for version 4
  - functionality related to sample reduction block
  - handling of channel hierarchy
  - full handling of bus logging measurements
  - handling of unfinished measurements (mdf 4)
  - full support for remaining mdf 4 channel arrays types
  - xml schema for MDBLOCK
  - full handling of event blocks
  - channels with default X axis
  - chanenls with reference to attachment

1.4 Dependencies

asammdf uses the following libraries

- numpy : the heart that makes all tick
- numexpr : for algebraic and rational channel conversions
- matplotlib : for Signal plotting
- 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

other optional dependencies

- chardet : to detect non-standard unicode encodings
- PyQt4 or PyQt5 : for GUI tool
- pyqtgraph : for GUI tool

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

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*, *memory='full'*, *version='4.10'*, *callback=None*, *queue=None*)

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] mdf file name, if provided it must be a real file name

memory [str] memory option; default *full*:

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

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'

static concatenate(*files*, *outversion='4.10'*, *memory='full'*, *sync=True*, *callback=None*)

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

outversion [str] merged file version

memory [str] memory option; default *full*

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 (*to*, *memory='full'*)

convert *MDF* to other version

Parameters

to [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'

memory [str] memory option; default *full*

Returns

out [MDF] new *MDF* object

cut (*start=None*, *stop=None*, *whence=0*)

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

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 'DataGroup_<cntr>_<channel name>'. The channel group master will be renamed to 'DataGroup_<cntr>_<channel name>_master' (<*cntr*> is the data group index starting from 0)
- *pandas* : export all channels as a single pandas DataFrame

filename [string] 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 *zeros*
- *format*: only valid for *mat* export; can be '4', '5' or '7.3', default is '5'

Returns

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

filter (*channels*, *memory='full'*)

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

memory [str] memory option for filtered *MDF*; default *full*

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)])
```

(continues on next page)

(continued from previous page)

```
>>> 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="">
```

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

static merge (*files*, *outversion='4.10'*, *memory='full'*, *sync=True*, *callback=None*)
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

outversion [str] merged file version

memory [str] memory option; default *full*

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]

resample (*raster*, *memory='full'*)
resample all channels using the given raster

Parameters

raster [float] time raster is seconds

memory [str] memory option; default *None*

Returns

mdf [MDF] new *MDF* with resampled channels

select (*channels*, *dataframe=False*)
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):
```

(continues on next page)

(continued from previous page)

```
...     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*, *outversion='4.10'*, *memory='full'*, *sync=True*, *callback=None*)

merge several files and return the merged *MDF* object

Parameters

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

outversion [str] merged file version

memory [str] memory option; default *full*

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

Returns

merged [MDF] new *MDF* object with merge channels

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

asammdf tries to emulate the mdf structure using Python builtin data types.

The *header* attibute is an OrderedDict that holds the file metadata.

The *groups* attribute is a dictionary list 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_conversions : list of ChannelConversion objects in 1-to-1 relation with the channel list
- channel_sources : list of SourceInformation objects in 1-to-1 relation with the channels list
- chanel_dependencies : list of ChannelDependency objects in a 1-to-1 relation with the channel list
- data_block : DataBlock object
- texts : dictionay containing TextBlock objects used throughout the mdf
  - channels : list of dictionaries that contain TextBlock objects ralated to each channel
    - long_name_addr : channel long name
    - comment_addr : channel comment
    - display_name_addr : channel display name
  - channel group : list of dictionaries that contain TextBlock objects ralated to each channel group
    - comment_addr : channel group comment
  - conversion_tab : list of dictionaries that contain TextBlock objects ralated to VATB and VTABR channel conversions
    - text_{n} : n-th text of the VTABR conversion
- sorted : bool flag to indicate if the source file was sorted; it is used when *memory* is *low* or *minimum*
- size : data block size; used for lazy laoding of measured data
- record_size : dict of record ID -> record size pairs

The *file_history* attribute is a TextBlock object.

The *channel_db* attibute is a dictionary that holds the *(data group index, channel index)* pair for all signals. This is used to speed up the *get_signal_by_name* method.

The *master_db* attibute is a dictionary that holds the *channel index* of the master channel for all data groups. This is used to speed up the *get_signal_by_name* method.

class asammdf.mdf_v3.**MDF3**(*name=None*, *memory='full'*, *version='3.30'*, *callback=None*)

If the *name* exist it will be loaded otherwise an empty file will be created that can be later saved to disk

Parameters

name [string] mdf file name

memory [str] memory optimization option; default *full*

- 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 memorised into RAM
- if *minimum* only minimal data is memorised into RAM

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

Attributes

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

file_history [TextBlock] file history text block; can be None

groups [list] list of data groups

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*)

Appends a new data group.

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

Parameters

signals [list] list on *Signal* objects

acquisition_info [str] acquisition information; default 'Python'

common_timebase [bool] flag to hint that the signals have the same timebase

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')
```

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*)
configure read and write fragment size for chuncked data access

Parameters

read_fragment_size [int] size hint of splitted 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 splitted 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

use_display_names [bool] use display name if available for the Signal's name returned by the get method

extend(*index*, *signals*)
Extend a group with new samples. The first signal is the master channel's samples, and the next signals must respect the same order in which they 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 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, s1, s2, s3])
```

get(*name=None*, *group=None*, *index=None*, *raster=None*, *samples_only=False*, *data=None*, *raw=False*)

Gets channel samples. 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

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*

Returns

res [(numpy.array | Signal)] returns *Signal* if *samples_only**=**False* (default option), otherwise returns numpy.array. 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
...
>>> 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.]
```

(continues on next page)

(continued from previous page)

```
        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*)
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

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 *dst* is not provided the the destination file name is the MDF name. 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] destination file name, Default ''

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.v2_v3_blocks.**Channel**(***kargs*)

CNBLOCK class derived from *dict*

The Channel object can be created in two modes:

- **using the *stream* and *address* keyword parameters - when reading** from file
- using any of the following presented keys - when creating a new Channel

The keys have the following meaning:

- id - Block type identifier, always "CN"
- block_len - Block size of this block in bytes (entire CNBLOCK)
- **next_ch_addr - Pointer to next channel block (CNBLOCK) of this channel** group (NIL allowed)
- **conversion_addr - Pointer to the conversion formula (CCBLOCK) of this** signal (NIL allowed)
- **source_depend_addr - Pointer to the source-depending extensions (CEBLOCK)** of this signal (NIL allowed)
- **ch_depend_addr - Pointer to the dependency block (CDBLOCK) of this signal** (NIL allowed)
- **comment_addr - Pointer to the channel comment (TXBLOCK) of this signal** (NIL allowed)
- channel_type - Channel type
  - 0 = data channel
  - **1 = time channel for all signals of this group (in each channel** group, exactly one channel must be defined as time channel). The time stamps recording in a time channel are always relative to the start time of the measurement defined in HDBLOCK.
- **short_name - Short signal name, i.e. the first 31 characters of the** ASAM-MCD name of the signal (end of text should be indicated by 0)
- description - Signal description (end of text should be indicated by 0)
- **start_offset - Start offset in bits to determine the first bit of the** signal in the data record. The start offset N is divided into two parts: a "Byte offset" (= N div 8) and a "Bit offset" (= N mod 8). The channel block can define an "additional Byte offset" (see below) which must be added to the Byte offset.
- **bit_count - Number of bits used to encode the value of this signal in a** data record
- data_type - Signal data type
- range_flag - Value range valid flag
- **min_raw_value - Minimum signal value that occurred for this signal** (raw value)
- **max_raw_value - Maximum signal value that occurred for this signal** (raw value)
- sampling_rate - Sampling rate for a virtual time channel. Unit [s]

- **long_name_addr - Pointer to TXBLOCK that contains the ASAM-MCD long** signal name
- **display_name_addr - Pointer to TXBLOCK that contains the signal's display** name (NIL allowed)
- **aditional_byte_offset - Additional Byte offset of the signal in the data** record (default value: 0).

Parameters

stream [file handle] mdf file handle

address [int] block address inside mdf file

Examples

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

Attributes

name [str] full channel name

address [int] block address inside mdf file

dependencies [list] lsit of channel dependencies

ChannelConversion Class

class asammdf.v2_v3_blocks.**ChannelConversion**(***kargs*)

CCBLOCK class derived from *dict*

The ChannelConversion object can be created in two modes:

- **using the *stream* and *address* keyword parameters - when reading** from file
- **using any of the following presented keys - when creating a new** ChannelConversion

The first keys are common for all conversion types, and are followed by conversion specific keys. The keys have the following meaning:

- common keys
  - id - Block type identifier, always "CC"
  - block_len - Block size of this block in bytes (entire CCBLOCK)
  - range_flag - Physical value range valid flag:
  - **min_phy_value - Minimum physical signal value that occurred for this** signal
  - **max_phy_value - Maximum physical signal value that occurred for this** signal
  - unit - Physical unit (string should be terminated with 0)
  - conversion_type - Conversion type (formula identifier)
  - ref_param_nr - Size information about additional conversion data

- specific keys
  - linear conversion
    - b - offset
    - a - factor
    - **CANapeHiddenExtra - sometimes CANape appends extra information;** not compliant with MDF specs
  - ASAM formula conversion
    - formula - ecuation as string
  - polynomial or rational conversion
    - P1 .. P6 - factors
  - exponential or logarithmic conversion
    - P1 .. P7 - factors
  - tabular with or without interpolation (grouped by *n*)
    - raw_{n} - n-th raw integer value (X axis)
    - phys_{n} - n-th physical value (Y axis)
  - text table conversion
    - param_val_{n} - n-th integers value (X axis)
    - text_{n} - n-th text value (Y axis)
  - text range table conversion
    - lower_{n} - n-th lower raw value
    - upper_{n} - n-th upper raw value
    - text_{n} - n-th text value

Parameters

stream [file handle] mdf file handle

address [int] block address inside mdf file

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
```

Attributes

address [int] block address inside mdf file

ChannelDependency Class

class asammdf.v2_v3_blocks.**ChannelDependency**(***kargs*)
CDBLOCK class derived from *dict*

Currently the ChannelDependency object can only be created using the *stream* and *address* keyword parameters when reading from file

The keys have the following meaning:

- id - Block type identifier, always "CD"
- block_len - Block size of this block in bytes (entire CDBLOCK)
- dependency_type - Dependency type
- sd_nr - Total number of signals dependencies (m)
- for each dependency there is a group of three keys:
  - **dg_{n} - Pointer to the data group block (DGBLOCK) of** signal dependency *n*
  - **cg_{n} - Pointer to the channel group block (DGBLOCK) of** signal dependency *n*
  - **ch_{n} - Pointer to the channel block (DGBLOCK) of** signal dependency *n*
- **there can also be optional keys which decribe dimensions for** the N-dimensional dependencies:
  - **dim_{n} - Optional: size of dimension *n* for N-dimensional** dependency

Parameters

stream [file handle] mdf file handle

address [int] block address inside mdf file

Attributes

address [int] block address inside mdf file

ChannelExtension Class

class asammdf.v2_v3_blocks.**ChannelExtension**(***kargs*)
CEBLOCK class derived from *dict*

The ChannelExtension object can be created in two modes:

- **using the *stream* and *address* keyword parameters - when reading** from file
- **using any of the following presented keys - when creating** a new ChannelExtension

The first keys are common for all conversion types, and are followed by conversion specific keys. The keys have the following meaning:

- common keys
  - id - Block type identifier, always "CE"
  - block_len - Block size of this block in bytes (entire CEBLOCK)
  - type - Extension type identifier
- specific keys
  - for DIM block
    - module_nr - Number of module

        * module_address - Address
        * description - Description
        * ECU_identification - Identification of ECU
        * reserved0' - reserved
    - for Vector CAN block
        * CAN_id - Identifier of CAN message
        * CAN_ch_index - Index of CAN channel
        * message_name - Name of message (string should be terminated by 0)
        * sender_name - Name of sender (string should be terminated by 0)
        * reserved0 - reserved

Parameters

stream [file handle] mdf file handle

address [int] block address inside mdf file

Attributes

address [int] block address inside mdf file

ChannelGroup Class

class asammdf.v2_v3_blocks.**ChannelGroup**(***kargs*)

CGBLOCK class derived from *dict*

The ChannelGroup object can be created in two modes:

- **using the *stream* and *address* keyword parameters - when reading** from file
- **using any of the following presented keys - when creating** a new ChannelGroup

The keys have the following meaning:

- id - Block type identifier, always "CG"
- block_len - Block size of this block in bytes (entire CGBLOCK)
- **next_cg_addr - Pointer to next channel group block (CGBLOCK) (NIL** allowed)
- first_ch_addr - Pointer to first channel block (CNBLOCK) (NIL allowed)
- **comment_addr - Pointer to channel group comment text (TXBLOCK)** (NIL allowed)
- **record_id - Record ID, i.e. value of the identifier for a record if** the DGBLOCK defines a number of record IDs > 0
- ch_nr - Number of channels (redundant information)
- **samples_byte_nr - Size of data record in Bytes (without record ID),** i.e. size of plain data for a each recorded sample of this channel group
- **cycles_nr - Number of records of this type in the data block** i.e. number of samples for this channel group
- **sample_reduction_addr - only since version 3.3. Pointer to** first sample reduction block (SRBLOCK) (NIL allowed) Default value: NIL

Parameters

stream [file handle] mdf file handle

address [int] block address inside mdf file

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'
```

Attributes

address [int] block address inside mdf file

DataGroup Class

class asammdf.v2_v3_blocks.**DataGroup**(***kargs*)

DGBLOCK class derived from *dict*

The DataGroup object can be created in two modes:

- **using the *stream* and *address* keyword parameters - when reading** from file
- using any of the following presented keys - when creating a new DataGroup

The keys have the following meaning:

- id - Block type identifier, always “DG”
- block_len - Block size of this block in bytes (entire DGBLOCK)
- next_dg_addr - Pointer to next data group block (DGBLOCK) (NIL allowed)
- **first_cg_addr - Pointer to first channel group block (CGBLOCK)** (NIL allowed)
- trigger_addr - Pointer to trigger block (TRBLOCK) (NIL allowed)
- **data_block_addr - Pointer to the data block (see separate chapter** on data storage)
- cg_nr - Number of channel groups (redundant information)
- record_id_nr - Number of record IDs in the data block
- reserved0 - since version 3.2; Reserved

Parameters

stream [file handle] mdf file handle

address [int] block address inside mdf file

Attributes

address [int] block address inside mdf file

FileIdentificationBlock Class

class asammdf.v2_v3_blocks.**FileIdentificationBlock**(**_kargs_)
IDBLOCK class derived from *dict*

The TriggerBlock object can be created in two modes:

- **using the *stream* and *address* keyword parameters - when reading** from file
- using the classmethod *from_text*

The keys have the following meaning:

- file_identification - file identifier
- version_str - format identifier
- program_identification - program identifier
- byte_order - default byte order
- float_format - default floating-point format
- mdf_version - version number of MDF format
- code_page - code page number
- reserved0 - reserved
- reserved1 - reserved
- unfinalized_standard_flags - Standard Flags for unfinalized MDF
- unfinalized_custom_flags - Custom Flags for unfinalized MDF

Parameters

stream [file handle] mdf file handle

version [int] mdf version in case of new file

Attributes

address [int] block address inside mdf file; should be 0 always

HeaderBlock Class

class asammdf.v2_v3_blocks.**HeaderBlock**(**_kargs_)
HDBLOCK class derived from *dict*

The TriggerBlock object can be created in two modes:

- using the *stream* - when reading from file
- using the classmethod *from_text*

The keys have the following meaning:

- id - Block type identifier, always “HD”
- block_len - Block size of this block in bytes (entire HDBLOCK)
- first_dg_addr - Pointer to the first data group block (DGBLOCK)
- **comment_addr - Pointer to the measurement file comment text (TXBLOCK)** (NIL allowed)
- program_addr - Pointer to program block (PRBLOCK) (NIL allowed)

- dg_nr - Number of data groups (redundant information)
- date - Date at which the recording was started in "DD:MM:YYYY" format
- time - Time at which the recording was started in "HH:MM:SS" format
- author - author name
- organization - organization
- project - project name
- subject - subject

Since version 3.2 the following extra keys were added:

- abs_time - Time stamp at which recording was started in nanoseconds.
- tz_offset - UTC time offset in hours (= GMT time zone)
- time_quality - Time quality class
- timer_identification - Timer identification (time source),

Parameters

stream [file handle] mdf file handle

Attributes

address [int] block address inside mdf file; should be 64 always

ProgramBlock Class

class asammdf.v2_v3_blocks.**ProgramBlock**(**kargs*)

PRBLOCK class derived from *dict*

The ProgramBlock object can be created in two modes:

- **using the *stream* and *address* keyword parameters - when reading** from file
- **using any of the following presented keys - when creating** a new ProgramBlock

The keys have the following meaning:

- id - Block type identifier, always "PR"
- block_len - Block size of this block in bytes (entire PRBLOCK)
- data - Program-specific data

Parameters

stream [file handle] mdf file handle

address [int] block address inside mdf file

Attributes

address [int] block address inside mdf file

SampleReduction Class

class asammdf.v2_v3_blocks.**SampleReduction**(**kargs*)
SRBLOCK class derived from *dict*

Currently the SampleReduction object can only be created by using the *stream* and *address* keyword parameters - when reading from file

The keys have the following meaning:

- id - Block type identifier, always "SR"
- block_len - Block size of this block in bytes (entire SRBLOCK)
- **next_sr_addr - Pointer to next sample reduction block (SRBLOCK)** (NIL allowed)
- data_block_addr - Pointer to the data block for this sample reduction
- cycles_nr - Number of reduced samples in the data block.
- **time_interval - Length of time interval [s] used to calculate** the reduced samples.

Parameters

stream [file handle] mdf file handle

address [int] block address inside mdf file

Attributes

address [int] block address inside mdf file

TextBlock Class

class asammdf.v2_v3_blocks.**TextBlock**(**kargs*)
TXBLOCK class derived from *dict*

The ProgramBlock object can be created in two modes:

- **using the *stream* and *address* keyword parameters - when reading** from file
- using the classmethod *from_text*

The keys have the following meaning:

- id - Block type identifier, always "TX"
- block_len - Block size of this block in bytes (entire TXBLOCK)
- text - Text (new line indicated by CR and LF; end of text indicated by 0)

Parameters

stream [file handle] mdf file handle

address [int] block address inside mdf file

text [bytes] bytes for creating a new TextBlock

Examples

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

Attributes

address [int] block address inside mdf file

text_str [str] text data as unicode string

TriggerBlock Class

class asammdf.v2_v3_blocks.**TriggerBlock**(***kargs*)

TRBLOCK class derived from *dict*

The TriggerBlock object can be created in two modes:

- **using the *stream* and *address* keyword parameters - when reading** from file
- using the classmethod *from_text*

The keys have the following meaning:

- id - Block type identifier, always "TR"
- block_len - Block size of this block in bytes (entire TRBLOCK)
- text_addr - Pointer to trigger comment text (TXBLOCK) (NIL allowed)
- trigger_events_nr - Number of trigger events n (0 allowed)
- trigger_{n}_time - Trigger time [s] of trigger event *n*
- trigger_{n}_pretime - Pre trigger time [s] of trigger event *n*
- trigger_{n}_posttime - Post trigger time [s] of trigger event *n*

Parameters

stream [file handle] mdf file handle

address [int] block address inside mdf file

Attributes

address [int] block address inside mdf file

2.3 MDF4

asammdf tries to emulate the mdf structure using Python builtin data types.

The *header* attibute is an OrderedDict that holds the file metadata.

The *groups* attribute is a dictionary list 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_conversions : list of ChannelConversion objects in 1-to-1 relation with the channel list
- channel_sources : list of SourceInformation objects in 1-to-1 relation with the channels list
- data_block : DataBlock object
- texts : dictionay containing TextBlock objects used throughout the mdf
  - channels : list of dictionaries that contain TextBlock objects ralated to each channel
    - name_addr : channel name
    - comment_addr : channel comment
  - channel group : list of dictionaries that contain TextBlock objects ralated to each channel group
    - acq_name_addr : channel group acquisition comment
    - comment_addr : channel group comment
  - conversion_tab : list of dictionaries that contain TextBlock objects related to TABX and RTABX channel conversions
    - text_{n} : n-th text of the VTABR conversion
    - default_addr : default text
  - conversions : list of dictionaries that containt TextBlock obejcts related to channel conversions
    - name_addr : converions name
    - unit_addr : channel unit_addr
    - comment_addr : converison comment
    - formula_addr : formula text; only valid for algebraic conversions
  - sources : list of dictionaries that containt TextBlock obejcts related to channel sources
    - name_addr : source name
    - path_addr : source path_addr
    - comment_addr : source comment

The *file_history* attribute is a list of (FileHistory, TextBlock) pairs .

The *channel_db* attibute is a dictionary that holds the *(data group index, channel index)* pair for all signals. This is used to speed up the *get_signal_by_name* method.

The *master_db* attibute is a dictionary that holds the *channel index* of the master channel for all data groups. This is used to speed up the *get_signal_by_name* method.

class asammdf.mdf_v4.**MDF4**(*name=None*, *memory='full'*, *version='4.10'*, *callback=None*, *queue=None*)

If the *name* exist it will be memorised otherwise an empty file will be created that can be later saved to disk

Parameters

name [string] mdf file name

memory [str] memory optimization option; default *full*

- 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'

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

file_comment [TextBlock] file comment TextBlock

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

groups [list] list of data groups

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

append(*signals*, *source_info='Python'*, *common_timebase=False*)

Appends a new data group.

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

Parameters

signals [list] list on *Signal* objects

source_info [str] source information; default 'Python'

common_timebase [bool] flag to hint that the signals have the same timebase

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]
```

(continues on next page)

(continued from previous page)

```
>>> mdf2 = MDF3('out.mdf')
>>> mdf2.append(sigs, 'created by asammdf v1.1.0')
```

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

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*)
configure read and write fragment size for chuncked data access

Parameters

read_fragment_size [int] size hint of splitted 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 splitted 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

use_display_names [bool] use display name if available for the Signal's name returned by the get method

extend(*index*, *signals*)
Extend a group with new samples. The first signal is the master channel's samples, and the next signals must respect the same order in which they 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 objects

Examples

```
>>> # case 1 conversion type None
>>> s1 = np.array([1, 2, 3, 4, 5])
>>> s2 = np.array([-1, -2, -3, -4, -5])
```

(continues on next page)

(continued from previous page)

```
>>> 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, s1, s2, s3])
```

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 | str] attachment data

get(*name=None*, *group=None*, *index=None*, *raster=None*, *samples_only=False*, *data=None*, *raw=False*)

Gets channel samples. 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

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*

Returns

res [(numpy.array | Signal)] returns *Signal* if *samples_only* = *False* (default option), otherwise returns numpy.array 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="">
```

(continues on next page)

(continued from previous page)

```
>>> # 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="">
```

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_master(*index*, *data=None*, *raster=None*)
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

Returns

t [numpy.array] master channel samples

get_valid_indexes(*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

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

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 *dst* is not provided the the destination file name is the MDF name. 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 [str] output file name

2.3.1 MDF version 4 blocks

The following classes implement different MDF version4 blocks.

AttachmentBlock Class

class asammdf.v4_blocks.**AttachmentBlock**(***kargs*)
ATBLOCK class

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

Channel Class

class asammdf.v4_blocks.**Channel**(***kargs*)
CNBLOCK class

ChannelConversion Class

class asammdf.v4_blocks.**ChannelConversion**(***kargs*)
CCBLOCK class

ChannelGroup Class

class asammdf.v4_blocks.**ChannelGroup**(***kargs*)
CGBLOCK class

DataGroup Class

class asammdf.v4_blocks.**DataGroup**(***kargs*)
DGBLOCK class

DataList Class

class asammdf.v4_blocks.**DataList**(**kargs)
DLBLOCK class

DataBlock Class

class asammdf.v4_blocks.**DataBlock**(**kargs)
DTBLOCK class

Parameters

address [int] DTBLOCK address inside the file

stream [int] file handle

FileIdentificationBlock Class

class asammdf.v4_blocks.**FileIdentificationBlock**(**kargs)
IDBLOCK class

HeaderBlock Class

class asammdf.v4_blocks.**HeaderBlock**(**kargs)
HDBLOCK class

start_time
get the measurement start timestamp

Returns

timestamp [datetime] start timestamp

SourceInformation Class

class asammdf.v4_blocks.**SourceInformation**(**kargs)
SIBLOCK class

FileHistory Class

class asammdf.v4_blocks.**FileHistory**(**kargs)
FHBLOCK class

TextBlock Class

class asammdf.v4_blocks.**TextBlock**(**kargs)
common TXBLOCK and MDBLOCK class

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*)

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

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*)

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

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*)
returns a new *Signal* interpolated using the *new_timestamps*

Parameters

new_timestamps [np.array] timestamps used for 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

CHAPTER 3

Bus logging

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

To handle this **cantools** package was added to the dependecies.

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 Impact of *memory* argument

By default when the *MDF* object is created all data is loaded into RAM (memory='full'). This will give you the best performance from *asammdf*.

However if you reach the physical memory limit *asammdf* gives you two options:

- memory='low' : only the metadata is loaded into RAM, the raw channel data is loaded when needed
- memory='minimum' : only minimal data is loaded into RAM.

4.1.1 *MDF* created with *memory='full'*

Advantages

- best performance if all channels are used (for example *cut*, *convert*, *export* or *merge* methods)

Disadvantages

- higher RAM usage, there is the chance of MemoryError for large files
- data is not accessed in chunks
- time can be wasted if only a small number of channels is retreived from the file (for example *filter*, *get* or *select* methods)

Use case

- when data fits inside the system RAM

4.1.2 *MDF* created with *memory='low'*

Advantages

- lower RAM usage than memory='full'

- can handle files that do not fit in the available physical memory
- middle ground between 'full' speed and 'minimum' memory usage

Disadvantages

- slower performance for retrieving channel data
- must call *close* method to release the temporary file used in case of appending.

Note: it is advised to use the *MDF* context manager in this case

Use case

- when 'full' data exceeds available RAM
- it is advised to avoid getting individual channels when using this option
- best performance / memory usage ratio when using *cut*, *convert*, *filter*, *merge* or *select* methods

Note: See benchmarks for the effects of using the flag

4.1.3 *MDF* created with *memory='minimum'*

Advantages

- lowest RAM usage
- the only choise when dealing with huge files (10's of thousands of channels and GB of sample data)
- handle big files on 32 bit Python ()

Disadvantages

- slightly slower performance compared to momeory='low'
- must call *close* method to release the temporary file used in case of appending.

Note: See benchmarks for the effects of using the flag

4.2 Chunked data access

When the *MDF* is created with the option "full" all the samples are loaded into RAM and are processed as a signle block. For large files this can lead to MemoryError exceptions (for example trying to merge several GB sized files).

asammdf optimizes memory usage for options "low" and "minimum" 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).

4.3 Optimized methods

The *MDF* methods (*cut*, *filter*, *select*) are optimized and should be used instead of calling *get* for several channels. For "low" and "minimum" options the time savings can be dramatic.

CHAPTER 5

Examples

5.1 Working with MDF

```
from __future__ import print_function, division
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
mdf4 = MDF(version='4.10')

# append the 3 signals to the new file
signals = [s_uint8, s_int32, s_float64]
```

(continues on next page)

(continued from previous page)

```
mdf4.append(signals, 'Created by Python')

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

# convert new file to mdf version 3.10 with lowest possible RAM usage
mdf3 = mdf4.convert(to='3.10', memory='minimum')
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 __future__ import print_function, division
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,
```

(continues on next page)

(continued from previous page)

```
                    name='Float64_Signal',
                    unit='f8')

# 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
```

(continues on next page)

(continued from previous page)

```
sigs = []

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(
```

(continues on next page)

(continued from previous page)

```
    np.ones(cycles, dtype=np.int64),
    t,
    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',
)
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 = {
```

(continues on next page)

(continued from previous page)

```
    'val_{}'.format(i): i
    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
```

(continues on next page)

(continued from previous page)

```
        for i in range(vals)
    }
)
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'),
]
```

(continues on next page)

(continued from previous page)

```
sig = Signal(
    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)


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

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

CHAPTER 6

Benchmarks

asammdf relies heavily on *dict* objects. Starting with Python 3.6 the *dict* objects are more compact and ordered (implementation detail); *asammdf* uses takes advantage of those changes so for best performance it is advised to use Python >= 3.6.

6.1 Test setup

The benchmarks were done using two test files (available here https://github.com/danielhrisca/asammdf/issues/14) (for mdf version 3 and 4) of around 170MB. The files contain 183 data groups and a total of 36424 channels.

asamdf 3.0.0 was compared against *mdfreader 2.7.5*. *mdfreader* seems to be the most used Python package to handle MDF files, and it also supports both version 3 and 4 of the standard.

The three benchmark cathegories are file open, file save and extracting the data for all channels inside the file(36424 calls). For each cathegory 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

- 3.6.4 (default, Jan 5 2018, 02:35:40) [GCC 7.2.1 20171224]
- Linux-4.15.0-1-MANJARO-x86_64-with-arch-Manjaro-Linux
- 4GB installed RAM

Notations used in the results

- full = asammdf MDF object created with memory=full (everything loaded into RAM)
- low = asammdf MDF object created with memory=low (raw channel data not loaded into RAM, but metadata loaded to RAM)
- minimum = asammdf MDF object created with memory=full (lowest possible RAM usage)
- compress = mdfreader mdf object created with compression=blosc
- noDataLoading = mdfreader mdf object read with noDataLoading=True

Files used for benchmark:

- 183 groups
- 36424 channels

6.2.1 Raw data

Open file	Time [ms]	RAM [MB]
asammdf 3.0.0 full mdfv3	706	256
asammdf 3.0.0 low mdfv3	637	103
asammdf 3.0.0 minimum mdfv3	612	64
mdfreader 2.7.5 mdfv3	2201	414
mdfreader 2.7.5 compress mdfv3	1871	281
mdfreader 2.7.5 noDataLoading mdfv3	948	160
asammdf 3.0.0 full mdfv4	2599	296
asammdf 3.0.0 low mdfv4	2485	131
asammdf 3.0.0 minimum mdfv4	1376	64
mdfreader 2.7.5 mdfv4	5706	435
mdfreader 2.7.5 compress mdfv4	5453	303
mdfreader 2.7.5 noDataLoading mdfv4	3904	181

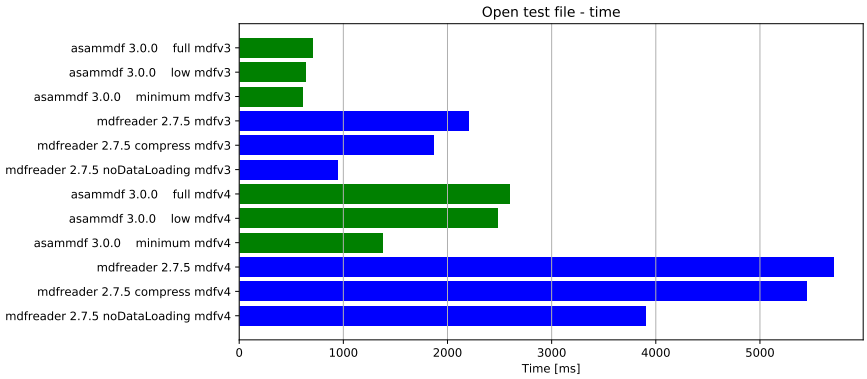
Save file	Time [ms]	RAM [MB]
asammdf 3.0.0 full mdfv3	468	258
asammdf 3.0.0 low mdfv3	363	110
asammdf 3.0.0 minimum mdfv3	919	80
mdfreader 2.7.5 mdfv3	6424	451
mdfreader 2.7.5 noDataLoading mdfv3	7364	510
mdfreader 2.7.5 compress mdfv3	6624	449
asammdf 3.0.0 full mdfv4	984	319
asammdf 3.0.0 low mdfv4	1028	156
asammdf 3.0.0 minimum mdfv4	2786	80
mdfreader 2.7.5 mdfv4	3355	460
mdfreader 2.7.5 noDataLoading mdfv4	5153	483
mdfreader 2.7.5 compress mdfv4	3773	457

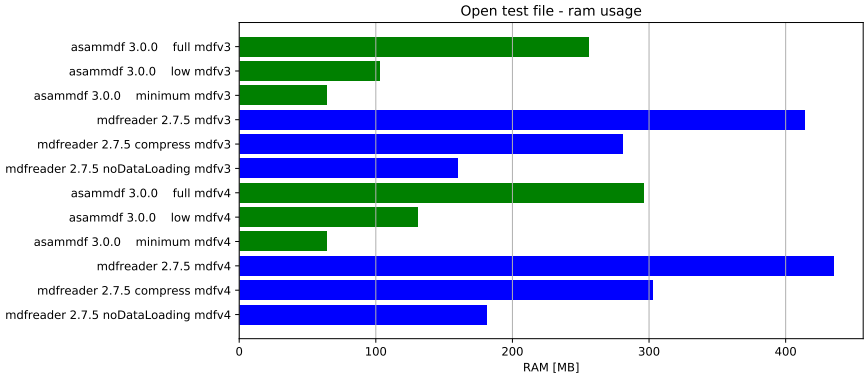
Get all channels (36424 calls)	Time [ms]	RAM [MB]
asammdf 3.0.0 full mdfv3	1196	269
asammdf 3.0.0 low mdfv3	5230	121
asammdf 3.0.0 minimum mdfv3	6871	85
mdfreader 2.7.5 mdfv3	77	414
mdfreader 2.7.5 noDataLoading mdfv3	13036	195
mdfreader 2.7.5 compress mdfv3	184	281
asammdf 3.0.0 full mdfv4	1207	305
asammdf 3.0.0 low mdfv4	5613	144
asammdf 3.0.0 minimum mdfv4	7725	80
mdfreader 2.7.5 mdfv4	74	435
mdfreader 2.7.5 noDataLoading mdfv4	14140	207
mdfreader 2.7.5 compress mdfv4	171	307

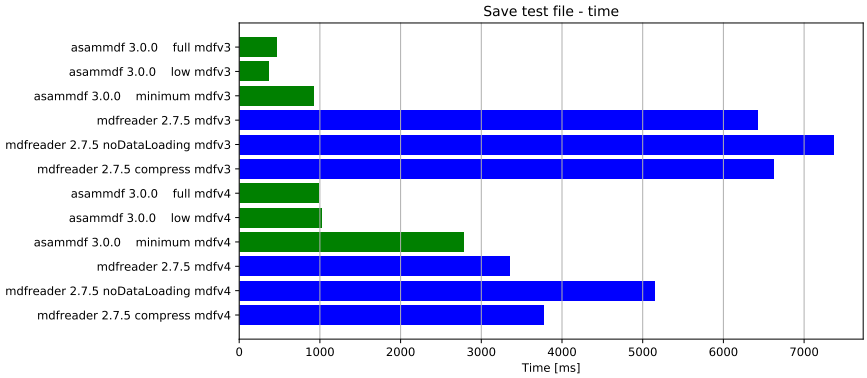
Convert file	Time [ms]	RAM [MB]
asammdf 3.0.0 full v3 to v4	3712	565
asammdf 3.0.0 low v3 to v4	4091	228
asammdf 3.0.0 minimum v3 to v4	6740	126
asammdf 3.0.0 full v4 to v3	3787	571
asammdf 3.0.0 low v4 to v3	4546	222
asammdf 3.0.0 minimum v4 to v3	8369	115

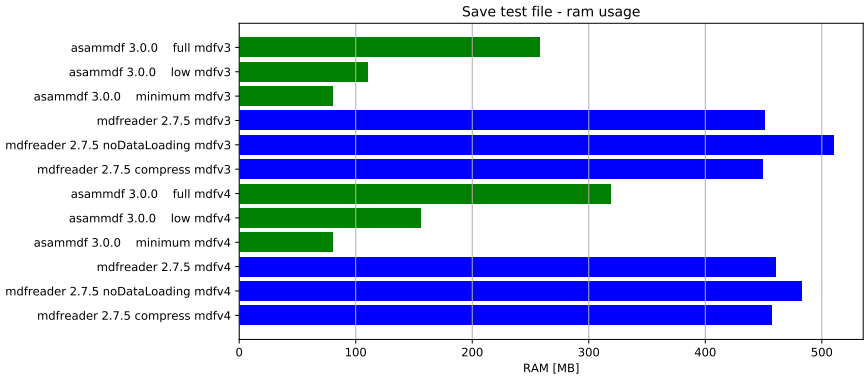
Merge files	Time [ms]	RAM [MB]
asammdf 3.0.0 full v3	7297	975
asammdf 3.0.0 low v3	7766	282
asammdf 3.0.0 minimum v3	11363	163
mdfreader 2.7.5 mdfv3	13039	1301
mdfreader 2.7.5 compress mdfv3	12877	1298
mdfreader 2.7.5 noDataLoading mdfv3	12981	1421
asammdf 3.0.0 full v4	11313	1025
asammdf 3.0.0 low v4	12155	322
asammdf 3.0.0 minimum v4	18787	152
mdfreader 2.7.5 mdfv4	21423	1309
mdfreader 2.7.5 noDataLoading mdfv4	20142	1352
mdfreader 2.7.5 compress mdfv4	20600	1309

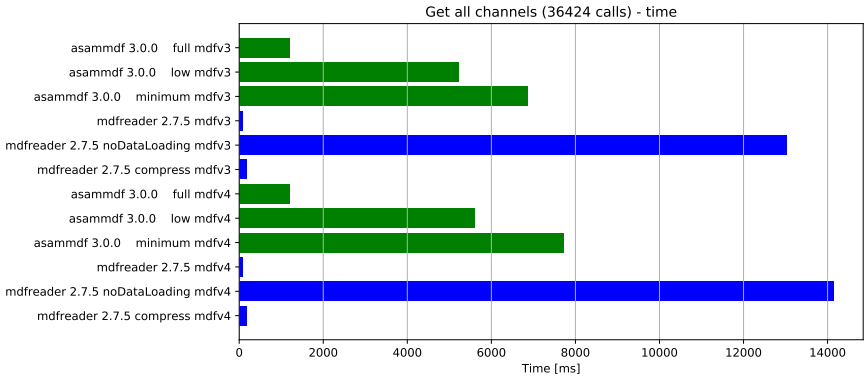
6.2.2 Graphical results

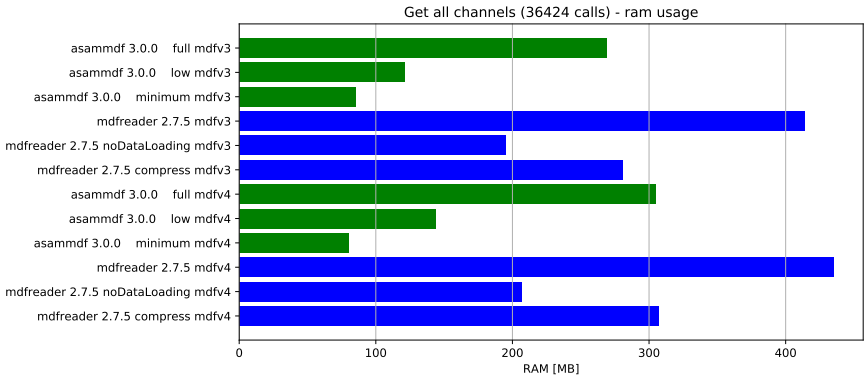


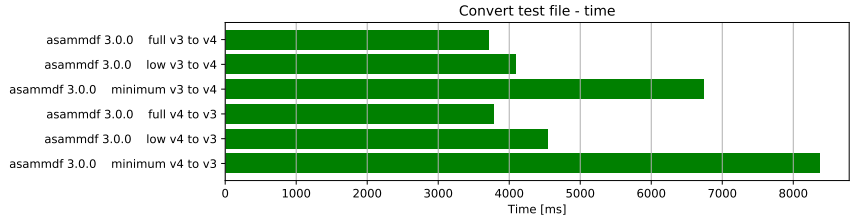


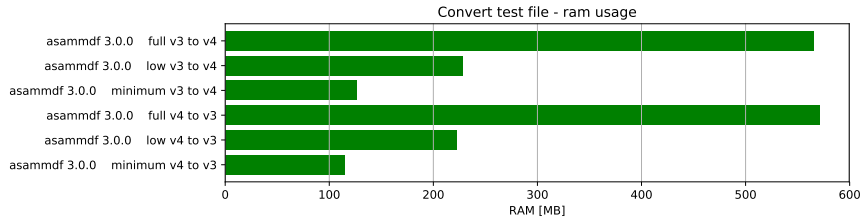

Save test file - time
asammdf 3.0.0 full mdfv3
asammdf 3.0.0 low mdfv3
asammdf 3.0.0 minimum mdfv3
mdfreader 2.7.5 mdfv3
mdfreader 2.7.5 noDataLoading mdfv3
mdfreader 2.7.5 compress mdfv3
asammdf 3.0.0 full mdfv4
asammdf 3.0.0 low mdfv4
asammdf 3.0.0 minimum mdfv4
mdfreader 2.7.5 mdfv4
mdfreader 2.7.5 noDataLoading mdfv4
mdfreader 2.7.5 compress mdfv4
0
1000
2000
3000
4000
5000
6000
7000
Time [ms]

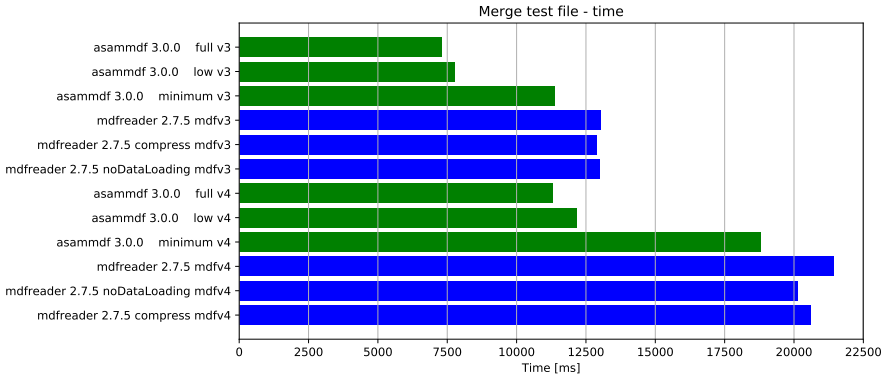

Save test file - ram usage
asammdf 3.0.0 full mdfv3
asammdf 3.0.0 low mdfv3
asammdf 3.0.0 minimum mdfv3
mdfreader 2.7.5 mdfv3
mdfreader 2.7.5 noDataLoading mdfv3
mdfreader 2.7.5 compress mdfv3
asammdf 3.0.0 full mdfv4
asammdf 3.0.0 low mdfv4
asammdf 3.0.0 minimum mdfv4
mdfreader 2.7.5 mdfv4
mdfreader 2.7.5 noDataLoading mdfv4
mdfreader 2.7.5 compress mdfv4
0
100
200
300
400
500
RAM [MB]

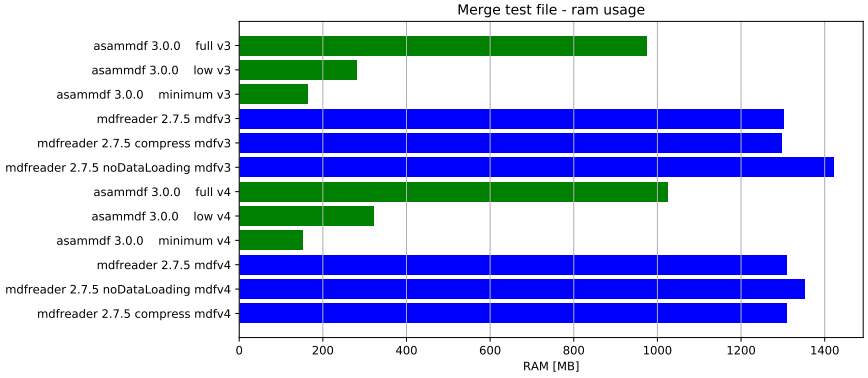

Get all channels (36424 calls) - time
asammdf 3.0.0 full mdfv3
asammdf 3.0.0 low mdfv3
asammdf 3.0.0 minimum mdfv3
mdfreader 2.7.5 mdfv3
mdfreader 2.7.5 noDataLoading mdfv3
mdfreader 2.7.5 compress mdfv3
asammdf 3.0.0 full mdfv4
asammdf 3.0.0 low mdfv4
asammdf 3.0.0 minimum mdfv4
mdfreader 2.7.5 mdfv4
mdfreader 2.7.5 noDataLoading mdfv4
mdfreader 2.7.5 compress mdfv4
0
2000
4000
6000
8000
10000
12000
14000
Time [ms]


Get all channels (36424 calls) - ram usage
asammdf 3.0.0 full mdfv3
asammdf 3.0.0 low mdfv3
asammdf 3.0.0 minimum mdfv3
mdfreader 2.7.5 mdfv3
mdfreader 2.7.5 noDataLoading mdfv3
mdfreader 2.7.5 compress mdfv3
asammdf 3.0.0 full mdfv4
asammdf 3.0.0 low mdfv4
asammdf 3.0.0 minimum mdfv4
mdfreader 2.7.5 mdfv4
mdfreader 2.7.5 noDataLoading mdfv4
mdfreader 2.7.5 compress mdfv4
0
100
200
300
400
RAM [MB]


Convert test file - time
asammdf 3.0.0 full v3 to v4
asammdf 3.0.0 low v3 to v4
asammdf 3.0.0 minimum v3 to v4
asammdf 3.0.0 full v4 to v3
asammdf 3.0.0 low v4 to v3
asammdf 3.0.0 minimum v4 to v3
0
1000
2000
3000
4000
5000
6000
7000
8000
Time [ms]


Convert test file - ram usage
asammdf 3.0.0 full v3 to v4
asammdf 3.0.0 low v3 to v4
asammdf 3.0.0 minimum v3 to v4
asammdf 3.0.0 full v4 to v3
asammdf 3.0.0 low v4 to v3
asammdf 3.0.0 minimum v4 to v3
0
100
200
300
400
500
600
RAM [MB]


Merge test file - time
asammdf 3.0.0 full v3
asammdf 3.0.0 low v3
asammdf 3.0.0 minimum v3
mdfreader 2.7.5 mdfv3
mdfreader 2.7.5 compress mdfv3
mdfreader 2.7.5 noDataLoading mdfv3
asammdf 3.0.0 full v4
asammdf 3.0.0 low v4
asammdf 3.0.0 minimum v4
mdfreader 2.7.5 mdfv4
mdfreader 2.7.5 noDataLoading mdfv4
mdfreader 2.7.5 compress mdfv4
0
2500
5000
7500
10000
12500
15000
17500
20000
22500
Time [ms]


Merge test file - ram usage
asammdf 3.0.0 full v3
asammdf 3.0.0 low v3
asammdf 3.0.0 minimum v3
mdfreader 2.7.5 mdfv3
mdfreader 2.7.5 compress mdfv3
mdfreader 2.7.5 noDataLoading mdfv3
asammdf 3.0.0 full v4
asammdf 3.0.0 low v4
asammdf 3.0.0 minimum v4
mdfreader 2.7.5 mdfv4
mdfreader 2.7.5 noDataLoading mdfv4
mdfreader 2.7.5 compress mdfv4
0
200
400
600
800
1000
1200
1400
RAM [MB]

CHAPTER 7

GUI

Starting with version 3.4.0 there is a graphical user interface that comes together with *asammdf*.

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

- PyQt4 or 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

- **Memory**: this specifies the *memory* argument when instantiating the *MDF* object. It can have one of the values:
  - `full`
  - `low`

  - `minimum`

  Changing this option does not affect already opened files; it will only apply later when openeing a new file.

- **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
- **Integer line style**: controls how integer type channels are displayed
  - `Step mode`: the line that joins the channel samples is a step line
  - `Direct connect mode`: the samples are joined using straight lines
- **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

Note:

- Changing the *Memory* option does not affect already opened files; it will only apply later when openeing a new file.

7.1.3 Plot

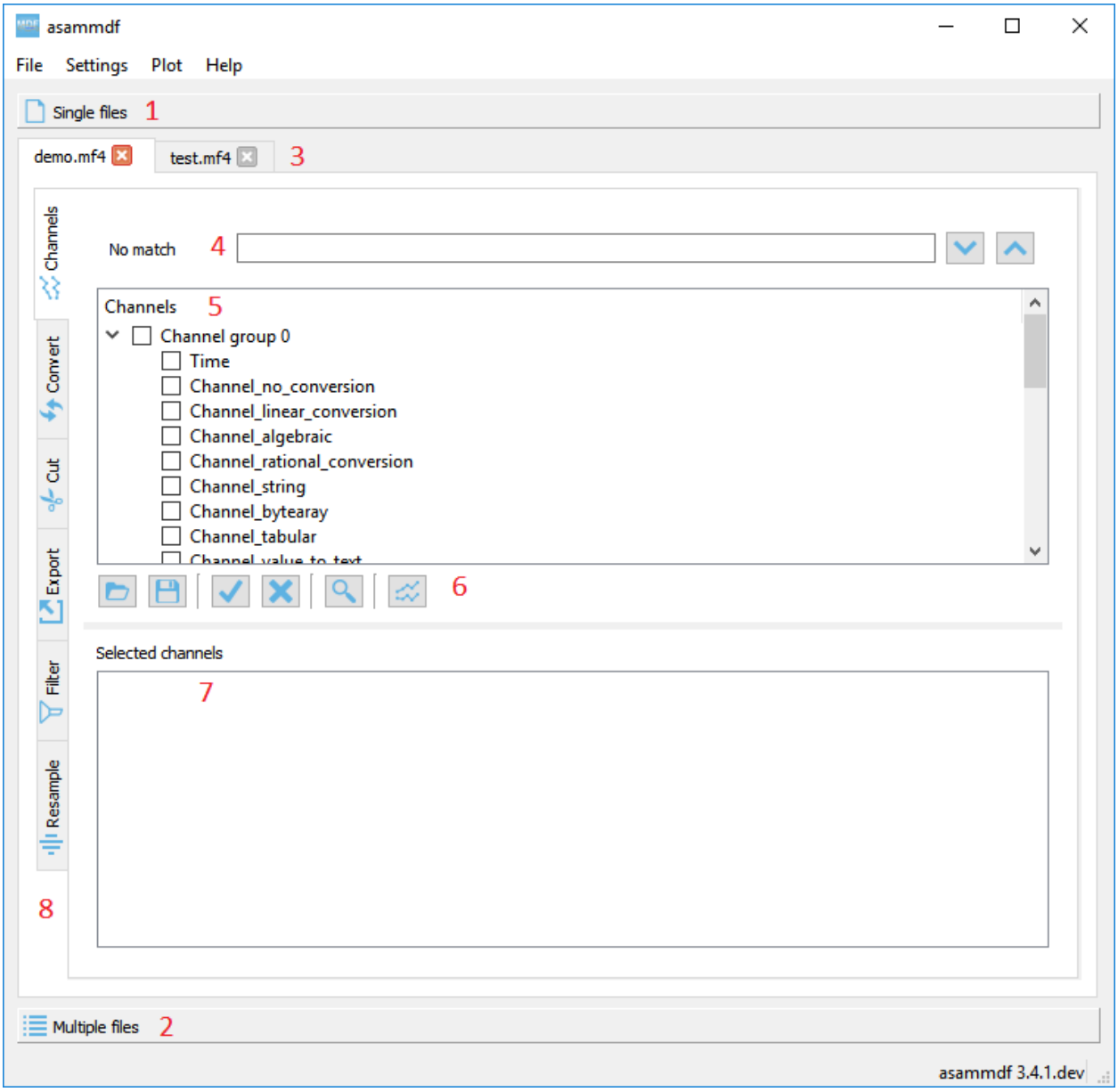
There are several keyboard shortcuts for handling the plots:

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
Ctrl+B	Bin	Toggle binary reprezentation of integer channels
Ctrl+H	Hex	Toggle hex reprezentation of integer channels
Ctrl+P	Physical	Toggle physical reprezentation of integer channels

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

7.2 Single files

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



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

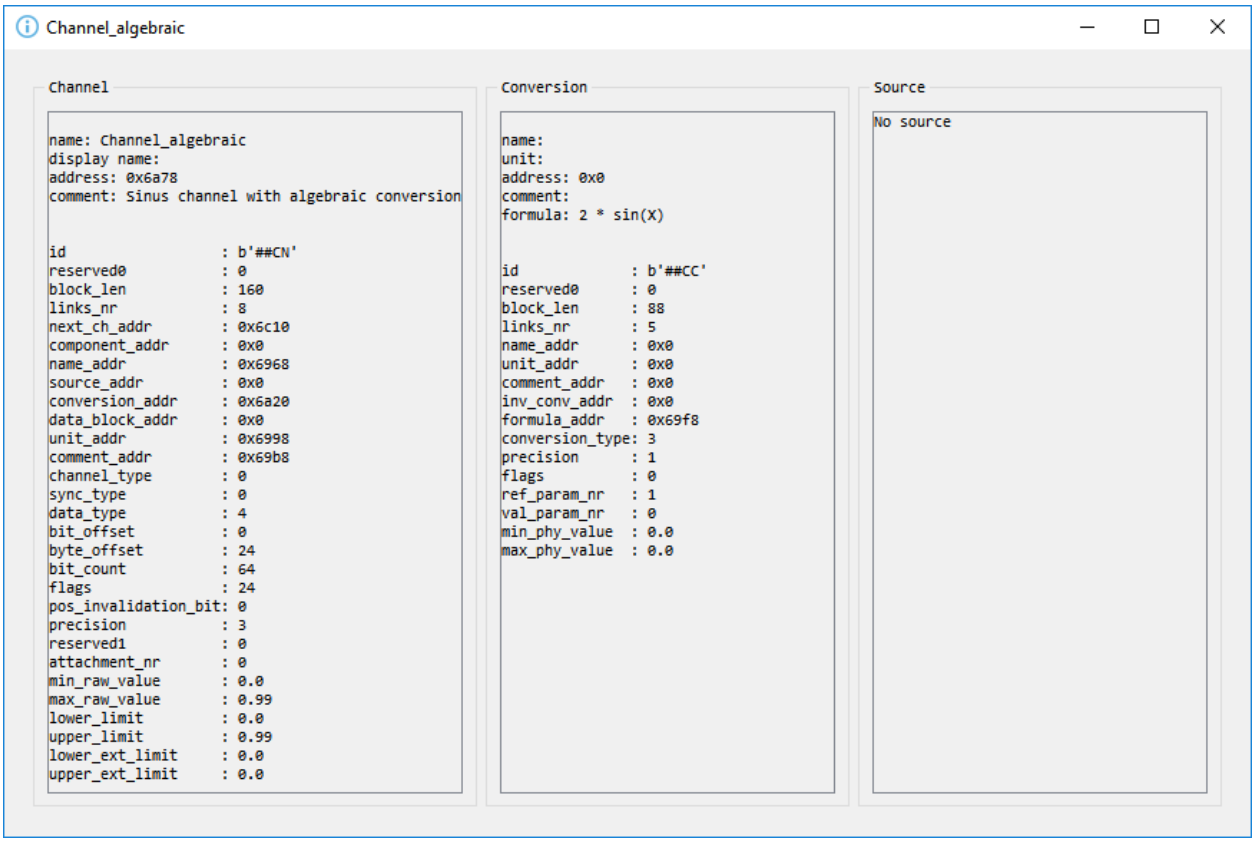
Using the *Seetings->Searh* menu option the user can choose how the search is performed. A positive search match will scroll the channels tree and highlight the channel entry.

When the same channel name exist several times in the file, you can switch between the occurences 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)

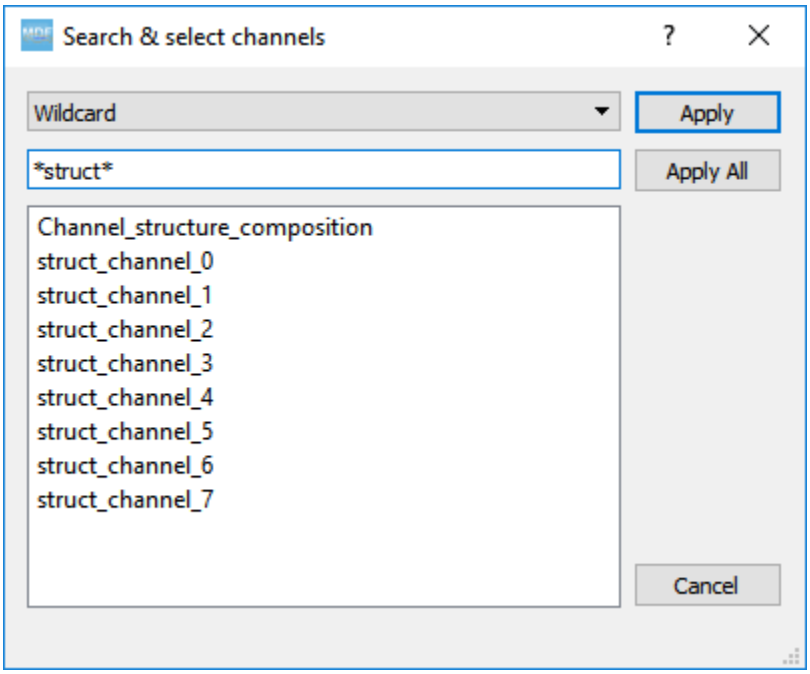


Only the channels that are checked in the channels tree will pe selected for plotting when the *Plot* button willbe pressed. Checking or unchecking channels will not affect the current plot.

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*.

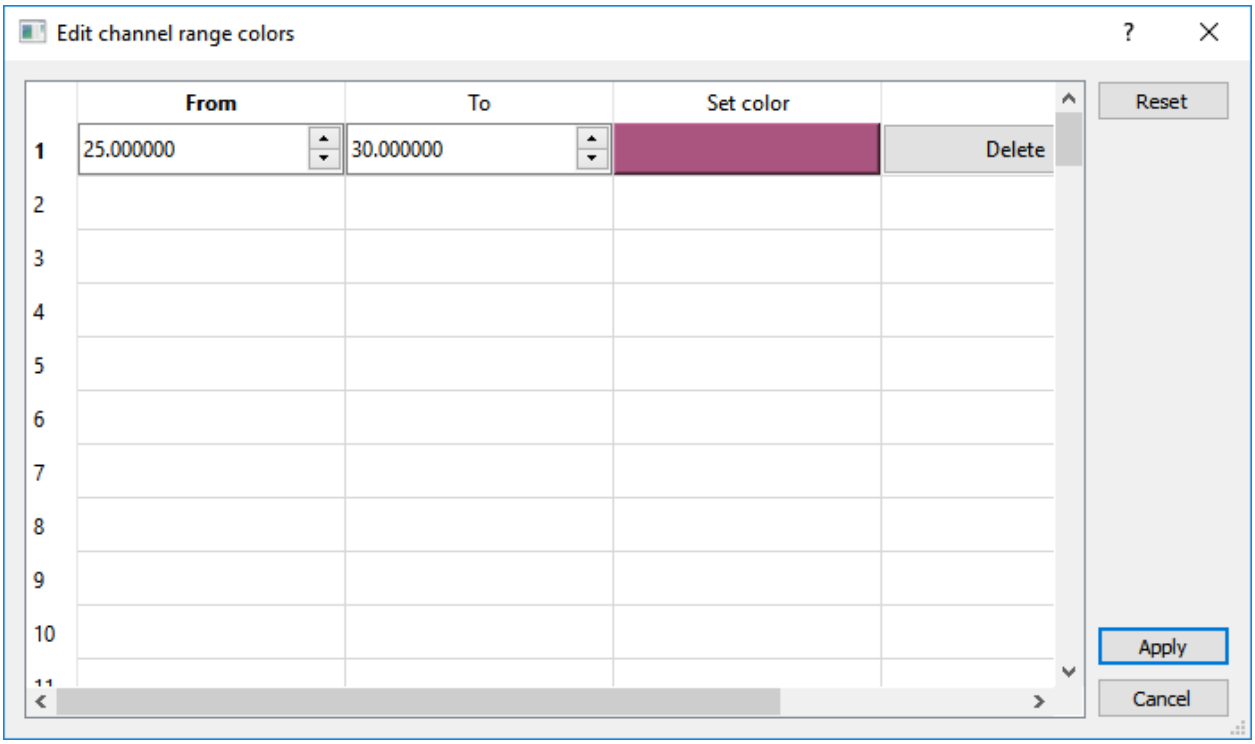
Selecting items from the *Selected channels list* will display their Y-axis on the right side of the plot, if the items are enabled for display.

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

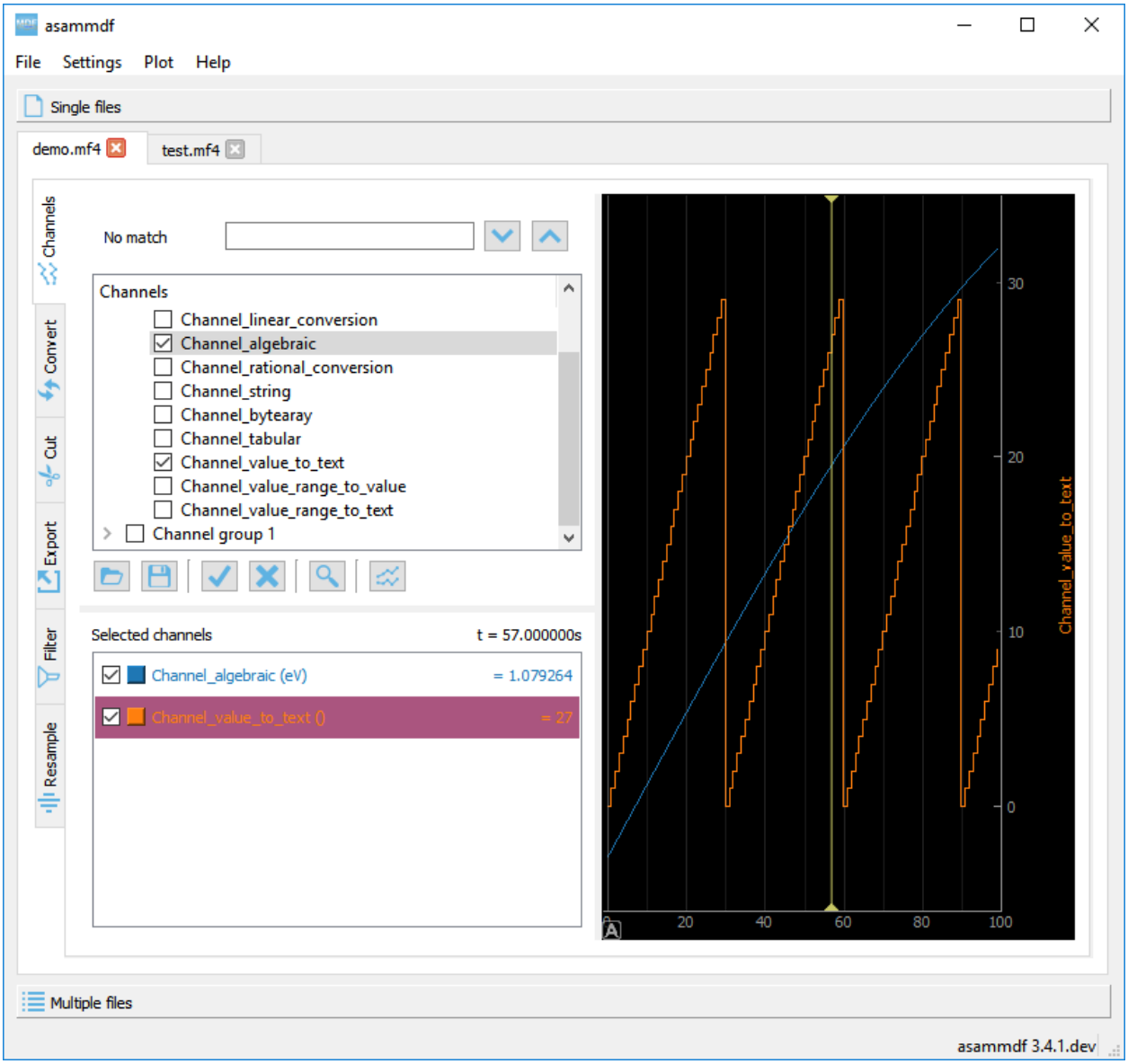
Each item has four elements

1. display enable checkbox
2. color select button
3. channel name and unit label
4. channel value label
    - 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 shwo the value delta between the range start and stop timestamps

Double clicking an item will open a range editor dialog

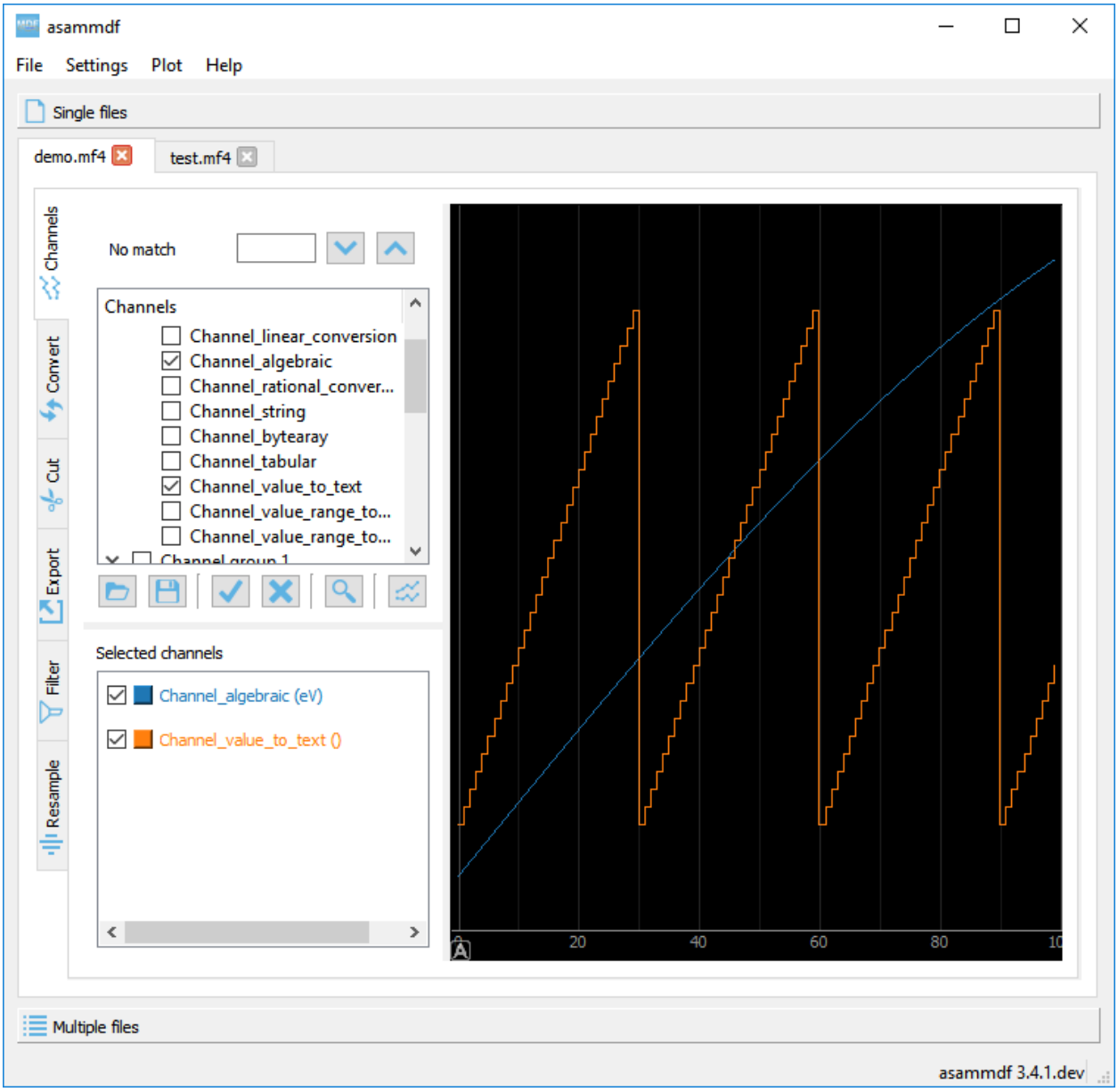


Here we can specify a range value visual alert. When the cursor is active and the current channel value is within the specified range, the item background will change to the selected color.

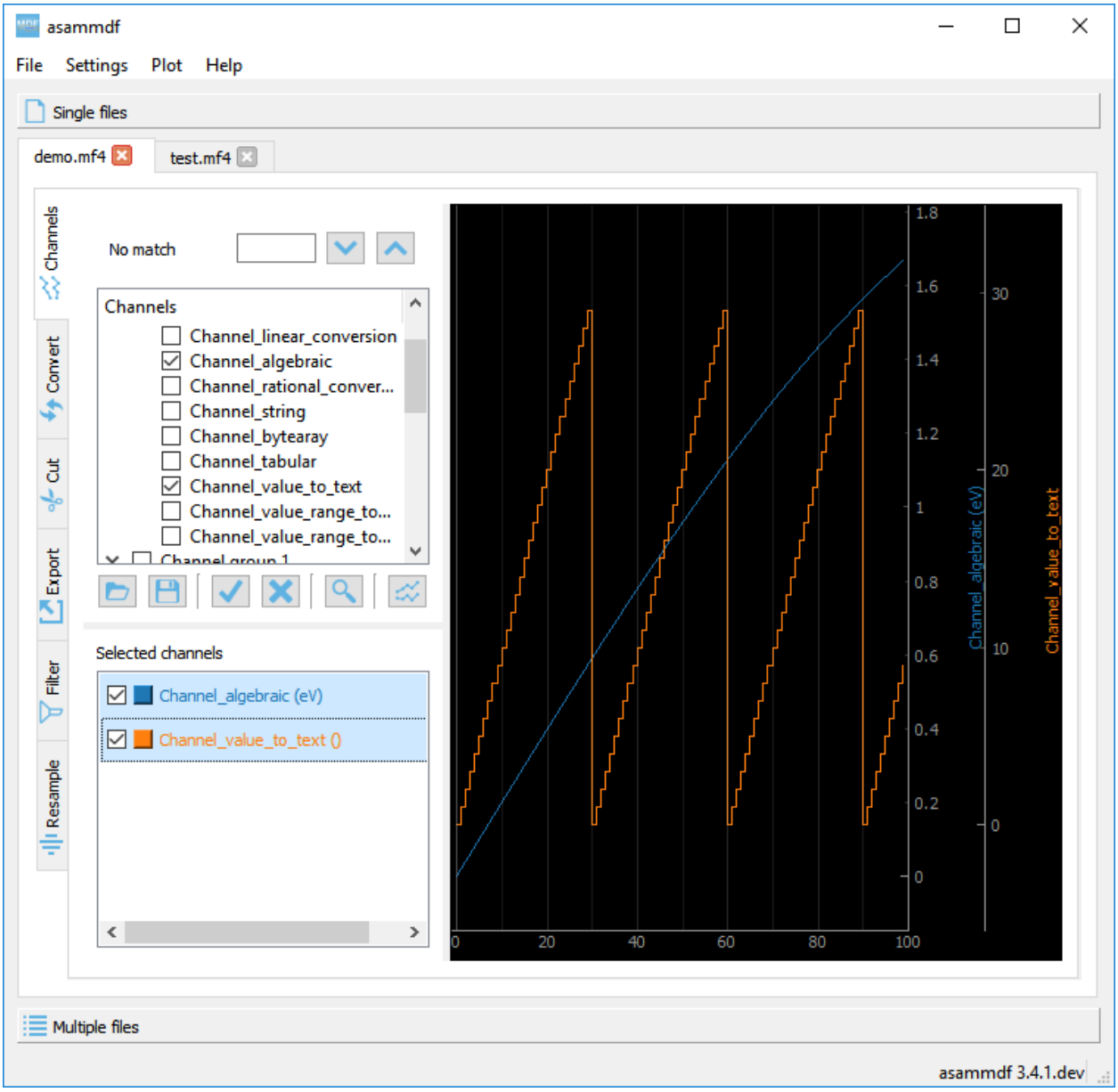


7.2.6 Plot

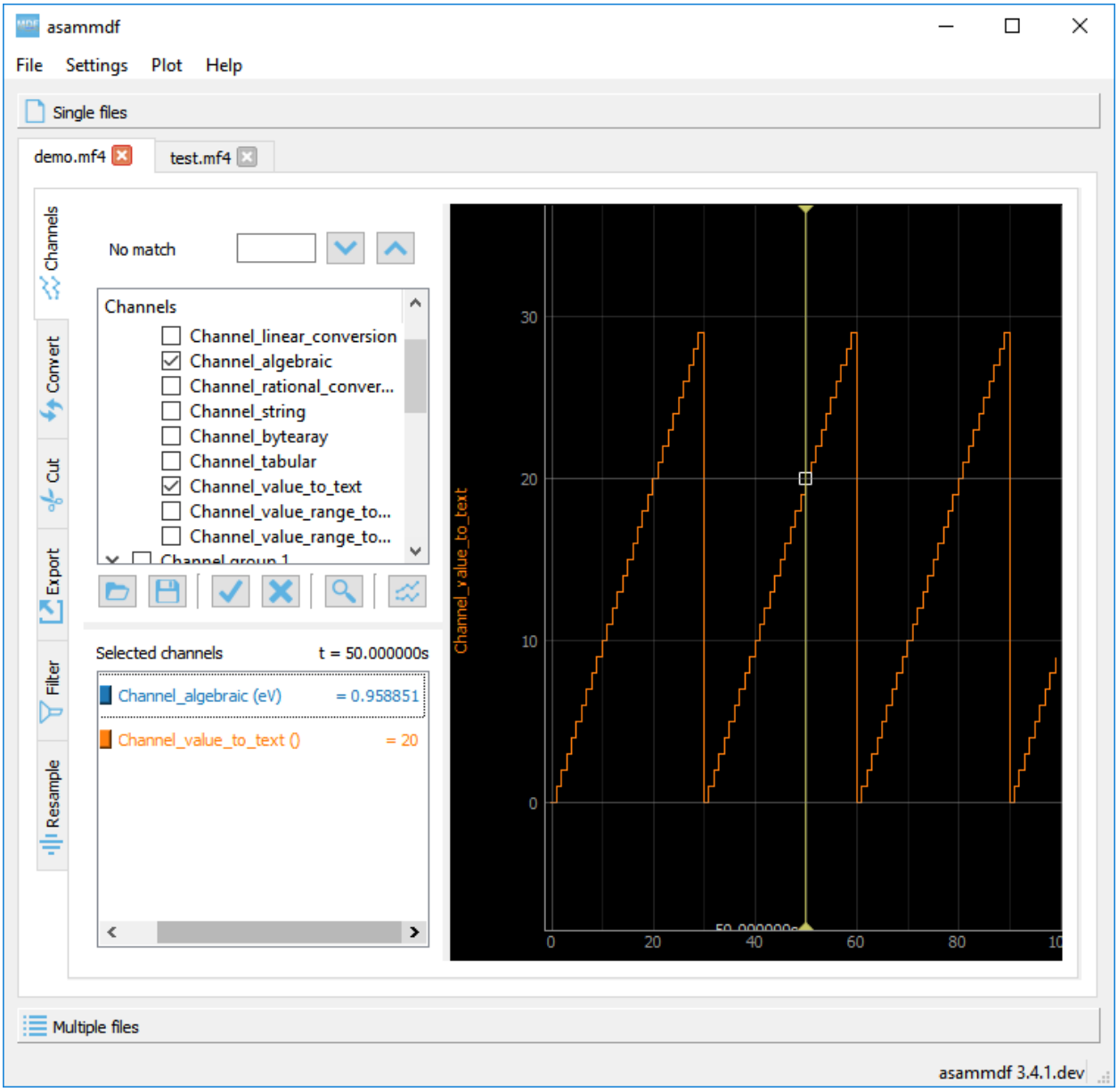
The initial plot will have all channels homed (see the *H* keyboard shortcut) and Y-axis disabled



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

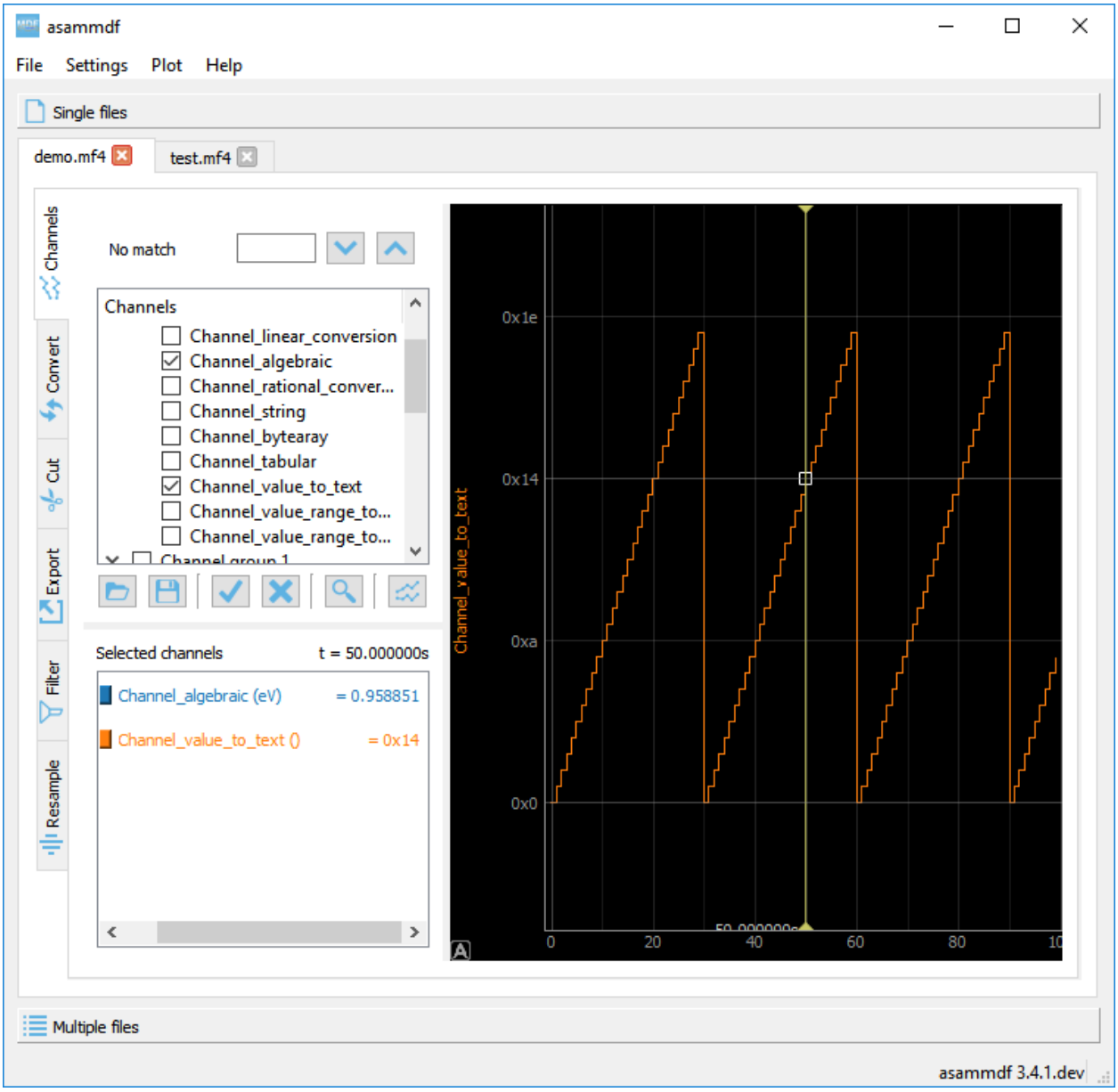


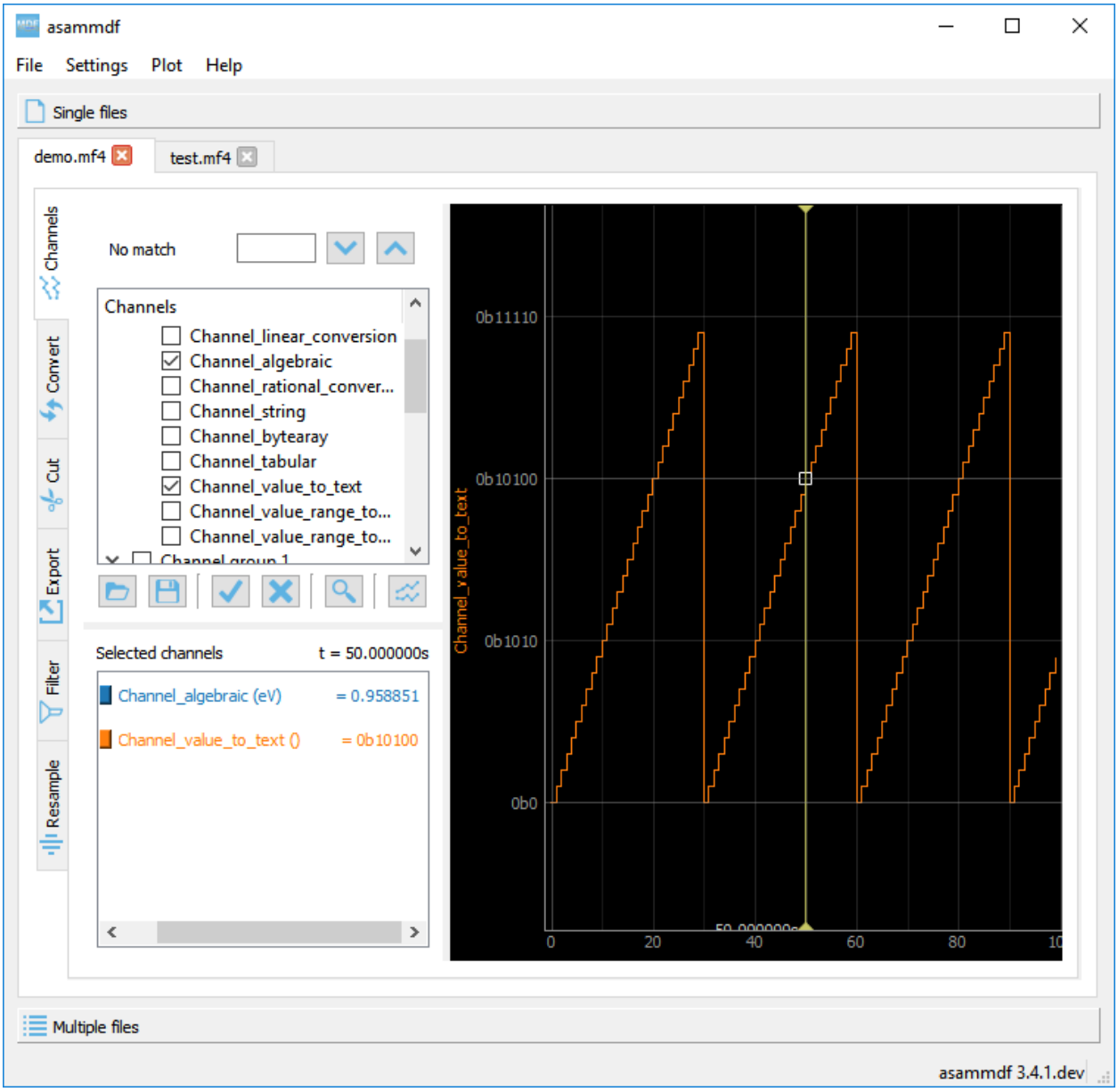
Using the *C* keyboard shotcut will toggle the cursor, and with it the channel values will be displayed for each item in the *Selected channels list*



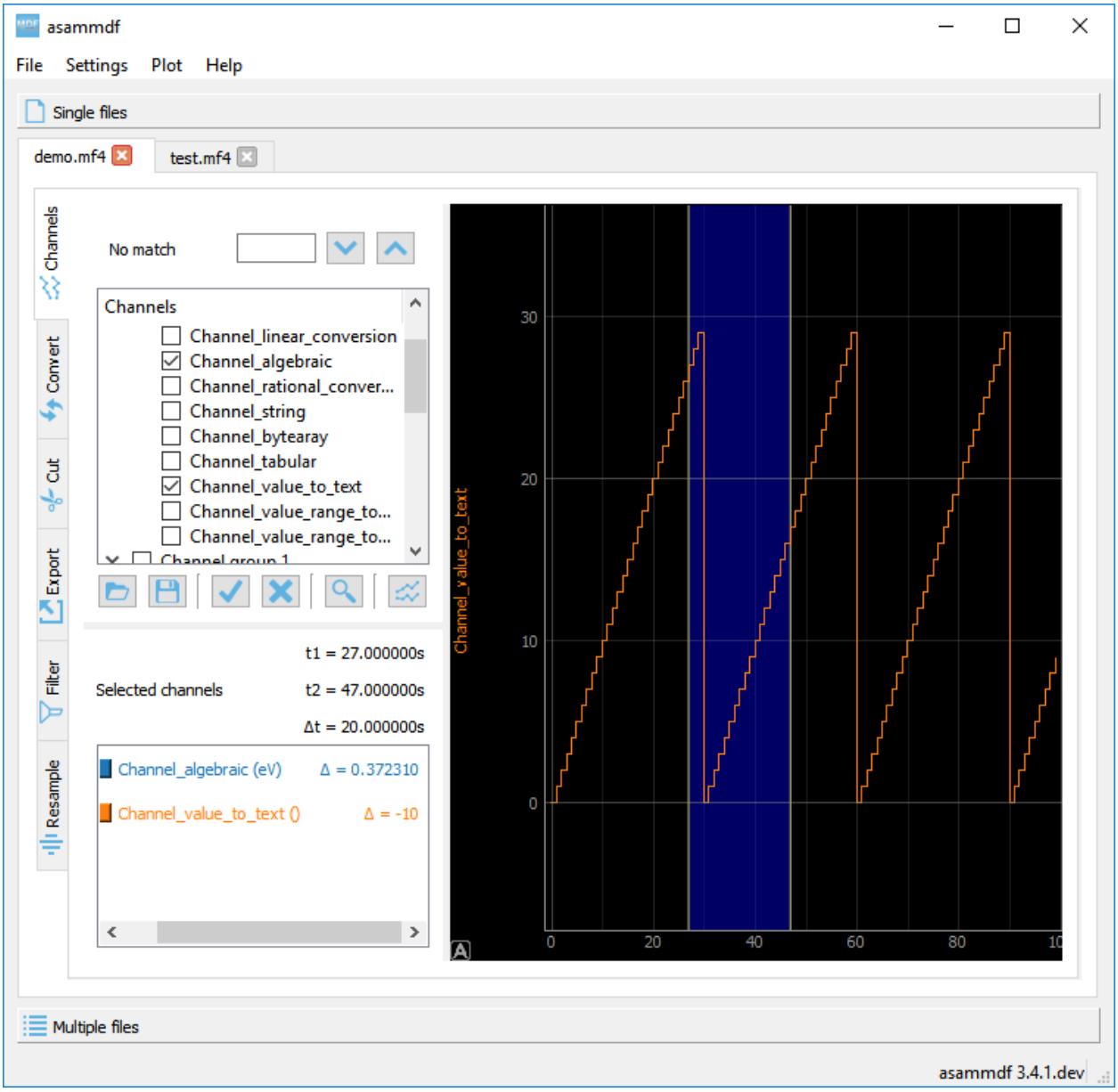
The *Ctrl+H* and *Ctrl+B* keyboard shortcuts will

- 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*

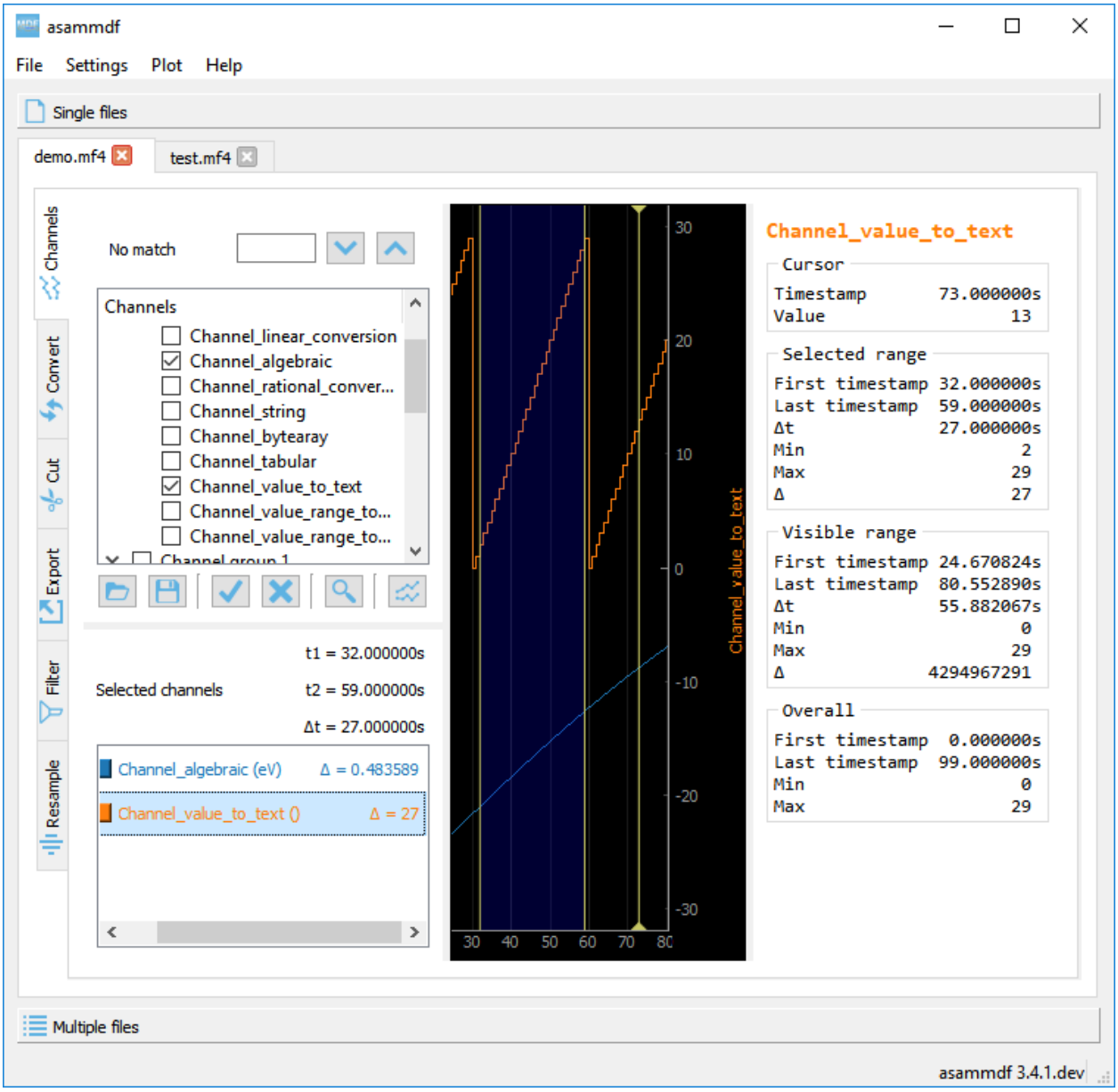

asammdf
File
Settings
Plot
Help
Single files
demo.mf4
test.mf4
Channels
Convert
Cut
Export
Filter
Resample
No match
Channels
Channel_linear_conversion
Channel_algebraic
Channel_rational_conver...
Channel_string
Channel_bytearay
Channel_tabular
Channel_value_to_text
Channel_value_range_to...
Channel_value_range_to...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



Using the *R* keyboard shotcut 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.

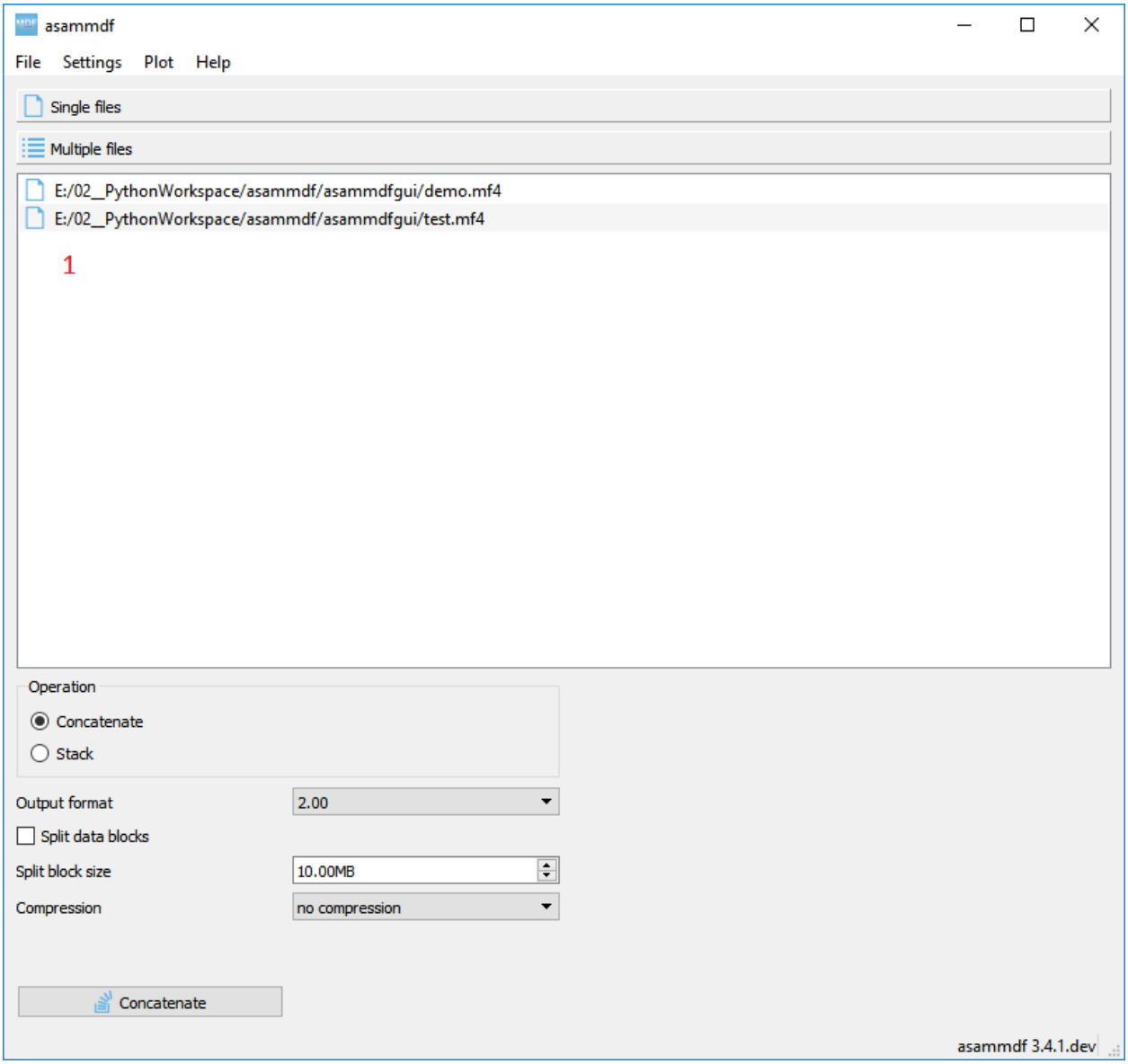


The *Statistics* panel is toggle using the *M* keyboard shortcut



7.3 Multiple files

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



The files list can be rearanged 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