

---

# **asammdf Documentation**

***Release 2.8.0***

**Daniel Hrisca**

**Dec 13, 2017**



---

## Contents

---

<b>1</b>	<b>Project goals</b>	<b>3</b>
<b>2</b>	<b>Features</b>	<b>5</b>
<b>3</b>	<b>Major features not implemented (yet)</b>	<b>7</b>
<b>4</b>	<b>Dependencies</b>	<b>9</b>
<b>5</b>	<b>Installation</b>	<b>11</b>
<b>6</b>	<b>API</b>	<b>13</b>
<b>7</b>	<b>Benchmarks</b>	<b>73</b>
<b>8</b>	<b>Indices and tables</b>	<b>83</b>



*asammdf* is a fast parser/editor for ASAM (Association 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  $\geq 3.4$



# CHAPTER 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





## CHAPTER 2

---

### Features

---

- create new mdf files from scratch
- append new channels
- read unsorted MDF v3 and v4 files
- filter a subset of channels from original mdf file
- cut measurement to specified time interval
- convert to different mdf version
- export to Excel, HDF5, Matlab and CSV
- merge multiple files sharing the same internal structure
- read and save mdf version 4.10 files containing zipped data blocks
- split large data blocks (configurable size) for mdf version 4
- disk space savings by compacting 1-dimensional integer channels (configurable)
- 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
- files are loaded in RAM for fast operations
- handle large files (exceeding the available RAM) 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
  - usually a measurement will have channels from different sources at different rates
  - the *Signal* class facilitates operations with such channels

---

### Major features not implemented (yet)

---

- for version 3
  - functionality related to sample reduction block (but the class is defined)
- for version 4
  - handling of bus logging measurements
  - handling of unfinished measurements (mdf 4)
  - full support for remaining mdf 4 channel arrays types
  - xml schema for TXBLOCK and MDBLOCK
  - partial conversions
  - event blocks
  - channels with default X axis
  - channels with reference to attachment



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

optional dependencies needed for exports

- h5py : for HDF5 export
- xlswriter : for Excel export
- scipy : for Matlab .mat export



## CHAPTER 5

---

### Installation

---

*asammdf* is available on

- github: <https://github.com/danielhrisca/asammdf/>
- PyPI: <https://pypi.org/project/asammdf/>

```
pip install asammdf
```





## 6.1 Package level

`asammdf.configure` (*integer\_compacting=None, split\_data\_blocks=None, split\_threshold=None, overwrite=None*)  
configure asammdf parameters

**Parameters** `integer_compacting` : bool

enable/disable compacting of integer channels on append. This has the potential to greatly reduce file size, but append speed is slower and further loading of the resulting file will also be slower.

**split\_data\_blocks** : bool

enable/disable splitting of large data blocks using data lists for mdx version 4

**split\_threshold** : int

size hint of splitted data blocks, default 2MB; if the initial size is smaller then no data list is used. The actual split size depends on the data groups' records size

**overwrite** : bool

default option for save method's overwrite argument

Enabling compacting of integer channels on append the file size of the resulting file can decrease up to a factor of ~0.5. Splitting the data blocks is usefull for large blocks. The recommended maximum threshold by ASAM is 4MB. *asammdf* uses a default of 2MB

## 6.2 MDF

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

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.

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'*)  
 Unified access to MDF v3 and v4 files.

**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 ('3.00', '3.10', '3.20', '3.30', '4.00', '4.10', '4.11'); default '4.10'

## Methods

<code>convert(to[, memory])</code>	convert MDF to other versions
<code>cut([start, stop, whence])</code>	convert MDF to other versions
<code>export(fmt[, filename])</code>	export MDF to other formats. The <i>MDF</i> file name is used is
<code>filter(channels[, memory])</code>	return new <i>MDF</i> object that contains only the channels listed in
<code>iter_to_pandas()</code>	generator that yields channel groups as pandas DataFrames
<code>merge(files[, outversion, memory])</code>	merge several files and return the merged MDF object. The files
<code>resample(raster[, memory])</code>	resample all channels to given raster
<code>select(channels[, dataframe])</code>	return the channels listed in <i>channels</i> argument

**convert** (*to*, *memory='full'*)  
 convert MDF to other versions

**Parameters** *to* : str

new mdf version from ('3.00', '3.10', '3.20', '3.30', '4.00', '4.10', '4.11')

**memory** : str

memory option; default *full*

**Returns** *out* : MDF

new MDF object

**cut** (*start=None, stop=None, whence=0*)  
convert MDF to other versions

**Parameters** **start** : float

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

**stop** : float

stop time, default . 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*)  
export MDF to other formats. The *MDF* file name is used is available, else the *filename* aragument 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 (<MDF-NAME>\_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 (<MDF-NAME>\_DataGroup\_<cntr>.xlsx)
- *mat* : Matlab .mat version 5 export, for Matlab >= 7.6. In the mat file the channels will be renamed 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)

**filename** : string

export file name

**filter** (*channels, memory=None*)  
return new *MDF* object that contains only the channels listed in *channels* argument

**Parameters** **channels** : list

list of channel names to be filtered

**memory** : str

memory option for filtered mdf; default None in which case the original file’s memory option is used

**Returns** **mdf** : MDF

new MDF file

**iter\_to\_pandas** ()

generator that yields channel groups as pandas DataFrames

**static merge** (*files*, *outversion*='4.10', *memory*='full')

merge several files and return the merged MDF object. 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

**outversion** : str

merged file version

**memory** : str

memory option; default *full*

**Returns** **merged** : MDF

new MDF object with merged channels

**Raises** **MdfException** : if there are inconsistencies between the files

merged MDF object

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

resample all channels to 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)

return the channels listed in *channels* argument

**Parameters** **channels** : list

list of channel names to be filtered

**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

## 6.2.1 MDF2, MDF3 and MDF4 classes

### MDF2

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

The *header* attribute 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
- *channel\_dependencies* : list of ChannelDependency objects in a 1-to-1 relation with the channel list
- *data\_block* : DataBlock object
- *texts* : dictionary containing TextBlock objects used throughout the mdf
  - *channels* : list of dictionaries that contain TextBlock objects related 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 related to each channel group
    - \* *comment\_addr* : channel group comment
  - *conversion\_tab* : list of dictionaries that contain TextBlock objects related 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 loading of measured data
- *record\_size* : dict of record ID -> record size pairs

The *file\_history* attribute is a TextBlock object.

The *channel\_db* attribute 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* attribute 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.

### API

**class** asammdf.mdf3.**MDF3** (*name=None, memory=2, version='3.30'*)

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 memoryed in RAM
- if *low* the channel data is read from disk on request, and the

metadata is memoryed into RAM \* if *minimum* only minimal data is memoryed into RAM

**version** : string

mdf file version ('3.00', '3.10', '3.20' or '3.30'); default '3.30'

## Attributes

<b>name</b>	(string) mdf file name
<b>groups</b>	(list) list of data groups
<b>header</b>	(OrderedDict) mdf file header
<b>file_history</b>	(TextBlock) file history text block; can be None
<b>memory</b>	(bool) load measured data option
<b>version</b>	(str) mdf version
<b>chan-nels_db</b>	(dict) used for fast channel access by name; for each name key the value is a list of (group index, channel index) tuples
<b>mas-ters_db</b>	(dict) used for fast master channel access; for each group index key the value is the master channel index

## Methods

<code>add_trigger(group, timestamp[, pre_time, ...])</code>	add trigger to data group
<code>append(signals[, acquisition_info, ...])</code>	Appends a new data group.
<code>close()</code>	if the MDF was created with memory='minimum' and new
<code>get([name, group, index, raster, ...])</code>	Gets channel samples.
<code>get_channel_comment([name, group, index])</code>	Gets channel comment.
<code>get_channel_unit([name, group, index])</code>	Gets channel unit.
<code>get_master(index[, data])</code>	returns master channel samples for given group
<code>info()</code>	get MDF information as a dict
<code>iter_get_triggers()</code>	generator that yields triggers
<code>save([dst, overwrite, compression])</code>	Save MDF to <i>dst</i> .

**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, timestamps=t, unit='+', name='Positive')
>>> s2 = Signal(samples=s2, timestamps=t, unit='-', name='Negative')
>>> s3 = Signal(samples=s3, timestamps=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

**get** (*name*=None, *group*=None, *index*=None, *raster*=None, *samples\_only*=False, *data*=None)

Gets channel samples. 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

**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

**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** *MdfError* :

- \* if the channel name is not found
- \* if the group index is out of range
- \* if the channel index is out of range

**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\_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*)

returns master channel samples for given group

**Parameters index** : int

group index

**data** : bytes

data block raw bytes; 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=None*, *compression=0*)

Save MDF to *dst*. If *dst* is not provided 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

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

## MDF version 2 blocks

The following classes implement different MDF version3 blocks.

### Channel Class

**class** `asammdf.mdf2.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” ( $= N \text{ div } 8$ ) and a “Bit offset” ( $= N \text{ mod } 8$ ). 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) \* `additional_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

<b>name</b>	(str) full channel name
<b>address</b>	(int) block address inside mdf file
<b>dependencies</b>	(list) list of channel dependencies

## Methods

<code>clear()</code>	-> None. Remove all items from D.)
<code>copy()</code>	-> a shallow copy of D)
<code>fromkeys(\$type, iterable[, value])</code>	Returns a new dict with keys from iterable and values equal to value.
<code>get((k[,d]) -&gt; D[k] if k in D, ...)</code>	
<code>items(...)</code>	
<code>keys(...)</code>	
<code>pop((k[,d]) -&gt; v, ...)</code>	If key is not found, d is returned if given, otherwise Key-Error is raised
<code>popitem()</code>	-> (k, v), ...)
<code>setdefault((k[,d]) -&gt; D.get(k,d), ...)</code>	2-tuple; but raise KeyError if D is empty.
<code>update([E, ...])</code>	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
<code>values(...)</code>	

## ChannelConversion Class

**class** `asammdf.mdf2.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* - equation 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

<b>address</b>	(int) block address inside mdf file
----------------	-------------------------------------

## Methods

<code>clear()</code> -> None. Remove all items from D.)	
<code>copy()</code> -> a shallow copy of D)	
<code>fromkeys(\$type, iterable[, value])</code>	Returns a new dict with keys from iterable and values equal to value.
<code>get((k[,d]) -&gt; D[k] if k in D, ...)</code>	
<code>items(...)</code>	

Continued on next page

Table 6.4 – continued from previous page

<code>keys(...)</code>	
<code>pop((k[,d]) -&gt; v, ...)</code>	If key is not found, d is returned if given, otherwise Key-Error is raised
<code>popitem() -&gt; (k, v), ...)</code>	2-tuple; but raise KeyError if D is empty.
<code>setdefault((k[,d]) -&gt; D.get(k,d), ...)</code>	
<code>update([E, ...])</code>	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
<code>values(...)</code>	

## ChannelDependency Class

**class** `asammdf.mdf2.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 describe 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

<b>address</b>	(int) block address inside mdf file
----------------	-------------------------------------

## Methods

<code>clear()</code> -> None. Remove all items from D.)
---

<code>copy()</code> -> a shallow copy of D)
---

Continued on next page

Table 6.5 – continued from previous page

<code>fromkeys(\$type, iterable[, value])</code>	Returns a new dict with keys from iterable and values equal to value.
<code>get((k[,d]) -&gt; D[k] if k in D, ...)</code>	
<code>items(...)</code>	
<code>keys(...)</code>	
<code>pop((k[,d]) -&gt; v, ...)</code>	If key is not found, d is returned if given, otherwise Key-Error is raised
<code>popitem() -&gt; (k, v), ...)</code>	2-tuple; but raise KeyError if D is empty.
<code>setdefault((k[,d]) -&gt; D.get(k,d), ...)</code>	
<code>update([E, ...])</code>	If E is present and has a <code>.keys()</code> method, then does: for k in E: <code>D[k] = E[k]</code>
<code>values(...)</code>	

## ChannelExtension Class

**class** `asammdf.mdf2.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

<b>address</b>	(int) block address inside mdf file
----------------	-------------------------------------

## Methods

clear() -> None. Remove all items from D.)	
copy() -> a shallow copy of D)	
fromkeys(\$type, iterable[, value])	Returns a new dict with keys from iterable and values equal to value.
get((k[,d]) -> D[k] if k in D, ...)	
items(...)	
keys(...)	
pop((k[,d]) -> v, ...)	If key is not found, d is returned if given, otherwise Key-Error is raised
popitem() -> (k, v), ...)	2-tuple; but raise KeyError if D is empty.
setdefault((k[,d]) -> D.get(k,d), ...)	
update((E, ...)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
values(...)	

## ChannelGroup Class

**class** asammdf.mdf2.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

<b>address</b>	(int) block address inside mdf file
----------------	-------------------------------------

## Methods

<code>clear()</code> -> None. Remove all items from D.)	
<code>copy()</code> -> a shallow copy of D)	
<code>fromkeys(\$type, iterable[, value])</code>	Returns a new dict with keys from iterable and values equal to value.
<code>get((k[,d]) -&gt; D[k] if k in D, ...)</code>	
<code>items(...)</code>	
<code>keys(...)</code>	
<code>pop((k[,d]) -&gt; v, ...)</code>	If key is not found, d is returned if given, otherwise Key-Error is raised
<code>popitem()</code> -> (k, v), ...)	2-tuple; but raise KeyError if D is empty.
<code>setdefault((k[,d]) -&gt; D.get(k,d), ...)</code>	
<code>update((E, ...)</code>	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
<code>values(...)</code>	

## DataGroup Class

**class** `asammdf.mdf2.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

<b>address</b>	(int) block address inside mdf file
----------------	-------------------------------------

## Methods

<code>clear()</code>	-> None. Remove all items from D.)
<code>copy()</code>	-> a shallow copy of D)
<code>fromkeys(\$type, iterable[, value])</code>	Returns a new dict with keys from iterable and values equal to value.
<code>get((k[,d])</code>	-> D[k] if k in D, ...)
<code>items(...)</code>	
<code>keys(...)</code>	
<code>pop((k[,d])</code>	-> v, ...)
	If key is not found, d is returned if given, otherwise Key-Error is raised
<code>popitem()</code>	-> (k, v), ...)
	2-tuple; but raise KeyError if D is empty.
<code>setdefault((k[,d])</code>	-> D.get(k,d), ...)
<code>update(([E, ...)</code>	
	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
<code>values(...)</code>	

## FileIdentificationBlock Class

**class** asammdf.mdf2.**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

<b>address</b>	(int) block address inside mdf file; should be 0 always
----------------	---

## Methods

<code>clear()</code> -> None. Remove all items from D.)	
<code>copy()</code> -> a shallow copy of D)	
<code>fromkeys(\$type, iterable[, value])</code>	Returns a new dict with keys from iterable and values equal to value.
<code>get((k[,d]) -&gt; D[k] if k in D, ...)</code>	
<code>items(...)</code>	
<code>keys(...)</code>	
<code>pop((k[,d]) -&gt; v, ...)</code>	If key is not found, d is returned if given, otherwise Key-Error is raised

Continued on next page

Table 6.9 – continued from previous page

<code>popitem()</code> -> (k, v), ...)	2-tuple; but raise <code>KeyError</code> if D is empty.
<code>setdefault((k[,d]) -&gt; D.get(k,d), ...)</code>	
<code>update([E, ...)</code>	If E is present and has a <code>.keys()</code> method, then does: for k in E: D[k] = E[k]
<code>values(...)</code>	

## HeaderBlock Class

**class** `asammdf.mdf2.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

<b>address</b>	(int) block address inside mdf file; should be 64 always
----------------	--

## Methods

<code>clear()</code>	-> None. Remove all items from D.)
<code>copy()</code>	-> a shallow copy of D)
<code>fromkeys(\$type, iterable[, value])</code>	Returns a new dict with keys from iterable and values equal to value.
<code>get((k[,d])</code>	-> D[k] if k in D, ...)
<code>items(...)</code>	
<code>keys(...)</code>	
<code>pop((k[,d])</code>	-> v, ...)
	If key is not found, d is returned if given, otherwise Key-Error is raised
<code>popitem()</code>	-> (k, v), ...)
	2-tuple; but raise KeyError if D is empty.
<code>setdefault((k[,d])</code>	-> D.get(k,d), ...)
<code>update([E, ...])</code>	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
<code>values(...)</code>	

## ProgramBlock Class

## SampleReduction Class

## TextBlock Class

**class** `asammdf.mdf2.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

<b>address</b>	(int) block address inside mdf file
<b>text_str</b>	(str) text data as unicode string

## Methods

<code>clear()</code>	-> None. Remove all items from D.)
<code>copy()</code>	-> a shallow copy of D)
<code>fromkeys(\$type, iterable[, value])</code>	Returns a new dict with keys from iterable and values equal to value.
<code>get((k[,d]) -&gt; D[k] if k in D, ...)</code>	
<code>items(...)</code>	
<code>keys(...)</code>	
<code>pop((k[,d]) -&gt; v, ...)</code>	If key is not found, d is returned if given, otherwise Key-Error is raised
<code>popitem() -&gt; (k, v), ...)</code>	2-tuple; but raise KeyError if D is empty.
<code>setdefault((k[,d]) -&gt; D.get(k,d), ...)</code>	
<code>update([E, ...])</code>	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
<code>values(...)</code>	

## TriggerBlock Class

**class** asammdf.mdf2.**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 “TX”
- *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

<b>address</b>	(int) block address inside mdf file
----------------	-------------------------------------

## Methods

<code>clear()</code>	-> None. Remove all items from D.)
<code>copy()</code>	-> a shallow copy of D)
<code>fromkeys(\$type, iterable[, value])</code>	Returns a new dict with keys from iterable and values equal to value.
<code>get((k[,d]) -&gt; D[k] if k in D, ...)</code>	
<code>items(...)</code>	
<code>keys(...)</code>	
<code>pop((k[,d]) -&gt; v, ...)</code>	If key is not found, d is returned if given, otherwise Key-Error is raised
<code>popitem()</code>	-> (k, v), ...)
<code>setdefault((k[,d]) -&gt; D.get(k,d), ...)</code>	2-tuple; but raise KeyError if D is empty.
<code>update([E, ...])</code>	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
<code>values(...)</code>	

## MDF3

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

The *header* attribute 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** : dictionary containing TextBlock objects used throughout the mdf
  - **channels** : list of dictionaries that contain TextBlock objects related 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 related to each channel group
  - \* `comment_addr` : channel group comment
- `conversion_tab` : list of dictionaries that contain TextBlock objects related 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 loading of measured data
- `record_size` : dict of record ID -> record size pairs

The `file_history` attribute is a TextBlock object.

The `channel_db` attribute 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` attribute 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.

## API

**class** `asammdf.mdf3.MDF3` (*name=None*, *memory=2*, *version='3.30'*)

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 memoryed in RAM
- if *low* the channel data is read from disk on request, and the

metadata is memoryed into RAM \* if *minimum* only minimal data is memoryed into RAM

**version** : string

mdf file version ('3.00', '3.10', '3.20' or '3.30'); default '3.30'



## Attributes

<b>name</b>	(string) mdf file name
<b>groups</b>	(list) list of data groups
<b>header</b>	(OrderedDict) mdf file header
<b>file_history</b>	(TextBlock) file history text block; can be None
<b>memory</b>	(bool) load measured data option
<b>version</b>	(str) mdf version
<b>chan-nels_db</b>	(dict) used for fast channel access by name; for each name key the value is a list of (group index, channel index) tuples
<b>mas-ters_db</b>	(dict) used for fast master channel access; for each group index key the value is the master channel index

## Methods

<code>add_trigger(group, timestamp[, pre_time, ...])</code>	add trigger to data group
<code>append(signals[, acquisition_info, ...])</code>	Appends a new data group.
<code>close()</code>	if the MDF was created with memory='minimum' and new
<code>get([name, group, index, raster, ...])</code>	Gets channel samples.
<code>get_channel_comment([name, group, index])</code>	Gets channel comment.
<code>get_channel_unit([name, group, index])</code>	Gets channel unit.
<code>get_master(index[, data])</code>	returns master channel samples for given group
<code>info()</code>	get MDF information as a dict
<code>iter_get_triggers()</code>	generator that yields triggers
<code>save([dst, overwrite, compression])</code>	Save MDF to <i>dst</i> .

**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, timestamps=t, unit='+', name='Positive')
>>> s2 = Signal(samples=s2, timestamps=t, unit='-', name='Negative')
>>> s3 = Signal(samples=s3, timestamps=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

**get** (*name=None, group=None, index=None, raster=None, samples\_only=False, data=None*)

Gets channel samples. 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

**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

**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** **MdfError** :

- \* if the channel name is not found
- \* if the group index is out of range
- \* if the channel index is out of range

**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\_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*)

returns master channel samples for given group

**Parameters** **index** : int

group index

**data** : bytes

data block raw bytes; 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=None*, *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

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

## MDF version 3 blocks

The following classes implement different MDF version3 blocks.

### Channel Class

**class** `asammdf.mdf3.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” ( $= N \bmod 8$ ). 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) \* additional\_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

<b>name</b>	(str) full channel name
<b>address</b>	(int) block address inside mdf file
<b>dependencies</b>	(list) list of channel dependencies

## Methods

<code>clear()</code> -> None. Remove all items from D.)	
<code>copy()</code> -> a shallow copy of D)	
<code>fromkeys(\$type, iterable[, value])</code>	Returns a new dict with keys from iterable and values equal to value.
<code>get((k[,d]) -&gt; D[k] if k in D, ...)</code>	
<code>items(...)</code>	
<code>keys(...)</code>	
<code>pop((k[,d]) -&gt; v, ...)</code>	If key is not found, d is returned if given, otherwise Key-Error is raised
<code>popitem()</code> -> (k, v), ...)	2-tuple; but raise KeyError if D is empty.
<code>setdefault((k[,d]) -&gt; D.get(k,d), ...)</code>	

Continued on next page

Table 6.14 – continued from previous page

<code>update((E, ...))</code>	If E is present and has a <code>.keys()</code> method, then does: for k in E: $D[k] = E[k]$
<code>values(...)</code>	

## ChannelConversion Class

**class** `asammdf.mdf3.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 (for-  
mula 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

<b>address</b>	(int) block address inside mdf file
----------------	-------------------------------------

## Methods

clear()	-> None. Remove all items from D.)
copy()	-> a shallow copy of D)
fromkeys(\$type, iterable[, value])	Returns a new dict with keys from iterable and values equal to value.
get((k[,d]) -> D[k] if k in D, ...)	
items(...)	
keys(...)	
pop((k[,d]) -> v, ...)	If key is not found, d is returned if given, otherwise Key-Error is raised
popitem() -> (k, v), ...)	2-tuple; but raise KeyError if D is empty.
setdefault((k[,d]) -> D.get(k,d), ...)	
update((E, ...)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
values(...)	

## ChannelDependency Class

**class** asammdf.mdf3.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 describe 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

<b>address</b>	(int) block address inside mdf file
----------------	-------------------------------------

## Methods

<code>clear()</code> -> None. Remove all items from D.)	
<code>copy()</code> -> a shallow copy of D)	
<code>fromkeys(\$type, iterable[, value])</code>	Returns a new dict with keys from iterable and values equal to value.
<code>get((k[,d]) -&gt; D[k] if k in D, ...)</code>	
<code>items(...)</code>	
<code>keys(...)</code>	
<code>pop((k[,d]) -&gt; v, ...)</code>	If key is not found, d is returned if given, otherwise Key-Error is raised
<code>popitem()</code> -> (k, v), ...)	2-tuple; but raise KeyError if D is empty.
<code>setdefault((k[,d]) -&gt; D.get(k,d), ...)</code>	
<code>update([E, ...])</code>	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
<code>values(...)</code>	

## ChannelExtension Class

**class** `asammdf.mdf3.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

<b>address</b>	(int) block address inside mdf file
----------------	-------------------------------------

## Methods

<code>clear()</code> -> None. Remove all items from D.)	
<code>copy()</code> -> a shallow copy of D)	
<code>fromkeys(\$type, iterable[, value])</code>	Returns a new dict with keys from iterable and values equal to value.
<code>get((k[,d]) -&gt; D[k] if k in D, ...)</code>	
<code>items(...)</code>	
<code>keys(...)</code>	
<code>pop((k[,d]) -&gt; v, ...)</code>	If key is not found, d is returned if given, otherwise KeyError is raised
<code>popitem()</code> -> (k, v), ...)	2-tuple; but raise KeyError if D is empty.
<code>setdefault((k[,d]) -&gt; D.get(k,d), ...)</code>	
<code>update([E, ...])</code>	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
<code>values(...)</code>	

## ChannelGroup Class

**class** `asammdf.mdf3.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

<b>address</b>	(int) block address inside mdf file
----------------	-------------------------------------

## Methods

<code>clear()</code>	-> None. Remove all items from D.)
<code>copy()</code>	-> a shallow copy of D)
<code>fromkeys(\$type, iterable[, value])</code>	Returns a new dict with keys from iterable and values equal to value.
<code>get((k[,d]) -&gt; D[k] if k in D, ...)</code>	
<code>items(...)</code>	
<code>keys(...)</code>	
<code>pop((k[,d]) -&gt; v, ...)</code>	If key is not found, d is returned if given, otherwise Key-Error is raised
<code>popitem() -&gt; (k, v), ...)</code>	2-tuple; but raise KeyError if D is empty.
<code>setdefault((k[,d]) -&gt; D.get(k,d), ...)</code>	
<code>update([E, ...])</code>	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
<code>values(...)</code>	

## DataGroup Class

**class** `asammdf.mdf3.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

<b>address</b>	(int) block address inside mdf file
----------------	-------------------------------------

## Methods

<code>clear()</code> -> None. Remove all items from D.)	
<code>copy()</code> -> a shallow copy of D)	
<code>fromkeys(\$type, iterable[, value])</code>	Returns a new dict with keys from iterable and values equal to value.
<code>get((k[,d]) -&gt; D[k] if k in D, ...)</code>	
<code>items(...)</code>	
<code>keys(...)</code>	
<code>pop((k[,d]) -&gt; v, ...)</code>	If key is not found, d is returned if given, otherwise Key-Error is raised
<code>popitem()</code> -> (k, v), ...)	2-tuple; but raise KeyError if D is empty.
<code>setdefault((k[,d]) -&gt; D.get(k,d), ...)</code>	
<code>update([E, ...])</code>	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
<code>values(...)</code>	

## FileIdentificationBlock Class

**class** `asammdf.mdf3.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

<b>address</b>	(int) block address inside mdf file; should be 0 always
----------------	---

## Methods

<code>clear()</code> -> None. Remove all items from D.)	
<code>copy()</code> -> a shallow copy of D)	
<code>fromkeys(\$type, iterable[, value])</code>	Returns a new dict with keys from iterable and values equal to value.
<code>get((k[,d]) -&gt; D[k] if k in D, ...)</code>	
<code>items(...)</code>	
<code>keys(...)</code>	
<code>pop((k[,d]) -&gt; v, ...)</code>	If key is not found, d is returned if given, otherwise Key-Error is raised
<code>popitem()</code> -> (k, v), ...)	2-tuple; but raise KeyError if D is empty.
<code>setdefault((k[,d]) -&gt; D.get(k,d), ...)</code>	
<code>update([E, ...])</code>	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
<code>values(...)</code>	

## HeaderBlock Class

**class** `asammdf.mdf3.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

<b>address</b>	(int) block address inside mdf file; should be 64 always
----------------	--

## Methods

<code>clear()</code> -> None. Remove all items from D.)	
<code>copy()</code> -> a shallow copy of D)	
<code>fromkeys(\$type, iterable[, value])</code>	Returns a new dict with keys from iterable and values equal to value.
<code>get((k[,d]) -&gt; D[k] if k in D, ...)</code>	
<code>items(...)</code>	
<code>keys(...)</code>	
<code>pop((k[,d]) -&gt; v, ...)</code>	If key is not found, d is returned if given, otherwise Key-Error is raised
<code>popitem()</code> -> (k, v), ...)	2-tuple; but raise KeyError if D is empty.
<code>setdefault((k[,d]) -&gt; D.get(k,d), ...)</code>	

Continued on next page

Table 6.21 – continued from previous page

---

<code>update((E, ...))</code>	If E is present and has a <code>.keys()</code> method, then does: for k in E: <code>D[k] = E[k]</code>
<code>values(...)</code>	

---

## ProgramBlock Class

## SampleReduction Class

## TextBlock Class

**class** `asammdf.mdf3.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

<b>address</b>	(int) block address inside mdf file
<b>text_str</b>	(str) text data as unicode string



## Methods

<code>clear()</code>	-> None. Remove all items from D.)
<code>copy()</code>	-> a shallow copy of D)
<code>fromkeys(\$type, iterable[, value])</code>	Returns a new dict with keys from iterable and values equal to value.
<code>get((k[,d]) -&gt; D[k] if k in D, ...)</code>	
<code>items(...)</code>	
<code>keys(...)</code>	
<code>pop((k[,d]) -&gt; v, ...)</code>	If key is not found, d is returned if given, otherwise KeyError is raised
<code>popitem()</code>	-> (k, v), ...)
<code>setdefault((k[,d]) -&gt; D.get(k,d), ...)</code>	2-tuple; but raise KeyError if D is empty.
<code>update([E, ...])</code>	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
<code>values(...)</code>	

## TriggerBlock Class

**class** `asammdf.mdf3.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 “TX”
- *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

<b>address</b>	(int) block address inside mdf file
----------------	-------------------------------------

## Methods

<code>clear()</code>	-> None. Remove all items from D.)
<code>copy()</code>	-> a shallow copy of D)
<code>fromkeys(\$type, iterable[, value])</code>	Returns a new dict with keys from iterable and values equal to value.
<code>get((k[,d]) -&gt; D[k] if k in D, ...)</code>	
<code>items(...)</code>	
<code>keys(...)</code>	
<code>pop((k[,d]) -&gt; v, ...)</code>	If key is not found, d is returned if given, otherwise Key-Error is raised
<code>popitem()</code>	-> (k, v), ...)
<code>setdefault((k[,d]) -&gt; D.get(k,d), ...)</code>	2-tuple; but raise KeyError if D is empty.
<code>update([E, ...])</code>	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
<code>values(...)</code>	

## MDF4

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

The *header* attribute 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` : dictionary containing TextBlock objects used throughout the mdf
  - `channels` : list of dictionaries that contain TextBlock objects related to each channel
    - \* `name_addr` : channel name
    - \* `comment_addr` : channel comment
  - `channel group` : list of dictionaries that contain TextBlock objects related 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 contain TextBlock objects related to channel conversions
    - \* `name_addr` : conversions name
    - \* `unit_addr` : channel unit\_addr

- \* `comment_addr` : converison comment
- \* `formula_addr` : formula text; only valid for algebraic conversions
- `sources` : list of dictionaries that contain TextBlock objects 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` attribute 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` attribute 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.

## API

**class** `asammdf.mdf4.MDF4` (*name=None*, *memory='full'*, *version='4.10'*)

If the *name* exist it will be memoryed 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 memoryed in RAM
- if *low* the channel data is read from disk on request, and the

metadata is memoryed into RAM \* if *minimum* only minimal data is memoryed into RAM

**version** : string

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

## Attributes

<b>name</b>	(string) mdf file name
<b>groups</b>	(list) list of data groups
<b>header</b>	(HeaderBlock) mdf file header
<b>file_history</b>	(list) list of (FileHistory, TextBlock) pairs
<b>comment</b>	(TextBlock) mdf file comment
<b>identification</b>	(FileIdentificationBlock) mdf file start block
<b>memory</b>	(str) memory optimization option
<b>version</b>	(str) mdf version
<b>channels_db</b>	(dict) used for fast channel access by name; for each name key the value is a list of (group index, channel index) tuples
<b>masters_db</b>	(dict) used for fast master channel access; for each group index key the value is the master channel index

## Methods

<code>append(signals[, source_info, common_timebase])</code>	Appends a new data group.
<code>attach(data[, file_name, comment, ...])</code>	attach embedded attachment as application/octet-stream
<code>close()</code>	if the MDF was created with memory=False and new
<code>extract_attachment(index)</code>	extract attachemnt <i>index</i> data. If it is an embedded attachment,
<code>get([name, group, index, raster, ...])</code>	Gets channel samples.
<code>get_channel_comment([name, group, index])</code>	Gets channel comment.
<code>get_channel_unit([name, group, index])</code>	Gets channel unit.
<code>get_master(index[, data])</code>	returns master channel samples for given group
<code>info()</code>	get MDF information as a dict
<code>save([dst, overwrite, compression])</code>	Save MDF to <i>dst</i> .

**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, timestamps=t, unit='+', name='Positive')
>>> s2 = Signal(samples=s2, timestamps=t, unit='-', name='Negative')
>>> s3 = Signal(samples=s3, timestamps=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')
```

**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

**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

**extract\_attachment** (*index*)

extract attachemnt *index* data. If it is an embedded attachment, then this method creates the new file according to the attachemnt file name information

**Parameters** *index* : int

attachment index

**Returns** *data* : bytes | str

attachment data

**get** (*name=None*, *group=None*, *index=None*, *raster=None*, *samples\_only=False*, *data=None*)

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

**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 *BYTEARRAY*, *CANOPENDATE*, *CANOPENTIME* \* numpy array for all the rest

**Raises** *MdfError* :

- \* if the channel name is not found
- \* if the group index is out of range
- \* if the channel index is out of range

**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\_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*)

returns master channel samples for given group

**Parameters** *index* : int

group index

**data** : bytes

data block raw bytes; default *None*

**Returns** *t* : numpy.array

master channel samples

**info** ()

get MDF information as a dict

## Examples

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

**save** (*dst=""*, *overwrite=None*, *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)

## MDF version 4 blocks

The following classes implement different MDF version3 blocks.

### AttachmentBlock Class

**class** `asammdf.mdf4.AttachmentBlock` (*\*\*kargs*)

ATBLOCK class

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

#### Methods

<code>clear()</code> -> None. Remove all items from D.)	
<code>copy()</code> -> a shallow copy of D)	
<code>extract()</code>	
<code>fromkeys(\$type, iterable[, value])</code>	Returns a new dict with keys from iterable and values equal to value.
<code>get((k[,d]) -&gt; D[k] if k in D, ...)</code>	
<code>items(...)</code>	
<code>keys(...)</code>	
<code>pop((k[,d]) -&gt; v, ...)</code>	If key is not found, d is returned if given, otherwise KeyError is raised
<code>popitem() -&gt; (k, v), ...)</code>	2-tuple; but raise KeyError if D is empty.
<code>setdefault((k[,d]) -&gt; D.get(k,d), ...)</code>	
<code>update((E, ...)</code>	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
<code>values(...)</code>	

### Channel Class

**class** `asammdf.mdf4.Channel` (*\*\*kargs*)

CNBLOCK class

#### Methods

<code>clear()</code> -> None. Remove all items from D.)	
<code>copy()</code> -> a shallow copy of D)	

Continued on next page



Table 6.26 – continued from previous page

<code>fromkeys(\$type, iterable[, value])</code>	Returns a new dict with keys from iterable and values equal to value.
<code>get((k[,d]) -&gt; D[k] if k in D, ...)</code>	
<code>items(...)</code>	
<code>keys(...)</code>	
<code>pop((k[,d]) -&gt; v, ...)</code>	If key is not found, d is returned if given, otherwise Key-Error is raised
<code>popitem() -&gt; (k, v), ...)</code>	2-tuple; but raise KeyError if D is empty.
<code>setdefault((k[,d]) -&gt; D.get(k,d), ...)</code>	
<code>update((E, ...)</code>	If E is present and has a <code>.keys()</code> method, then does: for k in E: <code>D[k] = E[k]</code>
<code>values(...)</code>	

**ChannelConversion Class**

**class** `asammdf.mdf4.ChannelConversion` (*\*\*kargs*)  
CCBLOCK class

**Methods**

<code>clear() -&gt; None. Remove all items from D.)</code>	
<code>copy() -&gt; a shallow copy of D)</code>	
<code>fromkeys(\$type, iterable[, value])</code>	Returns a new dict with keys from iterable and values equal to value.
<code>get((k[,d]) -&gt; D[k] if k in D, ...)</code>	
<code>items(...)</code>	
<code>keys(...)</code>	
<code>pop((k[,d]) -&gt; v, ...)</code>	If key is not found, d is returned if given, otherwise Key-Error is raised
<code>popitem() -&gt; (k, v), ...)</code>	2-tuple; but raise KeyError if D is empty.
<code>setdefault((k[,d]) -&gt; D.get(k,d), ...)</code>	
<code>update((E, ...)</code>	If E is present and has a <code>.keys()</code> method, then does: for k in E: <code>D[k] = E[k]</code>
<code>values(...)</code>	

**ChannelGroup Class**

**class** `asammdf.mdf4.ChannelGroup` (*\*\*kargs*)  
CGBLOCK class

**Methods**

<code>clear() -&gt; None. Remove all items from D.)</code>	
<code>copy() -&gt; a shallow copy of D)</code>	
<code>fromkeys(\$type, iterable[, value])</code>	Returns a new dict with keys from iterable and values equal to value.
<code>get((k[,d]) -&gt; D[k] if k in D, ...)</code>	

Continued on next page

Table 6.28 – continued from previous page

items(...)	
keys(...)	
pop((k[,d]) -> v, ...)	If key is not found, d is returned if given, otherwise Key-Error is raised
popitem() -> (k, v), ...)	2-tuple; but raise KeyError if D is empty.
setdefault((k[,d]) -> D.get(k,d), ...)	
update([E, ...])	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
values(...)	

## DataGroup Class

```
class asammdf.mdf4.DataGroup (**kargs)
    DGBLOCK class
```

### Methods

clear() -> None. Remove all items from D.)	
copy() -> a shallow copy of D)	
fromkeys(\$type, iterable[, value])	Returns a new dict with keys from iterable and values equal to value.
get((k[,d]) -> D[k] if k in D, ...)	
items(...)	
keys(...)	
pop((k[,d]) -> v, ...)	If key is not found, d is returned if given, otherwise Key-Error is raised
popitem() -> (k, v), ...)	2-tuple; but raise KeyError if D is empty.
setdefault((k[,d]) -> D.get(k,d), ...)	
update([E, ...])	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
values(...)	

## DataList Class

```
class asammdf.mdf4.DataList (**kargs)
    DLBLOCK class
```

### Methods

clear() -> None. Remove all items from D.)	
copy() -> a shallow copy of D)	
fromkeys(\$type, iterable[, value])	Returns a new dict with keys from iterable and values equal to value.
get((k[,d]) -> D[k] if k in D, ...)	
items(...)	
keys(...)	

Continued on next page

Table 6.30 – continued from previous page

<code>pop((k[,d]) -&gt; v, ...)</code>	If key is not found, d is returned if given, otherwise Key-Error is raised
<code>popitem() -&gt; (k, v), ...)</code>	2-tuple; but raise KeyError if D is empty.
<code>setdefault((k[,d]) -&gt; D.get(k,d), ...)</code>	
<code>update((E, ...)</code>	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
<code>values(...)</code>	

**DataBlock Class**

**class** `asammdf.mdf4.DataBlock` (*\*\*kargs*)  
DTBLOCK class

**Parameters** `address` : int

DTBLOCK address inside the file

**stream** : int

file handle

**Methods**

<code>clear() -&gt; None</code>	Remove all items from D.)
<code>copy() -&gt; a shallow copy of D)</code>	
<code>fromkeys(\$type, iterable[, value])</code>	Returns a new dict with keys from iterable and values equal to value.
<code>get((k[,d]) -&gt; D[k] if k in D, ...)</code>	
<code>items(...)</code>	
<code>keys(...)</code>	
<code>pop((k[,d]) -&gt; v, ...)</code>	If key is not found, d is returned if given, otherwise Key-Error is raised
<code>popitem() -&gt; (k, v), ...)</code>	2-tuple; but raise KeyError if D is empty.
<code>setdefault((k[,d]) -&gt; D.get(k,d), ...)</code>	
<code>update((E, ...)</code>	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
<code>values(...)</code>	

**FileIdentificationBlock Class**

**class** `asammdf.mdf4.FileIdentificationBlock` (*\*\*kargs*)  
IDBLOCK class

**Methods**

<code>clear() -&gt; None</code>	Remove all items from D.)
<code>copy() -&gt; a shallow copy of D)</code>	
<code>fromkeys(\$type, iterable[, value])</code>	Returns a new dict with keys from iterable and values equal to value.

Continued on next page

Table 6.32 – continued from previous page

get((k[,d]) -> D[k] if k in D, ...)	
items(...)	
keys(...)	
pop((k[,d]) -> v, ...)	If key is not found, d is returned if given, otherwise Key-Error is raised
popitem() -> (k, v), ...)	2-tuple; but raise KeyError if D is empty.
setdefault((k[,d]) -> D.get(k,d), ...)	
update([E, ...])	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
values(...)	

## HeaderBlock Class

```
class asammdf.mdf4.HeaderBlock (**kargs)
    HDBLOCK class
```

### Methods

clear() -> None. Remove all items from D.)	
copy() -> a shallow copy of D)	
fromkeys(\$type, iterable[, value])	Returns a new dict with keys from iterable and values equal to value.
get((k[,d]) -> D[k] if k in D, ...)	
items(...)	
keys(...)	
pop((k[,d]) -> v, ...)	If key is not found, d is returned if given, otherwise Key-Error is raised
popitem() -> (k, v), ...)	2-tuple; but raise KeyError if D is empty.
setdefault((k[,d]) -> D.get(k,d), ...)	
update([E, ...])	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
values(...)	

## SourceInformation Class

```
class asammdf.mdf4.SourceInformation (**kargs)
    SIBLOCK class
```

### Methods

clear() -> None. Remove all items from D.)	
copy() -> a shallow copy of D)	
fromkeys(\$type, iterable[, value])	Returns a new dict with keys from iterable and values equal to value.
get((k[,d]) -> D[k] if k in D, ...)	
items(...)	
keys(...)	

Continued on next page

Table 6.34 – continued from previous page

<code>pop((k[,d]) -&gt; v, ...)</code>	If key is not found, d is returned if given, otherwise Key-Error is raised
<code>popitem() -&gt; (k, v), ...)</code>	2-tuple; but raise KeyError if D is empty.
<code>setdefault((k[,d]) -&gt; D.get(k,d), ...)</code>	
<code>update((E, ...)</code>	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
<code>values(...)</code>	

## FileHistory Class

**class** `asammdf.mdf4.FileHistory` (*\*\*kargs*)  
 FHBLOCK class

### Methods

<code>clear() -&gt; None.</code>	Remove all items from D.)
<code>copy() -&gt; a shallow copy of D)</code>	
<code>fromkeys(\$type, iterable[, value])</code>	Returns a new dict with keys from iterable and values equal to value.
<code>get((k[,d]) -&gt; D[k] if k in D, ...)</code>	
<code>items(...)</code>	
<code>keys(...)</code>	
<code>pop((k[,d]) -&gt; v, ...)</code>	If key is not found, d is returned if given, otherwise Key-Error is raised
<code>popitem() -&gt; (k, v), ...)</code>	2-tuple; but raise KeyError if D is empty.
<code>setdefault((k[,d]) -&gt; D.get(k,d), ...)</code>	
<code>update((E, ...)</code>	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
<code>values(...)</code>	

## TextBlock Class

**class** `asammdf.mdf4.TextBlock` (*\*\*kargs*)  
 common TXBLOCK and MDBLOCK class

### Methods

<code>clear() -&gt; None.</code>	Remove all items from D.)
<code>copy() -&gt; a shallow copy of D)</code>	
<code>fromkeys(\$type, iterable[, value])</code>	Returns a new dict with keys from iterable and values equal to value.
<code>get((k[,d]) -&gt; D[k] if k in D, ...)</code>	
<code>items(...)</code>	
<code>keys(...)</code>	
<code>pop((k[,d]) -&gt; v, ...)</code>	If key is not found, d is returned if given, otherwise Key-Error is raised
<code>popitem() -&gt; (k, v), ...)</code>	2-tuple; but raise KeyError if D is empty.

Continued on next page

Table 6.36 – continued from previous page

<code>setdefault((k,d) -&gt; D.get(k,d), ...)</code>	
<code>update([E, ...])</code>	If E is present and has a <code>.keys()</code> method, then does: for k in E: <code>D[k] = E[k]</code>
<code>values(...)</code>	

## 6.2.2 Notes about *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.

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

Advantages

- best performance

Disadvantages

- higher RAM usage, there is the chance the file will exceed available RAM

Use case

- when data fits inside the system RAM

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

Advantages

- lower RAM usage than *memory=full*
- can handle files that do not fit in the available physical memory

Disadvantages

- slow performance for getting 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 *default* data exceeds available RAM
- it is advised to avoid getting individual channels when using this option.

Instead you can get performance close to *memory='full'* if you use the *select* method with the list of target channels.

---

**Note:** See benchmarks for the effects of using the flag

---

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

## Advantages

- lowest RAM usage
- can handle files that do not fit in the available physical memory
- 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

---

## 6.3 Signal

**class** `asammdf.signal.Signal` (*samples=None, timestamps=None, unit="", name="", info=None, comment=""*)

The Signal represents a signal described by it's samples and timestamps. It can do arithmetic operations against other Signal or numeric type. The operations are computed in respect to the timestamps (time correct). The integer signals are not interpolated, instead the last value relative to the current timestamp is used. *samples*, *timestamps* 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

**info** : dict

dict that contains extra information about the signal , default *None*

**comment** : str

signal comment, default ""

### Methods

<code>astype(np_type)</code>	returns new <i>Signal</i> with samples of dtype <i>np_type</i>
<code>cut([start, stop])</code>	Cuts the signal according to the <i>start</i> and <i>stop</i> values, by using the insertion indexes in the signal's <i>time</i> axis.
<code>extend(other)</code>	extend signal with samples from another signal

Continued on next page

Table 6.37 – continued from previous page

<code>interp(new_timestamps)</code>	returns a new <i>Signal</i> interpolated using the <i>new_timestamps</i>
<code>plot()</code>	plot <i>Signal</i> samples

**astype** (*np\_type*)

returns new *Signal* with samples of dtype *np\_type*

**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*

**interp** (*new\_timestamps*)

returns a new *Signal* interpolated using the *new\_timestamps*

**plot** ()

plot *Signal* samples

## 6.4 Examples

### 6.4.1 Working with MDF

```
from __future__ import print_function, division
from asammdf import MDF, Signal, configure
import numpy as np

# configure asammdf to optimize disk space usage
configure(integer_compacting=True)
# configure asammdf to split data blocks on 10KB blocks
configure(split_data_blocks=True, split_threshold=10*1024)

# create 3 Signal objects
```



```

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

# uint8
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]
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 lower 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)

```

## 6.4.2 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

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

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

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

# 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
print(phi.shape, samples.shape)
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()
```



*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  $\geq 3.6$ .

### 7.1 Intro

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.

*asammdf* 2.8.0 was compared against *mdfreader* 2.7.2 (latest versions from PyPI). *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 categories are file open, file save and extracting the data for all channels inside the file(36424 calls). For each category two aspect were noted: elapsed time and peak RAM usage.

### 7.2 Dependencies

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

- psutil
- mdfreader

### 7.3 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 )

## 7.4 x64 Python results

### Benchmark environment

- 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 18:41:36) [MSC v.1900 64 bit (AMD64)]
- Windows-10-10.0.14393-SP0
- Intel64 Family 6 Model 94 Stepping 3, GenuineIntel
- 16GB 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
- compression bcolz 6 = mdfreader mdf object created with compression=6
- noDataLoading = mdfreader mdf object read with noDataLoading=True

### Files used for benchmark:

- 183 groups
- 36424 channels

### 7.4.1 Raw data

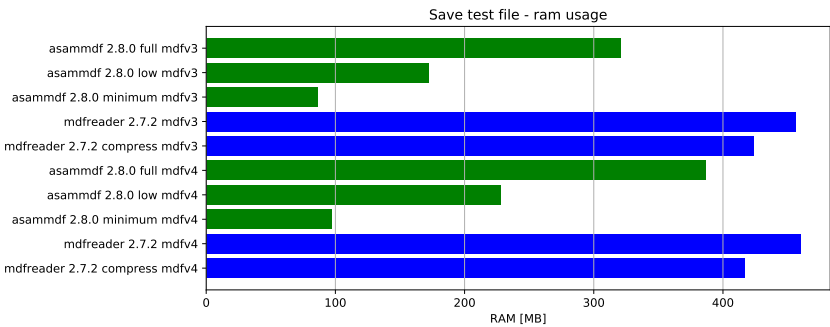
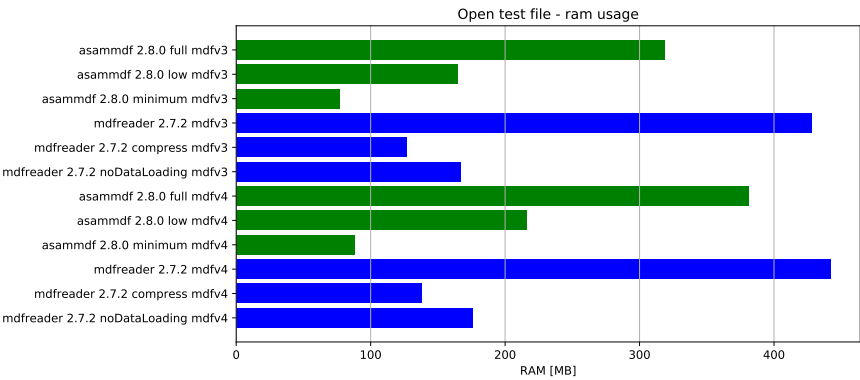
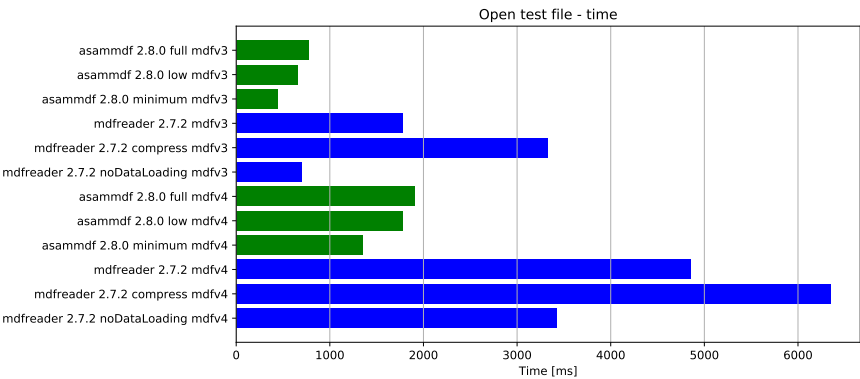
Open file	Time [ms]	RAM [MB]
asammdf 2.8.0 full mdfv3	772	319
asammdf 2.8.0 low mdfv3	656	165
asammdf 2.8.0 minimum mdfv3	441	77
mdfreader 2.7.2 mdfv3	1783	428
mdfreader 2.7.2 compress mdfv3	3330	127
mdfreader 2.7.2 noDataLoading mdfv3	699	167
asammdf 2.8.0 full mdfv4	1903	381
asammdf 2.8.0 low mdfv4	1783	216
asammdf 2.8.0 minimum mdfv4	1348	88
mdfreader 2.7.2 mdfv4	4849	442
mdfreader 2.7.2 compress mdfv4	6347	138
mdfreader 2.7.2 noDataLoading mdfv4	3425	176

Save file	Time [ms]	RAM [MB]
asammdf 2.8.0 full mdfv3	359	321
asammdf 2.8.0 low mdfv3	415	172
asammdf 2.8.0 minimum mdfv3	993	86
mdfreader 2.7.2 mdfv3	8402	456
mdfreader 2.7.2 compress mdfv3	8364	424
asammdf 2.8.0 full mdfv4	497	387
asammdf 2.8.0 low mdfv4	507	228
asammdf 2.8.0 minimum mdfv4	2179	97
mdfreader 2.7.2 mdfv4	7958	460
mdfreader 2.7.2 compress mdfv4	8170	417

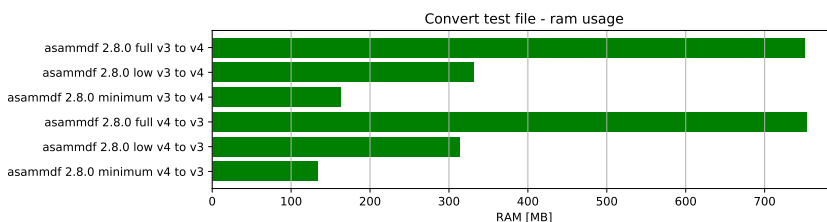
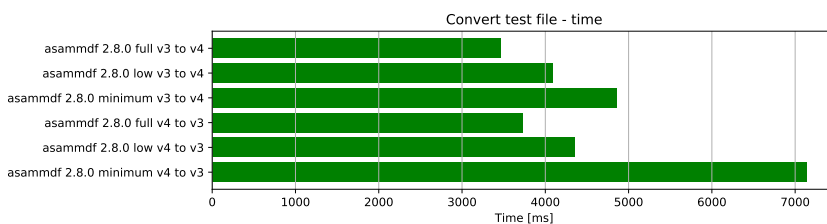
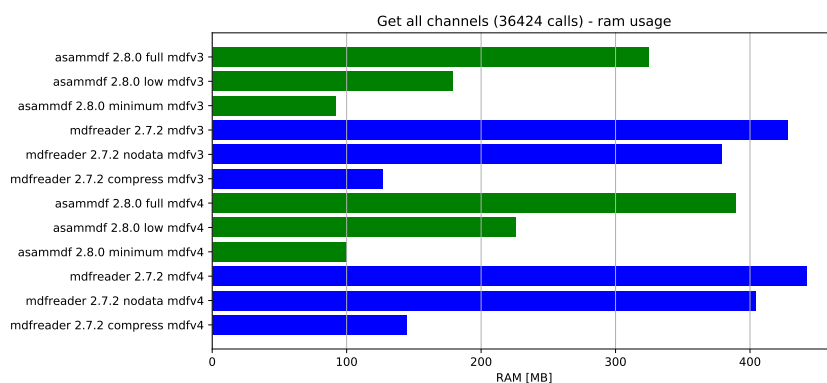
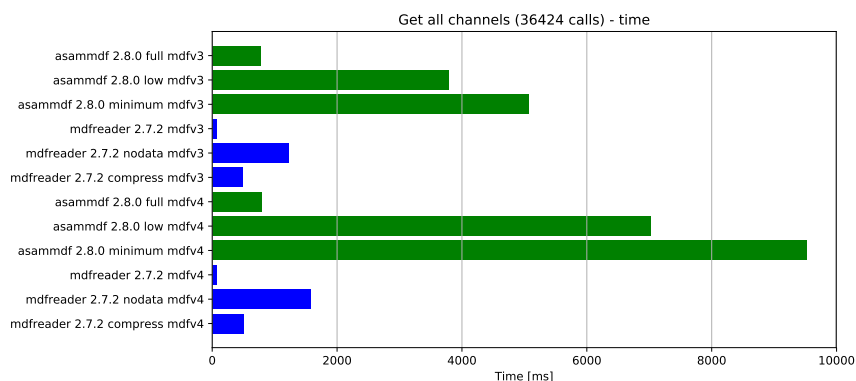
Get all channels (36424 calls)	Time [ms]	RAM [MB]
asammdf 2.8.0 full mdfv3	772	325
asammdf 2.8.0 low mdfv3	3784	179
asammdf 2.8.0 minimum mdfv3	5076	92
mdfreader 2.7.2 mdfv3	65	428
mdfreader 2.7.2 nodata mdfv3	1231	379
mdfreader 2.7.2 compress mdfv3	487	127
asammdf 2.8.0 full mdfv4	800	389
asammdf 2.8.0 low mdfv4	7025	226
asammdf 2.8.0 minimum mdfv4	9518	100
mdfreader 2.7.2 mdfv4	71	442
mdfreader 2.7.2 nodata mdfv4	1575	404
mdfreader 2.7.2 compress mdfv4	508	145

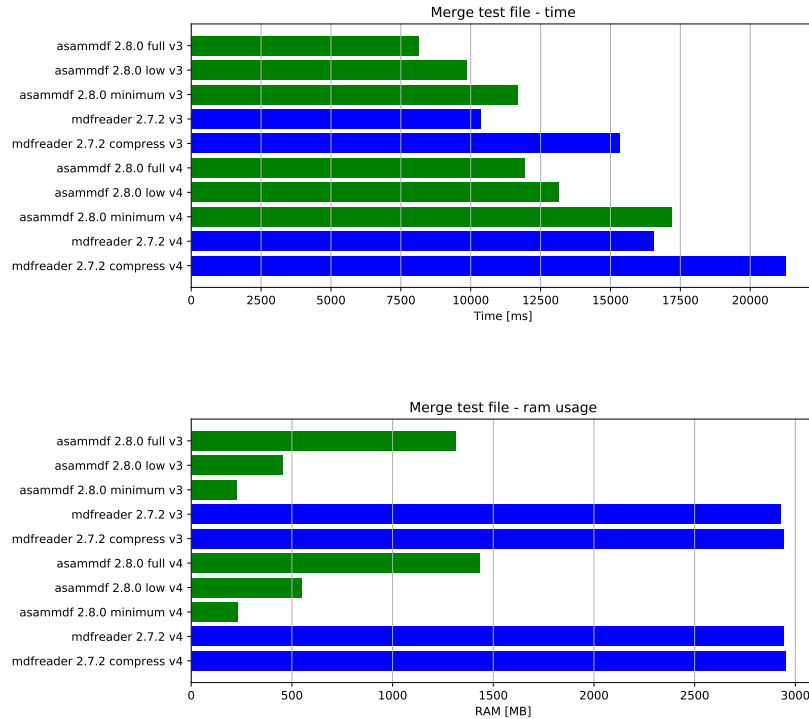
Convert file	Time [ms]	RAM [MB]
asammdf 2.8.0 full v3 to v4	3461	751
asammdf 2.8.0 low v3 to v4	4092	331
asammdf 2.8.0 minimum v3 to v4	4852	163
asammdf 2.8.0 full v4 to v3	3732	753
asammdf 2.8.0 low v4 to v3	4348	313
asammdf 2.8.0 minimum v4 to v3	7136	134

Merge files	Time [ms]	RAM [MB]
asammdf 2.8.0 full v3	8152	1312
asammdf 2.8.0 low v3	9839	456
asammdf 2.8.0 minimum v3	11694	228
mdfreader 2.7.2 v3	10352	2927
mdfreader 2.7.2 compress v3	15314	2940
asammdf 2.8.0 full v4	11938	1434
asammdf 2.8.0 low v4	13154	549
asammdf 2.8.0 minimum v4	17188	229
mdfreader 2.7.2 v4	16536	2941
mdfreader 2.7.2 compress v4	21261	2951









## 7.4.2 Graphical results

## 7.5 x86 Python results

### Benchmark environment

- 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)]
- Windows-10-10.0.14393-SP0
- Intel64 Family 6 Model 94 Stepping 3, GenuineIntel
- 16GB 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 mdx object created with compression=blosc
- noDataLoading = mdfreader mdx object read with noDataLoading=True

### Files used for benchmark:

- 183 groups
- 36424 channels

### 7.5.1 Raw data

Open file	Time [ms]	RAM [MB]
asammdf 2.8.0 full mdv3	918	264
asammdf 2.8.0 low mdv3	898	110
asammdf 2.8.0 minimum mdv3	577	56
mdfreader 2.7.2 mdv3	2462	395
mdfreader 2.7.2 compress mdv3	4174	97
mdfreader 2.7.2 noDataLoading mdv3	911	105
asammdf 2.8.0 full mdv4	2644	302
asammdf 2.8.0 low mdv4	2269	137
asammdf 2.8.0 minimum mdv4	1883	62
mdfreader 2.7.2 mdv4	5869	403
mdfreader 2.7.2 compress mdv4	7367	101
mdfreader 2.7.2 noDataLoading mdv4	3897	110

Save file	Time [ms]	RAM [MB]
asammdf 2.8.0 full mdv3	452	267
asammdf 2.8.0 low mdv3	495	118
asammdf 2.8.0 minimum mdv3	1206	62
mdfreader 2.7.2 mdv3	9258	415
asammdf 2.8.0 full mdv4	642	307
asammdf 2.8.0 low mdv4	693	146
asammdf 2.8.0 minimum mdv4	2642	71
mdfreader 2.7.2 mdv4	8548	422

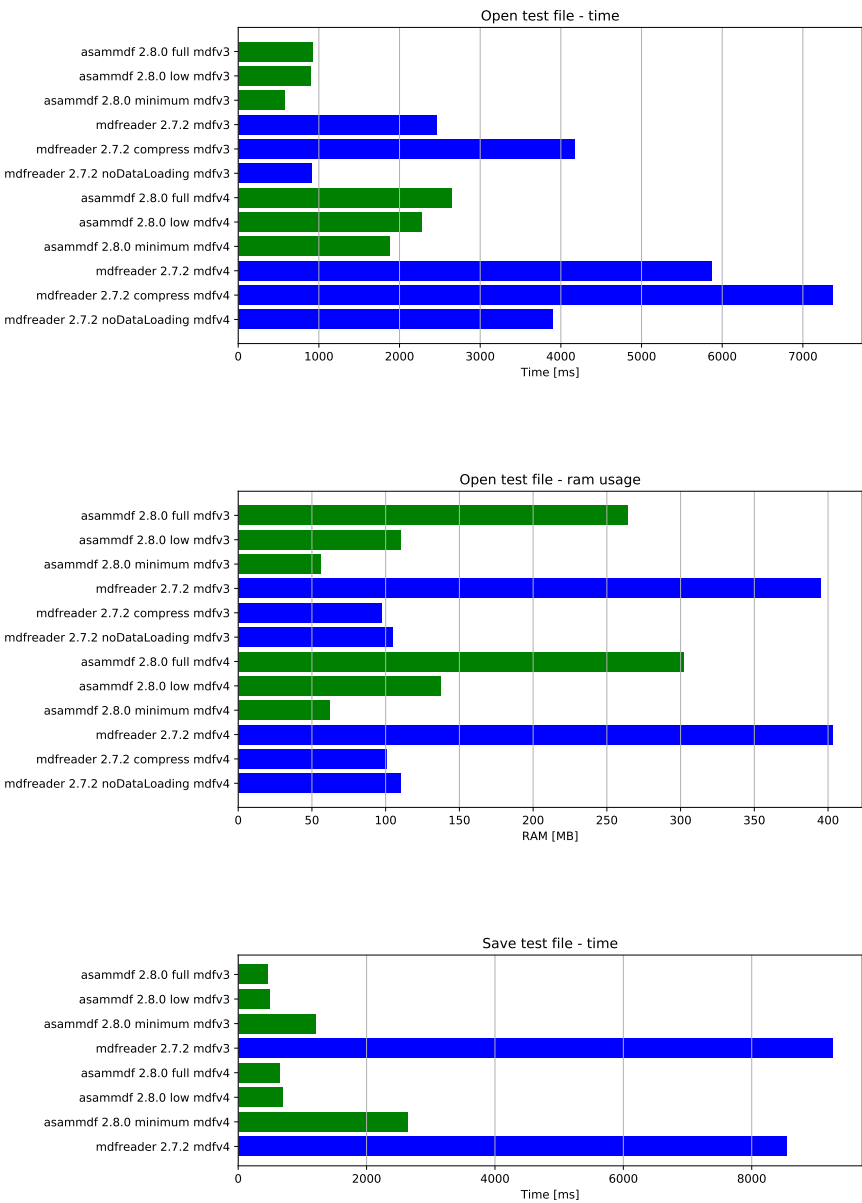
Get all channels (36424 calls)	Time [ms]	RAM [MB]
asammdf 2.8.0 full mdv3	889	268
asammdf 2.8.0 low mdv3	12707	120
asammdf 2.8.0 minimum mdv3	13644	66
mdfreader 2.7.2 mdv3	80	395
mdfreader 2.7.2 nodata mdv3	1413	310
mdfreader 2.7.2 compress mdv3	529	97
asammdf 2.8.0 full mdv4	968	307
asammdf 2.8.0 low mdv4	14475	144
asammdf 2.8.0 minimum mdv4	17057	69
mdfreader 2.7.2 mdv4	72	403
mdfreader 2.7.2 nodata mdv4	1806	325
mdfreader 2.7.2 compress mdv4	562	107

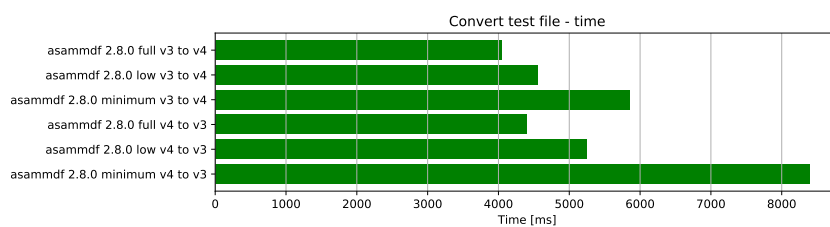
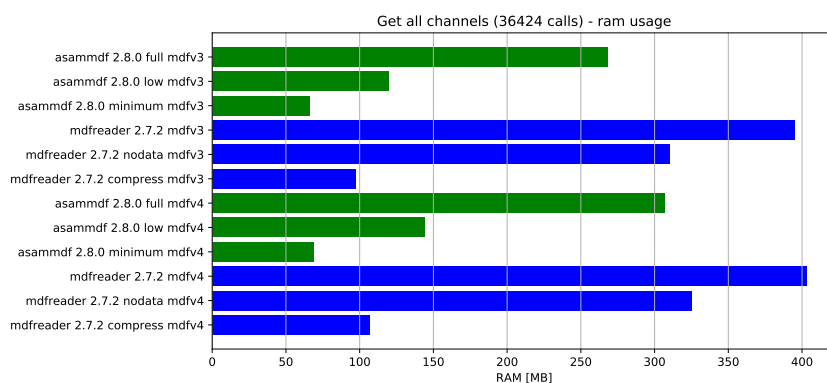
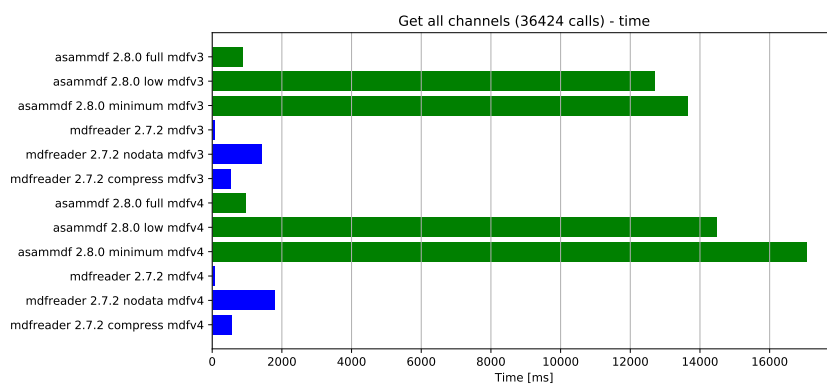
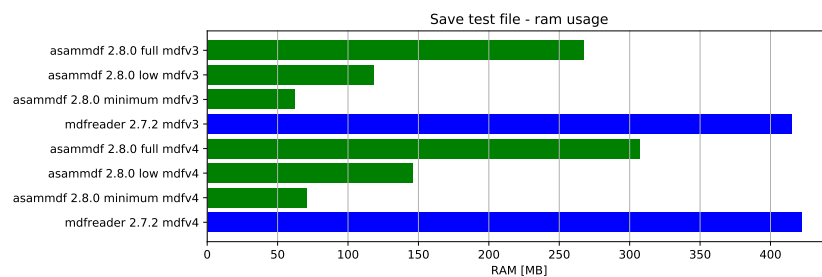
Convert file	Time [ms]	RAM [MB]
asammdf 2.8.0 full v3 to v4	4048	642
asammdf 2.8.0 low v3 to v4	4551	219
asammdf 2.8.0 minimum v3 to v4	5847	121
asammdf 2.8.0 full v4 to v3	4394	639
asammdf 2.8.0 low v4 to v3	5239	198
asammdf 2.8.0 minimum v4 to v3	8392	98

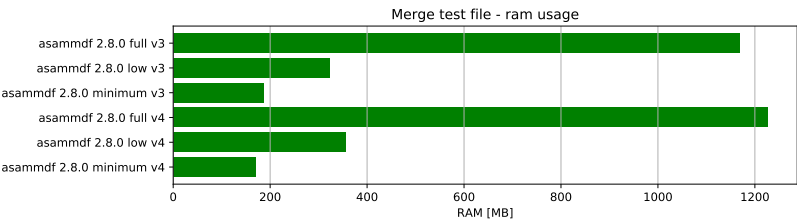
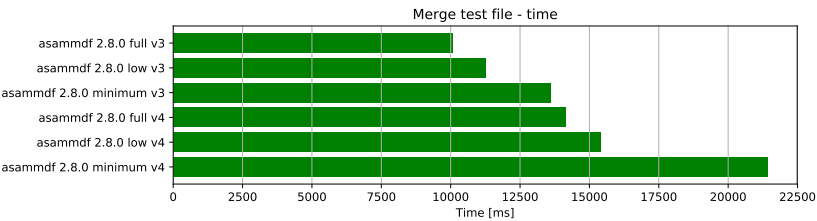
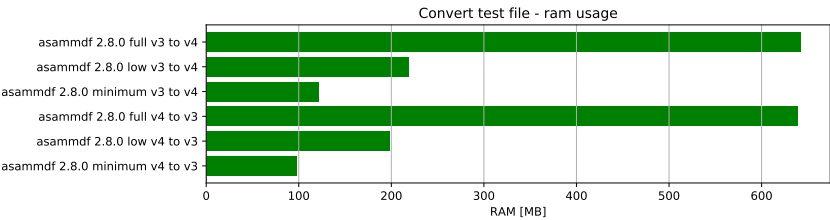
Merge files	Time [ms]	RAM [MB]
asammdf 2.8.0 full v3	10061	1168
asammdf 2.8.0 low v3	11245	323
asammdf 2.8.0 minimum v3	13618	186
asammdf 2.8.0 full v4	14144	1226
asammdf 2.8.0 low v4	15410	355
asammdf 2.8.0 minimum v4	21417	170

- mdfreader got a MemoryError

7.5.2 Graphical results







## CHAPTER 8

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`





## A

`add_trigger()` (asammdf.mdf3.MDF3 method), 18, 37  
`append()` (asammdf.mdf3.MDF3 method), 19, 37  
`append()` (asammdf.mdf4.MDF4 method), 56  
`astype()` (asammdf.signal.Signal method), 68  
`attach()` (asammdf.mdf4.MDF4 method), 56  
`AttachmentBlock` (class in asammdf.mdf4), 60

## C

`Channel` (class in asammdf.mdf2), 22  
`Channel` (class in asammdf.mdf3), 41  
`Channel` (class in asammdf.mdf4), 60  
`ChannelConversion` (class in asammdf.mdf2), 24  
`ChannelConversion` (class in asammdf.mdf3), 43  
`ChannelConversion` (class in asammdf.mdf4), 61  
`ChannelDependency` (class in asammdf.mdf2), 26  
`ChannelDependency` (class in asammdf.mdf3), 44  
`ChannelExtension` (class in asammdf.mdf2), 27  
`ChannelExtension` (class in asammdf.mdf3), 46  
`ChannelGroup` (class in asammdf.mdf2), 28  
`ChannelGroup` (class in asammdf.mdf3), 47  
`ChannelGroup` (class in asammdf.mdf4), 61  
`close()` (asammdf.mdf3.MDF3 method), 19, 38  
`close()` (asammdf.mdf4.MDF4 method), 57  
`configure()` (in module asammdf), 13  
`convert()` (asammdf.mdf.MDF method), 14  
`cut()` (asammdf.mdf.MDF method), 15  
`cut()` (asammdf.signal.Signal method), 68

## D

`DataBlock` (class in asammdf.mdf4), 63  
`DataGroup` (class in asammdf.mdf2), 29  
`DataGroup` (class in asammdf.mdf3), 48  
`DataGroup` (class in asammdf.mdf4), 62  
`DataList` (class in asammdf.mdf4), 62

## E

`export()` (asammdf.mdf.MDF method), 15  
`extend()` (asammdf.signal.Signal method), 68

`extract_attachment()` (asammdf.mdf4.MDF4 method), 57

## F

`FileHistory` (class in asammdf.mdf4), 65  
`FileIdentificationBlock` (class in asammdf.mdf2), 31  
`FileIdentificationBlock` (class in asammdf.mdf3), 49  
`FileIdentificationBlock` (class in asammdf.mdf4), 63  
`filter()` (asammdf.mdf.MDF method), 15

## G

`get()` (asammdf.mdf3.MDF3 method), 19, 38  
`get()` (asammdf.mdf4.MDF4 method), 57  
`get_channel_comment()` (asammdf.mdf3.MDF3 method), 20, 39  
`get_channel_comment()` (asammdf.mdf4.MDF4 method), 58  
`get_channel_unit()` (asammdf.mdf3.MDF3 method), 21, 39  
`get_channel_unit()` (asammdf.mdf4.MDF4 method), 58  
`get_master()` (asammdf.mdf3.MDF3 method), 21, 40  
`get_master()` (asammdf.mdf4.MDF4 method), 59

## H

`HeaderBlock` (class in asammdf.mdf2), 32  
`HeaderBlock` (class in asammdf.mdf3), 50  
`HeaderBlock` (class in asammdf.mdf4), 64

## I

`info()` (asammdf.mdf3.MDF3 method), 21, 40  
`info()` (asammdf.mdf4.MDF4 method), 59  
`interp()` (asammdf.signal.Signal method), 68  
`iter_get_triggers()` (asammdf.mdf3.MDF3 method), 22, 40  
`iter_to_pandas()` (asammdf.mdf.MDF method), 16

## M

`MDF` (class in asammdf.mdf), 14  
`MDF3` (class in asammdf.mdf3), 17, 36  
`MDF4` (class in asammdf.mdf4), 55

`merge()` (asammdf.mdf.MDF static method), [16](#)

## P

`plot()` (asammdf.signal.Signal method), [68](#)

## R

`resample()` (asammdf.mdf.MDF method), [16](#)

## S

`save()` (asammdf.mdf3.MDF3 method), [22](#), [41](#)

`save()` (asammdf.mdf4.MDF4 method), [59](#)

`select()` (asammdf.mdf.MDF method), [16](#)

Signal (class in asammdf.signal), [67](#)

SourceInformation (class in asammdf.mdf4), [64](#)

## T

TextBlock (class in asammdf.mdf2), [33](#)

TextBlock (class in asammdf.mdf3), [52](#)

TextBlock (class in asammdf.mdf4), [65](#)

TriggerBlock (class in asammdf.mdf2), [34](#)

TriggerBlock (class in asammdf.mdf3), [53](#)