asammdf Documentation

Release "7.3.11"

Daniel Hrisca

Apr 05, 2023

CONTENTS

1 Introduction **3**
- 1.1 Project goals . . . 3
- 1.2 Features . . . 3
- 1.3 Major features not implemented (yet) . . . 4
- 1.4 Dependencies . . . 4
- 1.5 Installation . . . 5
- 1.6 Contributing & Support . . . 6
  - 1.6.1 Contributors . . . 6

2 API **7**
- 2.1 MDF . . . 7
- 2.2 MDF3 . . . 29
  - 2.2.1 MDF version 2 & 3 blocks . . . 38
    - 2.2.1.1 Channel Class . . . 38
    - 2.2.1.2 ChannelConversion Class . . . 39
    - 2.2.1.3 ChannelDependency Class . . . 41
    - 2.2.1.4 ChannelExtension Class . . . 41
    - 2.2.1.5 ChannelGroup Class . . . 42
    - 2.2.1.6 DataGroup Class . . . 43
    - 2.2.1.7 FileIdentificationBlock Class . . . 44
    - 2.2.1.8 HeaderBlock Class . . . 44
    - 2.2.1.9 ProgramBlock Class . . . 45
    - 2.2.1.10 TextBlock Class . . . 46
    - 2.2.1.11 TriggerBlock Class . . . 46
- 2.3 MDF4 . . . 47
  - 2.3.1 MDF version 4 blocks . . . 59
    - 2.3.1.1 AttachmentBlock Class . . . 59
    - 2.3.1.2 Channel Class . . . 60
    - 2.3.1.3 ChannelConversion Class . . . 62
    - 2.3.1.4 ChannelGroup Class . . . 64
    - 2.3.1.5 DataGroup Class . . . 64
    - 2.3.1.6 DataList Class . . . 65
    - 2.3.1.7 DataBlock Class . . . 66
    - 2.3.1.8 FileIdentificationBlock Class . . . 66
    - 2.3.1.9 HeaderBlock Class . . . 67
    - 2.3.1.10 SourceInformation Class . . . 68
    - 2.3.1.11 FileHistory Class . . . 68
    - 2.3.1.12 TextBlock Class . . . 69
    - 2.3.1.13 EventBlock Class . . . 70
- 2.4 Signal . . . 71

3 Bus logging **75**

4 Tips **77**
4.1 Selective channel loading . . . 77
4.2 Data coupling . . . 77
4.3 Chunked data access . . . 77
4.4 Optimized methods . . . 77
4.5 Faster file loading . . . 78
4.5.1 Skip XML parsing for MDF4 files . . . 78

5 Examples **79**
5.1 Working with MDF . . . 79
5.2 Working with Signal . . . 80
5.3 MF4 demo file generator . . . 82

6 Benchmarks **89**
6.1 Test setup . . . 89
6.1.1 Dependencies . . . 89
6.1.2 Usage . . . 89
6.2 x64 Python results . . . 89
6.2.1 Graphical results . . . 91

7 GUI **95**
7.1 General shortcuts . . . 95
7.2 Menu . . . 95
7.2.1 File . . . 95
7.2.2 Mode . . . 96
7.2.3 Settings . . . 96
7.2.4 Plot . . . 97
7.3 Single files . . . 99
7.3.1 Layout elements . . . 99
7.3.1.1 1. Opened files tabs . . . 100
7.3.1.2 2. Channel tree display mode . . . 100
7.3.1.3 3. Complete channels tree . . . 100
7.3.1.4 4. Command buttons . . . 101
7.3.1.5 5. Windows area . . . 102
7.3.1.6 6. Numeric window . . . 102
7.3.1.7 7. Plot window . . . 103
7.3.1.8 8. Tabular window . . . 105
7.3.1.9 9. File operations . . . 106
7.3.1.10 10. CAN/LIN/FlexRay Bus Trace . . . 106
7.3.1.11 11. Drag & Drop . . . 107
7.4 Batch processing . . . 107
7.5 Comparison . . . 108

8 Functions manager view **109**
8.1 Function manager dialog . . . 109
8.2 Python function rules . . . 110
8.3 Virtual channel dialog . . . 111
8.4 Example functions . . . 114
8.4.1 Average of 3 channels . . . 114
8.4.2 Channel clipping . . . 114
8.4.3 Grey code to decimal . . . 114
8.4.4 Maximum of 3 channels . . . 115
8.4.5 rpm to rad/s . . . 115

8.4.6 gradient 115

9 Indices and tables **117**

Index **119**

asammdf is a fast parser and editor for ASAM (Association for Standardization 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 >= 3.7 (for Python 2.7, 3.4 and 3.5 see the 4.x.y releases)

CHAPTER

ONE

INTRODUCTION

1.1 Project goals

The main goals for this library are:

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

1.2 Features

- create new mdf files from scratch
- append new channels
- read unsorted MDF v3 and v4 files
- read CAN and LIN bus logging files
- extract CAN and LIN signals from anonymous bus logging measurements
- filter a subset of channels from original mdf file
- cut measurement to specified time interval
- convert to different mdf version
- export to pandas, HDF5, Matlab (v7.3), CSV and parquet
- merge multiple files sharing the same internal structure
- read and save mdf version 4.10 files containing zipped data blocks
- space optimizations for saved files (no duplicated blocks)
- split large data blocks (configurable size) for mdf version 4
- full support (read, append, save) for the following map types (multidimensional array channels):
  - mdf version 3 channels with CDBLOCK
  - mdf version 4 structure channel composition
  - mdf version 4 channel arrays with CNTemplate storage and one of the array types:
    - 0 - array
    - 1 - scaling axis
    - 2 - look-up

- add and extract attachments for mdf version 4
- handle large files (for example merging two files, each with 14000 channels and 5GB size, on a RaspberryPi)
- extract channel data, master channel and extra channel information as *Signal* objects for unified operations with v3 and v4 files
- time domain operation using the *Signal* class
  - Pandas data frames are good if all the channels have the same time base
  - a measurement will usually have channels from different sources at different rates
  - the *Signal* class facilitates operations with such channels

1.3 Major features not implemented (yet)

- for version 3
  - functionality related to sample reduction block: the sample reduction blocks are simply ignored
- for version 4
  - functionality related to sample reduction block: the sample reduction blocks are simply ignored
  - handling of channel hierarchy: channel hierarchy is ignored
  - full handling of bus logging measurements: currently only CAN and LIN bus logging are implemented with the ability to *get* signals defined in the attached CAN/LIN database (.arxml or .dbc). Signals can also be extracted from an anonymous bus logging measurement by providing a CAN or LIN database (.dbc or .arxml)
  - handling of unfinished measurements (mdf 4): warnings are logged based on the unfinished status flags but no further steps are taken to sanitize the measurement
  - full support for remaining mdf 4 channel arrays types
  - xml schema for MDBLOCK: most metadata stored in the comment blocks will not be available
  - full handling of event blocks: events are transferred to the new files (in case of calling methods that return new *MDF* objects) but no new events can be created
  - channels with default X axis: the default X axis is ignored and the channel group's master channel is used
  - attachment encryption/decryption using user provided encryption/decryption functions; this is not part of the MDF v4 spec and is only supported by this library

1.4 Dependencies

asammdf uses the following libraries

- numpy : the heart that makes all tick
- numexpr : for algebraic and rational channel conversions
- wheel : for installation in virtual environments
- pandas : for DataFrame export
- canmatrix : to handle CAN/LIN bus logging measurements
- natsort

- lxml : for canmatrix arxml support
- lz4 : to speed up the disk IO performance
- python-dateutil : measurement start time handling

optional dependencies needed for exports

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

other optional dependencies

- PySide6 : for GUI tool
- pyqtgraph : for GUI tool and Signal plotting (preferably the latest develop branch code)
- matplotlib : as fallback for Signal plotting
- cChardet : to detect non-standard Unicode encodings
- chardet : to detect non-standard Unicode encodings
- pyqtlet2 : for the GPS window
- isal : for faster zlib compression/decompression
- fsspec : access files stored in the cloud

1.5 Installation

asammdf is available on

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

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

In case a wheel is not present for you OS/Python versions and you lack the proper compiler setup to compile the c-extension code, then you can simply copy-paste the package code to your site-packages. In this way the python fallback code will be used instead of the compiled c-extension code.

1.6 Contributing & Support

Please have a look over the contributing guidelines

If you enjoy this library please consider making a donation to the numpy project or to danielhrisca using liberapay

1.6.1 Contributors

Thanks to all who contributed with commits to *asammdf*

Contributors Thanks to all who contributed with commits to *asammdf*:

CHAPTER

TWO

API

2.1 MDF

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

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

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

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

class asammdf.mdf.**MDF**(*name: InputType | None = None*, *version: str = '4.10'*, *channels: list[str] | None = None*, ***kwargs*)

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

Parameters

name
[string | BytesIO | zipfile.ZipFile | bz2.BZ2File | gzip.GzipFile] mdf file name (if provided it must be a real file name), file-like object or compressed file opened as Python object

Changed in version 6.2.0: added support for zipfile.ZipFile, bz2.BZ2File and gzip.GzipFile

version
[string] mdf file version from ('2.00', '2.10', '2.14', '3.00', '3.10', '3.20', '3.30', '4.00', '4.10', '4.11', '4.20'); default '4.10'. This argument is only used for MDF objects created from scratch; for MDF objects created from a file the version is set to file version

channels (None)
[iterable] channel names that will used for selective loading. This can dramatically improve the file loading time. Default None -> load all channels

New in version 6.1.0.

Changed in version 6.3.0: make the default None

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

remove_source_from_channel_names (kwargs)**
[bool] remove source from channel names ("SpeedXCP3" -> "Speed")

copy_on_get (kwargs)**
[bool] copy arrays in the get method; default *True*

expand_zippedfile (kwargs)**
[bool] only for bz2.BZ2File and gzip.GzipFile, load the file content into a BytesIO before parsing (avoids the huge performance penalty of doing random reads from the zipped file); default *True*

raise_on_multiple_occurrences (kwargs)**
[bool] raise exception when there are multiple channel occurrences in the file and the *get* call is ambiguous; default True

New in version 7.0.0.

temporary_folder (kwargs)**
[str | pathlib.Path] folder to use for temporary files

New in version 7.0.0.

Examples

```
>>> mdf = MDF(version='3.30') # new MDF object with version 3.30
>>> mdf = MDF('path/to/file.mf4') # MDF loaded from file
>>> mdf = MDF(BytesIO(data)) # MDF from file contents
>>> mdf = MDF(zipfile.ZipFile('data.zip')) # MDF creating using the first valid MDF␣
↪from archive
>>> mdf = MDF(bz2.BZ2File('path/to/data.bz2', 'rb')) # MDF from bz2 object
>>> mdf = MDF(gzip.GzipFile('path/to/data.gzip', 'rb')) # MDF from gzip object
```

cleanup_timestamps(*minimum: float*, *maximum: float*, *exp_min: int = -15*, *exp_max: int = 15*, *version: str | None = None*, *progress=None*) → *MDF*

convert *MDF* to other version

New in version 5.22.0.

Parameters

minimum
[float] minimum plausible time stamp

maximum
[float] maximum plausible time stamp

exp_min (-15)
[int] minimum plausible exponent used for the time stamps float values

exp_max (15)
[int] maximum plausible exponent used for the time stamps float values

version
[str] new mdf file version from ('2.00', '2.10', '2.14', '3.00', '3.10', '3.20', '3.30', '4.00', '4.10', '4.11', '4.20'); default the same as the input file

Returns

out
[MDF] new *MDF* object

static concatenate(*files: Sequence[MDF | InputType]*, *version: str = '4.10'*, *sync: bool = True*, *add_samples_origin: bool = False*, *direct_timestamp_continuation: bool = False*, *progress=None*, ***kwargs*) → *MDF*

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

The order of the input files is always preserved, only the samples timestamps are influenced by the `sync` argument.

Parameters

files
[list | tuple] list of *MDF* file names or *MDF*, zipfile.ZipFile, bz2.BZ2File or gzip.GzipFile instances

..versionchanged:: 6.2.0

added support for zipfile.ZipFile, bz2.BZ2File and gzip.GzipFile

version
[str] merged file version

sync
[bool] sync the files based on the start of measurement, default *True*. The order of the input files is preserved, only the samples timestamps are influenced by this argument

add_samples_origin
[bool] option to create a new "__samples_origin" channel that will hold the index of the measurement from where each timestamp originated

direct_timestamp_continuation (False)
[bool] the time stamps from the next file will be added right after the last time stamp from the previous file; default False

..versionadded:: 6.0.0

kwargs
use_display_names (False) : bool

Returns

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

Raises

MdfException
[if there are inconsistencies between the files]

Examples

```
>>> conc = MDF.concatenate(
    [
        'path/to/file.mf4',
        MDF(BytesIO(data)),
        MDF(zipfile.ZipFile('data.zip')),
        MDF(bz2.BZ2File('path/to/data.bz2', 'rb')),
        MDF(gzip.GzipFile('path/to/data.gzip', 'rb')),
    ],
```

(continues on next page)

(continued from previous page)

```
    version='4.00',
    sync=False,
)
```

configure(*, *from_other: MDF_v2_v3_v4 | None = None*, *read_fragment_size: int | None = None*, *write_fragment_size: int | None = None*, *use_display_names: bool | None = None*, *single_bit_uint_as_bool: bool | None = None*, *integer_interpolation: IntInterpolationModeType | IntegerInterpolation | None = None*, *copy_on_get: bool | None = None*, *float_interpolation: FloatInterpolationModeType | FloatInterpolation | None = None*, *raise_on_multiple_occurrences: bool | None = None*, *temporary_folder: str | None = None*, *fill_0_for_missing_computation_channels: bool | None = None*) → None

configure MDF parameters

The default values for the options are the following: * read_fragment_size = 0 * write_fragment_size = 4MB * use_display_names = False * single_bit_uint_as_bool = False * integer_interpolation = 0 (fill - use previous sample) * float_interpolation = 1 (linear interpolation) * copy_on_get = False * raise_on_multiple_occurrences = True * temporary_folder = "" * fill_0_for_missing_computation_channels = False

Parameters

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

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

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

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

integer_interpolation
[int] interpolation mode for integer channels:

- 0 - repeat previous sample
- 1 - use linear interpolation
- 2 - hybrid interpolation: channels with integer data type (raw values) that have a conversion that outputs float values will use linear interpolation, otherwise the previous sample is used

Changed in version 6.2.0: added hybrid mode interpolation

copy_on_get
[bool] copy arrays in the get method

float_interpolation
[int] interpolation mode for float channels:

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

New in version 6.2.0.

raise_on_multiple_occurrences
[bool] raise exception when there are multiple channel occurrences in the file and the *get* call is ambiguous; default True

New in version 6.2.0.

from_other
[MDF] copy configuration options from other MDF

New in version 6.2.0.

temporary_folder
[str] default folder for temporary files

New in version 7.0.0.

fill_0_for_missing_computation_channels
[bool] when a channel required by a computed channel is missing, then fill with 0 values. If false then the computation will fail and the computed channel will be marked as not existing.

New in version 7.1.0.

`convert`(*version: str*, *progress=None*) → *MDF*

convert *MDF* to other version

Parameters

version
[str] new mdf file version from ('2.00', '2.10', '2.14', '3.00', '3.10', '3.20', '3.30', '4.00', '4.10', '4.11', '4.20'); default '4.10'

Returns

out
[MDF] new *MDF* object

`cut`(*start: float | None = None*, *stop: float | None = None*, *whence: int = 0*, *version: str | None = None*, *include_ends: bool = True*, *time_from_zero: bool = False*, *progress=None*) → *MDF*

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

Parameters

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

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

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

- 0 : absolute
- 1 : relative to first timestamp

version
[str] new mdf file version from ('2.00', '2.10', '2.14', '3.00', '3.10', '3.20', '3.30', '4.00', '4.10', '4.11', 4.20'); default *None* and in this case the original file version is used

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

time_from_zero
[bool] start time stamps from 0s in the cut measurement

Returns

out
[MDF] new MDF object

`export`(*fmt: Literal['asc', 'csv', 'hdf5', 'mat', 'parquet']*, *filename: StrPathType | None = None*, *progress=None*, ***kwargs*) → None

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

The *pandas* export option was removed. you should use the method *to_dataframe* instead.

Parameters

fmt
[string] can be one of the following:

- *csv* : CSV export that uses the "," delimiter. This option will generate a new csv file for each data group (<MDFNAME>_DataGroup_<cntr>.csv)
- *hdf5* : HDF5 file output; each *MDF* data group is mapped to a *HDF5* group with the name 'DataGroup_<cntr>' (where <cntr> is the index)
- *mat* : Matlab .mat version 4, 5 or 7.3 export. If *single_time_base==False* the channels will be renamed in the mat file to 'D<cntr>_<channel name>'. The channel group master will be renamed to 'DM<cntr>_<channel name>' (*<cntr>* is the data group index starting from 0)
- *parquet* : export to Apache parquet format
- *asc*: Vector ASCII format for bus logging

  New in version 7.3.3.

filename
[string | pathlib.Path] export file name

****kwargs**

- *single_time_base*: resample all channels to common time base, default *False*
- *raster*: float time raster for resampling. Valid if *single_time_base* is *True*
- *time_from_zero*: adjust time channel to start from 0
- *use_display_names*: use display name instead of standard channel name, if available.
- *empty_channels*: behaviour for channels without samples; the options are *skip* or *zeros*; default is *skip*
- *format*: only valid for *mat* export; can be '4', '5' or '7.3', default is '5'
- *oned_as*: only valid for *mat* export; can be 'row' or 'column'

- *keep_arrays* : keep arrays and structure channels as well as the component channels. If *True* this can be very slow. If *False* only the component channels are saved, and their names will be prefixed with the parent channel.
- *reduce_memory_usage* : bool reduce memory usage by converting all float columns to float32 and searching for minimum dtype that can reprezent the values found in integer columns; default *False*
- *compression* : str compression to be used
  - for `parquet` : "GZIP" or "SNAPPY"
  - for `hfd5` : "gzip", "lzf" or "szip"
  - for `mat` : bool
- *time_as_date* (False) : bool export time as local timezone datetimee; only valid for CSV export

  New in version 5.8.0.
- *ignore_value2text_conversions* (False) : bool valid only for the channels that have value to text conversions and if *raw=False*. If this is True then the raw numeric values will be used, and the conversion will not be applied.

  New in version 5.8.0.
- raw (False) : bool export all channels using the raw values

  New in version 6.0.0.
- delimiter (',') : str only valid for CSV: see cpython documentation for csv.Dialect.delimiter

  New in version 6.2.0.
- doublequote (True) : bool only valid for CSV: see cpython documentation for csv.Dialect.doublequote

  New in version 6.2.0.
- escapechar (None) : str only valid for CSV: see cpython documentation for csv.Dialect.escapechar

  New in version 6.2.0.
- lineterminator ("\r\n") : str only valid for CSV: see cpython documentation for csv.Dialect.lineterminator

  New in version 6.2.0.
- quotechar ("'") : str only valid for CSV: see cpython documentation for csv.Dialect.quotechar

  New in version 6.2.0.
- quoting ("MINIMAL") : str only valid for CSV: see cpython documentation for csv.Dialect.quoting. Use the last part of the quoting constant name

  New in version 6.2.0.
- add_units (False) : bool only valid for CSV: add the channel units on the second row of the CSV file

  New in version 7.1.0.

`extract_bus_logging`(*database_files: dict[BusType, Iterable[DbcFileType]]*, *version: str | None = None*, *ignore_invalid_signals: bool | None = None*, *consolidated_j1939: bool | None = None*, *ignore_value2text_conversion: bool = True*, *prefix: str = ''*, *progress=None*) → *MDF*

extract all possible CAN signal using the provided databases.

Changed in version 6.0.0 from *extract_can_logging*

Parameters

database_files
[dict] each key will contain an iterable of database files for that bus type. The supported bus types are "CAN", "LIN". The iterables will contain the (databases, valid bus) pairs. The database can be a str, pathlib.Path or canamtrix.CanMatrix object. The valid bus is an integer specifying for which bus channel the database can be applied; 0 means any bus channel.

Changed in version 6.0.0: added canmatrix.CanMatrix type

Changed in version 6.3.0: added bus channel fileter

version (None)
[str] output file version

ignore_invalid_signals (None)
[bool | None] ignore signals that have all samples equal to their maximum value

New in version 5.7.0.

Deprecated since version 7.0.2: this argument is no longer used and will be removed in the future

consolidated_j1939 (None)
[bool | None] handle PGNs from all the messages as a single instance

New in version 5.7.0.

Deprecated since version 7.2.0: this argument is no longer used and will be removed in the future

ignore_value2text_conversion (True): bool
ignore value to text conversions

New in version 5.23.0.

prefix ("")
[str] prefix that will added to the channel group names and signal names in the output file

New in version 6.3.0.

Returns

mdf
[MDF] new MDF file that contains the succesfully extracted signals

Examples

```
>>> "extrac CAN and LIN bus logging"
>>> mdf = asammdf.MDF(r'bus_logging.mf4')
>>> databases = {
...     "CAN": [("file1.dbc", 0), ("file2.arxml", 2)],
...     "LIN": [("file3.dbc", 0)],
... }
>>> extracted = mdf.extract_bus_logging(database_files=database_files)
>>> ...
>>> "extrac just LIN bus logging"
>>> mdf = asammdf.MDF(r'bus_logging.mf4')
>>> databases = {
...     "LIN": [("file3.dbc", 0)],
... }
>>> extracted = mdf.extract_bus_logging(database_files=database_files)
```

filter(*channels: ChannelsType*, *version: str | None = None*, *progress=None*) → *MDF*

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

Parameters

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

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

version
[str] new mdf file version from ('2.00', '2.10', '2.14', '3.00', '3.10', '3.20', '3.30', '4.00', '4.10', '4.11', '4.20'); default *None* and in this case the original file version is used

Returns

mdf
[MDF] new *MDF* file

Examples

```
>>> from asammdf import MDF, Signal
>>> import numpy as np
>>> t = np.arange(5)
>>> s = np.ones(5)
>>> mdf = MDF()
>>> for i in range(4):
...     sigs = [Signal(s*(i*10+j), t, name='SIG') for j in range(1,4)]
...     mdf.append(sigs)
...
>>> filtered = mdf.filter(['SIG', ('SIG', 3, 1), ['SIG', 2], (None, 1, 2)])
>>> for gp_nr, ch_nr in filtered.channels_db['SIG']:
```

(continues on next page)

(continued from previous page)

```
...     print(filtered.get(group=gp_nr, index=ch_nr))
...
<Signal SIG:
        samples=[ 1.  1.  1.  1.  1.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
<Signal SIG:
        samples=[ 31.  31.  31.  31.  31.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
<Signal SIG:
        samples=[ 21.  21.  21.  21.  21.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
<Signal SIG:
        samples=[ 12.  12.  12.  12.  12.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
```

get_group(*index: int*, *raster: RasterType | None = None*, *time_from_zero: bool = True*, *empty_channels: EmptyChannelsType = 'skip'*, *keep_arrays: bool = False*, *use_display_names: bool = False*, *time_as_date: bool = False*, *reduce_memory_usage: bool = False*, *raw: bool = False*, *ignore_value2text_conversions: bool = False*, *only_basenames: bool = False*) → pd.DataFrame

get channel group as pandas DataFrames. If there are multiple occurrences for the same channel name, then a counter will be used to make the names unique (<original_name>_<counter>)

Parameters

index
[int] channel group index

use_display_names
[bool] use display name instead of standard channel name, if available.

reduce_memory_usage
[bool] reduce memory usage by converting all float columns to float32 and searching for minimum dtype that can reprezent the values found in integer columns; default *False*

raw (False)
[bool] the dataframe will contain the raw channel values

New in version 5.7.0.

ignore_value2text_conversions (False)
[bool] valid only for the channels that have value to text conversions and if *raw=False*. If this is True then the raw numeric values will be used, and the conversion will not be applied.

New in version 5.8.0.

keep_arrays (False)
[bool] keep arrays and structure channels as well as the component channels. If *True* this can be very slow. If *False* only the component channels are saved, and their names will be prefixed with the parent channel.

New in version 5.8.0.

empty_channels ("skip")
[str] behaviour for channels without samples; the options are *skip* or *zeros*; default is *skip*

New in version 5.8.0.

only_basenames (False)
[bool] use just the field names, without prefix, for structures and channel arrays

New in version 5.13.0.

raster
[float | np.array | str] new raster that can be

- a float step value
- a channel name who's timestamps will be used as raster (starting with asammdf 5.5.0)
- an array (starting with asammdf 5.5.0)

see *resample* for examples of using this argument

Returns

df
[pandas.DataFrame]

`iter_channels`(*skip_master: bool = True*, *copy_master: bool = True*, *raw: bool = False*) → Iterator[*Signal*]

generator that yields a *Signal* for each non-master channel

Parameters

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

copy_master
[bool] copy master for each yielded channel *True*

raw
[bool] return raw channels instead of converted; default *False*

`iter_get`(*name: str | None = None*, *group: int | None = None*, *index: int | None = None*, *raster: float | None = None*, *samples_only: Literal[False] = False*, *raw: bool = False*) → Iterator[*Signal*]

`iter_get`(*name: str | None = None*, *group: int | None = None*, *index: int | None = None*, *raster: float | None = None*, *samples_only: Literal[True] = False*, *raw: bool = False*) → Iterator[tuple[NDArray[Any], NDArray[Any] | None]]

iterator over a channel

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

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

Parameters

name
[string] name of channel

group
[int] 0-based group index

index
[int] 0-based channel index

raster
[float] time raster in seconds

samples_only
[bool]

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

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

`iter_groups`(*raster: RasterType | None = None*, *time_from_zero: bool = True*, *empty_channels: EmptyChannelsType = 'skip'*, *keep_arrays: bool = False*, *use_display_names: bool = False*, *time_as_date: bool = False*, *reduce_memory_usage: bool = False*, *raw: bool = False*, *ignore_value2text_conversions: bool = False*, *only_basenames: bool = False*) → Iterator[pd.DataFrame]

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

Parameters

use_display_names
[bool] use display name instead of standard channel name, if available.

New in version 5.21.0.

reduce_memory_usage
[bool] reduce memory usage by converting all float columns to float32 and searching for minimum dtype that can reprezent the values found in integer columns; default *False*

New in version 5.21.0.

raw (False)
[bool] the dataframe will contain the raw channel values

New in version 5.21.0.

ignore_value2text_conversions (False)
[bool] valid only for the channels that have value to text conversions and if *raw=False*. If this is True then the raw numeric values will be used, and the conversion will not be applied.

New in version 5.21.0.

keep_arrays (False)
[bool] keep arrays and structure channels as well as the component channels. If *True* this can be very slow. If *False* only the component channels are saved, and their names will be prefixed with the parent channel.

New in version 5.21.0.

empty_channels ("skip")
[str] behaviour for channels without samples; the options are *skip* or *zeros*; default is *skip*

New in version 5.21.0.

only_basenames (False)
[bool] use just the field names, without prefix, for structures and channel arrays

New in version 5.21.0.

raster
[float | np.array | str] new raster that can be

- a float step value
- a channel name who's timestamps will be used as raster (starting with asammdf 5.5.0)
- an array (starting with asammdf 5.5.0)

see *resample* for examples of using this argument

New in version 5.21.0.

`iter_to_dataframe`(*channels: ChannelsType | None = None*, *raster: RasterType | None = None*, *time_from_zero: bool = True*, *empty_channels: EmptyChannelsType = 'skip'*, *keep_arrays: bool = False*, *use_display_names: bool = False*, *time_as_date: bool = False*, *reduce_memory_usage: bool = False*, *raw: bool = False*, *ignore_value2text_conversions: bool = False*, *use_interpolation: bool = True*, *only_basenames: bool = False*, *chunk_ram_size: int = 209715200*, *interpolate_outwards_with_nan: bool = False*, *numeric_1D_only: bool = False*, *progress=None*) → Iterator[pd.DataFrame]

generator that yields pandas DataFrame's that should not exceed 200MB of RAM

New in version 5.15.0.

Parameters

channels
[list] list of items to be filtered (default None); each item can be :

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

raster
[float | np.array | str] new raster that can be

- a float step value
- a channel name who's timestamps will be used as raster (starting with asammdf 5.5.0)
- an array (starting with asammdf 5.5.0)

see *resample* for examples of using this argument

time_from_zero
[bool] adjust time channel to start from 0; default *True*

empty_channels
[str] behaviour for channels without samples; the options are *skip* or *zeros*; default is *skip*

use_display_names
[bool] use display name instead of standard channel name, if available.

keep_arrays
[bool] keep arrays and structure channels as well as the component channels. If *True* this can be very slow. If *False* only the component channels are saved, and their names will be prefixed with the parent channel.

time_as_date
[bool] the dataframe index will contain the datetime timestamps according to the measurement start time; default *False*. If *True* then the argument `time_from_zero` will be ignored.

reduce_memory_usage
[bool] reduce memory usage by converting all float columns to float32 and searching for minimum dtype that can reprezent the values found in integer columns; default *False*

raw (False)
[bool] the columns will contain the raw values

ignore_value2text_conversions (False)
[bool] valid only for the channels that have value to text conversions and if *raw=False*. If this is True then the raw numeric values will be used, and the conversion will not be applied.

use_interpolation (True)
[bool] option to perform interpoaltions when multiple timestamp raster are present. If *False* then dataframe columns will be automatically filled with NaN's were the dataframe index values are not found in the current column's timestamps

only_basenames (False)
[bool] use jsut the field names, without prefix, for structures and channel arrays

interpolate_outwards_with_nan
[bool] use NaN values for the samples that lie outside of the original signal's timestamps

chunk_ram_size
[int] desired data frame RAM usage in bytes; default 200 MB

numeric_1D_only (False)
[bool] only keep the 1D-columns that have numeric values

New in version 7.0.0.

Returns

dataframe
[pandas.DataFrame] yields pandas DataFrame's that should not exceed 200MB of RAM

`resample`(*raster: RasterType*, *version: str | None = None*, *time_from_zero: bool = False*, *progress=None*) → *MDF*

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

Parameters

raster
[float | np.array | str] new raster that can be

- a float step value
- a channel name who's timestamps will be used as raster (starting with asammdf 5.5.0)
- an array (starting with asammdf 5.5.0)

version
[str] new mdf file version from ('2.00', '2.10', '2.14', '3.00', '3.10', '3.20', '3.30', '4.00', '4.10', '4.11', '4.20'); default *None* and in this case the original file version is used

time_from_zero
[bool] start time stamps from 0s in the cut measurement

Returns

mdf
[MDF] new *MDF* with resampled channels

Examples

```
>>> from asammdf import MDF, Signal
>>> import numpy as np
>>> mdf = MDF()
>>> sig = Signal(name='S1', samples=[1,2,3,4], timestamps=[1,2,3,4])
>>> mdf.append(sig)
>>> sig = Signal(name='S2', samples=[1,2,3,4], timestamps=[1.1, 3.5, 3.7, 3.9])
>>> mdf.append(sig)
>>> resampled = mdf.resample(raster=0.1)
>>> resampled.select(['S1', 'S2'])
[<Signal S1:
        samples=[1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 4]
        timestamps=[1.  1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.  2.1 2.2 2.3 2.
↪4 2.5 2.6 2.7
 2.8 2.9 3.  3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 4. ]
        invalidation_bits=None
        unit=""
        conversion=None
        source=Source(name='Python', path='Python', comment='', source_type=4,␣
↪bus_type=0)
        comment=""
        mastermeta="('time', 1)"
        raw=True
        display_names={}
        attachment=()>
, <Signal S2:
        samples=[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 3 3 4 4]
        timestamps=[1.  1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.  2.1 2.2 2.3 2.
↪4 2.5 2.6 2.7
 2.8 2.9 3.  3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 4. ]
        invalidation_bits=None
        unit=""
```

(continues on next page)

(continued from previous page)

```
        conversion=None
        source=Source(name='Python', path='Python', comment='', source_type=4,␣
↪bus_type=0)
        comment=""
        mastermeta="('time', 1)"
        raw=True
        display_names={}
        attachment=()>
]
>>> resampled = mdf.resample(raster='S2')
>>> resampled.select(['S1', 'S2'])
[<Signal S1:
        samples=[1 3 3 3]
        timestamps=[1.1 3.5 3.7 3.9]
        invalidation_bits=None
        unit=""
        conversion=None
        source=Source(name='Python', path='Python', comment='', source_type=4,␣
↪bus_type=0)
        comment=""
        mastermeta="('time', 1)"
        raw=True
        display_names={}
        attachment=()>
, <Signal S2:
        samples=[1 2 3 4]
        timestamps=[1.1 3.5 3.7 3.9]
        invalidation_bits=None
        unit=""
        conversion=None
        source=Source(name='Python', path='Python', comment='', source_type=4,␣
↪bus_type=0)
        comment=""
        mastermeta="('time', 1)"
        raw=True
        display_names={}
        attachment=()>
]
>>> resampled = mdf.resample(raster=[1.9, 2.0, 2.1])
>>> resampled.select(['S1', 'S2'])
[<Signal S1:
        samples=[1 2 2]
        timestamps=[1.9 2.  2.1]
        invalidation_bits=None
        unit=""
        conversion=None
        source=Source(name='Python', path='Python', comment='', source_type=4,␣
↪bus_type=0)
        comment=""
        mastermeta="('time', 1)"
        raw=True
        display_names={}
```

(continues on next page)

(continued from previous page)

```
        attachment=()>
, <Signal S2:
        samples=[1 1 1]
        timestamps=[1.9 2.  2.1]
        invalidation_bits=None
        unit=""
        conversion=None
        source=Source(name='Python', path='Python', comment='', source_type=4,␣
↪bus_type=0)
        comment=""
        mastermeta="('time', 1)"
        raw=True
        display_names={}
        attachment=()>
]
>>> resampled = mdf.resample(raster='S2', time_from_zero=True)
>>> resampled.select(['S1', 'S2'])
[<Signal S1:
        samples=[1 3 3 3]
        timestamps=[0.  2.4 2.6 2.8]
        invalidation_bits=None
        unit=""
        conversion=None
        source=Source(name='Python', path='Python', comment='', source_type=4,␣
↪bus_type=0)
        comment=""
        mastermeta="('time', 1)"
        raw=True
        display_names={}
        attachment=()>
, <Signal S2:
        samples=[1 2 3 4]
        timestamps=[0.  2.4 2.6 2.8]
        invalidation_bits=None
        unit=""
        conversion=None
        source=Source(name='Python', path='Python', comment='', source_type=4,␣
↪bus_type=0)
        comment=""
        mastermeta="('time', 1)"
        raw=True
        display_names={}
        attachment=()>
]
```

static scramble(*name: str | PathLike[str]*, *skip_attachments: bool = False*, *progress=None*, ***kwargs*) → Path

scramble text blocks and keep original file structure

Parameters

name
[str | pathlib.Path] file name

skip_attachments
[bool] skip scrambling of attachments data if True

New in version 5.9.0.

Returns

name
[pathlib.Path] scrambled file name

search(*pattern: str*, *mode: Literal['plain', 'regex', 'wildcard'] | SearchMode = SearchMode.plain*, *case_insensitive: bool = False*) → list[str]

search channels

New in version 7.0.0.

Parameters

pattern
[str] search pattern

mode
[Literal[“plain”, “regex”, “wildcard”] or SearchMode, optional] search mode, by default SearchMode.plain

- *plain* : normal name search
- *regex* : regular expression based search
- *wildcard* : wildcard based search

case_insensitive
[bool, optional] case sensitivity for the channel name search, by default False

Returns

list[str]
name of the channels

Raises

ValueError
unsupported search mode

Examples

```
>>> mdf = MDF(file_name)
>>> mdf.search('*veh*speed*', case_insensitive=True, mode='wildcard') # case␣
↪insensitive wildcard based search
['vehicleAverageSpeed', 'vehicleInstantSpeed', 'targetVehicleAverageSpeed',
↪'targetVehicleInstantSpeed']
>>> mdf.search('^vehicle.*Speed$', case_insensitive=False, mode='regex') #␣
↪case sensitive regex based search
['vehicleAverageSpeed', 'vehicleInstantSpeed']
```

select(*channels: ChannelsType*, *record_offset: int = 0*, *raw: bool = False*, *copy_master: bool = True*, *ignore_value2text_conversions: bool = False*, *record_count: int | None = None*, *validate: bool = False*) → list[*Signal*]

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

Note: the *dataframe* argument was removed in version 5.8.0 use the `to_dataframe` method instead

Parameters

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

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

record_offset
[int] record number offset; optimization to get the last part of signal samples

raw
[bool] get raw channel samples; default *False*

copy_master
[bool] option to get a new timestamps array for each selected Signal or to use a shared array for channels of the same channel group; default *True*

ignore_value2text_conversions (False)
[bool] valid only for the channels that have value to text conversions and if *raw=False*. If this is True then the raw numeric values will be used, and the conversion will not be applied.

New in version 5.8.0.

validate (False)
[bool] consider the invalidation bits

New in version 5.16.0.

Returns

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

Examples

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

(continues on next page)

(continued from previous page)

```
[<Signal SIG:
        samples=[ 1.  1.  1.  1.  1.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
, <Signal SIG:
        samples=[ 31.  31.  31.  31.  31.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
, <Signal SIG:
        samples=[ 21.  21.  21.  21.  21.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
, <Signal SIG:
        samples=[ 12.  12.  12.  12.  12.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
]
```

static stack(*files: Sequence[*MDF *| InputType]*, *version: str = '4.10'*, *sync: bool = True*, *progress=None*, ***kwargs*) → *MDF*

stack several files and return the stacked *MDF* object

Parameters

files
[list | tuple] list of *MDF* file names or *MDF*, zipfile.ZipFile, bz2.BZ2File or gzip.GzipFile instances

..versionchanged:: 6.2.0

added support for zipfile.ZipFile, bz2.BZ2File and gzip.GzipFile

version
[str] merged file version

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

kwargs
use_display_names (False) : bool

Returns

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

Examples

```
>>> stacked = MDF.stack(
    [
        'path/to/file.mf4',
        MDF(BytesIO(data)),
        MDF(zipfile.ZipFile('data.zip')),
        MDF(bz2.BZ2File('path/to/data.bz2', 'rb')),
        MDF(gzip.GzipFile('path/to/data.gzip', 'rb')),
    ],
    version='4.00',
    sync=False,
)
```

property start_time: datetime

getter and setter the measurement start timestamp

Returns

timestamp
[datetime.datetime] start timestamp

to_dataframe(*channels: ChannelsType | None = None*, *raster: RasterType | None = None*, *time_from_zero: bool = True*, *empty_channels: EmptyChannelsType = 'skip'*, *keep_arrays: bool = False*, *use_display_names: bool = False*, *time_as_date: bool = False*, *reduce_memory_usage: bool = False*, *raw: bool = False*, *ignore_value2text_conversions: bool = False*, *use_interpolation: bool = True*, *only_basenames: bool = False*, *interpolate_outwards_with_nan: bool = False*, *numeric_1D_only: bool = False*, *progress=None*) → pd.DataFrame

generate pandas DataFrame

Parameters

channels
[list] list of items to be filtered (default None); each item can be :

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

raster
[float | np.array | str] new raster that can be

- a float step value
- a channel name who's timestamps will be used as raster (starting with asammdf 5.5.0)
- an array (starting with asammdf 5.5.0)

see *resample* for examples of using this argument

time_from_zero
[bool] adjust time channel to start from 0; default *True*

empty_channels
[str] behaviour for channels without samples; the options are *skip* or *zeros*; default is *skip*

use_display_names
[bool] use display name instead of standard channel name, if available.

keep_arrays
[bool] keep arrays and structure channels as well as the component channels. If *True* this can be very slow. If *False* only the component channels are saved, and their names will be prefixed with the parent channel.

time_as_date
[bool] the dataframe index will contain the datetime timestamps according to the measurement start time; default *False*. If *True* then the argument `time_from_zero` will be ignored.

reduce_memory_usage
[bool] reduce memory usage by converting all float columns to float32 and searching for minimum dtype that can reprezent the values found in integer columns; default *False*

raw (False)
[bool] the columns will contain the raw values

New in version 5.7.0.

ignore_value2text_conversions (False)
[bool] valid only for the channels that have value to text conversions and if *raw=False*. If this is True then the raw numeric values will be used, and the conversion will not be applied.

New in version 5.8.0.

use_interpolation (True)
[bool] option to perform interpoaltions when multiple timestamp raster are present. If *False* then dataframe columns will be automatically filled with NaN's were the dataframe index values are not found in the current column's timestamps

New in version 5.11.0.

only_basenames (False)
[bool] use just the field names, without prefix, for structures and channel arrays

New in version 5.13.0.

interpolate_outwards_with_nan
[bool] use NaN values for the samples that lie outside of the original signal's timestamps

New in version 5.15.0.

Returns

dataframe
[pandas.DataFrame]

`whereis`(*channel: str*, *source_name: str | None = None*, *source_path: str | None = None*, *acq_name: str | None = None*) → tuple[tuple[int, int], ...]

get occurrences of channel name in the file

Parameters

channel
[str] channel name string

source_name
[str, optional] filter occurrences on source name, by default None

source_path
[str, optional] filter occurrences on source path, by default None

acq_name
[str, optional] filter occurrences on channel group acquisition name, by default None

New in version 6.0.0.

Returns

tuple[tuple[int, int], ...]
(gp_idx, cn_idx) pairs

Examples

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

2.2 MDF3

class asammdf.blocks.mdf_v3.**MDF3**(*name: BufferedReader | BytesIO | StrPathType | None = None*, *version: str = '3.30'*, *channels: list[str] | None = None*, ***kwargs*)

The *header* attibute is a *HeaderBlock*.

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

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

Parameters

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

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

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

Attributes

attachments
[list] list of file attachments

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

groups
[list] list of data group dicts

header
[HeaderBlock] mdf file header

identification
[FileIdentificationBlock] mdf file start block

last_call_info
[dict | None] a dict to hold information about the last called method.

New in version 5.12.0.

masters_db
[dict]

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

memory
[str] memory optimization option

name
[string] mdf file name

version
[str] mdf version

`add_trigger`(*group: int*, *timestamp: float*, *pre_time: float = 0*, *post_time: float = 0*, *comment: str = ''*) → None

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: list[Signal] | Signal | DataFrame, acq_name: str | None = None, acq_source: Source | None = None, comment: str = 'Python', common_timebase: bool = False, units: dict[str, str | bytes] | None = None*) → int | None

Appends a new data group.

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

Parameters

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

acq_name
[str] channel group acquisition name

acq_source
[Source] channel group acquisition source

comment
[str] channel group comment; default 'Python'

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

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

Examples

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

(continues on next page)

(continued from previous page)

```
>>> mdf2 = MDF3('out.mdf')
>>> mdf2.append(sigs, comment='created by asammdf v1.1.0')
>>> df = pd.DataFrame.from_dict({'s1': np.array([1, 2, 3, 4, 5]), 's2': np.
↪array([-1, -2, -3, -4, -5])})
>>> units = {'s1': 'V', 's2': 'A'}
>>> mdf2.append(df, units=units)
```

close() → None

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

extend(*index: int*, *signals: list[tuple[NDArray[Any], None]]*) → None

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

Parameters

index
[int] group index

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

Examples

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

get(*name: str | None = None*, *group: int | None = None*, *index: int | None = None*, *raster: RasterType | None = None*, *samples_only: Literal[False] = False*, *data: bytes | None = None*, *raw: bool = False*, *ignore_invalidation_bits: bool = False*, *record_offset: int = 0*, *record_count: int | None = None*, *skip_channel_validation: bool = False*) → *Signal*

get(*name: str | None = None*, *group: int | None = None*, *index: int | None = None*, *raster: RasterType | None = None*, *samples_only: Literal[True] = False*, *data: bytes | None = None*, *raw: bool = False*, *ignore_invalidation_bits: bool = False*, *record_offset: int = 0*, *record_count: int | None = None*, *skip_channel_validation: bool = False*) → tuple[NDArray[Any], 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

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

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

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

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

skip_channel_validation (False)
[bool] skip validation of channel name, group index and channel index; defualt *False*. If *True*, the caller has to make sure that the *group* and *index* arguments are provided and are correct.

..versionadded:: 7.0.0

Returns

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

The *Signal* samples are

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

Raises

MdfException

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

Examples

```
>>> from asammdf import MDF, Signal
>>> import numpy as np
>>> t = np.arange(5)
>>> s = np.ones(5)
>>> mdf = MDF(version='3.30')
>>> for i in range(4):
...     sigs = [Signal(s*(i*10+j), t, name='Sig') for j in range(1, 4)]
...     mdf.append(sigs)
...
>>> # first group and channel index of the specified channel name
...
>>> mdf.get('Sig')
UserWarning: Multiple occurrences for channel "Sig". Using first occurrence␣
↪from data group 4. Provide both "group" and "index" arguments to select␣
↪another data group
<Signal Sig:
        samples=[ 1.  1.  1.  1.  1.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
>>> # first channel index in the specified group
...
>>> mdf.get('Sig', 1)
<Signal Sig:
        samples=[ 11.  11.  11.  11.  11.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
>>> # channel named Sig from group 1 channel index 2
...
>>> mdf.get('Sig', 1, 2)
<Signal Sig:
        samples=[ 12.  12.  12.  12.  12.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
>>> # channel index 1 or group 2
...
>>> mdf.get(None, 2, 1)
<Signal Sig:
```

(continues on next page)

(continued from previous page)

```
        samples=[ 21.  21.  21.  21.  21.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
>>> mdf.get(group=2, index=1)
<Signal Sig:
        samples=[ 21.  21.  21.  21.  21.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
```

get_channel_comment(*name: str | None = None*, *group: int | None = None*, *index: int | None = None*) → str

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

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

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

Parameters

name
[string] name of channel

group
[int] 0-based group index

index
[int] 0-based channel index

Returns

comment
[str] found channel comment

get_channel_name(*group: int*, *index: int*) → str

Gets channel name.

Parameters

group
[int] 0-based group index

index
[int] 0-based channel index

Returns

name
[str] found channel name

`get_channel_unit`(*name: str | None = None*, *group: int | None = None*, *index: int | None = None*) → str

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: int*, *data: bytes | None = None*, *raster: RasterType | None = None*, *record_offset: int = 0*, *record_count: int | None = None*, *one_piece: bool = False*) → NDArray[Any]

returns master channel samples for given group

Parameters

index
[int] group index

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

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

Deprecated since version 5.13.0.

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

Returns

t
[numpy.array] master channel samples

`info`() → dict[str, Any]

get MDF information as a dict

Examples

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

iter_get_triggers() → Iterator[TriggerInfoDict]

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: StrPathType*, *overwrite: bool = False*, *compression: CompressionType = 0*, *progress=None*, *add_history_block: bool = True*) → Path | None

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

Parameters

dst
[str | pathlib.Path] destination file name

overwrite
[bool] overwrite flag, default *False*

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

Returns

output_file
[str] output file name

property start_time: datetime

getter and setter the measurement start timestamp

Returns

timestamp
[datetime.datetime] start timestamp

2.2.1 MDF version 2 & 3 blocks

The following classes implement different MDF version3 blocks.

2.2.1.1 Channel Class

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

CNBLOCK class

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

CNBLOCK fields

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

Other attributes

- `address` - int : block address inside mdf file
- `comment` - str : channel comment
- `conversion` - ChannelConversion : channel conversion; *None* if the channel has no conversion
- `display_names` - dict : channel display names

..versionchanged:: 7.0.0

changed from str to dict

- `name` - str : full channel name
- `source` - SourceInformation : channel source information; *None* if the channel has no source information

Parameters

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

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

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

for dynamically created objects
see the key-value pairs

Examples

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

2.2.1.2 ChannelConversion Class

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

CCBLOCK class

ChannelConversion has the following common fields

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

ChannelConversion has the following specific fields

- linear conversion
  - `a` - float : factor
  - `b` - float : offset
  - `CANapeHiddenExtra` - bytes : sometimes CANape appends extra information; not compliant with MDF specs

- algebraic conversion
  - `formula` - bytes : equation as string
- polynomial or rational conversion
  - `P1` to `P6` - float : parameters
- exponential or logarithmic conversion
  - `P1` to `P7` - float : parameters
- tabular with or without interpolation (grouped by index)
  - `raw_<N>` - int : N-th raw value (X axis)
  - `phys_<N>` - float : N-th physical value (Y axis)
- text table conversion
  - `param_val_<N>` - int : N-th raw value (X axis)
  - `text_<N>` - N-th text physical value (Y axis)
- text range table conversion
  - `default_lower` - float : default lower raw value
  - `default_upper` - float : default upper raw value
  - `default_addr` - int : address of default text physical value
  - `lower_<N>` - float : N-th lower raw value
  - `upper_<N>` - float : N-th upper raw value
  - `text_<N>` - int : address of N-th text physical value

Other attributes

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

Parameters

address
[int] block address inside mdf file

raw_bytes
[bytes] complete block read from disk

stream
[file handle] mdf file handle

for dynamically created objects
see the key-value pairs

Examples

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

2.2.1.3 ChannelDependency Class

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

CDBLOCK class

CDBLOCK fields

- `id` - bytes : block ID; always b'CD'
- `block_len` - int : block bytes size
- `dependency_type` - int : integer code for dependency type
- `sd_nr` - int : total number of signals dependencies
- `dg_<N>` - address of data group block (DGBLOCK) of N-th signal dependency
- `dg_<N>` - address of channel group block (CGBLOCK) of N-th signal dependency
- `dg_<N>` - address of channel block (CNBLOCK) of N-th signal dependency
- `dim_<K>` - int : Optional size of dimension *K* for N-dimensional dependency

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

Parameters

stream
[file handle] mdf file handle

address
[int] block address inside mdf file

for dynamically created objects
see the key-value pairs

2.2.1.4 ChannelExtension Class

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

CEBLOCK class

CEBLOCK has the following common fields

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

CEBLOCK has the following specific fields

- for DIM block

- `module_nr` - int: module number
- `module_address` - int : module address
- `description` - bytes : module description
- `ECU_identification` - bytes : identification of ECU
- `reserved0` - bytes : reserved bytes

- for Vector CAN block
  - `CAN_id` - int : CAN message ID
  - `CAN_ch_index` - int : index of CAN channel
  - `message_name` - bytes : message name
  - `sender_name` - btyes : sender name
  - `reserved0` - bytes : reserved bytes

Other attributes

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

Parameters

stream
[file handle] mdf file handle

address
[int] block address inside mdf file

for dynamically created objects
see the key-value pairs

2.2.1.5 ChannelGroup Class

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

CGBLOCK class

CGBLOCK fields

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

- `cycles_nr` - int : number of cycles (records) of this type in the data block
- `sample_reduction_addr` - int : address to first sample reduction block

Other attributes

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

Parameters

stream
[file handle] mdf file handle

address
[int] block address inside mdf file

for dynamically created objects
see the key-value pairs

Examples

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

2.2.1.6 DataGroup Class

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

DGBLOCK class

DGBLOCK fields

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

Other attributes

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

Parameters

stream
[file handle] mdf file handle

address
[int] block address inside mdf file

for dynamically created objects
see the key-value pairs

2.2.1.7 FileIdentificationBlock Class

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

IDBLOCK class

IDBLOCK fields

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

Other attributes

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

Parameters

stream
[file handle] mdf file handle

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

2.2.1.8 HeaderBlock Class

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

HDBLOCK class

HDBLOCK fields

- `id` - bytes : block ID; always b'HD'
- `block_len` - int : block bytes size
- `first_dg_addr` - int : address of first data group block (DGBLOCK)
- `comment_addr` - int : address of TXBLOCK that contains the measurement file comment

- `program_addr` - int : address of program block (PRBLOCK)
- `dg_nr` - int : number of data groups
- `date` - bytes : date at which the recording was started in "DD:MM:YYYY" format
- `time` - btyes : time at which the recording was started in "HH:MM:SS" format
- `author_field` - bytes : author name
- `organization_field`` - bytes : organization name
- `project_field`` - bytes : project name
- `subject_field`` - bytes : subject

Since version 3.2 the following extra keys were added

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

Other attributes

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

Parameters

stream
[file handle] mdf file handle

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

2.2.1.9 ProgramBlock Class

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

PRBLOCK class

PRBLOCK fields

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

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

Parameters

stream
[file handle] mdf file handle

address
[int] block address inside mdf file

2.2.1.10 TextBlock Class

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

TXBLOCK class

TXBLOCK fields

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

Other attributes

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

Parameters

stream
[file handle] mdf file handle

address
[int] block address inside mdf file

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

Examples

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

2.2.1.11 TriggerBlock Class

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

TRBLOCK class

TRBLOCK fields

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

- `trigger_<N>_pretime` - float : pre trigger time [s] of trigger's N-th event
- `trigger_<N>_posttime` - float : post trigger time [s] of trigger's N-th event

Other attributes

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

Parameters

stream
[file handle] mdf file handle

address
[int] block address inside mdf file

2.3 MDF4

class asammdf.blocks.mdf_v4.**MDF4**(*name: BufferedReader | BytesIO | StrPathType | None = None*, *version: str = '4.10'*, *channels: list[str] | None = None*, ***kwargs*)

The *header* attibute is a *HeaderBlock*.

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

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

Parameters

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

version
[string] mdf file version ('4.00', '4.10', '4.11', '4.20'); default '4.10'

kwargs
use_display_names (True)
[bool] keyword only argument: for MDF4 files parse the XML channel comment to search

for the display name; XML parsing is quite expensive so setting this to *False* can decrease the loading times very much; default *True*

remove_source_from_channel_names (True)
[bool]

copy_on_get (True)
[bool] copy channel values (np.array) to avoid high memory usage

compact_vlsd (False)
[bool] use slower method to save the exact sample size for VLSD channels

column_storage (True)
[bool] use column storage for MDF version >= 4.20

password
[bytes | str] use this password to decode encrypted attachments

Attributes

attachments
[list] list of file attachments

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

events
[list] list event blocks

file_comment
[TextBlock] file comment TextBlock

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

groups
[list] list of data group dicts

header
[HeaderBlock] mdf file header

identification
[FileIdentificationBlock] mdf file start block

last_call_info
[dict | None] a dict to hold information about the last called method.

New in version 5.12.0.

masters_db
[dict]

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

name
[string] mdf file name

version
[str] mdf version

append(*signals: list[Signal] | Signal | DataFrame*, *acq_name: str | None = None*, *acq_source: Source | None = None*, *comment: str = 'Python'*, *common_timebase: bool = False*, *units: dict[str, str | bytes] | None = None*) → int | None

Appends a new data group.

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

Parameters

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

acq_name
[str] channel group acquisition name

acq_source
[Source] channel group acquisition source

comment
[str] channel group comment; default 'Python'

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

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

Examples

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

(continues on next page)

(continued from previous page)

```
>>> units = {'s1': 'V', 's2': 'A'}
>>> mdf2.append(df, units=units)
```

attach(*data: bytes*, *file_name: str | None = None*, *hash_sum: bytes | None = None*, *comment: str = ''*, *compression: bool = True*, *mime: str = 'application/octet-stream'*, *embedded: bool = True*, *password: str | bytes | None = None*) → int

attach embedded attachment as application/octet-stream.

Parameters

data
[bytes] data to be attached

file_name
[str] string file name

hash_sum
[bytes] md5 of the data

comment
[str] attachment comment

compression
[bool] use compression for embedded attachment data

mime
[str] mime type string

embedded
[bool] attachment is embedded in the file

password
[str | bytes | None , default None] password used to encrypt the data using AES256 encryption

New in version 7.0.0.

Returns

index
[int] new attachment index

close() → None

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

extend(*index: int*, *signals: list[tuple[NDArray[Any], NDArray[Any] | None]]*) → None

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

Parameters

index
[int] group index

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

Examples

```
>>> # case 1 conversion type None
>>> s1 = np.array([1, 2, 3, 4, 5])
>>> s2 = np.array([-1, -2, -3, -4, -5])
>>> s3 = np.array([0.1, 0.04, 0.09, 0.16, 0.25])
>>> t = np.array([0.001, 0.002, 0.003, 0.004, 0.005])
>>> names = ['Positive', 'Negative', 'Float']
>>> units = ['+', '-', '.f']
>>> s1 = Signal(samples=s1, 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 = MDF4('new.mdf')
>>> mdf.append([s1, s2, s3], comment='created by asammdf v1.1.0')
>>> t = np.array([0.006, 0.007, 0.008, 0.009, 0.010])
>>> # extend without invalidation bits
>>> mdf2.extend(0, [(t, None), (s1, None), (s2, None), (s3, None)])
>>> # some invaldiation btis
>>> s1_inv = np.array([0,0,0,1,1], dtype=np.bool)
>>> mdf2.extend(0, [(t, None), (s1.samples, None), (s2.samples, None), (s3.
↪samples, None)])
```

extract_attachment(*index: int | None = None*, *password: str | bytes | None = None*) → tuple[bytes, Path, bytes]

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

Parameters

index
[int] attachment index; default *None*

password
[str | bytes | None, default None] password used to encrypt the data using AES256 encryption

New in version 7.0.0.

Returns

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

get(*name: str | None = None*, *group: int | None = None*, *index: int | None = None*, *raster: RasterType | None = None*, *samples_only: Literal[False] = False*, *data: bytes | None = None*, *raw: bool = False*, *ignore_invalidation_bits: bool = False*, *record_offset: int = 0*, *record_count: int | None = None*, *skip_channel_validation: bool = False*) → *Signal*

get(*name: str | None = None*, *group: int | None = None*, *index: int | None = None*, *raster: RasterType | None = None*, *samples_only: Literal[True] = False*, *data: bytes | None = None*, *raw: bool = False*, *ignore_invalidation_bits: bool = False*, *record_offset: int = 0*, *record_count: int | None = None*, *skip_channel_validation: bool = False*) → tuple[NDArray[Any], NDArray[Any]]

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

Channel can be specified in two ways:

- using the first positional argument *name*

  - if 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

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

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

ignore_invalidation_bits
[bool] option to ignore invalidation bits

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

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

skip_channel_validation (False)
[bool] skip validation of channel name, group index and channel index; defualt *False*. If *True*, the caller has to make sure that the *group* and *index* arguments are provided and are correct.

..versionadded:: 7.0.0

Returns

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

The *Signal* samples are:

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

Raises

MdfException

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

Examples

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

(continues on next page)

(continued from previous page)

```
        comment="">
>>> # channel index 1 or group 2
...
>>> mdf.get(None, 2, 1)
<Signal Sig:
        samples=[ 21.  21.  21.  21.  21.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
>>> mdf.get(group=2, index=1)
<Signal Sig:
        samples=[ 21.  21.  21.  21.  21.]
        timestamps=[0 1 2 3 4]
        unit=""
        info=None
        comment="">
```

get_bus_signal(*bus: BusType*, *name: str*, *database: CanMatrix | StrPathType | None = None*, *ignore_invalidation_bits: bool = False*, *data: bytes | None = None*, *raw: bool = False*, *ignore_value2text_conversion: bool = True*) → *Signal*

get a signal decoded from a raw bus logging. The currently supported buses are CAN and LIN (LDF databases are not supported, they need to be converted to DBC and feed to this function)

New in version 6.0.0.

Parameters

bus
[str] "CAN" or "LIN"

name
[str] signal name

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

Changed in version 6.0.0: *db* and *database* arguments were merged into this single argument

ignore_invalidation_bits
[bool] option to ignore invalidation bits

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

ignore_value2text_conversion
[bool] return channel samples without values that have a description in .dbc or .arxml file *True*

Returns

sig
[Signal] Signal object with the physical values

get_can_signal(*name: str*, *database: CanMatrix | StrPathType | None = None*, *ignore_invalidation_bits: bool = False*, *data: bytes | None = None*, *raw: bool = False*, *ignore_value2text_conversion: bool = True*) → *Signal*

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

The signal name can be specified in the following ways

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

Parameters

name
[str] signal name

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

Changed in version 6.0.0: *db* and *database* arguments were merged into this single argument

ignore_invalidation_bits
[bool] option to ignore invalidation bits

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

ignore_value2text_conversion
[bool] return channel samples without values that have a description in .dbc or .arxml file *True*

Returns

sig
[Signal] Signal object with the physical values

`get_channel_comment`(*name: str | None = None*, *group: int | None = None*, *index: int | None = None*) → str

Gets channel comment.

Channel can be specified in two ways:

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

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

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

Parameters

name
[string] name of channel

group
[int] 0-based group index

index
[int] 0-based channel index

Returns

comment
[str] found channel comment

`get_channel_name`(*group: int*, *index: int*) → str

Gets channel name.

Parameters

group
[int] 0-based group index

index
[int] 0-based channel index

Returns

name
[str] found channel name

`get_channel_unit`(*name: str | None = None*, *group: int | None = None*, *index: int | None = None*) → str

Gets channel unit.

Channel can be specified in two ways:

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

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

Parameters

name
[string] name of channel

group
[int] 0-based group index

index
[int] 0-based channel index

Returns

unit
[str] found channel unit

`get_invalidation_bits`(*group_index: int*, *channel:* Channel, *fragment: tuple[bytes, int, int, ReadableBufferType | None]*) → NDArray[bool_]

get invalidation indexes for the channel

Parameters

group_index
[int] group index

channel
[Channel] channel object

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

Returns

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

`get_lin_signal`(*name: str*, *database: CanMatrix | StrPathType | None = None*, *ignore_invalidation_bits: bool = False*, *data: bytes | None = None*, *raw: bool = False*, *ignore_value2text_conversion: bool = True*) → *Signal*

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

The signal name can be specified in the following ways

- `LIN_Frame_<MESSAGE_ID>.<SIGNAL_NAME>` - Example: LIN_Frame_218.FL_WheelSpeed
- `<MESSAGE_NAME>.<SIGNAL_NAME>` - Example: Wheels.FL_WheelSpeed
- `<SIGNAL_NAME>` - Example: FL_WheelSpeed

New in version 6.0.0.

Parameters

name
[str] signal name

database
[str] path of external LIN database file (.dbc, .arxml or .ldf) or canmatrix.CanMatrix; default *None*

ignore_invalidation_bits
[bool] option to ignore invalidation bits

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

ignore_value2text_conversion
[bool] return channel samples without values that have a description in .dbc, .arxml or .ldf file *True*

Returns

sig
[Signal] Signal object with the physical values

get_master(*index: int*, *data: bytes | None = None*, *raster: RasterType | None = None*, *record_offset: int = 0*, *record_count: int | None = None*, *one_piece: bool = False*) → NDArray[Any]

returns master channel samples for given group

Parameters

index
[int] group index

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

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

Deprecated since version 5.13.0.

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

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

Returns

t
[numpy.array] master channel samples

info() → dict[str, Any]

get MDF information as a dict

Examples

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

save(*dst: WritableBufferType | StrPathType*, *overwrite: bool = False*, *compression: CompressionType = 0*, *progress=None*, *add_history_block: bool = True*) → Path

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

Parameters

dst
[str] 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)

add_history_block
[bool] option to add file historyu block

Returns

output_file
[pathlib.Path] path to saved file

`property start_time: datetime`

getter and setter the measurement start timestamp

Returns

timestamp
[datetime.datetime] start timestamp

2.3.1 MDF version 4 blocks

The following classes implement different MDF version4 blocks.

2.3.1.1 AttachmentBlock Class

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

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

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

ATBLOCK fields

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

Other attributes

- `address` - int : attachment address

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

Parameters

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

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

for dynamically created objects
see the key-value pairs

`extract()` → bytes

extract attachment data

Returns

data
[bytes]

2.3.1.2 Channel Class

`class asammdf.blocks.v4_blocks.Channel(**kwargs)`

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

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

CNBLOCK fields

- `id` - bytes : block ID; always b'##CN'
- `reserved0` - int : reserved bytes
- `block_len` - int : block bytes size
- `links_nr` - int : number of links
- `next_ch_addr` - int : next ATBLOCK address
- `component_addr` - int : address of first channel in case of structure channel composition, or ChannelArrayBlock in case of arrays file name
- `name_addr` - int : address of TXBLOCK that contains the channel name
- `source_addr` - int : address of channel source block
- `conversion_addr` - int : address of channel conversion block
- `data_block_addr` - int : address of signal data block for VLSD channels
- `unit_addr` - int : address of TXBLOCK that contains the channel unit
- `comment_addr` - int : address of TXBLOCK/MDBLOCK that contains the channel comment
- `attachment_<N>_addr` - int : address of N-th ATBLOCK referenced by the current channel; if no ATBLOCK is referenced there will be no such key-value pair

- `default_X_dg_addr` - int : address of DGBLOCK where the default X axis channel for the current channel is found; this key-value pair will not exist for channels that don't have a default X axis
- `default_X_cg_addr` - int : address of CGBLOCK where the default X axis channel for the current channel is found; this key-value pair will not exist for channels that don't have a default X axis
- `default_X_ch_addr` - int : address of default X axis channel for the current channel; this key-value pair will not exist for channels that don't have a default X axis
- `channel_type` - int : integer code for the channel type
- `sync_type` - int : integer code for the channel's sync type
- `data_type` - int : integer code for the channel's data type
- `bit_offset` - int : bit offset
- `byte_offset` - int : byte offset within the data record
- `bit_count` - int : channel bit count
- `flags` - int : CNBLOCK flags
- `pos_invalidation_bit` - int : invalidation bit position for the current channel if there are invalidation bytes in the data record
- `precision` - int : integer code for the precision
- `reserved1` - int : reserved bytes
- `min_raw_value` - int : min raw value of all samples
- `max_raw_value` - int : max raw value of all samples
- `lower_limit` - int : min physical value of all samples
- `upper_limit` - int : max physical value of all samples
- `lower_ext_limit` - int : min physical value of all samples
- `upper_ext_limit` - int : max physical value of all samples

Other attributes

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

Parameters

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

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

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

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

for dynamically created objects
see the key-value pairs

2.3.1.3 ChannelConversion Class

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

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

CCBLOCK common fields

- `id` - bytes : block ID; always b'##CG'
- `reserved0` - int : reserved bytes
- `block_len` - int : block bytes size
- `links_nr` - int : number of links
- `name_addr` - int : address of TXBLOCK that contains the conversion name
- `unit_addr` - int : address of TXBLOCK that contains the conversion unit
- `comment_addr` - int : address of TXBLOCK/MDBLOCK that contains the conversion comment
- `inv_conv_addr` int : address of inverse conversion
- `conversion_type` int : integer code for conversion type
- `precision` - int : integer code for precision
- `flags` - int : conversion block flags
- `ref_param_nr` - int : number fo referenced parameters (linked parameters)
- `val_param_nr` - int : number of value parameters
- `min_phy_value` - float : minimum physical channel value
- `max_phy_value` - float : maximum physical channel value

CCBLOCK specific fields

- linear conversion
  - `a` - float : factor
  - `b` - float : offset
- rational conversion
  - `P1` to `P6` - float : parameters
- algebraic conversion
  - `formula_addr` - address of TXBLOCK that contains the algebraic conversion formula
- tabular conversion with or without interpolation
  - `raw_<N>` - float : N-th raw value
  - `phys_<N>` - float : N-th physical value

- tabular range conversion
  - lower_<N> - float : N-th lower value
  - upper_<N> - float : N-th upper value
  - phys_<N> - float : N-th physical value
- tabular value to text conversion
  - val_<N> - float : N-th raw value
  - text_<N> - int : address of N-th TXBLOCK that contains the physical value
  - default - int : address of TXBLOCK that contains the default physical value
- tabular range to text conversion
  - lower_<N> - float : N-th lower value
  - upper_<N> - float : N-th upper value
  - text_<N> - int : address of N-th TXBLOCK that contains the physical value
  - default - int : address of TXBLOCK that contains the default physical value
- text to value conversion
  - val_<N> - float : N-th physical value
  - text_<N> - int : address of N-th TXBLOCK that contains the raw value
  - val_default - float : default physical value
- text transformation (translation) conversion
  - input_<N>_addr - int : address of N-th TXBLOCK that contains the raw value
  - output_<N>_addr - int : address of N-th TXBLOCK that contains the physical value
  - default_addr - int : address of TXBLOCK that contains the default physical value

Other attributes

- address - int : channel conversion address
- comment - str : channel conversion comment
- formula - str : algebraic conversion formula; default ‘’
- referenced_blocks - dict : dict of referenced blocks; can be TextBlock objects for value to text, and text to text conversions; for partial conversions the referenced blocks can be ChannelConversion object as well
- name - str : channel conversion name
- unit - str : channel conversion unit

2.3.1.4 ChannelGroup Class

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

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

CGBLOCK fields

- `id` - bytes : block ID; always b'##CG'
- `reserved0` - int : reserved bytes
- `block_len` - int : block bytes size
- `links_nr` - int : number of links
- `next_cg_addr` - int : next channel group address
- `first_ch_addr` - int : address of first channel of this channel group
- `acq_name_addr` - int : address of TextBLock that contains the channel group acquisition name
- `acq_source_addr` - int : address of SourceInformation that contains the channel group source
- `first_sample_reduction_addr` - int : address of first SRBLOCK; this is considered 0 since sample reduction is not yet supported
- `comment_addr` - int : address of TXBLOCK/MDBLOCK that contains the channel group comment
- `record_id` - int : record ID for the channel group
- `cycles_nr` - int : number of cycles for this channel group
- `flags` - int : channel group flags
- `path_separator` - int : ordinal for character used as path separator
- `reserved1` - int : reserved bytes
- `samples_byte_nr` - int : number of bytes used for channels samples in the record for this channel group; this does not contain the invalidation bytes
- `invalidation_bytes_nr` - int : number of bytes used for invalidation bits by this channel group

Other attributes

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

2.3.1.5 DataGroup Class

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

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

DGBLOCK fields

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

- next_dg_addr - int : address of next data group block
- first_cg_addr - int : address of first channel group for this data group
- data_block_addr - int : address of DTBLOCK, DZBLOCK, DLBLOCK or HLBLOCK that contains the raw samples for this data group
- comment_addr - int : address of TXBLOCK/MDBLOCK that contains the data group comment
- record_id_len - int : size of record ID used in case of unsorted data groups; can be 1, 2, 4 or 8
- reserved1 - int : reserved bytes

Other attributes

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

2.3.1.6 DataList Class

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

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

DLBLOCK common fields

- id - bytes : block ID; always b'##DL'
- reserved0 - int : reserved bytes
- block_len - int : block bytes size
- links_nr - int : number of links
- next_dl_addr - int : address of next DLBLOCK
- data_block_addr<N> - int : address of N-th data block
- flags - int : data list flags
- reserved1 - int : reserved bytes
- data_block_nr - int : number of data blocks referenced by this list

DLBLOCK specific fields

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

Other attributes

- address - int : data list address

2.3.1.7 DataBlock Class

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

Common implementation for DTBLOCK/RDBLOCK/SDBLOCK/DVBLOCK/DIBLOCK

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

DTBLOCK fields

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

Other attributes

- `address` - int : data block address

Parameters

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

stream
[int] file handle

reduction
[bool] sample reduction data block

2.3.1.8 FileIdentificationBlock Class

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

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

IDBLOCK fields

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

Other attributes

- `address` - int : should always be 0

2.3.1.9 HeaderBlock Class

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

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

HDBLOCK fields

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

Other attributes

- `address` - int : header address
- `comment` - str : file comment
- `author` - str : measurement author
- `department` - str : author's department
- `project` - str : working project
- `subject` - str : measurement subject

property start_time: datetime

getter and setter the measurement start timestamp

Returns

timestamp
[datetime.datetime] start timestamp

2.3.1.10 SourceInformation Class

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

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

SIBLOCK fields

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

Other attributes

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

2.3.1.11 FileHistory Class

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

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

FHBLOCK fields

- `id` - bytes : block ID; always b'##FH'
- `reserved0` - int : reserved bytes
- `block_len` - int : block bytes size
- `links_nr` - int : number of links
- `next_fh_addr` - int : address of next FHBLOCK
- `comment_addr` - int : address of TXBLOCK/MDBLOCK that contains the file history comment
- `abs_time` - int : time stamp at which the file modification happened
- `tz_offset` - int : UTC time offset in hours (= GMT time zone)
- `daylight_save_time` - int : daylight saving time
- `time_flags` - int : time flags
- `reserved1` - bytes : reserved bytes

Other attributes

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

property time_stamp: datetime

getter and setter the file history timestamp

Returns

timestamp
[datetime.datetime] start timestamp

2.3.1.12 TextBlock Class

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

common TXBLOCK and MDBLOCK class

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

TXBLOCK fields

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

Other attributes

- `address` - int : text block address

Parameters

address
[int] block address

stream
[handle] file handle

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

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

2.3.1.13 EventBlock Class

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

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

EVBLOCK fields

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

Other attributes

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

2.4 Signal

class asammdf.signal.**Signal**(*samples: ArrayLike | None = None*, *timestamps: ArrayLike | None = None*, *unit: str = ''*, *name: str = ''*, *conversion: dict[str, Any] | ChannelConversionType | None = None*, *comment: str = ''*, *raw: bool = True*, *master_metadata: tuple[str, SyncType] | None = None*, *display_names: dict[str, str] | None = None*, *attachment: tuple[bytes, str | None, str | None] | None = None*, *source: SourceType | None = None*, *bit_count: int | None = None*, *invalidation_bits: ArrayLike | None = None*, *encoding: str | None = None*, *group_index: int = -1*, *channel_index: int = -1*, *flags: Flags = SignalFlags.no_flags*)

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

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

comment
[str] signal comment, default ''

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

master_metadata
[list] master name and sync type

display_names
[dict] display names used by mdf version 3

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

source
[Source] source information named tuple

bit_count
[int] bit count; useful for integer channels

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

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

flags
[Signal.Flags] flags for user defined attributes and stream sync

`Flags`

alias of `SignalFlags`

`astype`(*np_type: dtype[Any] | None | Type[Any] | _SupportsDType[dtype[Any]] | str | Tuple[Any, int] | Tuple[Any, SupportsIndex | Sequence[SupportsIndex]] | List[Any] | _DTypeDict | Tuple[Any, Any]*) → *Signal*

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

Parameters

np_type
[np.dtype] new numpy dtye

Returns

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

`copy`() → *Signal*

copy all attributes to a new Signal

`cut`(*start: float | None = None*, *stop: float | None = None*, *include_ends: bool = True*, *integer_interpolation_mode: IntInterpolationModeType | IntegerInterpolation = IntegerInterpolation.REPEAT_PREVIOUS_SAMPLE*, *float_interpolation_mode: FloatInterpolationModeType | FloatInterpolation = FloatInterpolation.LINEAR_INTERPOLATION*) → *Signal*

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

Parameters

start
[float] start timestamp for cutting

stop
[float] stop timestamp for cutting

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

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

- 0 - repeat previous samples
- 1 - linear interpolation
- 2 - hybrid interpolation: channels with integer data type (raw values) that have a conversion that outputs float values will use linear interpolation, otherwise the previous sample is used

New in version 6.2.0.

float_interpolation_mode
[int] interpolation mode for float channels; default 1

- 0 - repeat previous sample

- 1 - use linear interpolation

New in version 6.2.0.

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:* Signal) → *Signal*

extend signal with samples from another signal

Parameters

other
[Signal]

Returns

signal
[Signal] new extended *Signal*

interp(*new_timestamps: NDArray[Any]*, *integer_interpolation_mode: IntInterpolationModeType | IntegerInterpolation = IntegerInterpolation.REPEAT_PREVIOUS_SAMPLE*, *float_interpolation_mode: FloatInterpolationModeType | FloatInterpolation = FloatInterpolation.LINEAR_INTERPOLATION*) → *Signal*

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

Parameters

new_timestamps
[np.array] timestamps used for interpolation

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

- 0 - repeat previous samples
- 1 - linear interpolation
- 2 - hybrid interpolation: channels with integer data type (raw values) that have a conversion that outputs float values will use linear interpolation, otherwise the previous sample is used

New in version 6.2.0.

float_interpolation_mode
[int] interpolation mode for float channels; default 1

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

New in version 6.2.0.

Returns

signal
[Signal] new interpolated *Signal*

`physical`() → *Signal*

get the physical samples values

Returns

phys
[Signal] new *Signal* with physical values

`plot`(*validate: bool = True*, *index_only: bool = False*) → None

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

Parameters

validate (True): bool
apply the invalidation bits

index_only (False)
[bool] use index based X axis. This can be useful if the master (usually time based) is corrupted with NaN, inf or if it is not strictly increasing

`validate`(*copy: bool = True*) → *Signal*

appply invalidation bits if they are available for this signal

Parameters

copy (True)
[bool] return a copy of the result

New in version 5.12.0.

CHAPTER

THREE

BUS LOGGING

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

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

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

There multiple ways to address this channel in this situation:

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

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

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

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

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

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

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

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

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

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

CHAPTER

FOUR

TIPS

4.1 Selective channel loading

Since asammdf 6.1.0 you can use the **channels** argument to perform a selective channel loading. Only the metadata of channels found in the selection iterable will be preserved after loading the file. This can yield a big speed improvement for loading the file, but also when performing operations with the MDF object (for example `select`, `get`, etc.).

```
required_channels = ["Speed", "Acceleration", "Force"]
mdf = MDF("input.mf4", channels=required_channels)
```

4.2 Data coupling

All the data returned by the MDF methods is decoupled from the raw data found in the original file. For example, if you modify the Signal returned by the `get` call the changes will not be seen in the raw data. A second `get` call of the same channel name will once again give you the data found in the original file.

4.3 Chunked data access

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

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

4.4 Optimized methods

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

Each *get* call will read all channel group raw samples from disk. If you need to extract multiple channels it is strongly advised to use the *select* method: for each channel group that contains channels submitted for selection, the raw samples will only be read once.

4.5 Faster file loading

4.5.1 Skip XML parsing for MDF4 files

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

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

CHAPTER

FIVE

EXAMPLES

5.1 Working with MDF

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


# create 3 Signal objects

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
    print(mdf3.version)

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

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

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

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

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

5.2 Working with Signal

```
from asammdf import Signal
import numpy as np


# create 3 Signal objects with different time stamps

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

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

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

(continues on next page)

(continued from previous page)

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

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

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


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

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

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

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

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

5.3 MF4 demo file generator

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

cycles = 100
sigs = []

mdf = MDF()

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

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

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


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

(continues on next page)

(continued from previous page)

```
# rational
conversion = {
    'P1': 0,
    'P2': 4,
    'P3': -0.5,
    'P4': 0,
    'P5': 0,
    'P6': 1,
}
sig = Signal(
    np.ones(cycles, dtype=np.int64),
    t,
    name='Channel_rational_conversion',
    unit='Nm',
    conversion=conversion,
    comment='Channel with rational conversion',
)
sigs.append(sig)

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

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

# tabular
vals = 20
conversion = {
    'raw_{}'.format(i): i
    for i in range(vals)
}
conversion.update(
    {
```

(continues on next page)

(continued from previous page)

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

# value to text
vals = 20
conversion = {
    'val_{}'.format(i): i
    for i in range(vals)
}
conversion.update(
    {
        'text_{}'.format(i): 'key_{}'.format(i).encode('ascii')
        for i in range(vals)
    }
)
conversion['default'] = b'default key'
sig = Signal(
    np.arange(cycles, dtype=np.uint32) % 30,
    t,
    name='Channel_value_to_text',
    conversion=conversion,
    comment='Value to text channel',
)
sigs.append(sig)

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

(continues on next page)

(continued from previous page)

```
    }
)
conversion['default'] = -1
sig = Signal(
    2 * np.arange(cycles, dtype=np.float64),
    t,
    name='Channel_value_range_to_value',
    unit='order',
    conversion=conversion,
    comment='Value range to value channel',
)
sigs.append(sig)

# value range to text
vals = 20
conversion = {
    'lower_{}'.format(i): i * 10
    for i in range(vals)
}
conversion.update(
    {
        'upper_{}'.format(i): (i + 1) * 10
        for i in range(vals)
    }
)
conversion.update(
    {
        'text_{}'.format(i): 'Level {}'.format(i)
        for i in range(vals)
    }
)
conversion['default'] = b'Unknown level'
sig = Signal(
    6 * np.arange(cycles, dtype=np.uint64) % 240,
    t,
    name='Channel_value_range_to_text',
    conversion=conversion,
    comment='Value range to text channel',
)
sigs.append(sig)


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



sigs = []

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

(continues on next page)

(continued from previous page)

```
    np.ones((cycles, 3), dtype=np.uint64) * 3,
]

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

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

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

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

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

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

types = [
    ('struct_channel_0', np.uint8),
    ('struct_channel_1', np.uint16),
    ('struct_channel_2', np.uint32),
```

(continues on next page)

(continued from previous page)

```
    ('struct_channel_3', np.uint64),
    ('struct_channel_4', np.int8),
    ('struct_channel_5', np.int16),
    ('struct_channel_6', np.int32),
    ('struct_channel_7', np.int64),
]

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


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

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

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

l3_arr = [
    l4_arr,
    l4_arr,
    l4_arr,
]

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

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


l2_arr = [
    l3_arr,
    l3_arr,
]
```

(continues on next page)

(continued from previous page)

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

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


l1_arr = [
    l2_arr,
]

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

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


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

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

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

CHAPTER

SIX

BENCHMARKS

6.1 Test setup

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

asamdf 7.0.1 was compared against *mdfreader 4.1*.

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

6.1.1 Dependencies

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

- psutil
- mdfreader

6.1.2 Usage

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

6.2 x64 Python results

Benchmark environment

- 3.9.4 (tags/v3.9.4:1f2e308, Apr 6 2021, 13:40:21) [MSC v.1928 64 bit (AMD64)]
- Windows-10-10.0.19041-SP0
- Intel64 Family 6 Model 158 Stepping 10, GenuineIntel
- numpy 1.21.2
- 16GB installed RAM

Notations used in the results

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

Files used for benchmark:

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

Open file	Time [ms]	RAM [MB]
asammdf 7.0.1 mdfv3	369	186
mdfreader 4.1 mdfv3	1741	498
mdfreader 4.1 no_data_loading mdfv3	646	248
mdfreader 4.1 compress mdfv3	1463	365
asammdf 7.0.1 mdfv4	468	199
mdfreader 4.1 mdfv4	4350	520
mdfreader 4.1 no_data_loading mdfv4	2892	310
mdfreader 4.1 compress mdfv4	4105	391

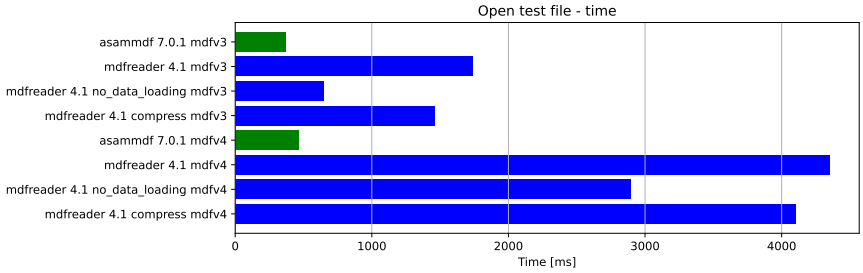
Save file	Time [ms]	RAM [MB]
asammdf 7.0.1 mdfv3	378	186
mdfreader 4.1 mdfv3	4310	527
mdfreader 4.1 no_data_loading mdfv3	5070	586
mdfreader 4.1 compress mdfv3	4456	525
asammdf 7.0.1 mdfv4	331	366
mdfreader 4.1 mdfv4	2254	539
mdfreader 4.1 no_data_loading mdfv4	3591	618
mdfreader 4.1 compress mdfv4	2400	535

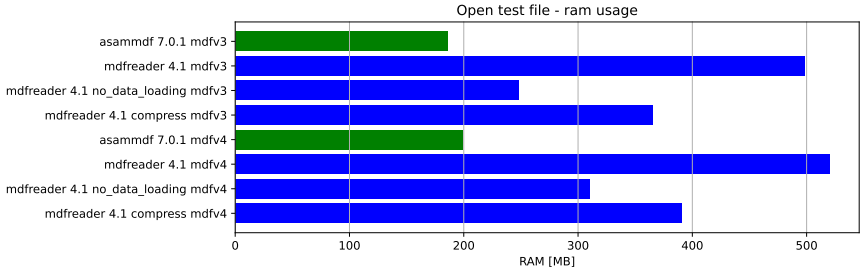
Get all channels (36424 calls)	Time [ms]	RAM [MB]
asammdf 7.0.1 mdfv3	3354	187
mdfreader 4.1 mdfv3	40	498
mdfreader 4.1 nodata mdfv3	12686	283
mdfreader 4.1 compress mdfv3	154	366
asammdf 7.0.1 mdfv4	5243	364
mdfreader 4.1 mdfv4	51	520
mdfreader 4.1 nodata mdfv4	20210	336
mdfreader 4.1 compress mdfv4	170	396

Convert file	Time [ms]	RAM [MB]
asammdf 7.0.1 v3 to v4	2186	232
asammdf 7.0.1 v4 to v410	2008	394
asammdf 7.0.1 v4 to v420	2359	438

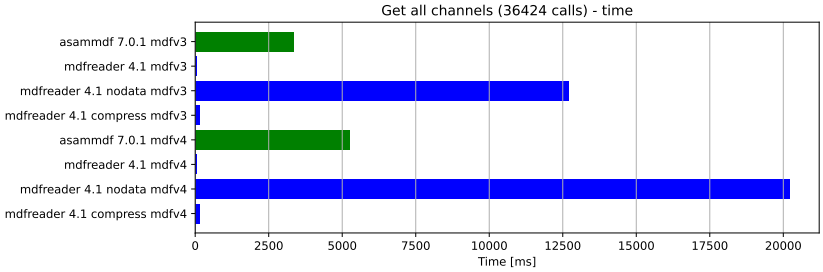
Merge 3 files	Time [ms]	RAM [MB]
asammdf 7.0.1 v3	6449	224
mdfreader 4.1 v3	0*	0*
mdfreader 4.1 nodata v3	0*	0*
mdfreader 4.1 compress v3	0*	0*
asammdf 7.0.1 v4	6713	409
mdfreader 4.1 v4	34746	1156
mdfreader 4.1 nodata v4	37608	1266
mdfreader 4.1 compress v4	34184	1151

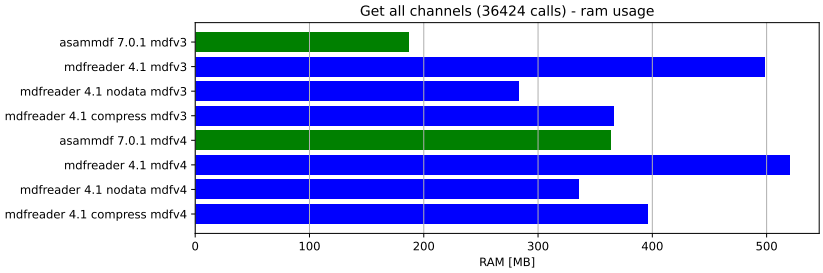
6.2.1 Graphical results

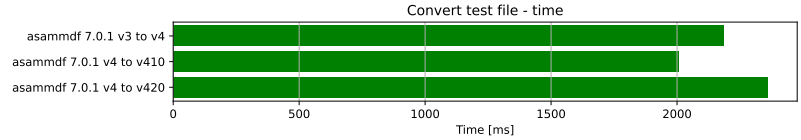


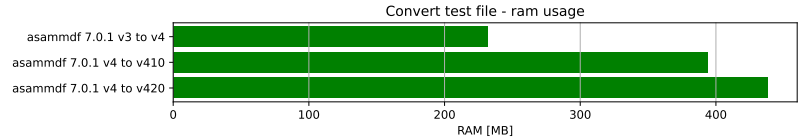


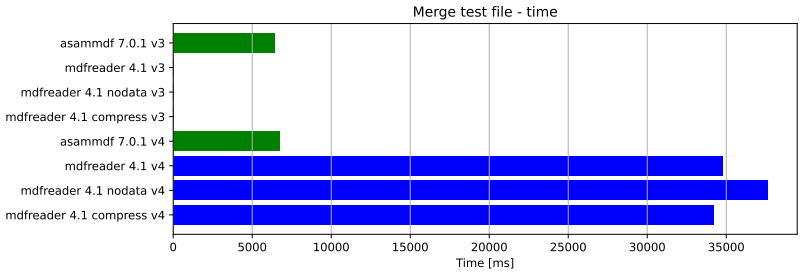

Save test file - ram usage
asammdf 7.0.1 mdfv3
mdfreader 4.1 mdfv3
mdfreader 4.1 no_data_loading mdfv3
mdfreader 4.1 compress mdfv3
asammdf 7.0.1 mdfv4
mdfreader 4.1 mdfv4
mdfreader 4.1 no_data_loading mdfv4
mdfreader 4.1 compress mdfv4
0
100
200
300
400
500
600
RAM [MB]

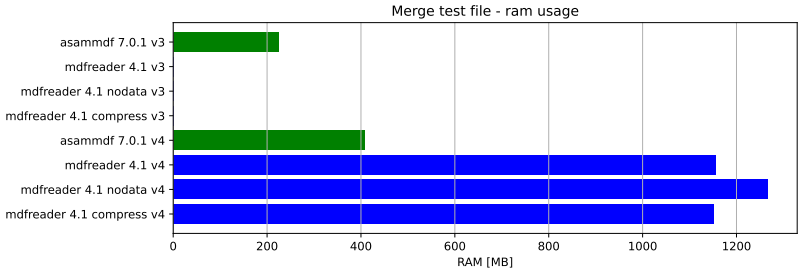

Get all channels (36424 calls) - time
asammdf 7.0.1 mdfv3
mdfreader 4.1 mdfv3
mdfreader 4.1 nodata mdfv3
mdfreader 4.1 compress mdfv3
asammdf 7.0.1 mdfv4
mdfreader 4.1 mdfv4
mdfreader 4.1 nodata mdfv4
mdfreader 4.1 compress mdfv4
0
2500
5000
7500
10000
12500
15000
17500
20000
Time [ms]


Get all channels (36424 calls) - ram usage
asammdf 7.0.1 mdfv3
mdfreader 4.1 mdfv3
mdfreader 4.1 nodata mdfv3
mdfreader 4.1 compress mdfv3
asammdf 7.0.1 mdfv4
mdfreader 4.1 mdfv4
mdfreader 4.1 nodata mdfv4
mdfreader 4.1 compress mdfv4
0
100
200
300
400
500
RAM [MB]


Convert test file - time
asammdf 7.0.1 v3 to v4
asammdf 7.0.1 v4 to v410
asammdf 7.0.1 v4 to v420
0
500
1000
1500
2000
Time [ms]


Convert test file - ram usage
asammdf 7.0.1 v3 to v4
asammdf 7.0.1 v4 to v410
asammdf 7.0.1 v4 to v420
0
100
200
300
400
RAM [MB]


Merge test file - time
asammdf 7.0.1 v3
mdfreader 4.1 v3
mdfreader 4.1 nodata v3
mdfreader 4.1 compress v3
asammdf 7.0.1 v4
mdfreader 4.1 v4
mdfreader 4.1 nodata v4
mdfreader 4.1 compress v4
0
5000
10000
15000
20000
25000
30000
35000
Time [ms]


Merge test file - ram usage
asammdf 7.0.1 v3
mdfreader 4.1 v3
mdfreader 4.1 nodata v3
mdfreader 4.1 compress v3
asammdf 7.0.1 v4
mdfreader 4.1 v4
mdfreader 4.1 nodata v4
mdfreader 4.1 compress v4
0
200
400
600
800
1000
1200
RAM [MB]

CHAPTER

SEVEN

GUI

With the GUI tool you can

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

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

The following dependencies are required by the GUI

- PyQt5
- pyqtgraph

7.1 General shortcuts

Shortcut	Action	Description
F1	Help	Opens this online help page
F2	Create plot	Create a new Plot window using the checked channels from the selection tree
F3	Create numeric	Create a new Numeric window using the checked channels from the selection tree
F4	Create tabular	Create a new Tabular window using the checked channels from the selection tree
F11	Toggle fullscreen	In the single files mode will display the current opened file in full screen
Ctrl+O	Open file(s)	

7.2 Menu

7.2.1 File

The first menu command is *Open*. Depending on the mode this allows to open files individually or for batch processing.

The second menu command is *Open folder*. If this is selected then, starting with the selected root folder, all sub-folders are searched recursively for MDF files.

Once a file has been opened, the user can load or save a display configuration using the *Open configuration* and *Save configuration* menu items.

7.2.2 Mode

- *Single files* : files are opened individually
- *Batch processing* : allows processing multiple files
- *Comparison* : show channels from all the opened files

7.2.3 Settings

The following settings are available

- **Sub-windows**: controls if multiple subplots will be created when the plot button is pressed
  - `Disabled`: a single plot is used that is overwritten
  - `Enabled`: a new subplot is added
- **Link sub-windows X-axis**: controls the subplots are linked on the X axis (zooming will affect all sub-windows)
  - `Disabled`
  - `Enabled`
- **Ignore value2text conversions**: do not apply the value to text conversions
  - `Disabled`
  - `Enabled`
- **Plot background**: switch plot background color (does not affect existing plots)
  - `Black`
  - `White`
- **Plot X axis**: select how the X axis will be displayed
  - **`seconds`**
    - `time` : values will be formatted as hours, minutes and seconds
  - `date` : the values will use the measurement start datetime
- **Theme**: switch application theme
  - `Dark`
  - `Light`
- **Step mode**: controls how the signal samples are interconnected visually in the Plot
- **Integer interpolation**: selects the way integer channels are interpolated
  - `0 - repeat previous sample`
  - `1 - linear interpolation`
  - `2 - hybrid interpolation`
- **Float interpolation**: selects the way float channels are interpolated
  - `0 - repeat previous sample`
  - `1 - linear interpolation`

The settings are saved and restored each time the GUI is started.

7.2.4 Plot

There are several keyboard shortcuts for handling the plots:

Shortcut	Action	Description
C	Edit signal color	Opens a dialog to select the signal color[7]
F	Fit all	Y-axis fit all active channels on the screen, keeping the current X-axis range
Shift+F	Fit selected	Y-axis fit all active selected channels on the screen; unselected channels are not fitted
G	Grid	Toggle grid lines
W	Home	Whole signals XY-axis fit
H	Honeywell	zomm in on the current cursor position using a constant zoom level of 0.1s/cm (real display width)
I	Zoom-in	X-axis zoom-in[1]
O	Zoom-out	X-axis zoom-out[Page 98, 1]
X	Zoom to range	If the region is enabled, it will zoom to it
M	Statistics	Toggle the display of the statistic panel
R	Range	Display a movable range that will trigger the display of the delta values for all plot channels[2]
S	Stack all	Y Stack all active channels so that they don't overlap, keeping the X-axis range
Shift+S	Stack selected	Y Stack all active selected channels so that they don't overlap; unselected channels are not stacked
Y	Lock/unlock region	Lock or unlock the left margin of the region
.	Toggle dots	Toggle the display of signal samples (using dots affects performance)
←	Move cursor left	Moves the cursor to the next sample on the left
→	Move cursor right	Moves the cursor to the next sample on the right
Ins	Insert computation	Insert new channel in the plot using functions and operations
F11	Toggle full screen	In the single files mode will display the current opened file in full screen
Alt+I	Toggle trigger texts	Toggle the text boxes for the triggers[6]
Alt+R	Raw samples	Toggle raw samples mode for the selected channels[Page 98, 6]
Alt+S	Scaled samples	Toggle scaled (physical) samples mode for the selected channels

continues on next page

Table 1 – continued from previous page

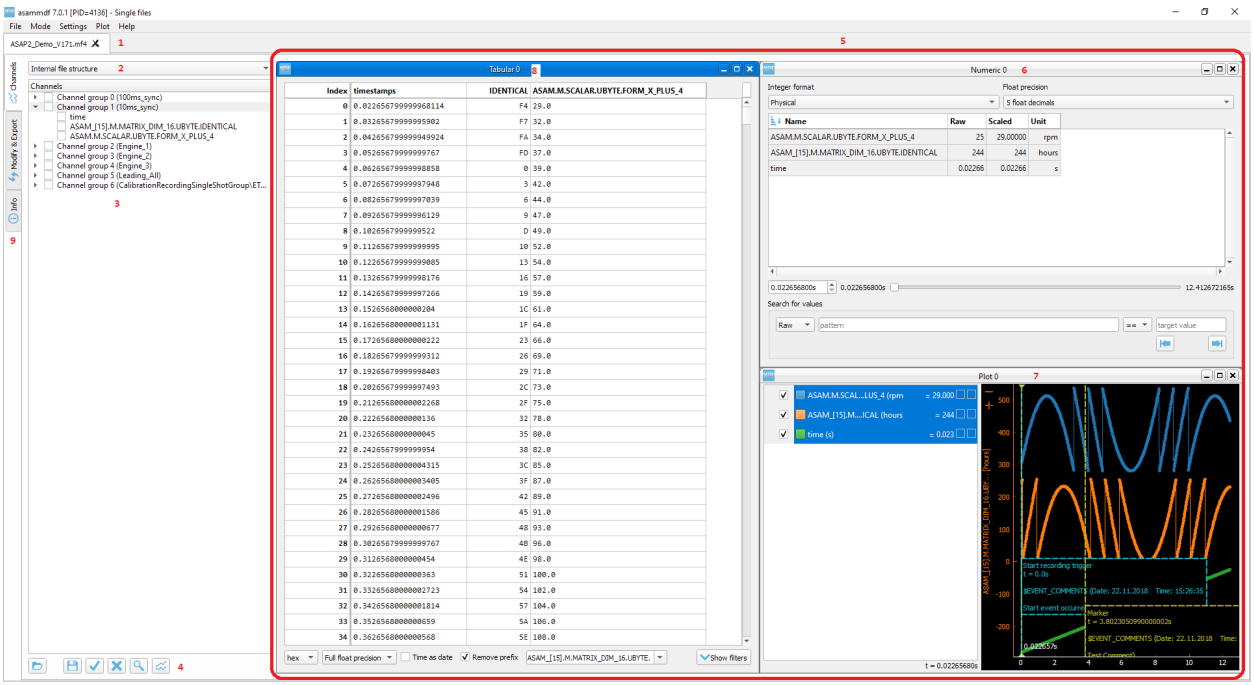
Shortcut	Action	Description
Ctrl+B	Bin	Toggle binary representation of integer channels
Ctrl+H	Hex	Toggle hex representation of integer channels
Ctrl+I	Insert cursor comment	Insert a visual vertical line and comment at the current cursor position[Page 98, 6]
Ctrl+P	Physical	Toggle physical representation of integer channels
Ctrl+R	Edit color ranges	Opens a dialog to edit the channel or channel group color ranges[7]
Ctrl+G	Edit Y axis scaling	Opens a dialog to visually edit the Y axis ranges acoording to the expected signal values[7]
Ctrl+S	Save plot channels	Save channels from current active subplot in a new MF4 file
Ctrl+Shift+S	Save all channels	Save all channels from all sub-windows in a new MF4 file
Shift+C	Cascade sub-windows	Cascade the sub plots
Shift+Alt+F	Toggle frames	Will toggle the sub plots MDI window frames
Shift+L	Toggle channel list	Will toggle the channel tree for the current opened file
Shift+T	Tile sub-windows	Tiles sub-windows in a grid
Shift+V	Tile vertically	Tiles sub-windows vertically[3]
Shift+H	Tile horizontally	Tiles sub-windows horizontally[3]
Shift+←	Shift channels left	Shifts the selected channels to the left in the time domain
Shift+→	Shift channels right	Shifts the selected channels to the right in the time domain
Shift+↑	Shift channels up	Shifts the selected channels to the up
Shift+↓	Shift channels down	Shifts the selected channels to the down

[7] New in *asammdf 7.1.0*
[1] If the cursor is present then zooming will center on it.
[2] Clicking the plot will move the left margin of the region. Pressing CTRL while clicking the plot will move the right margin of the region.
[6] New in *asammdf 5.20.0*
[3] New in *asammdf 5.7.0*

7.3 Single files

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

7.3.1 Layout elements



1. Opened files tabs
2. Channel tree display mode
3. Complete channels tree
4. Command buttons
5. Windows area
6. Numeric window
7. Plot window
8. Tabular window
9. File operations

7.3.1.1 1. Opened files tabs

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

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

7.3.1.2 2. Channel tree display mode

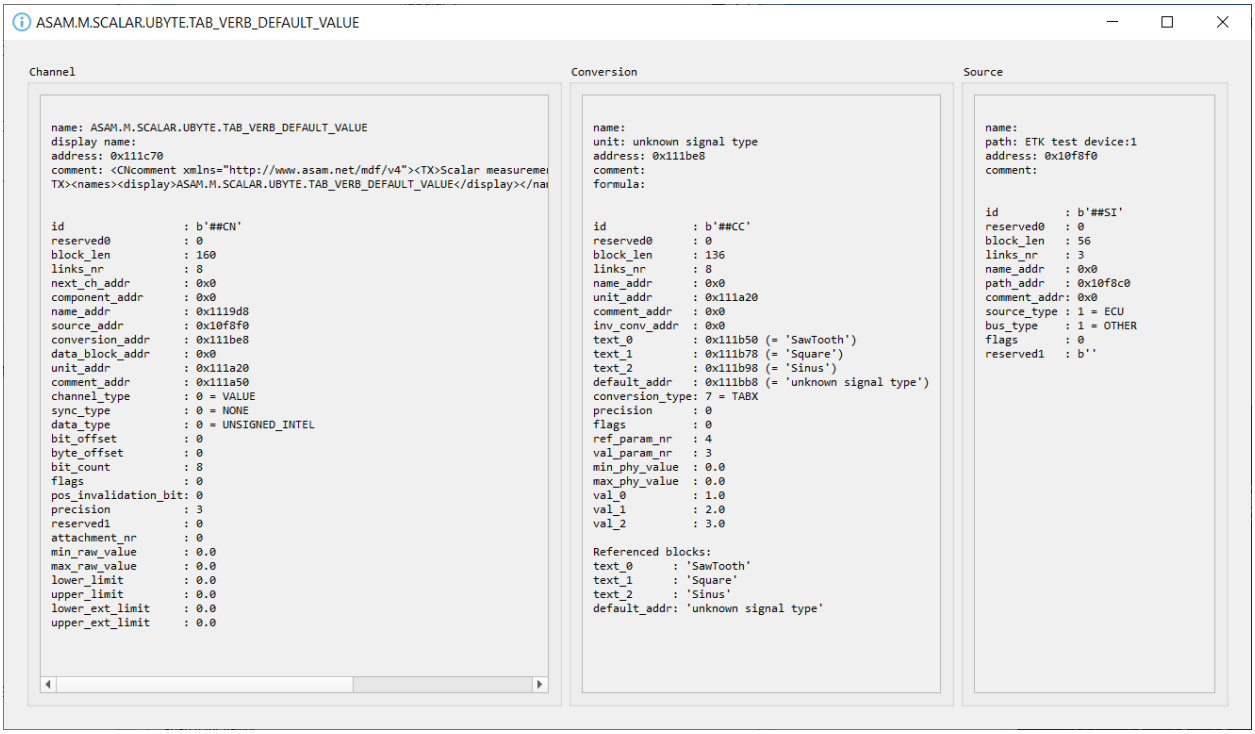
The channel tree can be displayed in three ways

- as a naturally sorted list
- grouped using the internal file structure
- only the selected channels

7.3.1.3 3. Complete channels tree

This tree contains all the channels found in the measurement.

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

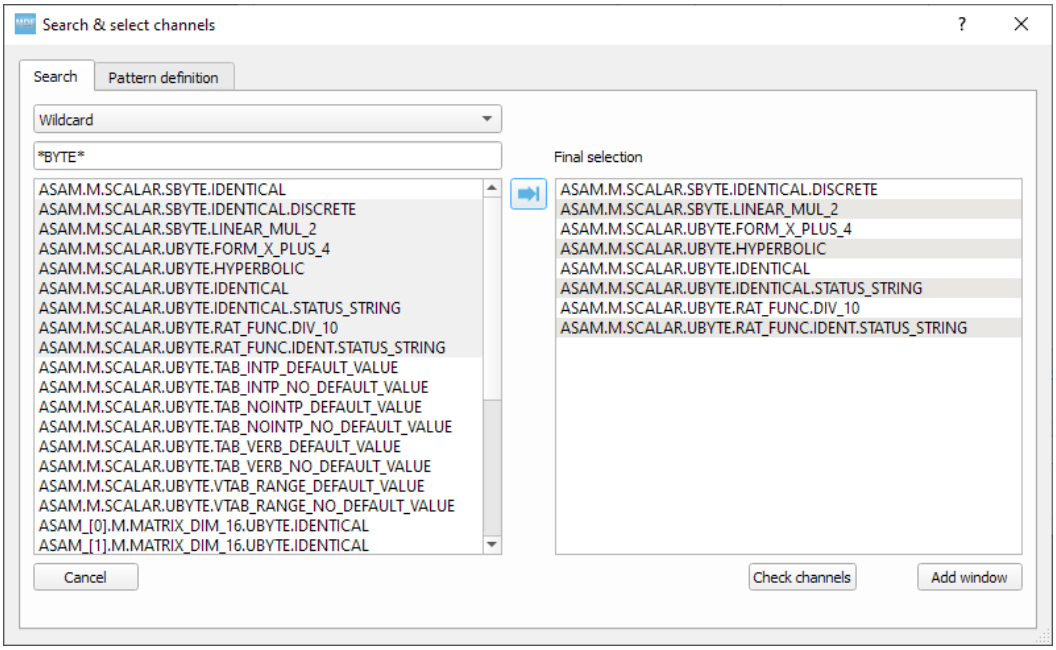


Only the channels that are checked in the channels tree will be selected for plotting when the *Create window* button is pressed. Checking or unchecking channels will not affect the current plot or sub-windows.

7.3.1.4 4. Command buttons

From left to right the buttons have the following functionality

- **Load configuration**: restores channels tree and all sub-plot windows from a saved configuration file
- **Save configuration**: saves all sub-windows (channels, colors, common axis and enable state) and channel tree
- **Select all channels**: checks all channels in the channels tree
- **Reset selection**: unchecks all channels in the channels tree
- **Advanced search & select**: will open an advanced search dialog
  - the dialog can use wildcard and regex patterns
  - multiple channels can be selected, and thus checked in the channels tree
  - in the "Pattern based window" tab the user can define a pattern that will be used to filter out the channels from the measurement file, and as a second filtering step some condition can be used based on the channels values. This information will be saved in the window configuration. The pattern based windows can be easily recognized by the title bar icon
  - the keyboard shortcut `Ctrl+F` can also be used to bring up the search dialog



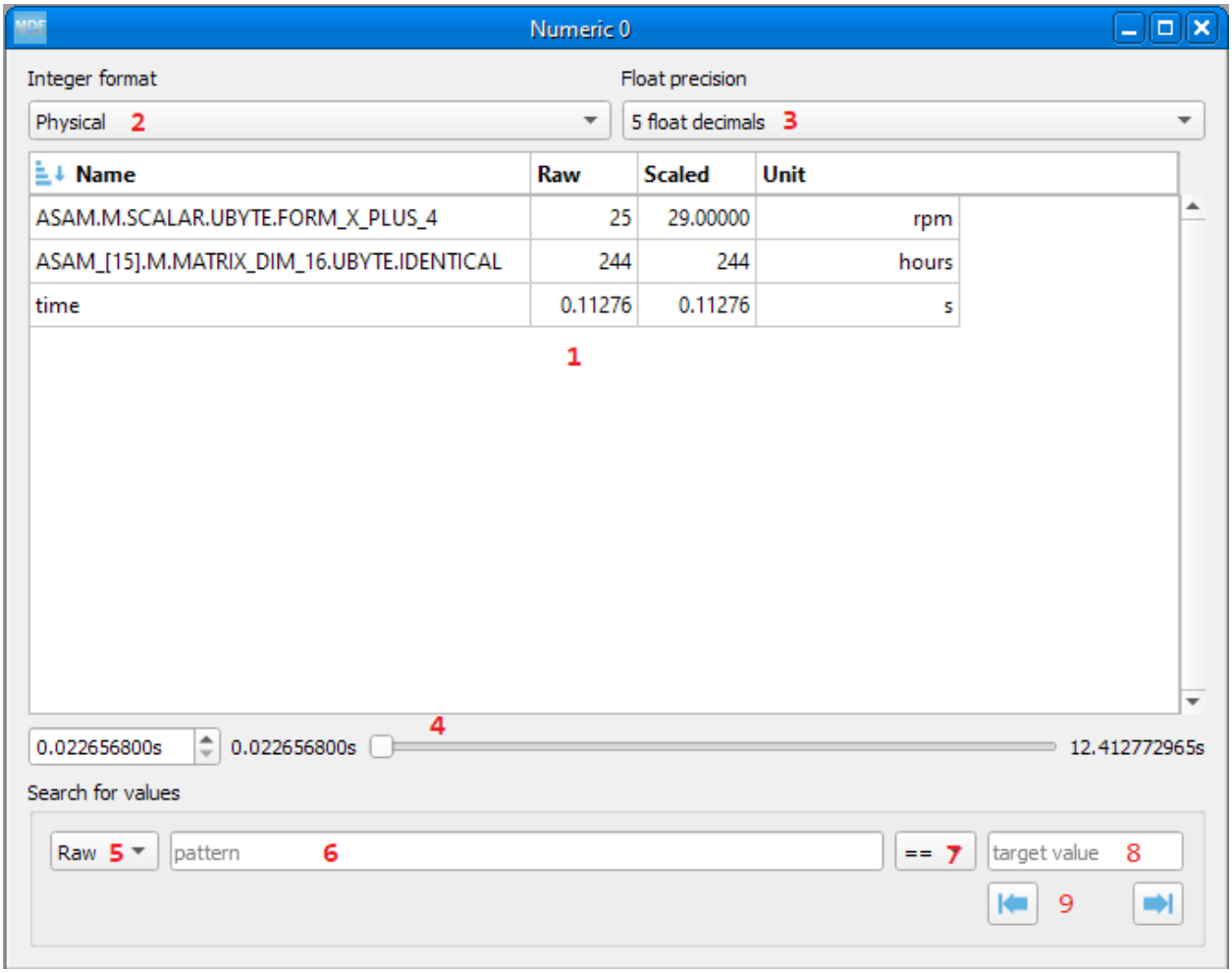
- **Create window**: generates a new window (Numeric, Plot, Tabular, GPS, CAN/LIN/FlexRay Bus Trace) based on the current checked channels from the channels tree. If sub-windows are disabled in the settings then the current window is replaced by the new plot. If sub-windows are enabled then a new sub-plot will be added, and the already existing sub-windows will not be affected. The same channel can be used in multiple sub-windows.

7.3.1.5 5. Windows area

If sub-windows are enabled then multiple plots can be used. The sub-windows can be re-arranged using drag & drop.

7.3.1.6 6. Numeric window

Numeric windows can handle a lot more channels than plot windows. You can use a numeric window to see the channel values at certain time stamps, and to search for certain channel values.



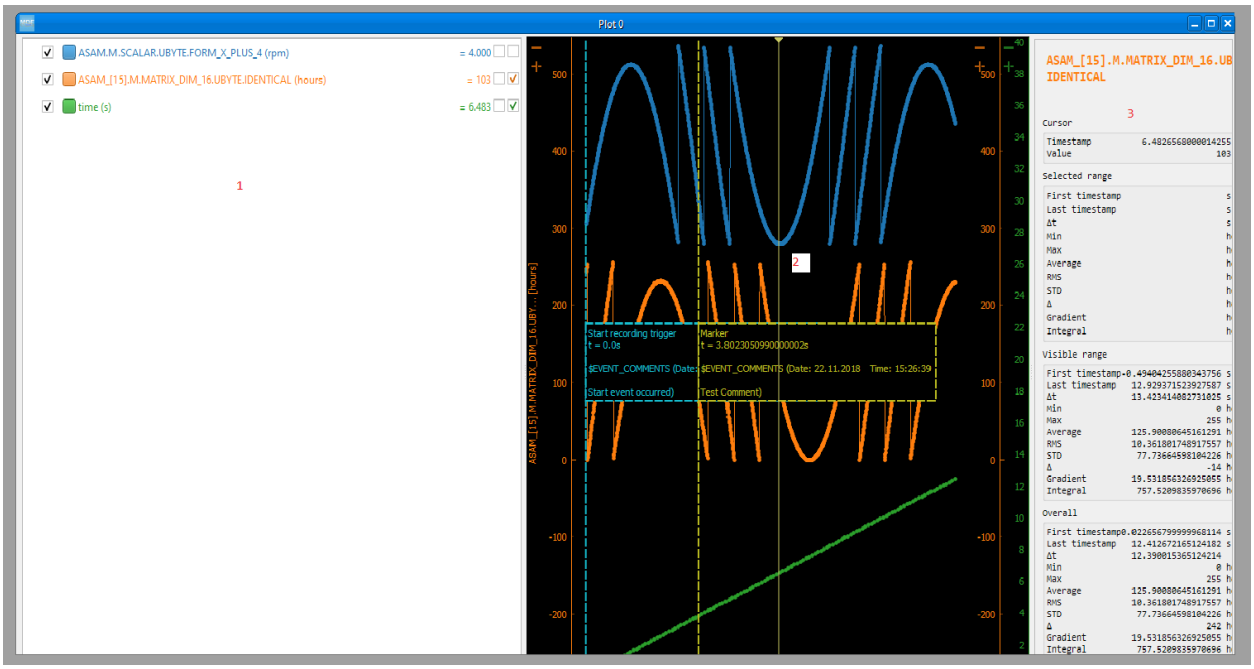
1. display area: here we can see the instantaneous signal values. The raw and scaled values are shown for each signal. Double clicking a column header will toggle the sorting on that column.
2. integer format: choose between physical, hex and binary format.
3. float decimals: choose the precision used for float display
4. timestamp selection: use the input box or the slider to adjust the timestamp
5. signal values search mode: choose between raw and scaled signal samples when searching for a certain value
6. signal name pattern: use a wildcard pattern to select the signals that will be used for value searching
7. operator: operator that will be used for the search
8. target value: search target value

9. direction: timebase direction for searching the values

Double clicking a row will bring up the range editor for associated signal.

7.3.1.7 7. Plot window

Plot windows are used to graphically display the channel samples.



pyqtgraph is used for the plots; to get the best performance consider the following tips.

- limit the number of channels: plotting hundreds of channels can get really slow
- disabling dots will make the plots a lot more responsive

The Plot window has three section 1. signal selection tree 2. graphical area 3. signal statistics panel (toggled using the `M` keyboard shortcut)

Each signal item from the signal selection tree has five elements

1. display enable checkbox
2. color select button
3. channel name and unit label
4. channel value label[4]
5. common axis checkbox
6. individual axis checkbox[5]

The user can also create channel groups in the selection tree. Simple channel groups are only used for grouping signals. Pattern based channel groups can be used to filter signals based on the name or samples values.

[4] the value is only displayed if the cursor or range are active. For the cursor is will show the current value, and for the range it will show the value delta between the range start and stop timestamps

[5] New in *asammdf 5.7.0*

The selection tree has an extended context menu accessible using the right mouse click.

Double clicking an item will open a range editor dialog, similar to the Numeric window range editor.

The initial graphics are view will have all the signal homed-in (see the *H* keyboard shortcut). The user is free to use the mouse to interact with the graphics area (zoom, pan).

The cursor is toggled using the *C* keyboard shortcut, and with it the channel values will be displayed for each item in the *Selected channels list*. The cursor can also be invoked by clicking the plot area.

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

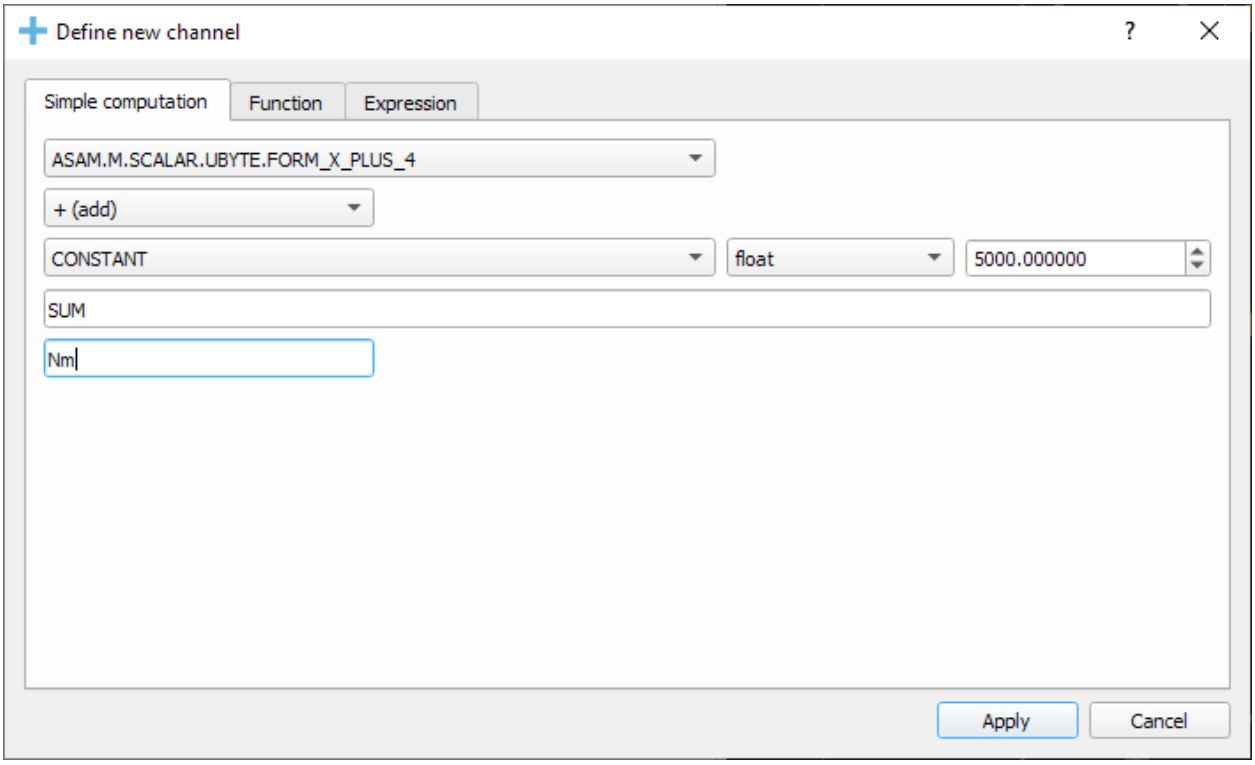
The *Ctrl+H*, *Ctrl+B* and *Ctrl+P* keyboard shortcuts will

- change the axis values for integer channels to hex, bin or physical mode
- change the channel value display mode for each integer channel item in the *Signal selection tree*

The *Alt+R* and *Alt+S* keyboard shortcuts will switch between the raw and scaled signal samples.

Each vertical axis width can be modified using the + and - buttons.

You can insert new computed channels by pressing the *insert* key. This will allow either to compute basic operations using the plot channels, to apply a function on one of the plot channels, or to specify a simple expression than uses multiple signals from the Plot window.

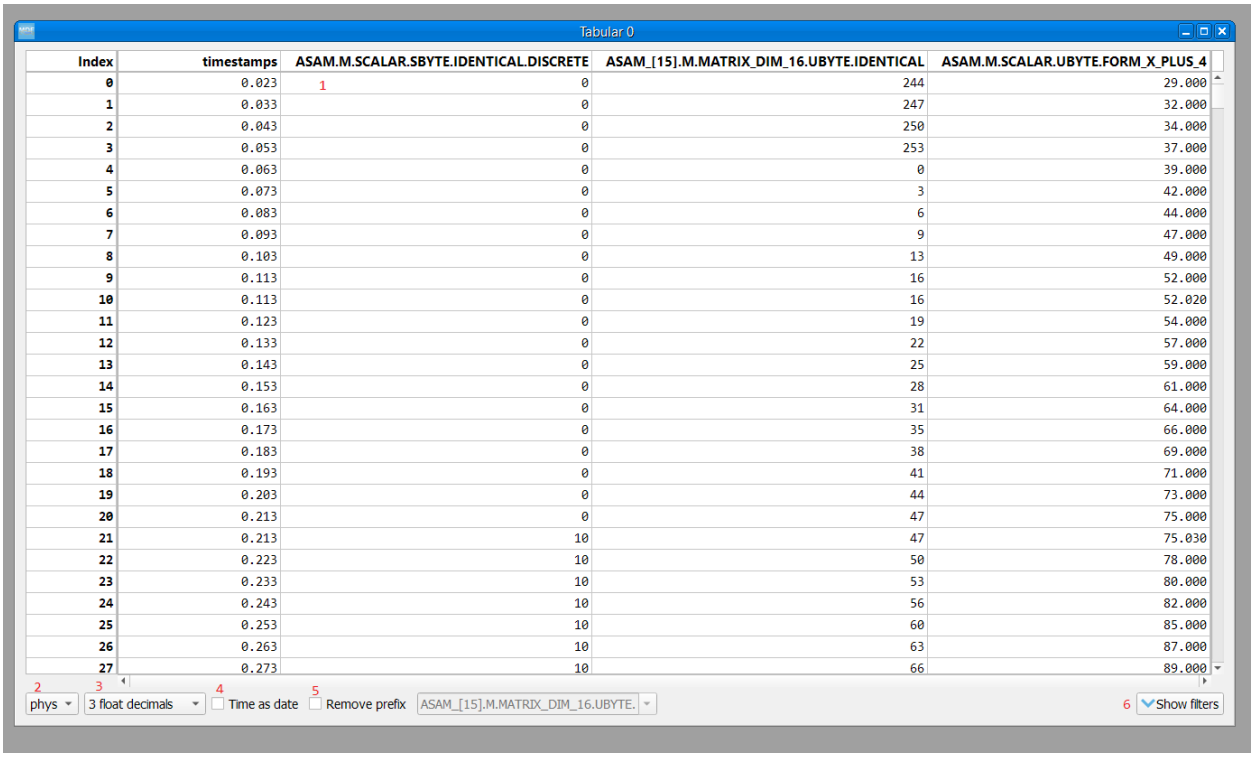


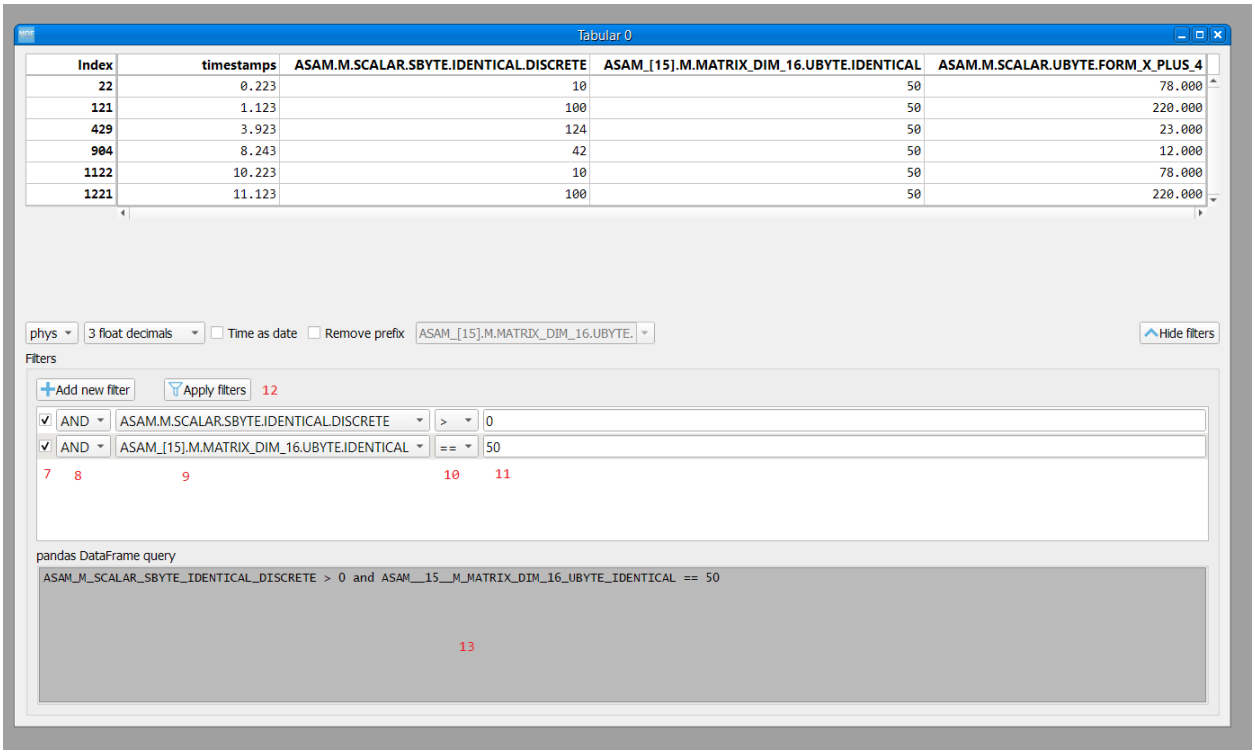
The currently active plot's channels can be saved to a new file by pressing *Ctrl+S*. The channels from all sub-windows can be saved to a new file by pressing *Ctrl+Shift+S*.

The sub-windows can be tiled as a grid, vertically or horizontally (see the keyboard shortcuts).

7.3.1.8 8. Tabular window

The tabular window is very similar to an Excel/CSV sheet. The most powerful feature of this window is that multiple filters can be defined for the signals values.





The tabular window has the following elements:

1. display area: here we can see the signal values. The raw and scaled values are shown for each signal. Right clicking a column header will show a pop-up window for controlling the sorting, defining signal ranges and adjusting the columns width.
2. integer format: choose between physical, hex and binary format.
3. float decimals: choose the precision used for float display
4. timestamp format: use the input box or the slider to adjust the timestamp display as float value or as datetime value
5. remove prefix: remove column names prefix; avoids unnecessary large column widths
6. toggle filters view: toggle the visibility of the filters (better vertical space if filters are not used)
7. filter enable
8. filter logical relation
9. filtering signal
10. filter operator
11. target value for filtering
12. apply filters: the actual filtering is done only after pressing the button. The user can modify the existing filters without changing the tabular view.
13. query: the Tabular window used a pandas dataframe as backend. The filtering is done by performing a query on the dataframe.

7.3.1.9 9. File operations

There are five aspects related to the measurement file that can be accessed using the tabs:

1. channels: here the user can visualize the signals using the available window types
2. modify & export: this tab contains the tools needed for processing the measurement file. The use can filter signals, cut and resample the measurement, or export it to other file formats.
3. bus logging: this tab is only visible for the measurements that contain CAN or LIN bus logging. The user can decode the raw bus logging using database files (.dbc, .ldf, .arxml)
4. attachments: this tab is only visible if the measurement contains attachments. The user can extract the attachment and save it to a new file.
5. info: this tab contains an overview of the measurement file content (channel groups, file header comments, total number of channels)

7.3.1.10 10. CAN/LIN/FlexRay Bus Trace

This window types can only be created by pressing the `Create window` button. If the measurement does not contain bus logging of the selected kind, then no window will be generated.

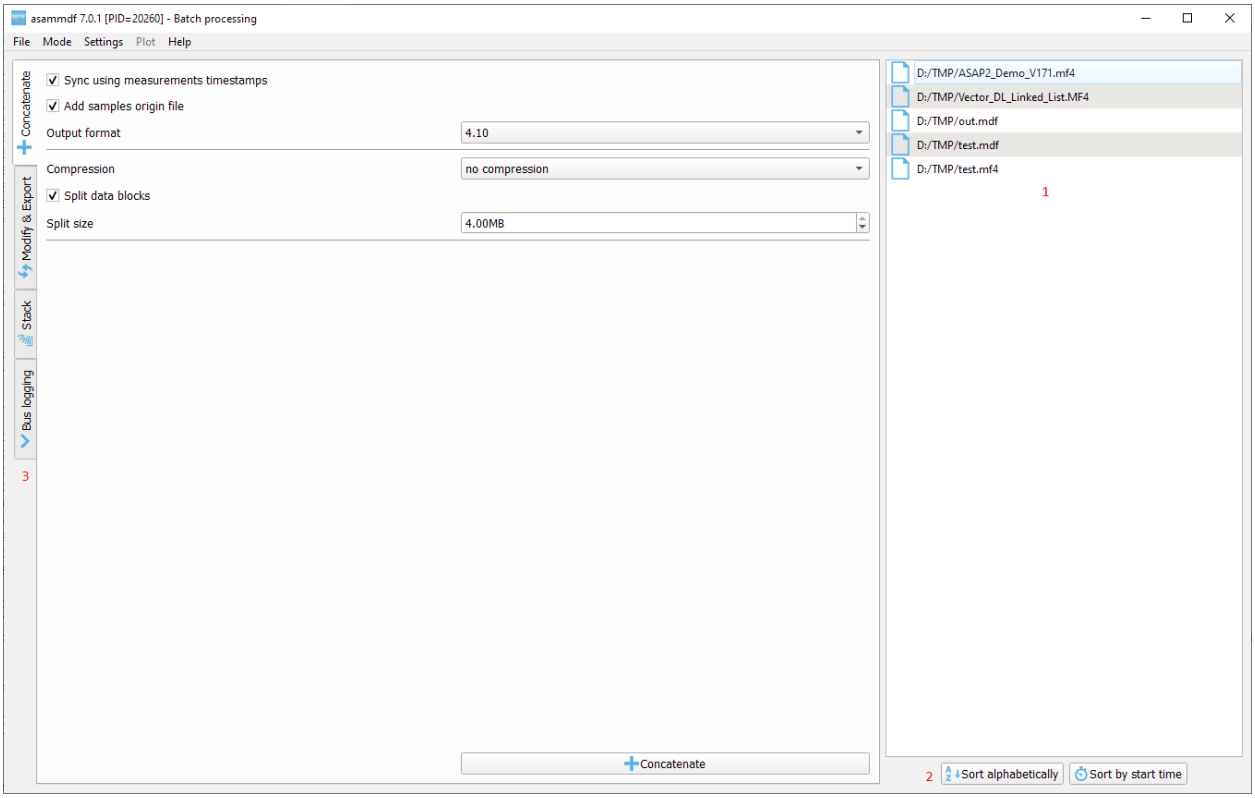
The filtering and signal ranges definition is done similar to the Tabular window.

7.3.1.11 11. Drag & Drop

Channels can be dragged and dropped between sub-windows for easier configuration. Drag and drop in the free MDI can be used to create new windows.

7.4 Batch processing

The *Batch processing* view is used to concatenate or stack multiple files, or to perform the same processing steps on multiple files. Keep in mind that the order of the input files is always preserved, only the samples timestamps are influenced by the `Sync using measurements timestamps` checkbox.



1. file list
2. file list sorting buttons
3. batch operations
    a. concatenate requires input files with matching internal structure (same number of channel groups and the same set of channels in each n-th group). Each signal in the output file will be the result of concatenation of the samples from the input files
    b. modify & export: similar to the single files view
    c. stack will create a single measurement that will contain all the channel groups from the input files. Identically named channels will not be concatenated, they will just appear multiple times in the output file
    d. bus logging: similar to the single files view

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

7.5 Comparison

Use *CTRL+F* to search channels from all the opened files. The channel names are prefixed with the measurement index.

CHAPTER

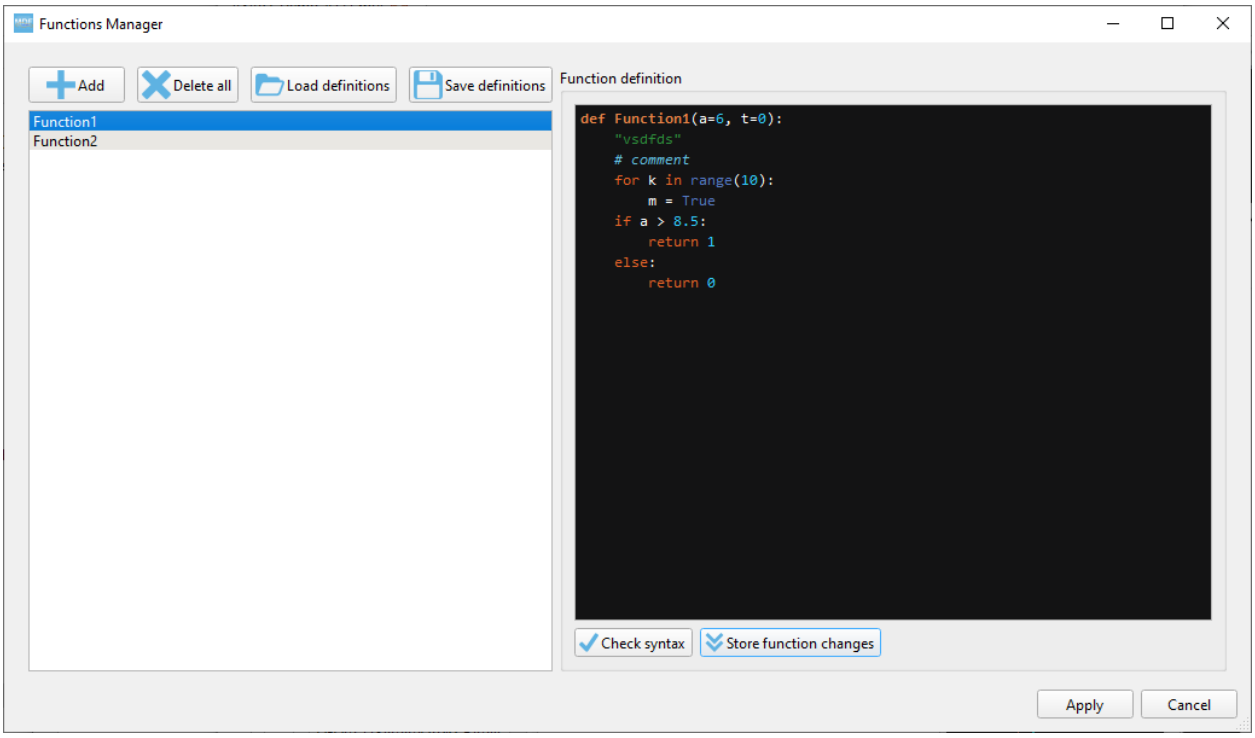
EIGHT

FUNCTIONS MANAGER VIEW

Virtual channels (computed channels) can be defined in the Plot windows. The user will define the functions used for the virtual channels using Python language syntax; the extra rules will be described lower. After this the user can insert virtual channels in the Plot window (keyboard shortcut `Ins`) and will map an existing function to the virtual channel, as well as a list of alternative channel names (ASAPs) for each function argument.

8.1 Function manager dialog

The user defined functions are associated with the display file. This means that there will be separate function managers for each opened measurement file. When a display file (.dspf) is saved by the user it will also contain the associated user defined functions. The functions manager dialog can be invoked using the keyboard shortcut `F6`.



Option	Description
`Add`	Adds a new function
`Delete all`	Deletes all the defined functions
`Load definitions`	Imports definitions found in `.def` files
`Save definitions`	Saves all the define function to a new file with the extension `.def`
`Function definition`	Python language function definition
`Check syntax`	Check if the current function definition syntax is valid
`Store function changes`	Updates the definition after the user has made changes. If this is not pressed and the user switches to another function then the changes will be lost.
`Apply`	Accept all the changes made by the user
`Cancel`	Rejects all the changes made by the user

8.2 Python function rules

The user defines the functions using Python syntax; this means that the function definition must be valid in Python code.

Further more the following rules and restrictions must be followed:

- all function arguments must have default numeric values: this means that the definition *def Function1(a, b, t=0)* is not accepted, instead the user must provide default values such as *def Function1(a=1.5, b=0, t=0)*
- the last argument must be `t=0`: the argument *t* is reserved for the time stamp and must not be otherwise used by the user. This means that the definition *def Function1(a=1.5, b=0)* is not valid, instead the user can write something like this *def Function1(a=1.5, b=0, t=0)*
- always return numeric values on all the branches: Python functions automatically return *None* if no *return* is explicitly written. The user must take care that all possible code braces return numeric values.

```
"this function will produce errors because not all branches have explicit␣
↪return statements with numeric values"
def Function1(a=0, t=0):
    if a > 5:
        return 1

"this function will produce errors because it returns non-numeric values"
def Function1(a=0, t=0):
    if a > 5:
        return "Activated"
    else:
        return "Disabled"

"this function is OK because it has explicit return statements with␣
↪numeric values"
def Function1(a=0, t=0):
    if a > 5:
        return 1
```

(continues on next page)

(continued from previous page)

```
    else:
        return 0
```

- the modules *math*, *pandas* and *numpy* are available inside the function as *math*, *pd* and *np*
- functions can be cross-referenced; one function can call another user defined function

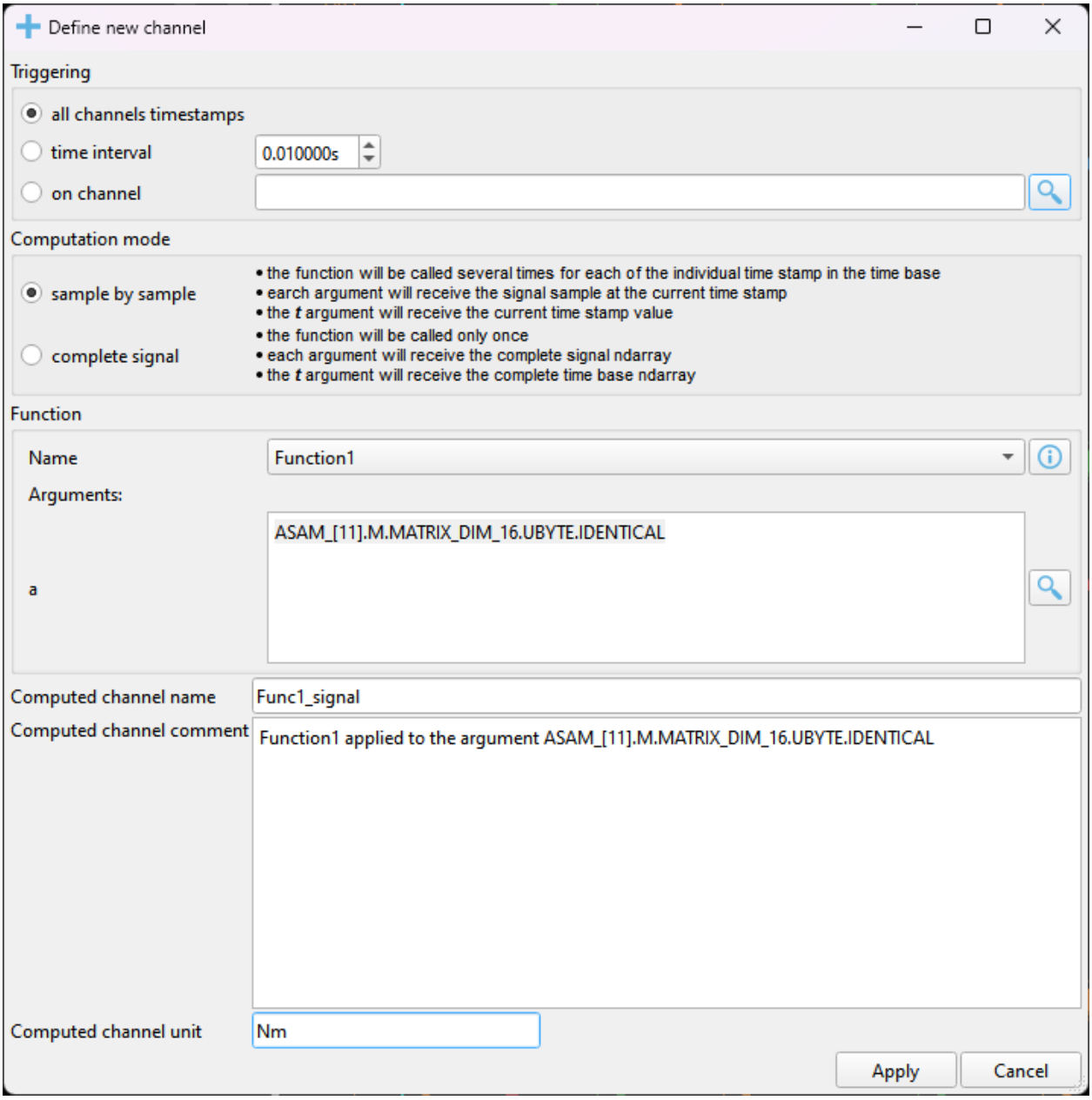
```
def Function1(a=0, t=0):
    if a > 5:
        return 1
    else:
        return 0

" Function2 can call Function 1"
def Function2(b=1, c=7, t=0):
    if b != 0:
        return Function1(c)
    else:
        return c / 2
```

8.3 Virtual channel dialog

Virtual channels can be added to Plot windows using the keyboard shortcut `Ins` (insert). The precondition for this is to have at least one function defined in the Functions manager.

In the *Define channel* dialog window the user can select one of the available function, and then for each of the function's arguments a list of alternative channel names can be filed by the user. The argument *t=0* is reserved for the time-stamp and is not shown in the dialog window. The alternative names must be placed on separate lines in the text field found next to the argument name.


Define new channel
Triggering
all channels timestamps
time interval
0.010000s
on channel
Computation mode
sample by sample
• the function will be called several times for each of the individual time stamp in the time base
• earch argument will receive the signal sample at the current time stamp
• the t argument will receive the current time stamp value
complete signal
• the function will be called only once
• each argument will receive the complete signal ndarray
• the t argument will receive the complete time base ndarray
Function
Name
Function1
Arguments:
a
ASAM_[11].M.MATRIX_DIM_16.UBYTE.IDENTICAL
Computed channel name
Func1_signal
Computed channel comment
Function1 applied to the argument ASAM_[11].M.MATRIX_DIM_16.UBYTE.IDENTICAL
Computed channel unit
Nm
Apply
Cancel

Option	Description
Triggering	the virtual channel samples will be calculated in the time stamps defined by the selected choice
Computatic mode	the virtual channel can be computed passing the signal arguments samples by sample or passing the signal arguments directly as numpy ndarrays
Function name	select one of the available functions
Function info	displays the function definition code
Arguments	alternative channel names for each argument. In the example argument *b* will use the first existing ASAP from the choices *MAIN_CLOCK* or *MAIN_CLOCK_5MS*. If no alternative exist in the measurement file then the argument's default value will be used (see the function definition)
Argument search	opens a search dialog were channel names cane be searched from the measurement file
Computed channel name	virtual channel name
Computed channel comment	virtual channel comment
Computed channel unit	virtual channel unit
Function definitior	Python language function definition
Check syntax	Check if the current function definition syntax is valid
Store function changes	Updates the definition after the user has made changes. If this is not pressed and the user switches to another function then the changes will be lost.
Apply	Accept the changes and inserts the new virtual channel
Cancel	Rejects all the changes made by the user

When the computation mode is set to *sample by sample* the following steps are executed:

- the signals linked to the arguments are searched in the measurement file
- using the signals found in the measurement, the union of all time stamps is computed (resulting an ndarray of length N)
- all found signals are interpolated using the union time base
- the function is called N times (once for each of the N time stamps) passing the signal value for the current time stamp and the time stamp (in the *t* argument). The missing linked signals will be replaced by the default value specified in the function definition
- the returned value are stored and finally the resulting Signal is constructed

When the computation mode is set to *complete signal* the following steps are executed:

- the signals linked to the arguments are searched in the measurement file
- using the signals found in the measurement, the union of all time stamps is computed (resulting an ndarray of length N)
- all found signals are interpolated using the union time base

- the function is called only 1 time passing the complete signal ndarray's and the union of the time stamps (in the *t* argument). The missing linked signals will be replaced by an ndarray of length N filled with the default value specified in the function definition
- the result Signal is constructed using the ndarray returned by the function

8.4 Example functions

8.4.1 Average of 3 channels

The same definition works for both *sample by sample* and *complete signal* computation modes

```
def average(channel1=0, channel2=-1, channel3=0, t=0):
    return (channel1 + channel2 + channel3) / 3
```

8.4.2 Channel clipping

Using *sample by sample* computation mode

```
def clip(channel1=0, t=0):
    if channel1 > 100:
        return 100
    elif channe1 < 0:
        return 0
    else:
        return channel1
```

Using *complete signal* computation mode

```
def clip(channel1=0, t=0):
    return np.clip(cahnnel1, 0, 100)
```

8.4.3 Grey code to decimal

The same definition works for both *sample by sample* and *complete signal* computation modes

```
def gray2dec(position_sensor_value=0, t=0):

    for shift in (8, 4, 2, 1):
        position_sensor_value = position_sensor_value ^ (position_sensor_value >> shift)

    return position_sensor_value
```

8.4.4 Maximum of 3 channels

Using *sample by sample* computation mode

```
def maximum(channel1=0, channel2=-1, channel3=0, t=0):
    return max(channel1, channel2, channel3)
```

Using *complete signal* computation mode

```
def maximum(channel1=0, channel2=-1, channel3=0, t=0):
    return np.maximum.reduce([channel1, channel2, channel3])
```

8.4.5 rpm to rad/s

The same definition works for both *sample by sample* and *complete signal* computation modes

```
def rpm_to_rad_per_second(speed=0, t=0):
    return 2 * np.pi * speed / 60
```

8.4.6 gradient

In the *Define channel* dialog the computation mode must be set to *complete signal*

```
def rpm_to_rad_per_second(speed=0, t=0):
    return np.diff(speed) / np.diff(t)
```

It is not possible to implement this function in the *sample by sample* mode because only the current samples are forwarded to the function.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

A

astype() (*asammdf.signal.Signal method*), 72
AttachmentBlock (*class in asammdf.blocks.v4_blocks*), 59

C

Channel (*class in asammdf.blocks.v2_v3_blocks*), 38
Channel (*class in asammdf.blocks.v4_blocks*), 60
ChannelConversion (*class in asammdf.blocks.v2_v3_blocks*), 39
ChannelConversion (*class in asammdf.blocks.v4_blocks*), 62
ChannelDependency (*class in asammdf.blocks.v2_v3_blocks*), 41
ChannelExtension (*class in asammdf.blocks.v2_v3_blocks*), 41
ChannelGroup (*class in asammdf.blocks.v2_v3_blocks*), 42
ChannelGroup (*class in asammdf.blocks.v4_blocks*), 64
cleanup_timestamps() (*asammdf.mdf.MDF method*), 8
concatenate() (*asammdf.mdf.MDF static method*), 8
configure() (*asammdf.mdf.MDF method*), 10
convert() (*asammdf.mdf.MDF method*), 11
copy() (*asammdf.signal.Signal method*), 72
cut() (*asammdf.mdf.MDF method*), 11
cut() (*asammdf.signal.Signal method*), 72

D

DataBlock (*class in asammdf.blocks.v4_blocks*), 66
DataGroup (*class in asammdf.blocks.v2_v3_blocks*), 43
DataGroup (*class in asammdf.blocks.v4_blocks*), 64
DataList (*class in asammdf.blocks.v4_blocks*), 65

E

EventBlock (*class in asammdf.blocks.v4_blocks*), 70
export() (*asammdf.mdf.MDF method*), 12
extend() (*asammdf.signal.Signal method*), 73
extract() (*asammdf.blocks.v4_blocks.AttachmentBlock method*), 60
extract_bus_logging() (*asammdf.mdf.MDF method*), 13

F

FileHistory (*class in asammdf.blocks.v4_blocks*), 68
FileIdentificationBlock (*class in asammdf.blocks.v2_v3_blocks*), 44
FileIdentificationBlock (*class in asammdf.blocks.v4_blocks*), 66
filter() (*asammdf.mdf.MDF method*), 15
Flags (*asammdf.signal.Signal attribute*), 72

G

get_group() (*asammdf.mdf.MDF method*), 16

H

HeaderBlock (*class in asammdf.blocks.v2_v3_blocks*), 44
HeaderBlock (*class in asammdf.blocks.v4_blocks*), 67

I

interp() (*asammdf.signal.Signal method*), 73
iter_channels() (*asammdf.mdf.MDF method*), 17
iter_get() (*asammdf.mdf.MDF method*), 17
iter_groups() (*asammdf.mdf.MDF method*), 18
iter_to_dataframe() (*asammdf.mdf.MDF method*), 19

M

MDF (*class in asammdf.mdf*), 7

P

physical() (*asammdf.signal.Signal method*), 74
plot() (*asammdf.signal.Signal method*), 74
ProgramBlock (*class in asammdf.blocks.v2_v3_blocks*), 45

R

resample() (*asammdf.mdf.MDF method*), 20

S

scramble() (*asammdf.mdf.MDF static method*), 23
search() (*asammdf.mdf.MDF method*), 24
select() (*asammdf.mdf.MDF method*), 24

Signal (*class in asammdf.signal*), 71
SourceInformation (*class in asammdf.blocks.v4_blocks*), 68
stack() (*asammdf.mdf.MDF static method*), 26
start_time (*asammdf.blocks.v4_blocks.HeaderBlock property*), 67
start_time (*asammdf.mdf.MDF property*), 27

T

TextBlock (*class in asammdf.blocks.v2_v3_blocks*), 46
TextBlock (*class in asammdf.blocks.v4_blocks*), 69
time_stamp (*asammdf.blocks.v4_blocks.FileHistory property*), 69
to_dataframe() (*asammdf.mdf.MDF method*), 27
TriggerBlock (*class in asammdf.blocks.v2_v3_blocks*), 46

V

validate() (*asammdf.signal.Signal method*), 74

W

whereis() (*asammdf.mdf.MDF method*), 28