# idelib Documentation

***Release 3...4...0***

**David Randall Stokes**

**Mar 30, 2026**

# CONTENTS:

*idelib* is the core Python API for accessing the contents of enDAQ IDE recordings. It provides a means of easily accessing time series sensor data, with all necessary calibration applied, as well as recording metadata.

The package also contains the command-line utilities `ideexport` and `ideinfo` for use outside of Python.

CHAPTER

ONE

# IDE FILE BASICS

## 1.1 What's an IDE file?

An IDE file is a read-only, EBML-based hierarchical data format that stores recording information generated by an enDAQ sensor device. It contains both time-indexed data from several different kinds of sensors (like acceleration, pressure, temperature, etc.), as well as metadata about the recording device (like device serial number, model number, device name, etc.) and recording settings.

## 1.2 Accessing an IDE file

The top-level interface for an IDE file is the `Dataset` object, through which one can access all of the above-listed information. When you open a file for reading, for example, this is the type of object that is returned.

### 1.2.1 Opening an IDE File

You can open an IDE file like so:

```
filename = "/path/to/your/file.IDE"
with idelib.importFile(filename) as ds:
    print(type(ds))
```

Note: a `Dataset` object perfoms *lazy-loading*, meaning that it only loads information as is needed. As a result, it internally retains a handle to the source file which needs to be closed after use. This can be accomplished by either:

- using `Dataset` as a *context manager* (as seen above; this is the recommended method), or
- by using `Dataset` as a normal object and calling the `close()` method manually:

  ```
  filename = "/path/to/your/file.IDE"
  ds = idelib.importFile(filename)
  # use `ds` here
  ds.close()  # remember to close the file after use!
  ```

## 1.3 Getting recording data

### 1.3.1 Channels & subchannels

IDE files organize recording data into *channels* and *subchannels*. A channel encapsulates data recorded by a particular individual sensor on the device (e.g., XYZ acceleration from the ADXL375 DC Accelerometer); a subchannel, if present, specifies a particular data stream within a channel (e.g., the X-coordinate acceleration from the ADXL375 DC Accelerometer).

At the top-level, a `Dataset` object has a `channels` member, which is a dict of all channels recorded in the file. The dict is keyed by channel ID numbers, with `Channel` objects in the values.

Each `Channel` object has a `subchannels` member, which is a list of `Subchannel` objects. If the channel has no subchannels, this member will be `None`.

The channels, channel ID numbers and subchannels that may appear in a given recording file depend on the physical sensors available on the recording device, which are indicated by the device's *product number*. Below are some product number abbreviations used herein:

| (Abbreviated) Product No. | Description | Example Product Nos. |
|---|---|---|
| S/W-D | enDAQ S-series & W-series devices with a single digital accelerometer | S3-D16, W5-D40 |
| S/W-DD | enDAQ S-series & W-series devices with dual digital accelerometers | S1-D100D40, S2-D25D16 |
| S/W-ED | enDAQ S-series & W-series devices with an analog electrocapacitive and a digital accelerometer | W8-E25D40, S4-E100D40 |
| S/W-RD | enDAQ S-series & W-series devices with an analog piezoresistive and a digital accelerometer | S4-R500D40, W8-R2000D40 |
| SSX | Mide Slam Stick X data recorders | SSX |
| SSC | Mide Slam Stick C data recorders | SSC |
| SSS | Mide Slam Stick S data recorders | SSS |

The below table lists channel ID numbers used in a recording file based on the recording device's product number (device series numbers and accelerometer sensitivity ranges are omitted when applicable to all such devices):

| Sensor | Channel ID | Valid Devices | Subchannels |
|---|---|---|---|
| Main Accelerometer | 8 | S/W-RD, S/W-ED, SSS, SSX | X-, Y-, Z-axis Acceleration |
| 16/200g Accelerometer | 32 | S/W-DD, SSX, SSS, SSC, S/W-D16, S/W-D200 | X-, Y-, Z-axis Acceleration |
| 8/40g Accelerometer | 80 | S/W-RD, S/W-DD, S/W-ED, S/W-D40, S/W-D8 | X-, Y-, Z-axis Acceleration |
| IMU Gyroscope | 47 | All[1] | X-, Y-, Z-axis Rotation |
| Absolute Orientation | 65 | All[Page 5, 1] | X-, Y-, Z-, W-axis Quaternion; Acc |
| Relative Orientation | 70 | All[Page 5, 1] | X-, Y-, Z-, W-axis Quaternion |
| MPL3115 | 36 | All[Page 5, 1] | Pressure, Temperature[2] |
| MS8607 | 59 | All[Page 5, 1] | Pressure, Temperature, Humidity[3] |
| SI1133 | 76 | All[Page 5, 1] | Lux, UV |

To simply use all recording data, we can iterate through each subchannel in a dataset like so:

```
for ch in ds.channels.values():
    for sch in ch.subchannels:
        print(sch)
```

### 1.3.2 EventArrays & raw data

Each `Channel` and `SubChannel` object has a `getSession()` method, which returns an `EventArray` object. `EventArray` is a wrapper around a channel's underlying recording data that loads data on demand from the source file. You can index an `EventArray` (e.g., `eventarray[i]` for some index `i`) to get a numpy `ndarray` of data. Data is organized in an n-dimensional array.

For subchannels, this will always be a 2-by-n array, where n is the number of samples recorded; `eventarray[1]` indexes the samples, `eventarray[0]` indexes the respective timestamps.

For channels, this will be a (c+1)-by-n array, where n is the number of samples recorded and c is the number of subchannels; `eventarray[1:]` indexes the samples, `eventarray[0]` indexes the respective timestamps.

## 1.4 Getting metadata

`Dataset` makes available some basic metadata. Some useful pieces of information are stored directly as members:

```
>>> ds.filename
'/home/enDAQ/recordings/test.IDE'
```

Other data is stored in the dict member `recorderInfo`:

```
>>> ds.recorderInfo['RecorderSerial']
10118
>>> ds.recorderInfo['PartNumber']
'W8-E100D40'
```

`EventArray` also stores some sample-specific metadata, like the data's units:

```
>>> eventarray.units
('Acceleration', 'g')
```

[1] excluding early SSC/SSS/SSX models
[2] 1 Hz Internal Measurements
[3] 10 Hz Control Pad Measurements

CHAPTER

TWO

# IDE FILE COMMAND-LINE UTILITIES

`idelib` provides some useful command-line utilities for converting IDE data to other formats, and for viewing general information about an IDE file. These are automatically installed when installing the `idelib` package.

## 2.1 Getting Started

### 2.1.1 Installing

The utilities are part of the `idelib` package, so they are installed along with it. The common way to install is via `pip` (which is typically bundled with Python):

```
pip install idelib
```

### 2.1.2 Running the scripts

These utilities can be run in two ways: via an executable script, or as a Python module.

#### Running as a Python module

You can run the utilities by running Python with the `-m` argument, followed by the name of the submodule and any arguments, e.g.:

- `python -m idelib.tools.ideexport --help`
- `python -m idelib.tools.ideinfo --help`
- `python -m idelib.tools.idesync --help`

While verbose, this is the most reliable way to run the utilities.

### Running the executable

When `idelib` is installed, it will create an executable file for each utility. The location of the scripts varies by operating system (Linux/Windows/MacOS/etc.) and by Python environment (e.g., standard Python from python.org or conda), but it should be in your 'path', so running it is done by typing its name, followed by any arguments:

- `ideexport --help`
- `ideinfo --help`
- `idesync --help`

Unfortunately, there are many factors specific to your computer and Python setup that can interfere with this. Most commonly, the executables are somewhere outside of your 'path' and could not be found, or security/authorization issues prevented the executables' creation. In either case, the result is the standard 'not found' error for your OS.

## 2.2 ideexport

`ideexport` converts one or more IDE files into text (CSV, etc.) or Matlab MAT v5 files.

```
usage: ideexport.py [-h] [-o OUTPUT] [-t {csv,mat,txt}] [-c CHANNEL] [-m] [-u] [-n] [-r]␣
↪[-d {comma,tab,pipe}] [-f] FILENAME.IDE [FILENAME.IDE ...]

Batch IDE Conversion Utility v3.4.0 - Copyright (c) 2026 Midé Technology

positional arguments:
  FILENAME.IDE          The source .IDE file(s) to convert. Wildcards permitted.

options:
  -h, --help            show this help message and exit
  -o OUTPUT, --output OUTPUT
                        The output path to which to save the exported files. Defaults to␣
↪the same location as the source file.
  -t {csv,mat,txt}, --type {csv,mat,txt}
                        The type of file to export.
  -c CHANNEL, --channel CHANNEL
                        Export the specific channel. Can be used multiple times. If not␣
↪used, all channels will export.
  -m, --removemean      Remove the mean from accelerometer data.
  -u, --utc             Write timestamps as UTC 'Unix epoch' time.
  -n, --names           Include channel names in exported filenames.
  -s, --sync            Apply recordings' sychronization data (if present in the file).

Text Export Options (CSV, TXT, etc.):
  -r, --headers         Write 'header' information (column names) as the first row of␣
↪text-based export.
  -d {comma,tab,pipe}, --delimiter {comma,tab,pipe}
                        The delimiting character, for exporting non-CSV text-based files.
  -f, --isoformat       Write timestamps as ISO-formatted UTC.
```

## 2.3 idesync

`idesync` generates synchronization metadata, synchronizing one or more files to a 'reference' recording. This information, which is inserted into the recording, can be applied when exporting with `ideexport --sync`.

```
usage: idesync.py [-h] [-g] [-i] FILENAME.IDE [FILENAME.IDE ...]

Batch IDE Synchronization Utility v3.4.0b1 - Copyright (c) 2025 Midé Technology

positional arguments:
  FILENAME.IDE   The recording to which to synchronize the others, and/or adjustits␣
↪starting time if --gps is used.
  FILENAME.IDE   Recordings to sync to the reference.

options:
  -h, --help     show this help message and exit
  -g, --gps      Update the reference recording's start time using its GPS/GNSS data.␣
↪Cannot be used in conjunction wtih --inherit.
  -i, --inherit  If the reference recording has been synced to another, sync the other␣
↪datasets to that (rather than the reference itself). Cannot be used in conjunction␣
↪with --gps.
```

Note: The applied synchronization is not currently displayed when viewing recordings in *enDAQ Lab* (verion 3.1.1 or earlier).

## 2.4 ideinfo

`ideinfo` gathers information about an IDE file, and either displays the text on the console or writes it to a file.

```
usage: ideinfo.py [-h] [-o OUTPUT] FILENAME.IDE [FILENAME.IDE ...]

IDE Info Viewer v3.4.0 - Copyright (c) 2026 Midé Technology

positional arguments:
  FILENAME.IDE          The source .IDE file(s) to convert. Wildcards permitted.

options:
  -h, --help            show this help message and exit
  -o OUTPUT, --output OUTPUT
                        The output path to which to save the info text.
```

CHAPTER

# THREE

# *IDELIB* API REFERENCE

## 3.1 idelib.dataset

Module for reading and analyzing Mide Instrumentation Data Exchange (MIDE) files.

Created on Sep 26, 2013

**author**
dstokes

### 3.1.1 Classes

`class idelib.dataset.Dataset(`*stream*, *name=None*, *quiet=True*, *attributes=None*`)`

A collection of sensor data and associated configuration info. Typically represents a single MIDE EMBL file.

Dictionary attributes are all keyed by the relevant ID (sensor ID, channel ID, etc.).

**Variables**

- **loading** – Boolean; *True* if a file is still loading (or has not yet been loaded).
- **fileDamaged** – Boolean; *True* if the file ended prematurely.
- **loadCancelled** – Boolean; *True* if the file loading was aborted part way through.
- **sessions** – A list of individual Session objects in the data set. A valid file will have at least one, even if there are no *Session* elements in the data.
- **sensors** – A dictionary of Sensors.
- **plots** – A dictionary of individual Plots, the modified output of a Channel (or even another plot).
- **transforms** – A dictionary of functions (or function-like objects) for adjusting/calibrating sensor data.

Constructor. Typically, these objects will be instantiated by functions in the *importer* module.

**Parameters**

- **stream** – A file-like stream object containing EBML data.
- **name** – An optional name for the Dataset. Defaults to the base name of the file (if applicable).
- **quiet** – If *True*, non-fatal errors (e.g. schema/file version mismatches) are suppressed.
- **attributes** – A dictionary of arbitrary attributes, e.g. `Attribute` elements parsed from the file. Typically used for diagnostic data.

addChannel(*channelId=None*, *parser=None*, *channelClass=None*, ***kwargs*)

Add a Channel to a Sensor. Note that the *channelId* and *parser* keyword arguments are *not* optional.

**Parameters**

- **channelId** – An unique ID number for the channel.
- **parser** – The Channel's data parser
- **channelClass** – An alternate (sub)class of channel. Defaults to *None*, which creates a standard *Channel*.

addSensor(*sensorId=None*, *name=None*, *sensorClass=None*, *sourceName=None*, *sourceId=None*, *relative=False*, *traceData=None*, *transform=None*, *attributes=None*, *bandwidthLimitId=None*)

Create a new Sensor object, and add it to the dataset, and return it. If the given sensor ID already exists, the existing sensor is returned instead. To modify a sensor or add a sensor object created elsewhere, use *Dataset.sensors[sensorId]* directly.

Note that the *sensorId* keyword argument is *not* optional.

**Parameters**

- **sensorId** (*Optional[int]*) – The ID of the new sensor.
- **name** (*Optional[str]*) – The new sensor's name
- **sensorClass** (*Optional[Type]*) – An alternate (sub)class of sensor. Defaults to *None*, which creates a *Sensor*.
- **sourceName** (*Optional[str]*) – Human-friendly source (authority, network, or other source-of-truth) name for generic/virtual sensor types, particularly 'time' sensors.
- **sourceId** (*Optional[str]*) – Machine-readable, uniquely identifying hash identifying source equivalency for comparing relative sources, particularly 'time' sensors.
- **relative** (*bool*) – *False* if sensor values are Absolute (default), *True* if values are Relative (have an unknown offset, e.g., AC-coupled measurements or time values with an unknown Epoch).
- **transform** (*Optional[Union[int, Transform]]*) – A sensor-level data pre-processing function.
- **attributes** (*Optional[Dict[str, Any]]*) – A dictionary of arbitrary attributes, e.g. `Attribute` elements parsed from the file.
- **traceData** (*Optional[Dict[str, Any]]*) – An optional dictionary of traceability data, such as sensor serial number, etc.
- **bandwidthLimitId** (*Optional[int]*) – The ID of the bandwidth limit (defined in the `BwLimitList` (not currently used).

**Returns**

The new sensor.

addSession(*startTime=None*, *endTime=None*, *utcStartTime=None*)

Create a new session, add it to the Dataset, and return it. Part of the import process.

addTransform(*transform*)

Add a transform (calibration, etc.) to the dataset. Various child objects will reference them by ID. Note: unlike the other *add* methods, this does not instantiate new objects.

addWarning(*warningId=None*, *channelId=None*, *subchannelId=None*, *low=None*, *high=None*, ***kwargs*)

Add a *WarningRange* to the dataset, which indicates when a sensor is reporting values outside of a given range.

**Parameters**

- **warningId** – A unique numeric ID for the *WarningRange*.
- **channelId** – The channel ID of the source being monitored.
- **subchannelId** – The monitored source's subchannel ID.
- **low** – The minimum value of the acceptable range.
- **high** – The maximum value of the acceptable range.

**Returns**

The new *WarningRange* instance.

property channels

A dictionary of individual Sensor channels.

close()

Close the recording file.

property closed

Has the recording file been closed?

endSession()

Set the current session's start/end times. Part of the import process.

property exitCondition

The numeric code number for the condition that stopped the recording.

property fingerprint: Optional[str]

A simple hash for identifying this *Dataset*, so it can be referenced elsewhere, even if moved/renamed. The fingerprint hash is generated from the recording's metadata, so modifying or truncating the sensor data will not affect it. As such, the fingerprint should not be used to verify file integrity.

getPlots(*subchannels=True*, *plots=True*, *debug=True*, *sort=True*, *visibility=0*)

Get all plotable data sources: sensor SubChannels and/or Plots.

**Parameters**

- **subchannels** – Include subchannels if *True*.
- **plots** – Include Plots if *True*.
- **debug** – If *False*, exclude debugging/diagnostic channels.
- **sort** – Sort the plots by name if *True*.
- **visibility** – The plots' maximum level of visibility, for display purposes. Standard visibility ranges: * 0: Standard data sources. Always visible by default. * 10: Optionally visible, of interest but only in certain situations. * 20: Hidden by default, but possibly of use to the user. * 30: Hidden by default, channel used indirectly (but human readable). * 40: Hidden by default, channel used indirectly (not very human readable). * 50: Diagnostic, always hide. Accessible via the idelib API.

hasSession(*sessionId*)

Does the Dataset contain a specific session number?

**hierarchy**()

Get a list of parents/grandparents all the way back to the root. The root is the first item in the list.

**property lastSession**

Retrieve the latest Session.

**path**()

Get the combined names of all the object's parents/grandparents.

**updateTransforms**()

Update the transforms (e.g. the calibration functions) in this dataset. This should be called before utilizing data in the set.

class idelib.dataset.**Channel**(*dataset*, *channelId=None*, *parser=None*, *sensor=None*, *name=None*, *units=None*, *transform=None*, *displayRange=None*, *sampleRate=None*, *cache=False*, *singleSample=None*, *attributes=None*)

Output from a Sensor, containing one or more SubChannels. A Sensor contains one or more Channels. SubChannels of a Channel can be accessed by index like a list or tuple.

**Variables**

- **types** – A tuple with the type of data in each of the Channel's Subchannels.
- **displayRange** – The possible ranges of each subchannel, dictated by the parser. Not necessarily the same as the range of actual values recorded in the file!

Constructor. This should generally be done indirectly via *Dataset.addChannel()*.

**Parameters**

- **sensor** – The parent sensor, if this Channel contains only data from a single sensor.
- **channelId** – The channel's ID, unique within the file.
- **parser** – The channel's EBML data parser.
- **name** – A custom name for this channel.
- **units** – The units measured in this channel, used if units are not explicitly indicated in the Channel's SubChannels.
- **transform** – A Transform object for adjusting sensor readings at the Channel level.
- **displayRange** – A 'hint' to the minimum and maximum values of data in this channel.
- **cache** – If *True*, this channel's data will be kept in memory rather than lazy-loaded.
- **singleSample** – A 'hint' that the data blocks for this channel each contain only a single sample (e.g. temperature/ pressure on an SSX). If *None*, this will be determined from the sample data.
- **attributes** – A dictionary of arbitrary attributes, e.g. `Attribute` elements parsed from the file.

**addSubChannel**(*subchannelId=None*, *channelClass=None*, ***kwargs*)

Create a new SubChannel of the Channel.

**getSession**(*sessionId=None*)

Retrieve data recorded in a Session.

**Parameters**

**sessionId** – The ID of the session to retrieve.

**Returns**
The recorded data.

**Return type**
*EventArray*

`getSubChannel`(*subchannelId*)

Retrieve one of the Channel's SubChannels. All Channels have at least one. A SubChannel object will be automatically generated if one hasn't already explicitly been defined.

**Parameters**
**subchannelId** –

**Returns**
The SubChannel matching the given ID.

`getTransforms`(*id_=None*, *_tlist=None*)

Get a list of all transforms applied to the data, from first (the lowest-level parent) to last (the transform, if any, on the object itself).

`hierarchy`()

Get a list of parents/grandparents all the way back to the root. The root is the first item in the list.

`parseBlock`(*block*, *start=None*, *end=None*, *step=1*, *subchannel=None*)

Parse subsamples out of a data block. Used internally.

**Parameters**

- **block** – The data block from which to parse subsamples.
- **start** – The first block index to retrieve.
- **end** – The last block index to retrieve.
- **step** – The number of steps between samples.
- **subchannel** – If supplied, return only the values for a specific subchannel (i.e. the method is being called by a SubChannel).

**Returns**
A list of tuples, one for each subsample.

`parseBlockByIndex`(*block*, *indices*, *subchannel=None*)

**Convert raw data into a set of subchannel values, returning only**
specific items from the result by index.

**Parameters**

- **block** – The data block element to parse.
- **indices** – A list of sample index numbers to retrieve.
- **subchannel** – If supplied, return only the values for a specific subchannel

**Returns**
A list of tuples, one for each subsample.

`path`()

Get the combined names of all the object's parents/grandparents.

setTransform(*transform*, *update=True*)

Set the transforming function/object. This does not change the value of *raw*, however; the new transform will not be applied unless it is *True*.

updateTransforms()

Recompute cached transform functions.

class idelib.dataset.SubChannel(*parent*, *subchannelId*, *name=None*, *units=(", ")*, *transform=None*, *displayRange=None*, *sensorId=None*, *warningId=None*, *axisName=None*, *attributes=None*, *color=None*, *visibility=0*)

Output from a sensor, derived from a channel containing multiple pieces of data (e.g. the Y from an accelerometer's XYZ). Looks like a 'real' channel.

Constructor. This should generally be done indirectly via *Channel.addSubChannel()*.

**Parameters**

- **parent** – The parent sensor.
- **subchannelId** – The channel's ID, unique within the file.
- **name** – A custom name for this channel.
- **units** – The units measured in this channel, used if units are not explicitly indicated in the Channel's SubChannels. A tuple containing the 'axis name' (e.g. 'Acceleration') and the unit symbol ('g').
- **transform** – A Transform object for adjusting sensor readings at the Channel level.
- **displayRange** – A 'hint' to the minimum and maximum values of data in this channel.
- **sensorId** – The ID of the sensor that generates this SubChannel's data.
- **warningId** – The ID of the *WarningRange* that indicates conditions that may adversely affect data recorded in this SubChannel.
- **axisName** – The name of the axis this SubChannel represents. Use if the *name* contains additional text (e.g. "X" if the name is "Accelerometer X (low-g)").
- **attributes** – A dictionary of arbitrary attributes, e.g. `Attribute` elements parsed from the file.
- **visibility** – The subchannel's level of visibility, for display purposes. The lower the value, the more 'visible' the subchannel.

addSubChannel(**args*, ***kwargs*)

Create a new SubChannel of the Channel.

getSession(*sessionId=None*)

Retrieve a session by ID. If none is provided, the last session in the Dataset is returned.

getSubChannel(**args*, ***kwargs*)

Retrieve one of the Channel's SubChannels. All Channels have at least one. A SubChannel object will be automatically generated if one hasn't already explicitly been defined.

**Parameters**
**subchannelId** –

**Returns**
The SubChannel matching the given ID.

**getTransforms**(*id_=None*, *_tlist=None*)

Get a list of all transforms applied to the data, from first (the lowest-level parent) to last (the transform, if any, on the object itself).

**hierarchy**()

Get a list of parents/grandparents all the way back to the root. The root is the first item in the list.

**parseBlock**(*block*, *start=None*, *end=None*, *step=1*)

Parse subsamples out of a data block. Used internally.

**Parameters**

- **block** – The data block from which to parse subsamples.
- **start** – The first block index to retrieve.
- **end** – The last block index to retrieve.
- **step** – The number of steps between samples.

**parseBlockByIndex**(*block*, *indices*)

Parse specific subsamples out of a data block. Used internally.

**Parameters**

- **block** – The data block from which to parse subsamples.
- **indices** – A list of individual index numbers to get.

**path**()

Get the combined names of all the object's parents/grandparents.

**setTransform**(*transform*, *update=True*)

Set the transforming function/object. This does not change the value of *raw*, however; the new transform will not be applied unless it is *True*.

**updateTransforms**()

Recompute cached transform functions.

class idelib.dataset.**EventArray**(*parentChannel*, *session=None*, *parentList=None*)

A list-like object containing discrete time/value pairs. Data is dynamically read from the underlying EBML file.

Constructor. This should almost always be done indirectly via the *getSession()* method of *Channel* and *SubChannel* objects.

**append**(*block*)

Add one data block's contents to the Channel's list of data. Note that this doesn't double-check the channel ID specified in the data, but it is inadvisable to include data from different channels.

**Attention**
Added elements must be in chronological order!

**arrayJitterySlice**(*start=None*, *end=None*, *step=1*, *jitter=0.5*, *display=False*)

Create an array of events within a range of indices.

**Parameters**

- **start** – The first index in the range, or a slice.
- **end** – The last index in the range. Not used if *start* is a slice.
- **step** – The step increment. Not used if *start* is a slice.

- **jitter** – The amount by which to vary the sample time, as a normalized percentage of the regular time between samples.
- **display** – If *True*, the *EventArray* transform (i.e. the 'display' transform) will be applied to the data.

**Returns**
a structured array of events in the specified index range.

**arrayMinMeanMax**(*startTime=None*, *endTime=None*, *padding=0*, *times=True*, *display=False*, *iterator=<built-in function iter>*)

Get the minimum, mean, and maximum values for blocks within a specified interval.

**Parameters**

- **startTime** – The first time (in microseconds by default), *None* to start at the beginning of the session.
- **endTime** – The second time, or *None* to use the end of the session.
- **times** – If *True* (default), the results include the block's starting time.
- **display** – If *True*, the final 'display' transform (e.g. unit conversion) will be applied to the results.
- **iterator** – A function that iterates the output. Intended for allowing iteration to be externally cancelled (e.g., in a GUI).

**Returns**
A structured array of data block statistics (min, mean, and max, respectively).

**arrayRange**(*startTime=None*, *endTime=None*, *step=1*, *display=False*)

Get a set of data occurring in a given time interval.

**Parameters**

- **startTime** – The first time (in microseconds by default), *None* to start at the beginning of the session.
- **endTime** – The second time, or *None* to use the end of the session.
- **step** – The number of steps between samples.
- **display** – If *True*, the final 'display' transform (e.g. unit conversion) will be applied to the results.

**Returns**
a structured array of events in the specified time interval.

**arrayResampledRange**(*startTime*, *stopTime*, *maxPoints*, *padding=0*, *jitter=0*, *display=False*)

Retrieve the events occurring within a given interval, undersampled as to not exceed a given length (e.g. the size of the data viewer's screen width).

**arraySlice**(*start=None*, *end=None*, *step=1*, *display=False*)

Create an array of events within a range of indices.

**Parameters**

- **start** – The first index in the range, or a slice.
- **end** – The last index in the range. Not used if *start* is a slice.
- **step** – The step increment. Not used if *start* is a slice.

- **display** – If *True*, the *EventArray* transform (i.e. the 'display' transform) will be applied to the data.

**Returns**
a structured array of events in the specified index range.

**arrayValues**(*start=None*, *end=None*, *step=1*, *subchannels=True*, *display=False*)

Get all values in the given index range (w/o times).

**Parameters**

- **start** – The first index in the range, or a slice.
- **end** – The last index in the range. Not used if *start* is a slice.
- **step** – The step increment. Not used if *start* is a slice.
- **subchannels** – A list of subchannel IDs or Boolean. *True* will return all subchannels in native order.
- **display** – If *True*, the *EventArray* transform (i.e. the 'display' transform) will be applied to the data.

**Returns**
a structured array of values in the specified index range.

**copy**(*newParent=None*)

Create a shallow copy of the event list.

**exportCsv**(*stream*, *start=None*, *stop=None*, *step=1*, *subchannels=None*, *callback=None*, *callbackInterval=0.01*, *timeScalar=1*, *raiseExceptions=False*, *dataFormat='%.6f'*, *delimiter=', '*, *useUtcTime=False*, *useIsoFormat=False*, *headers=False*, *removeMean=None*, *meanSpan=None*, *display=False*, *noBivariates=None*)

Export events as CSV to a stream (e.g. a file).

**Parameters**

- **stream** – The stream object to which to write CSV data.
- **start** (*Optional[int]*) – The first event index to export.
- **stop** (*Optional[int]*) – The last event index to export.
- **step** (*int*) – The number of events between exported lines.
- **subchannels** (*Optional[Iterable]*) – A sequence of individual subchannel numbers to export. Only applicable to objects with subchannels. *True* (default) exports them all.
- **callback** (*Optional[Callable]*) – A function (or function-like object) to notify as work is done. It should take four keyword arguments: *count* (the current line number), *total* (the total number of lines), *error* (an exception, if raised during the export), and *done* (will be *True* when the export is complete). If the callback object has a *cancelled* attribute that is *True*, the CSV export will be aborted. The default callback is *None* (nothing will be notified).
- **callbackInterval** (*float*) – The frequency of update, as a normalized percent of the total lines to export.
- **timeScalar** (*float*) – A scaling factor for the event times. The default is 1 (microseconds). Not applicable when exporting with UTC timestamps, which are always seconds.
- **raiseExceptions** (*bool*) – If *False*, all exceptions will be handled quietly, passed along to the callback.

- **dataFormat** (*str*) – The number of decimal places to use for the data. This is the same format as used when formatting floats.
- **delimiter** (*str*) – The characters separating columns in the output.
- **useUtcTime** (*bool*) – If *True*, times are written as the UTC timestamp. If *False*, times are relative to the recording.
- **useIsoFormat** (*bool*) – If *True*, the time column is written as the standard ISO date/time string. Only applies if *useUtcTime* is *True*.
- **headers** (*bool*) – If *True*, the first line of the CSV will contain the names of each column.
- **removeMean** (*Optional[bool]*) – Overrides the EventArray's mean removal for the export.
- **meanSpan** (*Optional[int]*) – The span of the mean removal for the export. -1 removes the total mean.
- **display** (*bool*) – If *True*, export using the EventArray's 'display' transform (e.g. unit conversion).
- **noBivariates** (*Optional[bool]*) – If *True*, do not apply the second value in bivariate calibration polynomials (e.g., temperature compensation).

**Returns**
Tuple: The number of rows exported and the elapsed time.

**Return type**
tuple[int, datetime.timedelta]

**getEventIndexBefore**(*t*)

Get the index of an event occurring on or immediately before the specified time.

**Parameters**
**t** – The time (in microseconds)

**Returns**
The index of the event preceding the given time, -1 if the time occurs before the first event.

**getEventIndexNear**(*t*)

The the event occurring closest to a specific time.

**Parameters**
**t** – The time (in microseconds)

**Returns**

**getInterval**()

Get the first and last event times in the set.

**getMax**(*startTime=None*, *endTime=None*, *display=False*, *iterator=<built-in function iter>*)

Get the event with the maximum value, optionally within a specified time range. For Channels, returns the maximum among all Subchannels.

**Parameters**

- **startTime** – The starting time. Defaults to the start.
- **endTime** – The ending time. Defaults to the end.
- **display** – If *True*, the final 'display' transform (e.g. unit conversion) will be applied to the results.

- **iterator** – A function that iterates the output. Intended for allowing iteration to be externally cancelled (e.g., in a GUI).

**Returns**
The event with the maximum value.

**getMean**(*startTime=None*, *endTime=None*, *display=False*, *iterator=<built-in function iter>*)

Get the mean value of all events, optionally within a specified time range. For Channels, returns the mean among all Subchannels.

**Parameters**

- **startTime** – The starting time. Defaults to the start.
- **endTime** – The ending time. Defaults to the end.
- **display** – If *True*, the final 'display' transform (e.g. unit conversion) will be applied to the results.
- **iterator** – A function that iterates the output. Intended for allowing iteration to be externally cancelled (e.g., in a GUI).

**Returns**
The event with the minimum value.

**getMeanNear**(*t*, *outOfRange=False*)

Retrieve the mean value near a given time.

**getMin**(*startTime=None*, *endTime=None*, *display=False*, *iterator=<built-in function iter>*)

Get the event with the minimum value, optionally within a specified time range. For Channels, returns the minimum among all Subchannels.

**Parameters**

- **startTime** – The starting time. Defaults to the start.
- **endTime** – The ending time. Defaults to the end.
- **display** – If *True*, the final 'display' transform (e.g. unit conversion) will be applied to the results.
- **iterator** – A function that iterates the output. Intended for allowing iteration to be externally cancelled (e.g., in a GUI).

**Returns**
The event with the minimum value.

**getMinMeanMax**(*startTime=None*, *endTime=None*, *padding=0*, *times=True*, *display=False*, *iterator=<built-in function iter>*)

Get the minimum, mean, and maximum values for blocks within a specified interval. (Currently an alias of *arrayMinMeanMax*.)

**Parameters**

- **startTime** – The first time (in microseconds by default), *None* to start at the beginning of the session.
- **endTime** – The second time, or *None* to use the end of the session.
- **times** – If *True* (default), the results include the block's starting time.
- **display** – If *True*, the final 'display' transform (e.g. unit conversion) will be applied to the results.

- **iterator** – A function that iterates the output. Intended for allowing iteration to be externally cancelled (e.g., in a GUI).

**Returns**
A structured array of data block statistics (min, mean, and max, respectively).

**getRange**(*startTime=None*, *endTime=None*, *display=False*)

Get a set of data occurring in a given time interval. (Currently an alias of *arrayRange*.)

**Parameters**

- **startTime** – The first time (in microseconds by default), *None* to start at the beginning of the session.
- **endTime** – The second time, or *None* to use the end of the session.
- **display** – If *True*, the final 'display' transform (e.g. unit conversion) will be applied to the results.

**Returns**
a collection of events in the specified time interval.

**getRangeIndices**(*startTime*, *endTime*)

Get the first and last event indices that fall within the specified interval.

**Parameters**

- **startTime** – The first time (in microseconds by default), *None* to start at the beginning of the session.
- **endTime** – The second time, or *None* to use the end of the session.

**getRangeMinMeanMax**(*startTime=None*, *endTime=None*, *subchannel=None*, *display=False*, *iterator=<built-in function iter>*)

Get the single minimum, mean, and maximum value for blocks within a specified interval. Note: Using this with a parent channel without specifying a subchannel number can produce meaningless data if the channels use different units or are on different scales.

**Parameters**

- **startTime** – The first time (in microseconds by default), *None* to start at the beginning of the session.
- **endTime** – The second time, or *None* to use the end of the session.
- **subchannel** – The subchannel ID to retrieve, if the EventArray's parent has subchannels.
- **display** – If *True*, the final 'display' transform (e.g. unit conversion) will be applied to the results.
- **iterator** – A function that iterates the output. Intended for allowing iteration to be externally cancelled (e.g., in a GUI).

**Returns**
A namedtuple of aggregated event statistics (min, mean, and max, respectively).

**getSampleRate**(*idx=None*)

Get the channel's sample rate. This is either supplied as part of the channel definition or calculated from the actual data and cached.

**Parameters**
**idx** – Because it is possible for sample rates to vary within a channel, an event index can be specified; the sample rate for that event and its siblings will be returned.

**Returns**
The sample rate, as samples per second (float)

`getSampleTime`(*idx=None*)

Get the time between samples.

**Parameters**
**idx** – Because it is possible for sample rates to vary within a channel, an event index can be specified; the time between samples for that event and its siblings will be returned.

**Returns**
The time between samples (us)

`getTransforms`(*id_=None*, *_tlist=None*)

Get a list of all transforms applied to the data, from first (the lowest-level parent) to last (the transform, if any, on the object itself).

`getValueAt`(*at*, *outOfRange=False*, *display=False*)

Retrieve the value at a specific time, interpolating between existing events.

**Parameters**

- **at** – The time at which to take the sample.
- **outOfRange** – If *False*, times before the first sample or after the last will raise an *IndexError*. If *True*, the first or last time, respectively, is returned.
- **display** – If *True*, export using the EventArray's 'display' transform (e.g. unit conversion).

`hierarchy`()

Get a list of parents/grandparents all the way back to the root. The root is the first item in the list.

`iterJitterySlice`(*start=None*, *end=None*, *step=1*, *jitter=0.5*, *display=False*)

Create an iterator producing events for a range of indices.

**Parameters**

- **start** – The first index in the range, or a slice.
- **end** – The last index in the range. Not used if *start* is a slice.
- **step** – The step increment. Not used if *start* is a slice.
- **jitter** – The amount by which to vary the sample time, as a normalized percentage of the regular time between samples.
- **display** – If *True*, the *EventArray* transform (i.e. the 'display' transform) will be applied to the data.

**Returns**
an iterable of events in the specified index range.

`iterMinMeanMax`(*startTime=None*, *endTime=None*, *padding=0*, *times=True*, *display=False*)

Get the minimum, mean, and maximum values for blocks within a specified interval.

**Parameters**

- **startTime** – The first time (in microseconds by default), *None* to start at the beginning of the session.
- **endTime** – The second time, or *None* to use the end of the session.
- **times** – If *True* (default), the results include the block's starting time.

- **display** – If *True*, the final 'display' transform (e.g. unit conversion) will be applied to the results.

**Returns**
An iterator producing sets of three events (min, mean, and max, respectively).

**iterRange**(*startTime=None*, *endTime=None*, *step=1*, *display=False*)

Get a set of data occurring in a given interval.

**Parameters**

- **startTime** – The first time (in microseconds by default), *None* to start at the beginning of the session.
- **endTime** – The second time, or *None* to use the end of the session.
- **step** – The number of steps between samples.
- **display** – If *True*, the final 'display' transform (e.g. unit conversion) will be applied to the results.

**iterResampledRange**(*startTime*, *stopTime*, *maxPoints*, *padding=0*, *jitter=0*, *display=False*)

Retrieve the events occurring within a given interval, undersampled as to not exceed a given length (e.g. the size of the data viewer's screen width).

**iterSlice**(*start=None*, *end=None*, *step=1*, *display=False*)

Create an iterator producing events for a range of indices.

**Parameters**

- **start** – The first index in the range, or a slice.
- **end** – The last index in the range. Not used if *start* is a slice.
- **step** – The step increment. Not used if *start* is a slice.
- **display** – If *True*, the *EventArray* transform (i.e. the 'display' transform) will be applied to the data.

**Returns**
an iterable of events in the specified index range.

**itervalues**(*start=None*, *end=None*, *step=1*, *subchannels=True*, *display=False*)

Iterate all values in the given index range (w/o times).

**Parameters**

- **start** – The first index in the range, or a slice.
- **end** – The last index in the range. Not used if *start* is a slice.
- **step** – The step increment. Not used if *start* is a slice.
- **subchannels** – A list of subchannel IDs or Boolean. *True* will return all subchannels in native order.
- **display** – If *True*, the *EventArray* transform (i.e. the 'display' transform) will be applied to the data.

**Returns**
an iterable of structured array value blocks in the specified index range.

**path**()

Get the combined names of all the object's parents/grandparents.

**setTransform**(*transform*, *update=True*)

Set the transforming function/object. This does not change the value of *raw*, however; the new transform will not be applied unless it is *True*.

**updateTransforms**(*recurse=True*)

(Re-)Build and (re-)apply the transformation functions.

**class** idelib.dataset.**WarningRange**(*dataset*, *warningId=None*, *channelId=None*, *subchannelId=None*, *low=None*, *high=None*, *attributes=None*)

An object for indicating when a set of events goes outside of a given range. Originally created for flagging periods of extreme temperatures that will affect accelerometer readings.

For efficiency, the source data should have relatively few samples (e.g. a low sample rate).

Constructor.

**property displayName**

A nice, human-readable description of this warning range, for use with user interfaces.

**getRange**(*start=None*, *end=None*, *sessionId=None*, *iterator=<built-in function iter>*)

Retrieve the invalid periods within a given range of events.

**Returns**

A list of invalid periods' [start, end] times.

**getSessionSource**(*sessionId=None*)

**getValueAt**(*at*, *sessionId=None*, *source=None*)

Retrieve the value at a specific time.

## 3.2 idelib.importer

idelib.importer.**importFile**(*filename=''*, *startTime=None*, *endTime=None*, *channels=None*, *updater=None*, *parserTypes=None*, *defaults=None*, *name=None*, *quiet=False*, ***kwargs*)

Create a new Dataset object and import the data from a MIDE file. Primarily for testing purposes. The GUI does the file creation and data loading in two discrete steps, as it will need a reference to the new document before the loading starts. :see: *readData()*

idelib.importer.**openFile**(*stream*, *updater=None*, *parserTypes=None*, *defaults=None*, *name=None*, *quiet=False*)

Create a *Dataset* instance and read the header data (i.e. non-sample- data). When called by a GUI, this function should be considered 'modal,' in that it shouldn't run in a background thread, unlike *readData()*.

**Parameters**

- **stream** – The file or file-like object containing the EBML data.
- **updater** – A function (or function-like object) to notify as work is done. It should take four keyword arguments: *count* (the current line number), *total* (the total number of samples), *error* (an unexpected exception, if raised during the import), and *done* (will be *True* when the export is complete). If the updater object has a *cancelled* attribute that is *True*, the import will be aborted. The default callback is *None* (nothing will be notified).
- **parserTypes** – A collection of *parsers.ElementHandler* classes.
- **defaults** – A nested dictionary containing a default set of sensors, channels, and subchannels. These will only be used if the dataset contains no sensor/channel/subchannel definitions.

- **name** – An optional name for the Dataset. Defaults to the base name of the file (if applicable).
- **quiet** – If *True*, non-fatal errors (e.g. schema/file version mismatches) are suppressed.

**Returns**
The opened (but still 'empty') *dataset.Dataset*

idelib.importer.**readData**(*doc*, *source=None*, *startTime=None*, *endTime=None*, *channels=None*, *updater=None*, *total=None*, *bytesRead=0*, *samplesRead=0*, *parserTypes=None*, ***kwargs*)

Import the data from a file into a Dataset.

**Parameters**

- **doc** – The Dataset document into which to import the data.
- **source** – An alternate Dataset to merge into the main one.
- **startTime** – The start of the extraction range, relative to the recording's start.
- **endTime** – The end of the extraction range, relative to the recording's end.
- **channels** – A list of channel IDs to import. If *None* (the default), all channels are imported.
- **updater** – A function (or function-like object) to notify as work is done. It should take four keyword arguments: *count* (the current line number), *total* (the total number of samples), *error* (an unexpected exception, if raised during the import), and *done* (will be *True* when the export is complete). If the updater object has a *cancelled* attribute that is *True*, the import will be aborted. The default callback is *None* (nothing will be notified).
- **total** – The total number of bytes in the file(s) being imported. Defaults to the size of the current file, but can be used to display an overall progress when merging multiple recordings. For display purposes.
- **bytesRead** – The number of bytes already imported. Mainly for merging multiple recordings. For display purposes.
- **samplesRead** – The total number of samples imported. Mainly for merging multiple recordings.
- **parserTypes** – A collection of *parsers.ElementHandler* classes.

**Returns**
The total number of samples read.

## 3.3 idelib.sync

Functions to assist in syncing one file to another, and functions to adjust files' timestamps based on the GPS/GNSS satelite time; two separate but related operations.

### 3.3.1 Syncing

Syncing modifies the start time and timestamps of one or more `Dataset` objects' recording session to match a 'reference' `Dataset`. Syncing is non-destructive; the sync can be repeatedly changed (i.e., a `Dataset` can be synced to a different reference) or removed without any cumulative effect to the timing.

Syncing can only be done with IDE files containing a common time reference channel; currently, only Wi-Fi enabled devices (i.e., the enDAQ W series) connected to the same Wi-Fi access point can create these. Note that the time sync reference channels are not shown in *enDAQ Lab*, but can be seen in `Dataset.channels`.

### 3.3.2 GPS/GNSS Time Adjustment

Adjusting a recording's starting timestamp using GPS/GNSS time requires the file was recorded on a device with a GPS module (e.g., an enDAQ W series recorder) and contains GPS/GNSS timing data. Please note that the former does not guarantee the latter; the GPS timing data may be missing if satellite reception was poor, or the recording ended before the signal was acquired.

Syncing and GPS/GNSS time adjustment can be used together by first applying the GPS/GNSS adjustment to the 'reference' recording before syncing other files to it. All recordings must have the same sync time channel, but only the 'reference' recording needs the GPS/GNSS data as well.

### 3.3.3 Usage

While this module implements several functions, the primary ones are *idelib.sync.sync()* and *idelib.sync.applyGNSSTime()*. *idelib.sync.updateUserdata()* (in conjunction with `idelib.userdata.writeUserData()`)can be used to save a recording's calculated time/sync info into itself for later use.

**exception** idelib.sync.**SyncError**

Exception raised when files cannot be synchronized.

idelib.sync.**applyGNSSTime**(*data*, *clear=False*)

Modify the recording's UTC start time using GPS/GNSS time data. Note that recordings with GPS/GNSS time bases cannot be synchronized to another recording; they should be the 'reference' recording to which others are synchronized.

**Parameters**

- **data** (Dataset) – The recording to modify. It must contain GPS/GNSS time data.
- **clear** – If *True*, remove any previous synchronization before updating the recording's time-base. If *False* and the recording has been synced to another, a *SyncError* will be raised.

**Returns**

The recording's original initial starting time and the new GNSS-corrected time (UNIX epoch seconds).

**Return type**

*Tuple*[float, float]

idelib.sync.**applySyncInfo**(*dataset*, *info*, *validate=True*)

Apply a dictionary of sync info to a dataset.

Note that this function runs synchronously, and can block/be blocked by other functions affecting the contents of the `Dataset`.

**Parameters**

- **dataset** (Dataset) – The *Dataset* to which the sync info will be applied.

- **info** (*Dict[str, Any]*) – The dictionary of synchronization info to validate. The keys match the names of `SyncInfo` child elements in the `mide_ide.xml` EBML schema.
- **validate** (*bool*) – If *True*, validate the sync info before applying.

idelib.sync.**getCommonSensorIds**(**datasets*)

Get the Sensor ID of the time reference shared by two or more recordings. This does *not* verify that any time data has been recorded from it, only that the sensor exists.

idelib.sync.**getGNSSTimebase**(*data*)

Get the recording's fine-grained UTC start time from its GPS/GNSS data.

**Parameters**
**data** (*Union[Dataset, EventArray]*) – The data from which to get the timebase. It can be either a *Dataset* (in which case it uses the first GNSS time source found) or an *EventArray* (to get the timebase from a specific subchannel).

**Returns**
The recording's updated UTC start time with fractional seconds (UNIX epoch).

**Return type**
float

idelib.sync.**getSyncInfo**(*dataset*)

Get a dictionary of sync info from a recording.

idelib.sync.**getSyncSensor**(*dataset*, *sourceId=None*, *sourceName=None*)

Get a specific time reference sensor from a *Dataset* by name and/or identifier.

**Parameters**

- **dataset** (Dataset) – The *Dataset* from which to get the sensor.
- **sourceId** (*Optional[str]*) – The time reference sensor's unique ID (e.g., the MAC address of a Wi-Fi access point generating TSF data).
- **sourceName** (*Optional[str]*) – The name of the time reference sensor (e.g., the name of a Wi-Fi access point generating TSF data).

idelib.sync.**getSyncSensors**(*dataset*)

Get all the 'sensors' in a *Dataset* that can be used for time synchronization.

idelib.sync.**getSyncSources**(*dataset*)

Get the *SubChannel* instances that can be used for time synchronization.

**Parameters**
**dataset** (Dataset) – The *Dataset* from which to get the sources.

idelib.sync.**getSyncTimeZero**(*data*, *start=None*, *end=None*, *sensorId=None*, *clear=False*)

Get the sync reference's time corresponding to the recording's relative timestamp zero.

**Parameters**

- **data** (*Union[Dataset, EventArray]*) – The data from which to get the reference time. It can be either a *Dataset* (in which case it uses either the first sync source found or the one indicated by *sensorId*) or an *EventArray* (to use a specific subchannel as the time reference).

- **start** (*Optional[int]*) – The starting index into the time reference data, to use a limited range to compute the zero offset. For use with large and/or noisy Datasets. Defaults to the beginning of the data.
- **end** (*Optional[int]*) – The ending index into the time reference data, to use a limited range to compute the zero offset. For use with large and/or noisy Datasets. Defaults to the end of the data.
- **sensorId** (*Optional[int]*) – The ID of a specific time reference sensor. Only used if *data* is a *Dataset*. If *None*, the first time reference sensor found is used.
- **clear** (*bool*) – If *False*, use cached zero offset (if present).

**Returns**
The sync reference time (in microseconds) corresponding to the recording's relative timestamp zero.

**Return type**
int

idelib.sync.**hasSyncReferenceInfo**(*info*)

Check if a dictionary of sync info contains the information needed to sync to a reference recording (e.g., copied from a recording that has already been synced).

**Parameters**
**info** (*Dict[str, Any]*) – The dictionary of synchronization info to check. The keys match the names of `SyncInfo` child elements in the `mide_ide.xml` EBML schema.

idelib.sync.**isSynced**(*data*)

Has the data been synchronized to another recording?

idelib.sync.**loadSyncInfo**(*dataset*, *refresh=False*)

Read and apply sync info from a file's userdata.

**Parameters**
- **dataset** (Dataset) – The *Dataset* from which to load the sync info.
- **refresh** (*bool*) – If *True*, ignore any cached user data and reload from the file.

**Returns**
*True* if sync info was present, *False* otherwise. Errors in the sync data will raise exceptions (e.g., *SyncError*).

**Return type**
bool

idelib.sync.**makeSyncReferenceInfo**(*info*)

Create a dictionary of sync 'reference' info from a dictionary of sync 'target' info. For use with *applySyncInfo()* when applying sync info copied from another recording, to sync to that recording.

idelib.sync.**removeGNSSTime**(*data*)

Remove the GPS/GNSS UTC start time from a recording, reverting to the original, device-generated initial timestamp.

idelib.sync.**removeSync**(*data*, *clean=False*)

Remove sync info from a `Dataset` recording session. The UTC start time and timestamp offsets will revert to those originally in the file.

Note that this function runs synchronously, and can block/be blocked by other functions affecting the contents of the `Dataset`.

**Parameters**

- **data** (*Union[Dataset, EventArray, Session]*) – The data from which to remove the sync info.
- **clean** (*bool*) – If *False*, the information syncing this recording to another is removed, but metadata about the file itself is kept. If *True*, all sync-related info is completely removed.

idelib.sync.**sync**(*reference*, **datasets*, *sensorId=None*, *inherit=True*)

Synchronize one or more recordings with a canonical 'reference' recording. The 'reference' is not modified. Synched recordings will have their timestamps and UTC start time offset to match the reference.

Note that this function runs synchronously, and can block/be blocked by other functions affecting the contents of the `Dataset`.

**Parameters**

- **reference** (`Dataset`) – The reference *Dataset* to which to synchronize the others.
- **datasets** (`Dataset`) – One or more *Dataset* objects to synchronize.
- **sensorId** (*Optional[int]*) – The time reference's sensor ID. If *None*, the first common time reference sensor detected is used. Use if the recordings have multiple time references in common.
- **inherit** (*bool*) – If *True* and the *reference* Dataset has been synced to another recording, sync the *datasets* to the same recording *reference* has been synced to. If *False*, clear existing sync info from the *reference* and sync the *datasets* to the reference's zero time.

idelib.sync.**updateUserdata**(*dataset*)

Create or update sync info in a Dataset's userdata. Note that this does not save the updated user data to the file; the function `idelib.userdata.writeUserData()` must be called explicitly.

**Parameters**

**dataset** (`Dataset`) – The *Dataset* to update.

idelib.sync.**validateSyncInfo**(*dataset*, *info*)

Check that a dictionary of sync info is valid for a dataset.

**Parameters**

- **dataset** (`Dataset`) – The *Dataset* to which the sync info will be applied.
- **info** (*Dict[str, Any]*) – The dictionary of synchronization info to validate. The keys match the names of `SyncInfo` child elements in the `mide_ide.xml` EBML schema.

## 3.4 idelib.util

Utility functions for doing low-level, general-purpose EBML reading and writing.

idelib.util.**extractTime**(*doc*, *out*, *startTime=0*, *endTime=None*, *channels=None*, *updater=None*)

Efficiently extract data within a certain interval from an IDE file. Note that due to the way data is stored in an IDE, the exported interval will be slightly wider than the specified start and end times; this ensures the data is copied verbatim and without loss.

**Parameters**

- **doc** – A loaded *Dataset* or the name of an IDE file.
- **out** – A filename or stream to which to save the extracted data.
- **startTime** – The start of the extraction range, relative to the recording's start.
- **endTime** – The end of the extraction range, relative to the recording's end.
- **channels** – A list of channel IDs to specifically export. If *None*, all channels will be exported. Note excluded channels will still appear in the new IDE's *channels* dictionary, but the file will contain no data for them.
- **updater** – A function (or function-like object) to notify as work is done. It should take four keyword arguments: *count* (the current line number), *total* (the total number of samples), *error* (an unexpected exception, if raised during the import), and *done* (will be *True* when the split is complete). If the updater object has a *cancelled* attribute that is *True*, the import will be aborted. The default callback is *None* (nothing will be notified).

**Returns**

The total number of bytes written, and total number of ChannelDataBlock elements copied.

idelib.util.**getExitCondition**(*recording*)

**Get the ExitCond Attribute from the end of a recording, if present.**

The result will be an integer:

- 1: Button press
- 2: USB connection
- 3: Recording time limit reached
- 4: Low battery
- 5: File size limit reached
- 128: I/O error (can occur if disk is full or 4GB FAT32 size limit reached.

**Parameters**

**recording** – The IDE filename, an *idelib.dataset.Dataset*, an *ebmlite.core.Dataset*, or stream containing IDE data.

idelib.util.**getLength**(*doc*)

Efficiently retrieve the start and end times of an IDE file, without it having to be fully imported.

**Parameters**

**doc** – The IDE filename, an *idelib.dataset.Dataset*, or stream containing IDE data.

**Returns**

The timestamps of the first and last samples in the file, in microseconds, relative to the start of the recording.

idelib.util.**verify**(*data*, *schema=None*)

Basic sanity-check of data validity. If the data is bad an exception will be raised. The specific exception varies depending on the problem in the data.

**Parameters**

**schema** – The full module name of the EBML schema.

**Returns**

*True*. Any problems will raise exceptions.

CHAPTER

FOUR

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## i

# INDEX

## H



## I



## L



## M



## O



## P



## R



## S



## U



## V



## W