asammdf Documentation

Release 2.2.0

Daniel Hrisca

Aug 21, 2017

Contents

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

asammdf supports both MDF version 3 and 4 formats.

asammdf works on Python 2.7, and Python >= 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

- read sorted and unsorted MDF v3 and v4 files
- files are loaded in RAM for fast operations
  - for low memory computers or for large data files there is the option to load only the metadata and leave the raw channel data (the samples) unread; this of course will mean slower channel data access speed
- 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 measuremetn will have channels from different sources at different rates
  - the *Signal* class facilitates operations with such channels
- remove data group by index or by specifing a channel name inside the target data group
- append new channels
- filter a subset of channels from original mdf file
- convert to different mdf version
- add and extract attachments
- mdf 4.10 zipped blocks

CHAPTER 3

Major features still not implemented

- functionality related to sample reduction block (but the class is defined)
- mdf 3 channel dependency save and append (only reading is implemented)
- handling of unfinnished measurements (mdf 4)
- mdf 4 channel arrays
- xml schema for TXBLOCK and MDBLOCK

CHAPTER 4

Dependencies

asammdf uses the following libraries

- numpy : the heart that makes all tick
- numexpr : for algebraic and rational channel conversions
- blosc : optionally used for in memmory raw channel data compression
- matplotlib : for Signal plotting
- pandas : for DataFrame export

CHAPTER 5

Features

- read sorted and unsorted MDF v3 and v4 files
- files are loaded in RAM for fast operations
  - for low memory computers or for large data files there is the option to load only the metadata and leave the raw channel data (the samples) unread; this of course will mean slower channel data access speed
- 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 measuremetn will have channels from different sources at different rates
  - the *Signal* class facilitates operations with such channels
- remove data group by index or by specifing a channel name inside the target data group
- create new mdf files from scratch
- append new channels
- convert to different mdf version
- add and extract attachments
- mdf 4.10 zipped blocks
- mdf 4 structure channels

CHAPTER 6

Major features still not implemented

- functionality related to sample reduction block (but the class is defined)
- mdf 3 channel dependency functionality
- functionality related to trigger blocks (but the class is defined)
- handling of unfinnished measurements (mdf 4)
- mdf 4 channel arrays
- xml schema for TXBLOCK and MDBLOCK

CHAPTER 7

Installation

asammdf is available on

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

```
pip install asammdf
```

CHAPTER 8

API

MDF

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

class `asammdf.mdf.`**`MDF`**(*name=None*, *load_measured_data=True*, *compression=False*, *version='3.20'*)
Unified access to MDF v3 and v4 files.

Parameters **name** : string

mdf file name

load_measured_data : bool

load data option; default *True*

- if *True* the data group binary data block will be loaded in RAM
- if *False* the channel data is read from disk on request

compression : bool

compression option for data group binary data block; default *False*

version : string

mdf file version ('3.00', '3.10', '3.20', '3.30', '4.00', '4.10', '4.11'); default '3.20'

Methods

`convert`
`filter`
`iter_to_pandas`

convert (*to*, *compression=False*)
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')

compression : bool

enable raw channel data compression for out MDF; default *False*

Returns **out** : MDF

new MDF object

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

Parameters **channels** : list

list of channel names to be filtered

Returns **mdf** : MDF

new MDF file

iter_to_pandas ()
generator that yields channel groups as pandas DataFrames

MDF3 and MDF4 classes

MDF3

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

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

The *groups* attribute is a dictionary list with the following keys:

- data_group : DataGroup object
- channel_group : ChannelGroup object
- channels : list of Channel objects with the same order as found in the mdf file
- channel_conversions : list of ChannelConversion objects in 1-to-1 relation with the channel list
- channel_sources : list of SourceInformation objects in 1-to-1 relation with the channels list
- data_block : DataBlock object
- texts : dictionay containing TextBlock objects used throughout the mdf
  - channels : list of dictionaries that contain TextBlock objects ralated to each channel
    - long_name_addr : channel long name
    - comment_addr : channel comment
    - display_name_addr : channel display name
  - channel group : list of dictionaries that contain TextBlock objects ralated to each channel group
    - comment_addr : channel group comment
  - conversion_tab : list of dictionaries that contain TextBlock objects ralated to VATB and VTABR channel conversions

  - text_{n} : n-th text of the VTABR conversion

The *file_history* attribute is a TextBlock object.

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

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

API

class `asammdf.mdf3.`**`MDF3`**(*name=None*, *load_measured_data=True*, *compression=False*, *version='3.20'*)

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

load_measured_data : bool

load data option; default *True*

- if *True* the data group binary data block will be loaded in RAM
- if *False* the channel data is read from disk on request

compression : bool

compression option for data group binary data block; default *False*

version : string

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

Attributes

name	(string) mdf file name
groups	(list) list of data groups
header	(OrderedDict) mdf file header
file_history	(TextBlock) file history text block; can be None
load_measured_data	(bool) load measured data option
compression	(bool) measured data compression option
version	(int) mdf version
channels_db	(dict) used for fast channel access by name; for each name key the value is a (group index, channel index) tuple
masters_db	(dict) used for fast master channel access; for each group index key the value is the master channel index

Methods

`add_trigger`
`append`
`get`

Continued on next page

Table 8.2 – continued from previous page

`info`
`iter_get_triggers`
`remove`
`save`

add_trigger (*group*, *time*, *pre_time=0*, *post_time=0*, *comment=''*)

add trigger to data group

Parameters **group** : int

group index

time : 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'*)

Appends a new data group.

Parameters **signals** : list

list on *Signal* objects

acquisition_info : str

acquisition information; default 'Python'

Examples

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

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

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

- using the first positional argument *name*
  - if there are multiple occurances for this channel then the *group* argument can be used to select a specific group.
  - if there are multiple occurances for this channel and the *group* argument 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 **vals, t, unit, conversion** : (numpy.array, numpy.array, string, dict | None)

The conversion is *None* exept for the VTAB and VTABR conversions. The conversion keys are:

- for VTAB conversion:
  - raw - numpy.array for X-axis
  - phys - numpy.array of strings for Y-axis
  - type - conversion type = CONVERSION_TYPE_VTAB
- for VTABR conversion:
  - lower - numpy.array for lower range
  - upper - numpy.array for upper range
  - phys - numpy.array of strings for Y-axis
  - type - conversion type = COONVERSION_TYPE_VTABR

The conversion information can be used by the *append* method for the *info* argument

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**

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

remove(*group=None*, *name=None*)

Remove data group. Use *group* or *name* keyword arguments to identify the group's index. *group* has priority

Parameters name : string

name of the channel inside the data group to be removed

group : int

data group index to be removed

Examples

```
>>> mdf = MDF3('test.mdf')
>>> mdf.remove(group=3)
>>> mdf.remove(name='VehicleSpeed')
```

save(*dst=None*)

Save MDF to *dst*. If *dst* is *None* the original file is overwritten

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

The keys have the following meaning:

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

Parameters **file_stream** : file handle

mdf file handle

address : int

block address inside mdf file

Examples

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

Attributes

name	(str) full channel name
address	(int) block address inside mdf file
dependencies	(list) lsit of channel dependencies

Methods

`clear`	
`copy`	Generic (shallow and deep) copying operations.
`fromkeys`	
`get`	
`items`	
`keys`	
`pop`	
`popitem`	
`setdefault`	
`update`	
`values`	

ChannelConversion Class

class `asammdf.mdf3.`**`ChannelConversion`**(***kargs*)

CCBLOCK class derived from *dict*

The ChannelConversion object can be created in two modes:

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

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

- common keys
  - id - Block type identifier, always “CC”
  - block_len - Block size of this block in bytes (entire CCBLOCK)

- range_flag - Physical value range valid flag:
- min_phy_value - Minimum physical signal value that occurred for this signal
- max_phy_value - Maximum physical signal value that occurred for this signal
- unit - Physical unit (string should be terminated with 0)
- conversion_type - Conversion type (formula identifier)
- ref_param_nr - Size information about additional conversion data

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

Parameters **file_stream** : file handle

mdf file handle

address : int

block address inside mdf file

Examples

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

Attributes

address	(int) block address inside mdf file

Methods

`clear`	
`copy`	Generic (shallow and deep) copying operations.
`fromkeys`	
`get`	
`items`	
`keys`	
`pop`	
`popitem`	
`setdefault`	
`update`	
`values`	

ChannelDependency Class

class `asammdf.mdf3.`**`ChannelDependency`**(***kargs*)

CDBLOCK class derived from *dict*

Currently the ChannelDependency object can only be created using the *file_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)
- data - Dependency type
- sd_nr - Total number of signals dependencies (m)
- for each dependency there is a group of three keys:
  - dg_{n} - Pointer to the data group block (DGBLOCK) of signal dependency *n*
  - cg_{n} - Pointer to the channel group block (DGBLOCK) of signal dependency *n*
  - ch_{n} - Pointer to the channel block (DGBLOCK) of signal dependency *n*
- there can also be optional keys which decribe dimensions for the N-dimensional dependencies:
  - dim_{n} - Optional: size of dimension *n* for N-dimensional dependency

Parameters **file_stream** : file handle mdf file handle

address : int

block address inside mdf file

Attributes

address	(int) block address inside mdf file

Methods

`clear`	
`copy`	Generic (shallow and deep) copying operations.
`fromkeys`	
`get`	
`items`	
`keys`	
`pop`	
`popitem`	
`setdefault`	
`update`	
`values`	

ChannelExtension Class

class `asammdf.mdf3.`**`ChannelExtension`**`(`*****kargs***`)`

CEBLOCK class derived from *dict*

The ChannelExtension object can be created in two modes:

- using the *file_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 **file_stream** : file handle

mdf file handle

address : int

block address inside mdf file

Attributes

address	(int) block address inside mdf file

Methods

`clear`	
`copy`	Generic (shallow and deep) copying operations.
`fromkeys`	
`get`	
`items`	
`keys`	
`pop`	
`popitem`	
`setdefault`	
`update`	
`values`	

ChannelGroup Class

class `asammdf.mdf3.`**`ChannelGroup`**(***kargs*)

CGBLOCK class derived from *dict*

The ChannelGroup object can be created in two modes:

- using the *file_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 **file_stream** : file handle

mdf file handle

address : int

block address inside mdf file

Examples

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

Attributes

address	(int) block address inside mdf file

Methods

`clear`	
`copy`	Generic (shallow and deep) copying operations.
`fromkeys`	
`get`	
`items`	
`keys`	
`pop`	
`popitem`	
`setdefault`	
`update`	
`values`	

DataGroup Class

class `asammdf.mdf3.DataGroup(**kargs)`

DGBLOCK class derived from *dict*

The DataGroup object can be created in two modes:

- using the *file_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 **file_stream** : file handle

mdf file handle

address : int

block address inside mdf file

Attributes

address	(int) block address inside mdf file

Methods

`clear`	
`copy`	Generic (shallow and deep) copying operations.
`fromkeys`	
`get`	
`items`	
`keys`	
`pop`	
`popitem`	
`setdefault`	
`update`	
`values`	

FileIdentificationBlock Class

class `asammdf.mdf3.`**`FileIdentificationBlock`**`(`***kargs*`)`

IDBLOCK class derived from *dict*

The TriggerBlock object can be created in two modes:

- using the *file_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 **file_stream** : file handle

mdf file handle

version : int

mdf version in case of new file

Attributes

address	(int) block address inside mdf file; should be 0 always

Methods

`clear`	
`copy`	Generic (shallow and deep) copying operations.
`fromkeys`	
`get`	
`items`	
`keys`	
`pop`	
`popitem`	
`setdefault`	

Continued on next page

Table 8.9 – continued from previous page

`update`
`values`

HeaderBlock Class

class `asammdf.mdf3.`**`HeaderBlock`**`(`***kargs*`)`

HDBLOCK class derived from *dict*

The TriggerBlock object can be created in two modes:

- using the *file_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 **file_stream** : file handle

mdf file handle

Attributes

address	(int) block address inside mdf file; should be 64 always

Methods

`clear`	
`copy`	Generic (shallow and deep) copying operations.
`fromkeys`	
`get`	
`items`	
`keys`	
`pop`	
`popitem`	
`setdefault`	
`update`	
`values`	

ProgramBlock Class

class `asammdf.mdf3.`**`ProgramBlock`**`(`***kargs*`)`

PRBLOCK class derived from *dict*

The ProgramBlock object can be created in two modes:

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

The keys have the following meaning:

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

Parameters **file_stream** : file handle

mdf file handle

address : int

block address inside mdf file

Attributes

address	(int) block address inside mdf file

Methods

`clear`	
`copy`	Generic (shallow and deep) copying operations.
`fromkeys`	
`get`	
`items`	
`keys`	
`pop`	

Continued on next page

Table 8.11 – continued from previous page

`popitem`	
`setdefault`	
`update`	
`values`	

SampleReduction Class

class `asammdf.mdf3.`**`SampleReduction`**`(`***kargs*`)`

SRBLOCK class derived from *dict*

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

The keys have the following meaning:

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

Parameters **file_stream** : file handle

mdf file handle

address : int

block address inside mdf file

Attributes

address	(int) block address inside mdf file

Methods

`clear`	
`copy`	Generic (shallow and deep) copying operations.
`fromkeys`	
`get`	
`items`	
`keys`	
`pop`	
`popitem`	
`setdefault`	
`update`	
`values`	

TextBlock Class

class `asammdf.mdf3.`**`TextBlock`**`(`***kargs*`)`

TXBLOCK class derived from *dict*

The ProgramBlock object can be created in two modes:

- using the *file_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 **file_stream** : file handle

mdf file handle

address : int

block address inside mdf file

text : bytes

bytes for creating a new TextBlock

Examples

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

Attributes

address	(int) block address inside mdf file
text_str	(str) text data as unicode string

Methods

`clear`	
`copy`	Generic (shallow and deep) copying operations.
`from_text`	
`fromkeys`	
`get`	
`items`	
`keys`	
`pop`	

Continued on next page

Table 8.13 – continued from previous page

`popitem`	
`setdefault`	
`update`	
`values`	

TriggerBlock Class

class `asammdf.mdf3.`**`TriggerBlock`**(***kargs*)

TRBLOCK class derived from *dict*

The TriggerBlock object can be created in two modes:

- using the *file_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 **file_stream** : file handle

mdf file handle

address : int

block address inside mdf file

Attributes

address	(int) block address inside mdf file

Methods

`clear`	
`copy`	Generic (shallow and deep) copying operations.
`fromkeys`	
`get`	
`items`	
`keys`	
`pop`	
`popitem`	

Continued on next page

Table 8.14 – continued from previous page

`setdefault`
`update`
`values`

MDF4

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

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

The *groups* attribute is a dictionary list with the following keys:

- data_group : DataGroup object
- channel_group : ChannelGroup object
- channels : list of Channel objects with the same order as found in the mdf file
- channel_conversions : list of ChannelConversion objects in 1-to-1 relation with the channel list
- channel_sources : list of SourceInformation objects in 1-to-1 relation with the channels list
- data_block : DataBlock object
- texts : dictionay containing TextBlock objects used throughout the mdf
  - channels : list of dictionaries that contain TextBlock objects ralated to each channel
    - name_addr : channel name
    - comment_addr : channel comment
  - channel group : list of dictionaries that contain TextBlock objects ralated to each channel group
    - acq_name_addr : channel group acquisition comment
    - comment_addr : channel group comment
  - conversion_tab : list of dictionaries that contain TextBlock objects related to TABX and RTABX channel conversions
    - text_{n} : n-th text of the VTABR conversion
    - default_addr : default text
  - conversions : list of dictionaries that containt TextBlock obejcts related to channel conversions
    - name_addr : converions name
    - unit_addr : channel unit_addr
    - comment_addr : converison comment
    - formula_addr : formula text; only valid for algebraic conversions
  - sources : list of dictionaries that containt TextBlock obejcts related to channel sources
    - name_addr : source name
    - path_addr : source path_addr
    - comment_addr : source comment

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

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

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

API

class `asammdf.mdf4.`**`MDF4`**(*name=None*, *load_measured_data=True*, *compression=False*, *version='4.00'*)

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

load_measured_data : bool

load data option; default *True*

- if *True* the data group binary data block will be loaded in RAM
- if *False* the channel data is read from disk on request

compression : bool

compression option for data group binary data block; default *False*

version : string

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

Attributes

name	(string) mdf file name
groups	(list) list of data groups
header	(HeaderBlock) mdf file header
file_history	(list) list of (FileHistory, TextBlock) pairs
comment	(TextBlock) mdf file comment
identification	(FileIdentificationBlock) mdf file start block
load_measured_data	(bool) load measured data option
compression	(bool) measured data compression option
version	(int) mdf version
channels_db	(dict) used for fast channel access by name; for each name key the value is a (group index, channel index) tuple
masters_db	(dict) used for fast master channel access; for each group index key the value is the master channel index

Methods

`append`
`attach`
`extract_attachment`

Continued on next page

Table 8.15 – continued from previous page

`get`
`info`
`remove`
`save`

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

Appends a new data group.

Parameters **signals** : list

list on *Signal* objects

acquisition_info : str

acquisition information; default 'Python'

Examples

```
>>> # case 1 conversion type None
>>> s1 = np.array([1, 2, 3, 4, 5])
>>> s2 = np.array([-1, -2, -3, -4, -5])
>>> s3 = np.array([0.1, 0.04, 0.09, 0.16, 0.25])
>>> t = np.array([0.001, 0.002, 0.003, 0.004, 0.005])
>>> names = ['Positive', 'Negative', 'Float']
>>> units = ['+', '-', '.f']
>>> info = {}
>>> s1 = Signal(samples=s1, timstamps=t, unit='+', name='Positive')
>>> s2 = Signal(samples=s2, timstamps=t, unit='-', name='Negative')
>>> s3 = Signal(samples=s3, timstamps=t, unit='flts', name='Floats')
>>> mdf = MDF4('new.mf4')
>>> mdf.append([s1, s2, s3], 'created by asammdf v1.1.0')
>>> # case 2: VTAB conversions from channels inside another file
>>> mdf1 = MDF4('in.mf4')
>>> ch1 = mdf1.get("Channel1_VTAB")
>>> ch2 = mdf1.get("Channel2_VTABR")
>>> sigs = [ch1, ch2]
>>> mdf2 = MDF4('out.mf4')
>>> mdf2.append(sigs, 'created by asammdf 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

`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*)

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* argument can be used to select a specific group.
  - if there are multiple occurances for this channel and the *group* argument 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 **vals, t, unit, conversion** : (numpy.array, numpy.array, string, dict | None)

The conversion is *None* exept for the VTAB and VTABR conversions. The conversion keys are:

- for TABX conversion:
  - raw - numpy.array for X-axis
  - phys - numpy.array of strings for Y-axis
  - type - conversion type = CONVERSION_TYPE_TABX
  - default - default bytes value
- for RTABX conversion:

- lower - numpy.array for lower range
- upper - numpy.array for upper range
- phys - numpy.array of strings for Y-axis
- type - conversion type =
- default - default bytes value

The conversion information can be used by the *append* method for the *info* argument

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**

info()

get MDF information as a dict

Examples

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

remove(*group=None*, *name=None*)

Remove data group. Use *group* or *name* keyword arguments to identify the group's index. *group* has priority

Parameters name : string

name of the channel inside the data group to be removed

group : int

data group index to be removed

Examples

```
>>> mdf = MDF4('test.mdf')
>>> mdf.remove(group=3)
>>> mdf.remove(name='VehicleSpeed')
```

save(*dst=None*)

Save MDF to *dst*. If *dst* is *None* the original file is overwritten

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 attachemnts are allowed, with keyword argument *data* of type bytes

Methods

`clear`	
`copy`	Generic (shallow and deep) copying operations.
`extract`	
`fromkeys`	
`get`	
`items`	
`keys`	
`pop`	
`popitem`	
`setdefault`	
`update`	
`values`	

Channel Class

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

CNBLOCK class

Methods

`clear`	
`copy`	Generic (shallow and deep) copying operations.
`fromkeys`	
`get`	
`items`	
`keys`	
`pop`	
`popitem`	
`setdefault`	
`update`	
`values`	

ChannelConversion Class

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

CCBLOCK class

Methods

`clear`	
`copy`	Generic (shallow and deep) copying operations.
`fromkeys`	
`get`	
`items`	
`keys`	
`pop`	
`popitem`	
`setdefault`	
`update`	
`values`	

ChannelGroup Class

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

CGBLOCK class

Methods

`clear`	
`copy`	Generic (shallow and deep) copying operations.
`fromkeys`	
`get`	
`items`	
`keys`	
`pop`	
`popitem`	
`setdefault`	
`update`	
`values`	

DataGroup Class

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

DGBLOCK class

Methods

`clear`	
`copy`	Generic (shallow and deep) copying operations.
`fromkeys`	
`get`	
`items`	
`keys`	

Continued on next page

Table 8.20 – continued from previous page

`pop`	
`popitem`	
`setdefault`	
`update`	
`values`	

DataList Class

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

DLBLOCK class

Methods

`clear`	
`copy`	Generic (shallow and deep) copying operations.
`fromkeys`	
`get`	
`items`	
`keys`	
`pop`	
`popitem`	
`setdefault`	
`update`	
`values`	

DataBlock Class

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

DTBLOCK class Raw channel dta can be compressed to save RAM; set the *compression* keyword argument to True when instantiating the object

Parameters **compression** : bool

enable raw channel data compression in RAM

address : int

DTBLOCK address inside the file

file_stream : int

file handle

Methods

`clear`	
`copy`	Generic (shallow and deep) copying operations.
`fromkeys`	
`get`	

Continued on next page

Table 8.22 – continued from previous page

`items`	
`keys`	
`pop`	
`popitem`	
`setdefault`	
`update`	
`values`	

FileIdentificationBlock Class

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

IDBLOCK class

Methods

`clear`	
`copy`	Generic (shallow and deep) copying operations.
`fromkeys`	
`get`	
`items`	
`keys`	
`pop`	
`popitem`	
`setdefault`	
`update`	
`values`	

HeaderBlock Class

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

HDBLOCK class

Methods

`clear`	
`copy`	Generic (shallow and deep) copying operations.
`fromkeys`	
`get`	
`items`	
`keys`	
`pop`	
`popitem`	
`setdefault`	
`update`	
`values`	

SourceInformation Class

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

SIBLOCK class

Methods

`clear`	
`copy`	Generic (shallow and deep) copying operations.
`fromkeys`	
`get`	
`items`	
`keys`	
`pop`	
`popitem`	
`setdefault`	
`update`	
`values`	

FileHistory Class

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

FHBLOCK class

Methods

`clear`	
`copy`	Generic (shallow and deep) copying operations.
`fromkeys`	
`get`	
`items`	
`keys`	
`pop`	
`popitem`	
`setdefault`	
`update`	
`values`	

TextBlock Class

class `asammdf.mdf4.`**`TextBlock`**`(`***kargs*`)`

common TXBLOCK and MDBLOCK class

Methods

`clear`	
`copy`	Generic (shallow and deep) copying operations.
`from_text`	
`fromkeys`	
`get`	
`items`	
`keys`	
`pop`	
`popitem`	
`setdefault`	
`update`	
`values`	

classmethod `from_text` (*text*, *meta=False*)
Create a TextBlock from a str or bytes

Parameters **text** : str | bytes

input text

meta : bool

enable meta text block

Examples

```
>>> t = TextBlock.from_text(b'speed')
>>> t['id']
b'##TX'
>>> t.text_str
speed
>>> t = TextBlock.from_text('mass', meta=True)
>>> t['id']
b'##MD'
```

Notes about *compression* and *load_measured_data* arguments

By default *MDF* object use no compression and the raw channel data is loaded into RAM. This will give you the best performance from *asammdf*.

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

1. use the *compression* flag: raw channel data is loaded into RAM but it is compressed. The default compression library is *blosc* and as a fallback *zlib* is used (slower). The advange is that you save RAM, but in return you will pay the compression/decompression time penalty in all operations (file open, getting channel data, saving to disk, converting).
2. use the *load_measured_data* flag: raw channel data is not read.

MDF defaults

Advantages

- best performance

Disadvantages

- highest RAM usage

Use case

- when data fits inside the system RAM

MDF with *compression*

Advantages

- lower RAM usage than *default*
- alows saving to disk and appending new data

Disadvantages

- slowest

Use case

- when *default* data exceeds RAM and you need to append and save

MDF with *load_measured_data*

Advantages

- lowest RAM usage
- faster than *compression*

Disadvantages

- ReadOnly mode: appending and saving is not possible

Use case

- when *default* data exceeds RAM and you only want to extract information from the file

Note: See benchmarks for the effects of using the flags.

Signal

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

The Signal represents a signal described by it's samples and timestamps. It can do aritmethic operations agains 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*, *timstamps* and *name* are mandatory arguments.

Parameters **samples** : numpy.array

signal samples

timestamps : numpy.array

signal timestamps

unit : str

signal unit

name : str

signal name

conversion : dict

dict describing the channel conversion , default *None*

comment : str

signal comment, default ''

Methods

astype
cut
interp
plot

astype (*np_type*)
returns new *Signal* with samples of dtype *np_type*

cut (*start*, *stop*)
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 **outsig** : 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
```

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

plot ()
plot Signal samples

Examples

Working with MDF

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


# create 3 Signal objects

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

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

# float64
s_float64 = Signal(samples=np.array([-20, -10, 0, 10, 20], dtype=np.int32),
                   timestamps=timestamps,
                   name='Float64_Signal',
                   unit='f8')

# create empty MDf version 4.00 file
mdf4 = MDF(version='4.00')

# 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')

# convert new file to mdf version 3.10 with compression of raw channel data
mdf3 = mdf4.convert(to='3.10', compression=True)
print(mdf3.version)
# prints >>> 3.10

# get the float signal
sig = mdf3.get('Float64_Signal')
print(sig)
# prints >>> Signal { name="Float64_Signal":        s=[-20 -10   0  10  20] t=[ 0.1  ␣
↪      0.2         0.30000001  0.40000001  0.5       ] unit="f8"      ␣
↪conversion=None }
```

Working with Signal

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

```
# create 3 Signal objects with different time stamps

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

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

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

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()
```

CHAPTER 9

Benchmarks

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

Intro

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

asamdf 2.1.0 was compared against *mdfreader 0.2.5*. *mdfreader* seems to be the most used Python package to handle MDF files, and it also supports both version 3 and 4 of the standard.

The three benchmark cathegories are file open, file save and extracting the data for all channels inside the file(36424 calls). For each cathegory two aspect were noted: elapsed time and peak RAM usage.

Dependencies

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

- psutil
- mdfreader

x64 Python results

The test environment used for 64 bit tests had:

- 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [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

- nodata = MDF object created with load_measured_data=False (raw channel data not loaded into RAM)
- compression = MDF object created with compression=True/blosc
- compression bcolz 6 = MDF object created with compression=6
- noDataLoading = MDF object read with noDataLoading=True

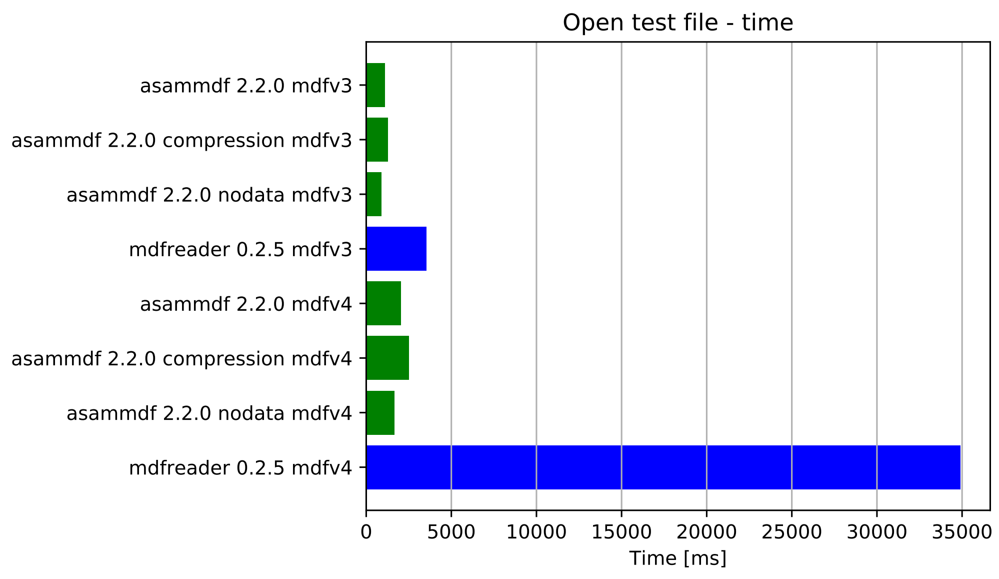
Files used for benchmark: * 183 groups * 36424 channels

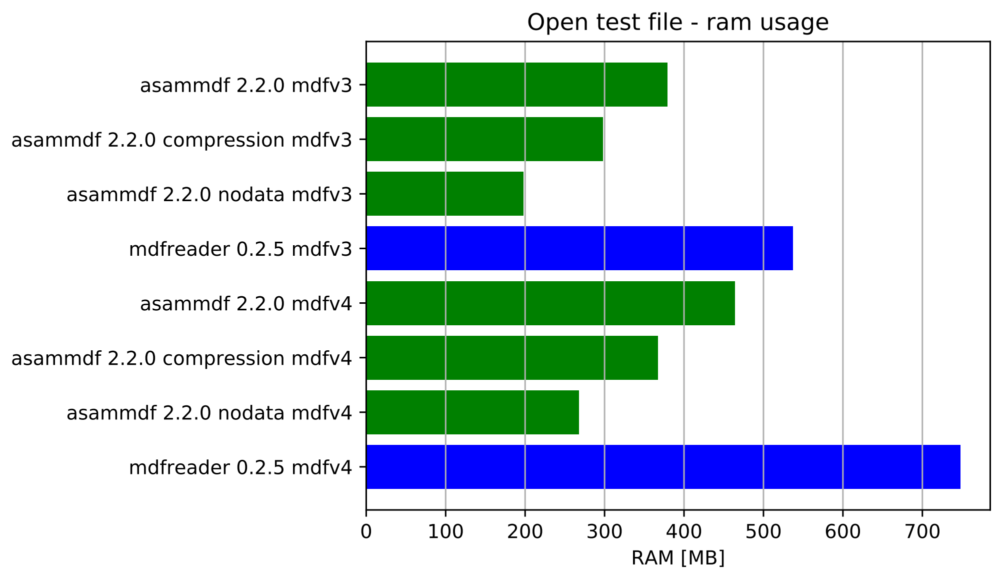
Open file	Time [ms]	RAM [MB]
asammdf 2.2.0 mdfv3	1088	379
asammdf 2.2.0 compression mdfv3	1287	298
asammdf 2.2.0 nodata mdfv3	896	198
mdfreader 0.2.5 mdfv3	3533	537
asammdf 2.2.0 mdfv4	2027	464
asammdf 2.2.0 compression mdfv4	2504	367
asammdf 2.2.0 nodata mdfv4	1668	268
mdfreader 0.2.5 mdfv4	34908	748

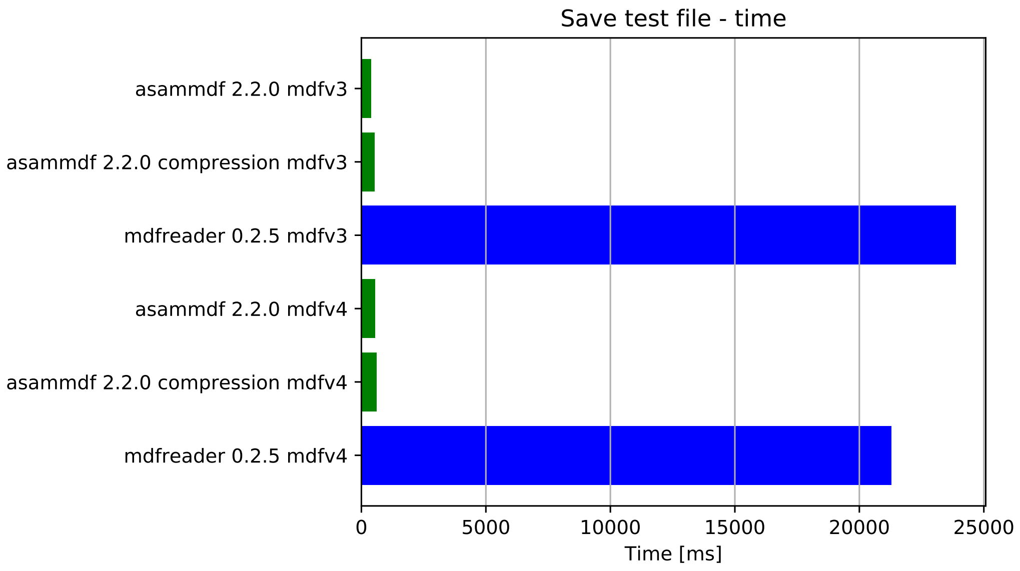
Save file	Time [ms]	RAM [MB]
asammdf 2.2.0 mdfv3	398	379
asammdf 2.2.0 compression mdfv3	523	302
mdfreader 0.2.5 mdfv3	23881	1997
asammdf 2.2.0 mdfv4	554	471
asammdf 2.2.0 compression mdfv4	615	373
mdfreader 0.2.5 mdfv4	21288	2795

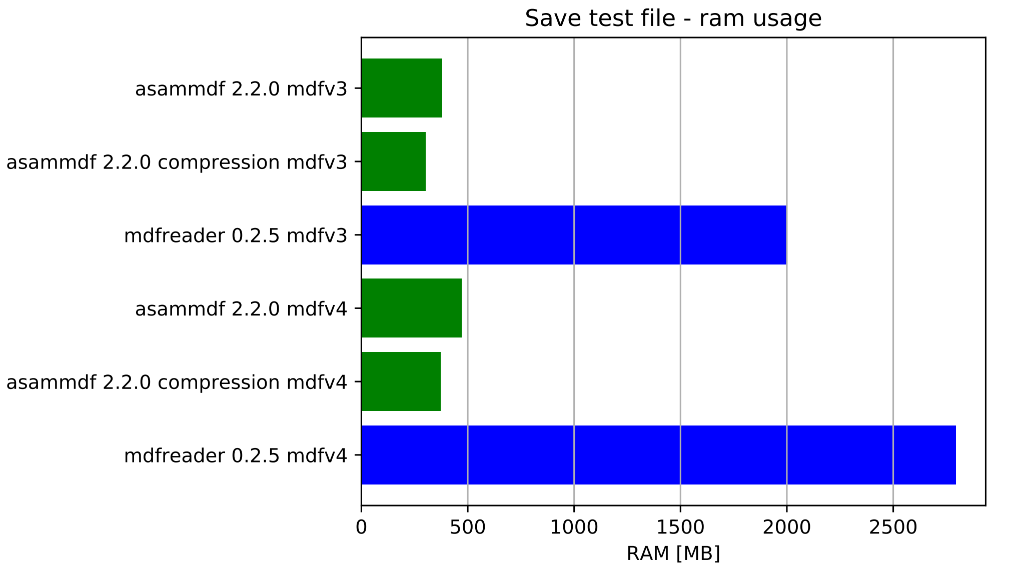
Get all channels (36424 calls)	Time [ms]	RAM [MB]
asammdf 2.2.0 mdfv3	577	383
asammdf 2.2.0 compression mdfv3	13504	306
asammdf 2.2.0 nodata mdfv3	9506	210
mdfreader 0.2.5 mdfv3	30	536
asammdf 2.2.0 mdfv4	498	469
asammdf 2.2.0 compression mdfv4	15310	377
asammdf 2.2.0 nodata mdfv4	12565	280
mdfreader 0.2.5 mdfv4	40	748

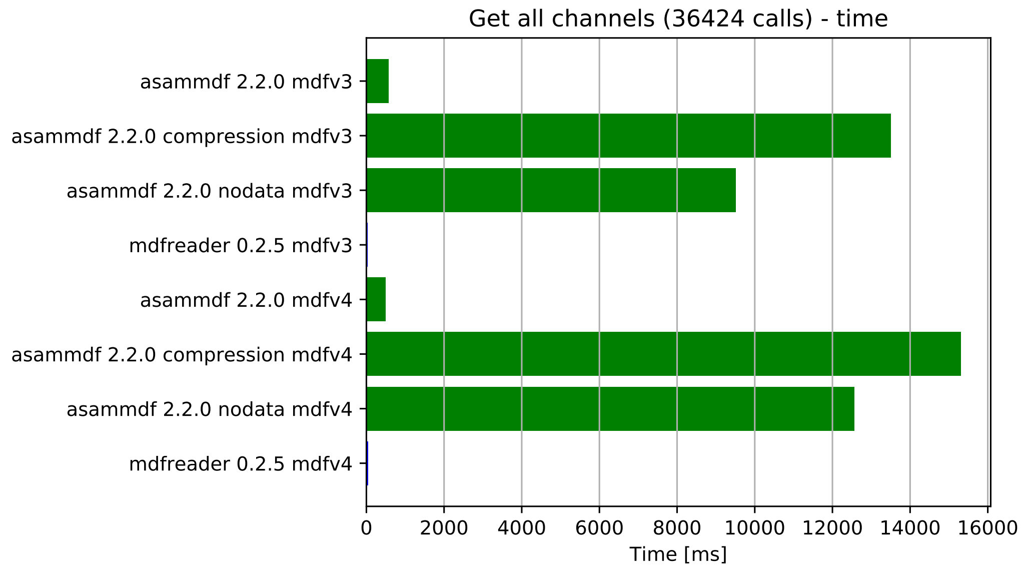
Graphical results

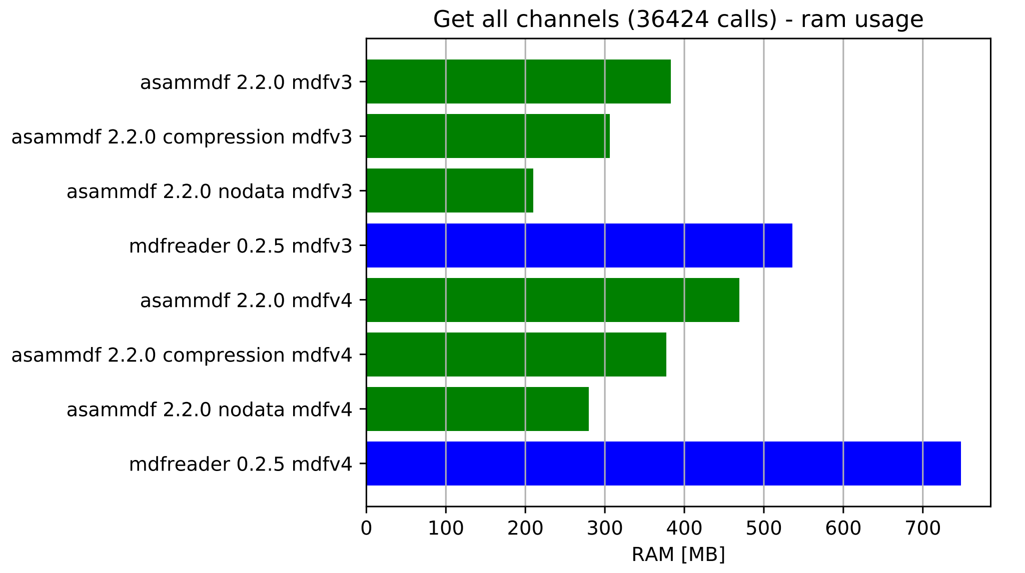





Save test file - time
asammdf 2.2.0 mdfv3
asammdf 2.2.0 compression mdfv3
mdfreader 0.2.5 mdfv3
asammdf 2.2.0 mdfv4
asammdf 2.2.0 compression mdfv4
mdfreader 0.2.5 mdfv4
0
5000
10000
15000
20000
25000
Time [ms]


Save test file - ram usage
asammdf 2.2.0 mdfv3
asammdf 2.2.0 compression mdfv3
mdfreader 0.2.5 mdfv3
asammdf 2.2.0 mdfv4
asammdf 2.2.0 compression mdfv4
mdfreader 0.2.5 mdfv4
0
500
1000
1500
2000
2500
RAM [MB]





x86 Python results

The test environment used for 32 bit tests had:

- Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)]
- Windows-7-6.1.7601-SP1
- Intel64 Family 6 Model 94 Stepping 3, GenuineIntel (i7-6820Q)
- 16GB installed RAM

The notations used in the results have the following meaning:

- nodata = MDF object created with load_measured_data=False (raw channel data no loaded into RAM)
- compression = MDF object created with compression=True (raw channel data loaded into RAM and compressed)
- noconvert = MDF object created with convertAfterRead=False

Raw data

- 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

- nodata = MDF object created with load_measured_data=False (raw channel data not loaded into RAM)
- compression = MDF object created with compression=True/blosc
- compression bcolz 6 = MDF object created with compression=6
- noDataLoading = MDF object read with noDataLoading=True

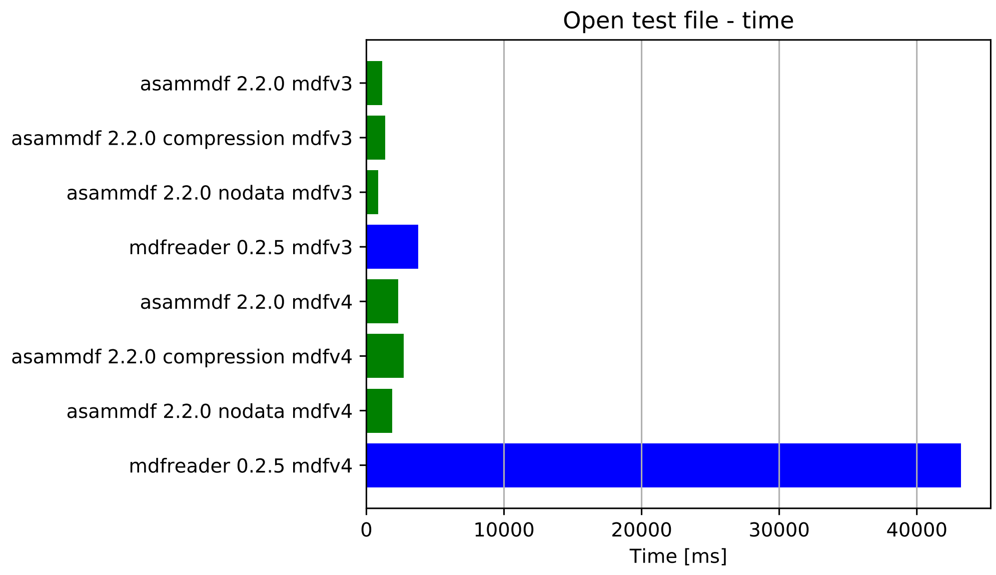
Files used for benchmark: * 183 groups * 36424 channels

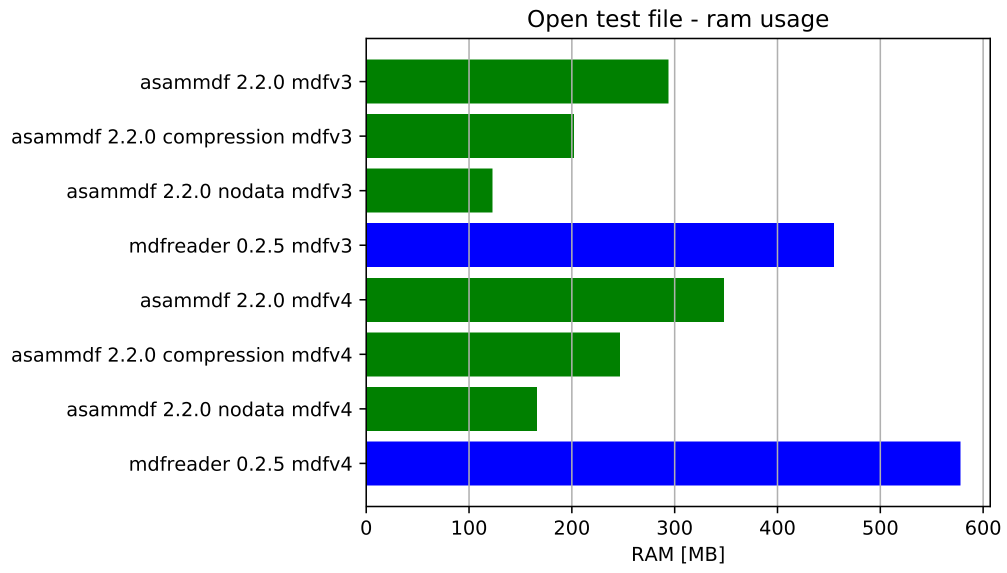
Open file	Time [ms]	RAM [MB]
asammdf 2.2.0 mdfv3	1149	294
asammdf 2.2.0 compression mdfv3	1368	202
asammdf 2.2.0 nodata mdfv3	861	123
mdfreader 0.2.5 mdfv3	3755	455
asammdf 2.2.0 mdfv4	2316	348
asammdf 2.2.0 compression mdfv4	2694	247
asammdf 2.2.0 nodata mdfv4	1886	166
mdfreader 0.2.5 mdfv4	43210	578

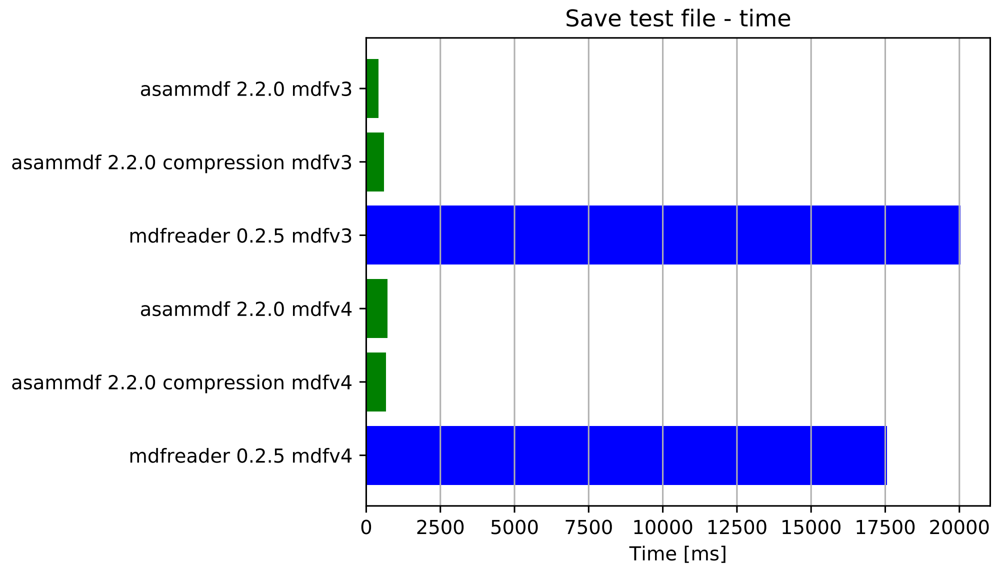
Save file	Time [ms]	RAM [MB]
asammdf 2.2.0 mdfv3	413	297
asammdf 2.2.0 compression mdfv3	592	204
mdfreader 0.2.5 mdfv3	20038	1224
asammdf 2.2.0 mdfv4	720	357
asammdf 2.2.0 compression mdfv4	674	253
mdfreader 0.2.5 mdfv4	17553	1687

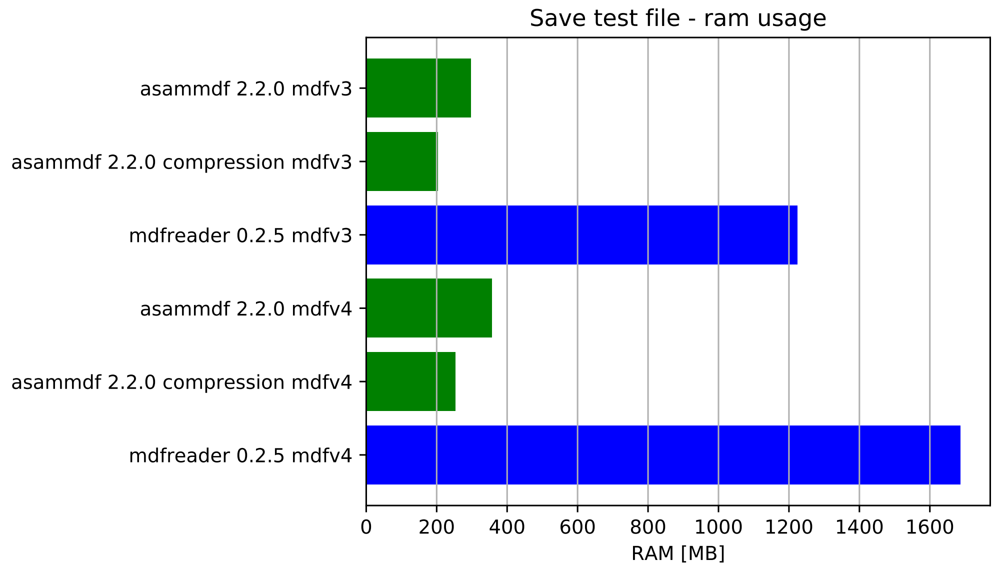
Get all channels (36424 calls)	Time [ms]	RAM [MB]
asammdf 2.2.0 mdfv3	784	299
asammdf 2.2.0 compression mdfv3	25345	207
asammdf 2.2.0 nodata mdfv3	18657	133
mdfreader 0.2.5 mdfv3	35	455
asammdf 2.2.0 mdfv4	695	354
asammdf 2.2.0 compression mdfv4	24325	255
asammdf 2.2.0 nodata mdfv4	20745	176
mdfreader 0.2.5 mdfv4	50	578

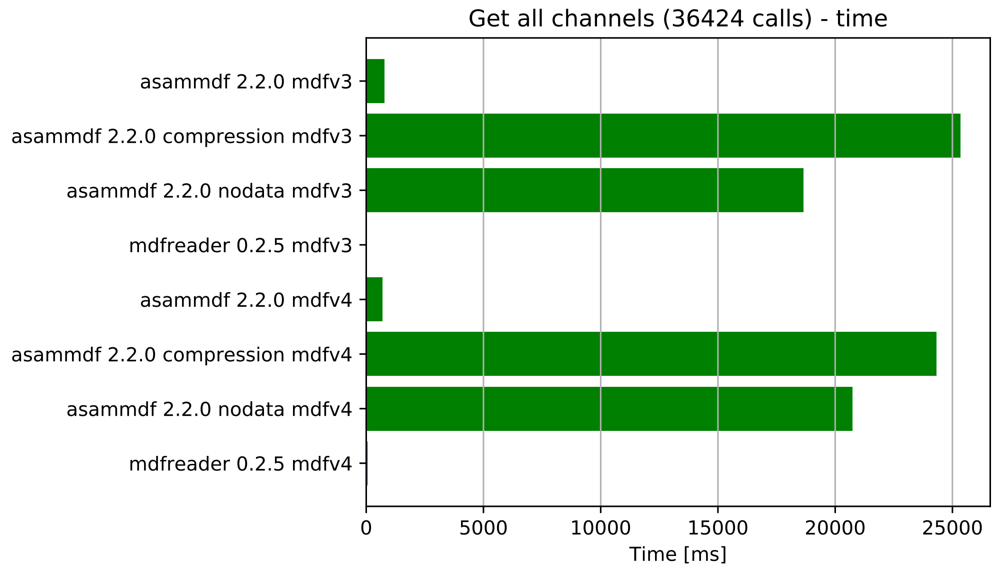
Graphical results

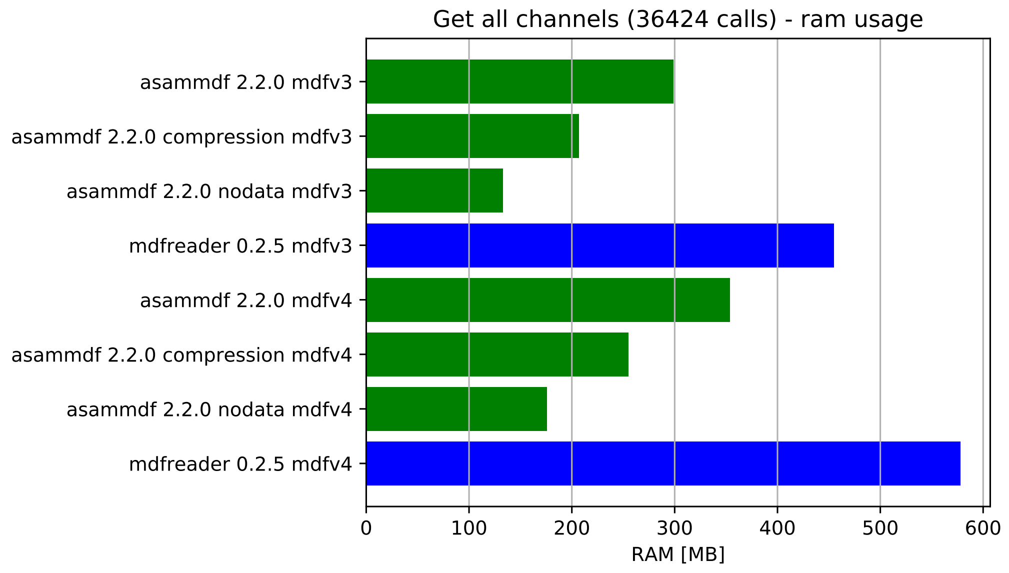





Save test file - time
asammdf 2.2.0 mdfv3
asammdf 2.2.0 compression mdfv3
mdfreader 0.2.5 mdfv3
asammdf 2.2.0 mdfv4
asammdf 2.2.0 compression mdfv4
mdfreader 0.2.5 mdfv4
0
2500
5000
7500
10000
12500
15000
17500
20000
Time [ms]


Save test file - ram usage
asammdf 2.2.0 mdfv3
asammdf 2.2.0 compression mdfv3
mdfreader 0.2.5 mdfv3
asammdf 2.2.0 mdfv4
asammdf 2.2.0 compression mdfv4
mdfreader 0.2.5 mdfv4
0
200
400
600
800
1000
1200
1400
1600
RAM [MB]


Get all channels (36424 calls) - time
asammdf 2.2.0 mdfv3
asammdf 2.2.0 compression mdfv3
asammdf 2.2.0 nodata mdfv3
mdfreader 0.2.5 mdfv3
asammdf 2.2.0 mdfv4
asammdf 2.2.0 compression mdfv4
asammdf 2.2.0 nodata mdfv4
mdfreader 0.2.5 mdfv4
0
5000
10000
15000
20000
25000
Time [ms]


Get all channels (36424 calls) - ram usage
asammdf 2.2.0 mdfv3
asammdf 2.2.0 compression mdfv3
asammdf 2.2.0 nodata mdfv3
mdfreader 0.2.5 mdfv3
asammdf 2.2.0 mdfv4
asammdf 2.2.0 compression mdfv4
asammdf 2.2.0 nodata mdfv4
mdfreader 0.2.5 mdfv4
0
100
200
300
400
500
600
RAM [MB]

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Index