baseband Documentation

Release 1.1.1

Marten H. van Kerkwijk, Chenchong Zhu

Aug 01, 2018

Contents

Welcome to the Baseband documentation! Baseband is a package for reading and writing VLBI and other radio baseband files, with the aim of simplifying and streamlining data conversion and standardization. It provides:

- File input/output objects for supported radio baseband formats, enabling selective decoding of data into `Numpy arrays`, and encoding user-defined arrays into baseband formats. Supported formats are listed under *specific file formats*.
- Helper objects for reading from and writing to an ordered sequence of files as if it was a single file.

Part I

Overview

CHAPTER 1

Installation

1.1 Requirements

Baseband requires:

- Astropy v2.0 or later
- Numpy v1.9 or later

1.2 Installing Baseband

1.2.1 Using pip

To install Baseband with pip, run:

```
pip3 install baseband
```

Note: To run without pip potentially updating Numpy and Astropy, run, include the `--no-deps` flag.

1.2.2 Obtaining source code

The source code and latest development version of Baseband can found on its GitHub repo. You can get your own clone using:

```
git clone git@github.com:mhvk/baseband.git
```

Of course, it is even better to fork it on GitHub, and then clone your own repository, so that you can more easily contribute!

1.2.3 Running code without installing

As Baseband is purely Python, it can be used without being built or installed, by appending the directory it is located in to the PYTHON_PATH environment variable. Alternatively, you can use sys.path within Python to append the path:

```
import sys
sys.path.append(BASEBAND_PATH)
```

where BASEBAND_PATH is the directory you downloaded or cloned Baseband into.

1.2.4 Installing source code

If you want Baseband to be more broadly available, either to all users on a system, or within, say, a virtual environment, use setup.py in the root directory by calling:

```
python3 setup.py install
```

For general information on setup.py, see its documentation . Many of the setup.py options are inherited from Astropy (specifically, from Astropy -affiliated package manager) and are described further in Astropy's installation documentation .

1.3 Testing the installation

The root directory setup.py can also be used to test if Baseband can successfully be run on your system:

```
python3 setup.py test
```

or, inside of Python:

```
import baseband
baseband.test()
```

These tests require pytest to be installed. Further documentation can be found on the Astropy running tests documentation .

1.4 Building documentation

Note: As with Astropy, building the documentation is unnecessary unless you are writing new documentation or do not have internet access, as Baseband's documentation is available online at baseband.readthedocs.io.

The Baseband documentation can be built again using setup.py from the root directory:

```
python3 setup.py build_docs
```

This requires to have Sphinx installed (and its dependencies).

CHAPTER 2

Getting Started

For most file formats, one can simply import baseband and use `baseband.open` to access the file. This gives one a filehandle from which one can read decoded samples:

```
>>> import baseband
>>> from baseband.data import SAMPLE_DADA
>>> fh = baseband.open(SAMPLE_DADA)
>>> fh.read(3)
array([[ -38.-38.j,  -38.-38.j],
       [ -38.-38.j,  -40. +0.j],
       [-105.+60.j,   85.-15.j]], dtype=complex64)
>>> fh.close()
```

For other file formats, a bit more information is needed. Below, we cover the basics of *inspecting files*, *reading* from and *writing* to files, and *converting* from one format to another. We assume that Baseband as well as NumPy and the Astropy units module have been imported:

```
>>> import baseband
>>> import numpy as np
>>> import astropy.units as u
```

2.1 Inspecting Files

Baseband allows you to quickly determine basic properties of a file, including what format it is, using the `baseband.file_info` function. For instance, it shows that the sample VDIF file that comes with Baseband is very short (sample files can all be found in the `baseband.data` module):

```
>>> import baseband.data
>>> baseband.file_info(baseband.data.SAMPLE_VDIF)
Stream information:
start_time = 2014-06-16T05:56:07.000000000
stop_time = 2014-06-16T05:56:07.001250000
```

(continues on next page)

(continued from previous page)

```
sample_rate = 32.0 MHz
shape = (40000, 8)
format = vdif
bps = 2
complex_data = False

File information:
edv = 3
frame_rate = 1600.0 Hz
samples_per_frame = 20000
sample_shape = (8, 1)
```

The same function will also tell you when more information is needed. For instance, for Mark 5B files one needs the number of channels used, as well as (roughly) when the data were taken:

```
>>> baseband.file_info(baseband.data.SAMPLE_MARK5B)
File information:
format = mark5b
frame_rate = 6400.0 Hz
bps = 2
complex_data = False

missing:  nchan: needed to determine sample shape and rate.
          kday, ref_time: needed to infer full times.
>>> from astropy.time import Time
>>> baseband.file_info(baseband.data.SAMPLE_MARK5B, nchan=8, ref_time=Time('2014-01-01'))
Stream information:
start_time = 2014-06-13T05:30:01.000000000
stop_time = 2014-06-13T05:30:01.000625000
sample_rate = 32.0 MHz
shape = (20000, 8)
format = mark5b
bps = 2
complex_data = False

File information:
frame_rate = 6400.0 Hz
samples_per_frame = 5000
sample_shape = (8,)
```

The information is gleaned from `info` properties on the various file and stream readers (see below).

Note: The one format for which `file_info` works a bit differently is GSB, as this format requires separate time-stamp and raw data files. Only the timestamp file can be inspected usefully.

2.2 Reading Files

2.2.1 Opening Files

As shown at the very start, files can be opened with the general `baseband.open` function. This will try to determine the file type using `file_info`, load the corresponding baseband module, and then open the file using that module's master input/output function.

Generally, if one knows the file type, one might as well work with the corresponding module directly. For instance, to explicitly use the DADA reader to open the sample DADA file included in Baseband, one can use the DADA module's `open` function:

```
>>> from baseband import dada
>>> from baseband.data import SAMPLE_DADA
>>> fh = dada.open(SAMPLE_DADA, 'rs')
>>> fh.read(3)
array([[ -38.-38.j,  -38.-38.j],
       [ -38.-38.j,  -40. +0.j],
       [-105.+60.j,   85.-15.j]], dtype=complex64)
>>> fh.close()
```

In general, file I/O and data manipulation use the same syntax across all file formats. When opening Mark 4 and Mark 5B files, however, some additional arguments may need to be passed (as was the case above for inspecting a Mark 5B file, and indeed this is a good way to find out what is needed). Notes on such features and quirks of individual formats can be found in the API entries of their open functions, and within the *Specific file format* documentation.

For the rest of this section, we will stick to VDIF files.

2.2.2 Decoding Data and the Sample File Pointer

By giving the openers a `'rs'` flag, which is the default, we open files in "stream reader" mode, where a file is accessed as if it were a stream of samples. For VDIF, `open` will then return an instance of `VDIFStreamReader`, which wraps a raw data file with methods to decode the binary *data frames* and seek to and read data *samples*. To decode the first 12 samples into a `ndarray`, we would use the `read` method:

```
>>> from baseband import vdif
>>> from baseband.data import SAMPLE_VDIF
>>> fh = vdif.open(SAMPLE_VDIF, 'rs')
>>> d = fh.read(12)
>>> type(d)
<... 'numpy.ndarray'>
>>> d.shape
(12, 8)
>>> d[:, 0].astype(int)    # First thread.
array([-1, -1,  3, -1,  1, -1,  3, -1,  1,  3, -1,  1])
```

As discussed in detail in the *VDIF section*, VDIF files are sequences of data frames, each of which is comprised of a *header* (which holds information like the time at which the data was taken) and a *payload*, or block of data. Multiple concurrent time streams can be stored within a single frame; each of these is called a "*channel*". Moreover, groups of channels can be stored over multiple frames, each of which is called a "*thread*". Our sample file is an "8-thread, single-channel file" (8 concurrent time streams with 1 stream per frame), and in the example above, `fh.read` decoded the first 12 samples from all 8 threads, mapping thread number to the second axis of the decoded data array. Reading files with multiple threads and channels will produce 3-dimensional arrays.

`fh` includes `shape`, `size` and `ndim`, which give the shape, total number of elements and dimensionality of the file's entire dataset if it was decoded into an array. The number of *complete samples* - the set of samples from all available threads and channels for one point in time - in the file is given by the first element in `shape`:

```
>>> fh.shape    # Shape of all data from the file in decoded array form.
(40000, 8)
>>> fh.shape[0] # Number of complete samples.
40000
>>> fh.size
320000
```

(continues on next page)

(continued from previous page)

```
>>> fh.ndim
2
```

The shape of a single complete sample, including names indicating the meaning of shape dimensions, is retrievable using:

```
>>> fh.sample_shape
SampleShape(nthread=8)
```

By default, dimensions of length unity are *squeezed*, or removed from the sample shape. To retain them, we can pass `squeeze=False` to `open`:

```
>>> fhns = vdif.open(SAMPLE_VDIF, 'rs', squeeze=False)
>>> fhns.sample_shape    # Sample shape now keeps channel dimension.
SampleShape(nthread=8, nchan=1)
>>> fhns.ndim            # fh.shape and fh.ndim also change with squeezing.
3
>>> d2 = fhns.read(12)
>>> d2.shape             # Decoded data has channel dimension.
(12, 8, 1)
>>> fhns.close()
```

Basic information about the file is obtained by either by `fh.info` or simply `fh` itself:

```
>>> fh.info
Stream information:
start_time = 2014-06-16T05:56:07.000000000
stop_time = 2014-06-16T05:56:07.001250000
sample_rate = 32.0 MHz
shape = (40000, 8)
format = vdif
bps = 2
complex_data = False

File information:
edv = 3
frame_rate = 1600.0 Hz
samples_per_frame = 20000
sample_shape = (8, 1)

>>> fh
<VDIFStreamReader name=... offset=12
    sample_rate=32.0 MHz, samples_per_frame=20000,
    sample_shape=SampleShape(nthread=8),
    bps=2, complex_data=False, edv=3, station=65532,
    start_time=2014-06-16T05:56:07.000000000>
```

Not coincidentally, the first is identical to what we *found above* using `file_info`.

The filehandle itself also shows the `offset`, the current location of the sample file pointer. Above, it is at 12 since we have read in 12 (complete) samples. If we called `fh.read (12)` again we would get the next 12 samples. If we instead called `fh.read()`, it would read from the pointer's *current* position to the end of the file. If we wanted all the data in one array, we would move the file pointer back to the start of file, using `fh.seek`, before reading:

```
>>> fh.seek(0)      # Seek to sample 0.  Seek returns its offset in counts.
0
```

(continues on next page)

(continued from previous page)

```
>>> d_complete = fh.read()
>>> d_complete.shape
(40000, 8)
```

We can also move the pointer with respect to the end of file by passing `2` as a second argument:

```
>>> fh.seek(-100, 2)    # Second arg is 0 (start of file) by default.
39900
>>> d_end = fh.read(100)
>>> np.array_equal(d_complete[-100:], d_end)
True
```

`-100` means 100 samples before the end of file, so `d_end` is equal to the last 100 entries of `d_complete`. Baseband only keeps the most recently accessed data frame in memory, making it possible to analyze (normally large) files through selective decoding using `seek` and `read`.

Note: As with file pointers in general, `fh.seek` will not return an error if one seeks beyond the end of file. Attempting to read beyond the end of file, however, will result in an `EOFError`.

To determine where the pointer is located, we use `fh.tell()`:

```
>>> fh.tell()
40000
>>> fh.close()
```

Caution should be used when decoding large blocks of data using `fh.read`. For typical files, the resulting arrays are far too large to hold in memory.

2.2.3 Seeking and Telling in Time With the Sample Pointer

We can use `seek` and `tell` with units of time rather than samples. To do this with `tell`, we can pass an appropriate `astropy.units.Unit` object to its optional `unit` parameter:

```
>>> fh = vdif.open(SAMPLE_VDIF, 'rs')
>>> fh.seek(40000)
40000
>>> fh.tell(unit=u.ms)
<Quantity 1.25 ms>
```

Passing the string `'time'` reports the pointer's location in absolute time:

```
>>> fh.tell(unit='time')
<Time object: scale='utc' format='isot' value=2014-06-16T05:56:07.001250000>
```

We can also pass an absolute `astropy.time.Time`, or a positive or negative time difference `TimeDelta` or `astropy.units.Quantity` to `seek`. If the offset is a `Time` object, the second argument to seek is ignored.:

```
>>> from astropy.time.core import TimeDelta
>>> from astropy.time import Time
>>> fh.seek(TimeDelta(-5e-4, format='sec'), 2)  # Seek -0.5 ms from end.
24000
>>> fh.seek(0.25*u.ms, 1)  # Seek 0.25 ms from current position.
32000
```

(continues on next page)

(continued from previous page)

```
>>> # Seek to specific time.
>>> fh.seek(Time('2014-06-16T05:56:07.001125'))
36000
```

We can retrieve the time of the first sample in the file using `start_time`, the time immediately after the last sample using `stop_time`, and the time of the pointer's current location (equivalent to `fh.tell(unit='time')`) using `time`:

```
>>> fh.start_time
<Time object: scale='utc' format='isot' value=2014-06-16T05:56:07.000000000>
>>> fh.stop_time
<Time object: scale='utc' format='isot' value=2014-06-16T05:56:07.001250000>
>>> fh.time
<Time object: scale='utc' format='isot' value=2014-06-16T05:56:07.001125000>
>>> fh.close()
```

2.2.4 Extracting Header Information

The first header of the file is stored as the `header0` attribute of the stream reader object; it gives direct access to header properties via keyword lookup:

```
>>> with vdif.open(SAMPLE_VDIF, 'rs') as fh:
...     header0 = fh.header0
>>> header0['frame_length']
629
```

The full list of keywords is available by printing out `header0`:

```
>>> header0
<VDIFHeader3 invalid_data: False,
             legacy_mode: False,
             seconds: 14363767,
             _1_30_2: 0,
             ref_epoch: 28,
             frame_nr: 0,
             vdif_version: 1,
             lg2_nchan: 0,
             frame_length: 629,
             complex_data: False,
             bits_per_sample: 1,
             thread_id: 1,
             station_id: 65532,
             edv: 3,
             sampling_unit: True,
             sampling_rate: 16,
             sync_pattern: 0xacabfeed,
             loif_tuning: 859832320,
             _7_28_4: 15,
             dbe_unit: 2,
             if_nr: 0,
             subband: 1,
             sideband: True,
             major_rev: 1,
             minor_rev: 5,
             personality: 131>
```

A number of derived properties, such as the time (as a `Time` object), are also available through the header object.

```
>>> header0.time
<Time object: scale='utc' format='isot' value=2014-06-16T05:56:07.000000000>
```

These are listed in the API for each header class. For example, the sample VDIF file's headers are of class:

```
>>> type(header0)
<class 'baseband.vdif.header.VDIFHeader3'>
```

and so its attributes can be found `here`.

2.2.5 Reading Specific Components of the Data

By default, `fh.read()` returns complete samples, i.e. with all available threads, polarizations or channels. If we were only interested in decoding a *subset* of the complete sample, we can select specific components by passing indexing objects to the `subset` keyword in open. For example, if we only wanted thread 3 of the sample VDIF file:

```
>>> fh = vdif.open(SAMPLE_VDIF, 'rs', subset=3)
>>> fh.sample_shape
()
>>> d = fh.read(20000)
>>> d.shape
(20000,)
>>> fh.subset
(3,)
>>> fh.close()
```

Since by default data are squeezed, one obtains a data stream with just a single dimension. If one would like to keep all information, one has to pass `squeeze=False` and also make `subset` a list (or slice):

```
>>> fh = vdif.open(SAMPLE_VDIF, 'rs', subset=[3], squeeze=False)
>>> fh.sample_shape
SampleShape(nthread=1, nchan=1)
>>> d = fh.read(20000)
>>> d.shape
(20000, 1, 1)
>>> fh.close()
```

Data with multi-dimensional samples can be subset by passing a `tuple` of indexing objects with the same dimensional ordering as the (possibly squeezed) sample shape; in the case of the sample VDIF with `squeeze=False`, this is threads, then channels. For example, if we wished to select threads 1 and 3, and channel 0:

```
>>> fh = vdif.open(SAMPLE_VDIF, 'rs', subset=([1, 3], 0), squeeze=False)
>>> fh.sample_shape
SampleShape(nthread=2)
>>> fh.close()
```

Generally, `subset` accepts any object that can be used to index a `numpy.ndarray`, including advanced indexing (as done above, with `subset=([1, 3], 0)`). If possible, slices should be used instead of list of integers, since indexing with them returns a view rather than a copy and thus avoid unnecessary processing and memory allocation. (An exception to this is VDIF threads, where the subset is used to selectively read specific threads, and thus is not used for actual slicing of the data.)

2.3 Writing to Files and Format Conversion

2.3.1 Writing to a File

To write data to disk, we again use open. Writing data in a particular format requires both the header and data samples. For modifying an existing file, we have both the old header and old data handy.

As a simple example, let's read in the 8-thread, single-channel sample VDIF file and rewrite it as an single-thread, 8-channel one, which, for example, may be necessary for compatibility with DSPSR:

```
>>> import baseband.vdif as vdif
>>> from baseband.data import SAMPLE_VDIF
>>> fr = vdif.open(SAMPLE_VDIF, 'rs')
>>> fw = vdif.open('test_vdif.vdif', 'ws',
...                sample_rate=fr.sample_rate,
...                samples_per_frame=fr.samples_per_frame // 8,
...                nthread=1, nchan=fr.sample_shape.nthread,
...                complex_data=fr.complex_data, bps=fr.bps,
...                edv=fr.header0.edv, station=fr.header0.station,
...                time=fr.start_time)
```

The minimal parameters needed to generate a file are listed under the documentation for each format's `open`, though comprehensive lists can be found in the documentation for each format's stream writer class (eg. for VDIF, it's under `VDIFStreamWriter`). In practice we specify as many relevant header properties as available to obtain a particular file structure. If we possess the *exact* first header of the file, it can simply be passed to `open` via the `header` keyword. In the example above, though, we manually switch the values of `nthread` and `nchan`. Because VDIF EDV = 3 requires each frame's payload to contain 5000 bytes, and `nchan` is now a factor of 8 larger, we decrease `samples_per_frame`, the number of complete (i.e. all threads and channels included) samples per frame, by a factor of 8.

Encoding samples and writing data to file is done by passing data arrays into `fw`'s `write` method. The first dimension of the arrays is sample number, and the remaining dimensions must be as given by `fw.sample_shape`:

```
>>> fw.sample_shape
SampleShape(nchan=8)
```

In this case, the required dimensions are the same as the arrays from `fr.read`. We can thus write the data to file using:

```
>>> while fr.tell() < fr.shape[0]:
...     fw.write(fr.read(fr.samples_per_frame))
>>> fr.close()
>>> fw.close()
```

For our sample file, we could simply have written

```
fw.write(fr.read())
```

instead of the loop, but for large files, reading and writing should be done in smaller chunks to minimize memory usage. Baseband stores only the data frame or frame set being read or written to in memory.

We can check the validity of our new file by re-opening it:

```
>>> fr = vdif.open(SAMPLE_VDIF, 'rs')
>>> fh = vdif.open('test_vdif.vdif', 'rs')
>>> fh.sample_shape
SampleShape(nchan=8)
>>> np.all(fr.read() == fh.read())
True
```

(continues on next page)

(continued from previous page)

```
>>> fr.close()
>>> fh.close()
```

Note: One can also use the top-level `open` function for writing, with the file format passed in via its `format` argument.

2.3.2 File Format Conversion

It is often preferable to convert data from one file format to another that offers wider compatibility, or better fits the structure of the data. As an example, we convert the sample Mark 4 data to VDIF.

Since we don't have a VDIF header handy, we pass the relevant Mark 4 header values into `vdif.open` to create one.

```
>>> import baseband.mark4 as mark4
>>> from baseband.data import SAMPLE_MARK4
>>> fr = mark4.open(SAMPLE_MARK4, 'rs', ntrack=64, decade=2010)
>>> spf = 640       # fanout * 160 = 640 invalid samples per Mark 4 frame
>>> fw = vdif.open('m4convert.vdif', 'ws', sample_rate=fr.sample_rate,
...                samples_per_frame=spf, nthread=1,
...                nchan=fr.sample_shape.nchan,
...                complex_data=fr.complex_data, bps=fr.bps,
...                edv=1, time=fr.start_time)
```

We choose `edv = 1` since it's the simplest VDIF EDV whose header includes a sampling rate. The concept of threads does not exist in Mark 4, so the file effectively has `nthread = 1`. As discussed in the *Mark 4 documentation*, the data at the start of each frame is effectively overwritten by the header and are represented by invalid samples in the stream reader. We set `samples_per_frame` to `640` so that each section of invalid data is captured in a single frame.

We now write the data to file, manually flagging each invalid data frame:

```
>>> while fr.tell() < fr.shape[0]:
...     d = fr.read(fr.samples_per_frame)
...     fw.write(d[:640], valid=False)
...     fw.write(d[640:])
>>> fr.close()
>>> fw.close()
```

Lastly, we check our new file:

```
>>> fr = mark4.open(SAMPLE_MARK4, 'rs', ntrack=64, decade=2010)
>>> fh = vdif.open('m4convert.vdif', 'rs')
>>> np.all(fr.read() == fh.read())
True
>>> fr.close()
>>> fh.close()
```

For file format conversion in general, we have to consider how to properly scale our data to make the best use of the dynamic range of the new encoded format. For VLBI formats like VDIF, Mark 4 and Mark 5B, samples of the same size have the same scale, which is why we did not have to rescale our data when writing 2-bits-per-sample Mark 4 data to a 2-bits-per-sample VDIF file. Rescaling is necessary, though, to convert DADA or GSB to VDIF. For examples of rescaling, see the `baseband/tests/test_conversion.py` file.

2.4 Reading or Writing to a Sequence of Files

Data from one continuous observation is often spread over a sequence of files. The `sequentialfile` module is available for reading in a sequence as if it were one contiguous file. Simple usage examples can be found in the *Sequential File* section. DADA data is so often stored in a file sequence that reading a time-ordered list of filenames is built into `open`; for details, see the `its API entry`.

CHAPTER 3

Glossary

channel
A single component of the *complete sample*, or a *stream* thereof. They typically represent one frequency sub-band, the output from a single antenna, or (for channelized data) one spectral or Fourier channel, ie. one part of a Fourier spectrum.

complete sample
Set of all component samples - ie. from all threads, polarizations, channels, etc. - for one point in time. Its dimensions are given by the *sample shape*.

component
One individual *thread* and *channel*, or one polarization and channel, etc. Component samples each occupy one element in decoded data arrays. A component sample is composed of one *elementary sample* if it is real, and two if it is complex.

data frame
A block of time-sampled data, or *payload*, accompanied by a *header*. “Frame” for short.

data frameset
In the VDIF format, the set of all *data frames* representing the same segment of time. Each data frame consists of sets of *channels* from different *threads*.

elementary sample
The smallest subdivision of a complete sample, i.e. the real / imaginary part of one *component* of a *complete sample*.

header
Metadata accompanying a *data frame*.

payload
The data within a *data frame*.

sample
Data from one point in time. *Complete samples* contain samples from all *components*, while *elementary samples* are one part of one component.

sample rate
Rate of complete samples.

sample shape
The lengths of the dimensions of the complete sample.

squeezing
The removal of any dimensions of length unity from decoded data.

stream
Timeseries of *samples*; may refer to all of, or a subsection of, the dataset.

subset
A subset of a complete sample, in particular one defined by the user for selective decoding.

thread
A collection of *channels* from the *complete sample*, or a *stream* thereof. For VDIF, each thread is carried by a separate (set of) *data frame(s)*.

Part II

Specific file formats

Baseband's code is subdivided into its supported file formats, and the following sections contain format specifications, usage notes, troubleshooting help and APIs for each.

CHAPTER 4

VDIF

The VLBI Data Interchange Format (VDIF) was introduced in 2009 to standardize VLBI data transfer and storage. Detailed specifications are found in VDIF's specification document.

4.1 File Structure

A VDIF file is composed of *data frames*. Each has a *header* of eight 32-bit words (32 bytes; the exception is the "legacy VDIF" format, which is four words, or 16 bytes, long), and a *payload* that ranges from 32 bytes to ~134 megabytes. Both are little-endian. The first four words of a VDIF header hold the same information in all VDIF files, but the last four words hold optional user-defined data. The layout of these four words is specified by the file's **extended-data version**, or EDV. More detailed information on the header can be found in the *tutorial for supporting a new VDIF EDV*.

A data frame may carry one or multiple *channels*, and a *stream* of data frames all carrying the same (set of) channels is known as a *thread* and denoted by its thread ID. The collection of frames representing the same time segment (and all possible thread IDs) is called a *data frameset* (or just "frameset").

Strict time and thread ID ordering of frames in the stream, while considered part of VDIF best practices, is not mandated, and cannot be guaranteed during data transmission over the internet.

4.2 Usage Notes

This section covers reading and writing VDIF files with Baseband; general usage can be found under the *Getting Started* section. For situations in which one is unsure of a file's format, Baseband features the general `baseband.open` and `baseband.file_info` functions, which are also discussed in *Getting Started*. The examples below use the small sample file `baseband/data/sample.vdif`, and the `numpy`, `astropy.units`, and `baseband.vdif` modules:

```
>>> import numpy as np
>>> from baseband import vdif
>>> import astropy.units as u
>>> from baseband.data import SAMPLE_VDIF
```

Simple reading and writing of VDIF files can be done entirely using open. Opening in binary mode provides a normal file reader, but extended with methods to read a VDIFFrameSet data container for storing a frame set as well as VDIFFrame one for storing a single frame:

```
>>> fh = vdif.open(SAMPLE_VDIF, 'rb')
>>> fs = fh.read_frameset()
>>> fs.data.shape
(20000, 8, 1)
>>> fr = fh.read_frame()
>>> fr.data.shape
(20000, 1)
>>> fh.close()
```

(As with other formats, `fr.data` is a read-only property of the frame.)

Opening in stream mode wraps the low-level routines such that reading and writing is in units of samples. It also provides access to header information:

```
>>> fh = vdif.open(SAMPLE_VDIF, 'rs')
>>> fh
<VDIFStreamReader name=... offset=0
    sample_rate=32.0 MHz, samples_per_frame=20000,
    sample_shape=SampleShape(nthread=8),
    bps=2, complex_data=False, edv=3, station=65532,
    start_time=2014-06-16T05:56:07.000000000>
>>> d = fh.read(12)
>>> d.shape
(12, 8)
>>> d[:, 0].astype(int)  # first thread
array([-1, -1,  3, -1,  1, -1,  3, -1,  1,  3, -1,  1])
>>> fh.close()
```

To set up a file for writing needs quite a bit of header information. Not coincidentally, what is given by the reader above suffices:

```
>>> from astropy.time import Time
>>> fw = vdif.open('try.vdif', 'ws', sample_rate=32*u.MHz,
...                samples_per_frame=20000, nchan=1, nthread=2,
...                complex_data=False, bps=2, edv=3, station=65532,
...                time=Time('2014-06-16T05:56:07.000000000'))
>>> with vdif.open(SAMPLE_VDIF, 'rs', subset=[1, 3]) as fh:
...     d = fh.read(20000)  # Get some data to write
>>> fw.write(d)
>>> fw.close()
>>> fh = vdif.open('try.vdif', 'rs')
>>> d2 = fh.read(12)
>>> np.all(d[:12] == d2)
True
>>> fh.close()
```

Here is a simple example to copy a VDIF file. We use the `sort=False` option to ensure the frames are written exactly in the same order, so the files should be identical:

```
>>> with vdif.open(SAMPLE_VDIF, 'rb') as fr, vdif.open('try.vdif', 'wb') as fw:
...     while True:
...         try:
...             fw.write_frameset(fr.read_frameset(sort=False))
```

(continues on next page)

(continued from previous page)

```
...         except:
...             break
```

For small files, one could just do:

```
>>> with vdif.open(SAMPLE_VDIF, 'rs') as fr, \
...         vdif.open('try.vdif', 'ws', header0=fr.header0,
...                   sample_rate=fr.sample_rate,
...                   nthread=fr.sample_shape.nthread) as fw:
...     fw.write(fr.read())
```

This copies everything to memory, though, and some header information is lost.

4.3 Troubleshooting

In situations where the VDIF files being handled are corrupted or modified in an unusual way, using `open` will likely lead to an exception being raised or to unexpected behavior. In such cases, it may still be possible to read in the data. Below, we provide a few solutions and workarounds to do so.

Note: This list is certainly incomplete. If you have an issue (solved or otherwise) you believe should be on this list, please e-mail the *contributors*.

4.3.1 AssertionError when checking EDV in header `verify` function

All VDIF header classes (other than `VDIFLegacyHeader`) check, using their `verify` function, that the EDV read from file matches the class EDV. If they do not, the following line

```
assert self.edv is None or self.edv == self['edv']
```

returns an AssertionError. If this occurs because the VDIF EDV is not yet supported by Baseband, support can be added by implementing a custom header class. If the EDV is supported, but the header deviates from the format found in the VLBI.org EDV registry, the best solution is to create a custom header class, then override the subclass selector in `VDIFHeader`. Tutorials for doing either can be found *here*.

4.3.2 EOFError encountered in `_get_frame_rate` when reading

When the sample rate is not input by the user and cannot be deduced from header information (if EDV = 1 or, the sample rate is found in the header), Baseband tries to determine the frame rate using the private method `_get_frame_rate` in `VDIFStreamReader` (and then multiply by the samples per frame to obtain the sample rate). This function raises `EOFError` if the file contains less than one second of data, or is corrupt. In either case the file can be opened still by explicitly passing in the sample rate to `open` via the `sample_rate` keyword.

4.4 Reference/API

4.4.1 baseband.vdif Package

VLBI Data Interchange Format (VDIF) reader/writer

For the VDIF specification, see http://www.vlbi.org/vdif

Functions

open(name[, mode])	Open VDIF file for reading or writing.

open

baseband.vdif.**open**(*name*, *mode=u'rs'*, ***kwargs*)

Open VDIF file for reading or writing.

Opened as a binary file, one gets a wrapped filehandle that adds methods to read/write a frame. Opened as a stream, the handle is wrapped further, with methods such as reading and writing to the file as if it were a stream of samples.

Parameters

name : str or filehandle

File name or handle.

mode : {'rb', 'wb', 'rs', or 'ws'}, optional

Whether to open for reading or writing, and as a regular binary file or as a stream. Default: 'rs', for reading a stream.

****kwargs**

Additional arguments when opening the file as a stream.

— For reading a stream : (see VDIFStreamReader)

sample_rate : Quantity, optional

Number of complete samples per second, i.e. the rate at which each channel in each thread is sampled. If None (default), will be inferred from the header or by scanning one second of the file.

squeeze : bool, optional

If True (default), remove any dimensions of length unity from decoded data.

subset : indexing object or tuple of objects, optional

Specific components of the complete sample to decode (after possible squeezing). If a single indexing object is passed, it selects threads. If a tuple is passed, the first selects threads and the second selects channels. If the tuple is empty (default), all components are read.

fill_value : float or complex, optional

Value to use for invalid or missing data. Default: 0.

verify : bool, optional

Whether to do basic checks of frame integrity when reading. The first frameset of the stream is always checked. Default: True.

— For writing a stream : (see VDIFStreamWriter)

header0 : VDIFHeader

Header for the first frame, holding time information, etc. Can instead give keyword arguments to construct a header (see `**kwargs`).

sample_rate : `Quantity`

Number of complete samples per second, i.e. the rate at which each channel in each thread is sampled. For EDV 1 and 3, can alternatively set `sample_rate` within the header.

nthread : int, optional

Number of threads (e.g., 2 for 2 polarisations). Default: 1.

squeeze : bool, optional

If `True` (default), writer accepts squeezed arrays as input, and adds any dimensions of length unity.

****kwargs**

If the header is not given, an attempt will be made to construct one with any further keyword arguments. See `VDIFStreamWriter`.

Returns

Filehandle

`VDIFFileReader` or `VDIFFileWriter` (binary), or `VDIFStreamReader` or `VDIFStreamWriter` (stream).

Classes

`VDIFFrame`(header, payload[, valid, verify])	Representation of a VDIF data frame, consisting of a header and payload.
`VDIFFrameSet`(frames[, header0])	Representation of a set of VDIF frames, combining different threads.
`VDIFHeader`(words[, edv, verify])	VDIF Header, supporting different Extended Data Versions.
`VDIFPayload`(words[, header, nchan, bps, . . .])	Container for decoding and encoding VDIF payloads.

VDIFFrame

class baseband.vdif.**VDIFFrame**(*header*, *payload*, *valid=None*, *verify=True*)

Bases: `baseband.vlbi_base.frame.VLBIFrameBase`

Representation of a VDIF data frame, consisting of a header and payload.

Parameters

header : `VDIFHeader`

Wrapper around the encoded header words, providing access to the header information.

payload : `VDIFPayload`

Wrapper around the payload, provding mechanisms to decode it.

valid : bool or None

Whether the data are valid. If `None` (default), is inferred from header. Note that `header` is changed in-place if `True` or `False`.

verify : bool

Whether or not to do basic assertions that check the integrity (e.g., that channel information and whether or not data are complex are consistent between header and data). Default: True.

Notes

The Frame can also be instantiated using class methods:

fromfile : read header and payload from a filehandle

fromdata : encode data as payload

Of course, one can also do the opposite:

tofile : method to write header and payload to filehandle

data : property that yields full decoded payload

One can decode part of the payload by indexing or slicing the frame. If the frame does not contain valid data, all values returned are set to `self.fill_value`.

A number of properties are defined: `shape`, `dtype` and `size` are the shape, type and number of complete samples of the data array, and `nbytes` the frame size in bytes. Furthermore, the frame acts as a dictionary, with keys those of the header. Any attribute that is not defined on the frame itself, such as `.time` will be looked up on the header as well.

Attributes Summary

`data`	Full decoded frame.
`dtype`	Numeric type of the frame data.
`fill_value`	Value to replace invalid data in the frame.
`nbytes`	Size of the encoded frame in bytes.
`ndim`	Number of dimensions of the frame data.
`sample_shape`	Shape of a sample in the frame (nchan,).
`shape`	Shape of the frame data.
`size`	Total number of component samples in the frame data.
`valid`	Whether frame contains valid data.

Methods Summary

`from_mark5b_frame`(mark5b_frame[, verify])	Construct an Mark5B over VDIF frame (EDV=0xab).
`fromdata`(data[, header, verify])	Construct frame from data and header.
`fromfile`(fh[, edv, verify])	Read a frame from a filehandle.
`keys`()	
`tofile`(fh)	Write encoded frame to filehandle.
`verify`()	Verify integrity.

Attributes Documentation

data
Full decoded frame.

dtype
Numeric type of the frame data.

fill_value
Value to replace invalid data in the frame.

nbytes
Size of the encoded frame in bytes.

ndim
Number of dimensions of the frame data.

sample_shape
Shape of a sample in the frame (nchan,).

shape
Shape of the frame data.

size
Total number of component samples in the frame data.

valid
Whether frame contains valid data.

This is just the opposite of the `invalid_data` item in the header. If set, that header item is adjusted correspondingly.

Methods Documentation

classmethod from_mark5b_frame(*mark5b_frame*, *verify=True*, ***kwargs*)
Construct an Mark5B over VDIF frame (EDV=0xab).

Any additional keywords can be used to set VDIF header properties not found in the Mark 5B header (such as station).

See http://www.vlbi.org/vdif/docs/vdif_extension_0xab.pdf

classmethod fromdata(*data*, *header=None*, *verify=True*, ***kwargs*)
Construct frame from data and header.

Parameters

data : `ndarray`

Array holding complex or real data to be encoded.

header : `VDIFHeader` or None

If not given, will attempt to generate one using the keywords.

verify : bool, optional

Whether or not to do basic assertions that check the integrity (e.g., that channel information and whether or not data are complex are consistent between header and data). Default: `True`.

****kwargs**

If `header` is not given, these are used to initialize one.

classmethod fromfile(*fh*, *edv=None*, *verify=True*)
Read a frame from a filehandle.

Parameters
fh : filehandle

From which the header and payload are read.

edv : int, False, or None, optional

Extended Data Version. `False` is for legacy headers. If `None` (default), it will be determined from the words themselves.

verify : bool, optional

Whether or not to do basic assertions that check the integrity (e.g., that channel information and whether or not data are complex are consistent between header and data). Default: `True`.

keys()

tofile(*fh*)
Write encoded frame to filehandle.

verify()
Verify integrity.

Checks consistency between the header information and payload data shape and type.

VDIFFrameSet

class baseband.vdif.**VDIFFrameSet**(*frames*, *header0=None*)
Bases: `object`

Representation of a set of VDIF frames, combining different threads.

Parameters
frames : list of `VDIFFrame`

Should all cover the same time span.

header0 : `VDIFHeader`

First header of the frame set. If `None` (default), is extracted from `frames[0]`.

Notes

The FrameSet can also be read instantiated using class methods:

fromfile : read frames from a filehandle, optionally selecting threads

fromdata : encode data as a set of frames

Of course, one can also do the opposite:

tofile : write frames to filehandle

data : property that yields full decoded frame payloads

One can decode part of the payload by indexing or slicing the frame. If the frame does not contain valid data, all values returned are set to `self.fill_value`.

A number of properties are defined: `shape`, `dtype` and `size` are the shape, type and number of complete samples of the data array, and `nbytes` the frame size in bytes. Like a VDIFFrame, the frame set acts as a dictionary, with keys those of the header of the first frame (available via `.header0`). Any attribute that is not defined on the frame set itself, such as `.time` will also be looked up on the header.

Attributes Summary

`data`	Full decoded frame.
`dtype`	Numeric type of the frameset data.
`fill_value`	Value to replace invalid data in the frameset.
`nbytes`	Size of the encoded frame in bytes.
`ndim`	Number of dimensions of the frameset data.
`sample_shape`	Shape of a sample in the frameset (nthread, nchan).
`shape`	Shape of the frameset data.
`size`	Total number of component samples in the frameset data.
`valid`	Whether frameset contains valid data.

Methods Summary

`fromdata`(data[, headers, verify])	Construct a set of frames from data and headers.
`fromfile`(fh[, thread_ids, edv, verify])	Read a frame set from a file, starting at the current location.
`keys`()	
`tofile`(fh)	Write all encoded frames to filehandle.

Attributes Documentation

data
Full decoded frame.

dtype
Numeric type of the frameset data.

fill_value
Value to replace invalid data in the frameset.

nbytes
Size of the encoded frame in bytes.

ndim
Number of dimensions of the frameset data.

sample_shape
Shape of a sample in the frameset (nthread, nchan).

shape
Shape of the frameset data.

size
Total number of component samples in the frameset data.

valid
Whether frameset contains valid data.

Methods Documentation

classmethod fromdata(*data*, *headers=None*, *verify=True*, ***kwargs*)
Construct a set of frames from data and headers.

Parameters

data : ndarray

Array holding complex or real data to be encoded. Dimensions should be (samples_per_frame, nthread, nchan).

headers : VDIFHeader, list of same, or None

If a single header, a list with increasing `thread_id` is generated. If not given, will attempt to generate a header from the keyword arguments.

verify : bool

Whether or not to do basic assertions that check the integrety (e.g., that channel information and whether or not data are complex are consistent between header and data). Default: `True`.

****kwargs**

If `header` is not given, these are used to initialize one.

Returns

frameset : VDIFFrameSet

classmethod fromfile(*fh*, *thread_ids=None*, *edv=None*, *verify=True*)
Read a frame set from a file, starting at the current location.

Parameters

fh : filehandle

Handle to the VDIF file. Should be at the location where the frames are read from.

thread_ids : list or None, optional

The thread ids that should be read. If `None` (default), continue reading threads as long as the frame number does not increase.

edv : int or None, optional

The expected extended data version for the VDIF Header. If `None` (default), use that of the first frame. (Passing it in slightly improves file integrity checking.)

verify : bool, optional

Whether to do (light) sanity checks on the header. Default: `True`.

Returns

frameset : VDIFFrameSet

Its `frames` property holds a list of frames (in order of either their `thread_id` or following the input `thread_ids` list). Use the `data` attribute to convert to an array.

keys()

tofile(*fh*)

Write all encoded frames to filehandle.

VDIFHeader

class baseband.vdif.**VDIFHeader**(*words*, *edv=None*, *verify=True*, ***kwargs*)

Bases: baseband.vlbi_base.header.VLBIHeaderBase

VDIF Header, supporting different Extended Data Versions.

Will initialize a header instance appropriate for a given EDV. See http://www.vlbi.org/vdif/docs/VDIF_specification_Release_1.1.1.pdf

Parameters

words : tuple of int, or None

Eight (or four for legacy VDIF) 32-bit unsigned int header words. If `None`, set to a tuple of zeros for later initialisation.

edv : int, False, or None, optional

Extended data version. If `False`, a legacy header is used. If `None` (default), it is determined from the header. (Given it explicitly is mostly useful for a slight speed-up.)

verify : bool

Whether to do basic verification of integrity. Default: `True`.

Returns

header : `VDIFHeader` subclass

As appropriate for the extended data version.

Attributes Summary

`bps`	Bits per elementary sample.
`edv`	VDIF Extended Data Version (EDV).
`frame_nbytes`	Size of the frame in bytes.
`mutable`	Whether the header can be modified.
`nbytes`	Size of the header in bytes.
`nchan`	Number of channels in the frame.
`payload_nbytes`	Size of the payload in bytes.
`samples_per_frame`	Number of complete samples in the frame.
`station`	Station ID: two ASCII characters, or 16-bit int.
`time`	Converts ref_epoch, seconds, and frame_nr to Time object.

Methods Summary

`copy`()	Create a mutable and independent copy of the header.
`from_mark5b_header`(mark5b_header, bps, …)	Construct an Mark5B over VDIF header (EDV=0xab).
`fromfile`(fh[, edv, verify])	Read VDIF Header from file.

Continued on next page

Table 8 – continued from previous page

fromkeys(**kwargs)	Initialise a header from parsed values.
fromvalues([edv])	Initialise a header from parsed values.
get_time([frame_rate])	Converts ref_epoch, seconds, and frame_nr to Time object.
keys()	
same_stream(other)	Whether header is consistent with being from the same stream.
set_time(time[, frame_rate])	Converts Time object to ref_epoch, seconds, and frame_nr.
tofile(fh)	Write VLBI frame header to filehandle.
update(**kwargs)	Update the header by setting keywords or properties.
verify()	Verify that the length of the words is consistent.

Attributes Documentation

bps
Bits per elementary sample.

edv
VDIF Extended Data Version (EDV).

frame_nbytes
Size of the frame in bytes.

mutable
Whether the header can be modified.

nbytes
Size of the header in bytes.

nchan
Number of channels in the frame.

payload_nbytes
Size of the payload in bytes.

samples_per_frame
Number of complete samples in the frame.

station
Station ID: two ASCII characters, or 16-bit int.

time
Converts ref_epoch, seconds, and frame_nr to Time object.

Uses 'ref_epoch', which stores the number of half-years from 2000, and 'seconds'. By default, it also calculates the offset using the current frame number. For non-zero 'frame_nr', this requires the frame rate, which is calculated from the sample rate in the header.

Parameters
frame_rate : Quantity, optional

For non-zero 'frame_nr', this is required to calculate the corresponding offset.

Returns
time : Time

Methods Documentation

copy()

Create a mutable and independent copy of the header.

Keyword arguments can be passed on as needed by possible subclasses.

classmethod from_mark5b_header(*mark5b_header*, *bps*, *nchan*, ***kwargs*)

Construct an Mark5B over VDIF header (EDV=0xab).

See http://www.vlbi.org/vdif/docs/vdif_extension_0xab.pdf

Note that the Mark 5B header does not encode the bits-per-sample and the number of channels used in the payload, so these need to be given separately. A complete frame can be encapsulated with `from_mark5b_frame`.

Parameters

mark5b_header : `Mark5BHeader`

Used to set time, etc.

bps : int

Bits per elementary sample.

nchan : int

Number of channels carried in the Mark 5B payload.

****kwargs**

Any further arguments. Strictly, none are necessary to create a valid VDIF header, but this can be used to pass on, e.g., `invalid_data`.

classmethod fromfile(*fh*, *edv=None*, *verify=True*)

Read VDIF Header from file.

Parameters

fh : filehandle

To read data from.

edv : int, False, or None, optional

Extended data version. If `False`, a legacy header is used. If `None` (default), it is determined from the header. (Given it explicitly is mostly useful for a slight speed-up.)

verify : bool, optional

Whether to do basic verification of integrity. Default: `True`.

classmethod fromkeys(***kwargs*)

Initialise a header from parsed values.

Like `fromvalues()`, but without any interpretation of keywords.

Raises

KeyError : if not all keys required are pass in.

classmethod fromvalues(*edv=False*, ***kwargs*)

Initialise a header from parsed values.

Here, the parsed values must be given as keyword arguments, i.e., for any `header = cls(<data>)`, `cls.fromvalues(**header) == header`.

However, unlike for the `fromkeys()` class method, data can also be set using arguments named after methods, such as `bps` and `time`.

Given defaults:

invalid_data : `False` legacy_mode : `False` vdif_version : 1 thread_id : 0 frame_nr : 0 sync_pattern : 0xACABFEED for EDV 1 and 3, 0xa5ea5 for EDV 2

Values set by other keyword arguments (if present):

bits_per_sample : from `bps` frame_length : from `samples_per_frame` or `frame_nbytes` lg2_nchan : from `nchan` station_id : from `station` sampling_rate, sampling_unit : from `sample_rate` ref_epoch, seconds, frame_nr : from `time`

Note that to set `time` to non-integer seconds one also needs to pass in `frame_rate` or `sample_rate`.

get_time(*frame_rate=None*)

Converts ref_epoch, seconds, and frame_nr to Time object.

Uses 'ref_epoch', which stores the number of half-years from 2000, and 'seconds'. By default, it also calculates the offset using the current frame number. For non-zero 'frame_nr', this requires the frame rate, which is calculated from the sample rate in the header.

Parameters

frame_rate : `Quantity`, optional

For non-zero 'frame_nr', this is required to calculate the corresponding offset.

Returns

time : `Time`

keys()

same_stream(*other*)

Whether header is consistent with being from the same stream.

set_time(*time*, *frame_rate=None*)

Converts Time object to ref_epoch, seconds, and frame_nr.

For non-integer seconds, a frame rate is needed to calculate the 'frame_nr'.

Parameters

time : `Time`

The time to use for this header.

frame_rate : `Quantity`, optional

For calculating 'frame_nr' from the fractional seconds.

tofile(*fh*)

Write VLBI frame header to filehandle.

update(***kwargs*)

Update the header by setting keywords or properties.

Here, any keywords matching header keys are applied first, and any remaining ones are used to set header properties, in the order set by the class (in `_properties`).

Parameters

verify : bool, optional

If `True` (default), verify integrity after updating.

****kwargs**

Arguments used to set keywords and properties.

verify()

Verify that the length of the words is consistent.

Subclasses should override this to do more thorough checks.

VDIFPayload

class baseband.vdif.**VDIFPayload**(*words*, *header=None*, *nchan=1*, *bps=2*, *complex_data=False*)

Bases: `baseband.vlbi_base.payload.VLBIPayloadBase`

Container for decoding and encoding VDIF payloads.

Parameters

words : `ndarray`

Array containing LSB unsigned words (with the right size) that encode the payload.

header : `VDIFHeader`

If given, used to infer the number of channels, bps, and whether the data are complex.

nchan : int, optional

Number of channels, used if `header` is not given. Default: 1.

bps : int, optional

Bits per elementary sample, used if `header` is not given. Default: 2.

complex_data : bool, optional

Whether the data are complex, used if `header` is not given. Default: `False`.

Attributes Summary

`data`	Full decoded payload.
`dtype`	Numeric type of the decoded data array.
`nbytes`	Size of the payload in bytes.
`ndim`	Number of dimensions of the decoded data array.
`shape`	Shape of the decoded data array.
`size`	Total number of component samples in the decoded data array.

Methods Summary

`fromdata`(data[, header, bps, edv])	Encode data as payload, using header information.
`fromfile`(fh, header)	Read payload from filehandle and decode it into data.
`tofile`(fh)	Write payload to filehandle.

Attributes Documentation

data

Full decoded payload.

dtype
Numeric type of the decoded data array.

nbytes
Size of the payload in bytes.

ndim
Number of dimensions of the decoded data array.

shape
Shape of the decoded data array.

size
Total number of component samples in the decoded data array.

Methods Documentation

classmethod fromdata(*data*, *header=None*, *bps=2*, *edv=None*)
Encode data as payload, using header information.

Parameters

data : `ndarray`

Values to be encoded.

header : `VDIFHeader`, optional

If given, used to infer the encoding, and to verify the number of channels and whether the data are complex.

bps : int, optional

Bits per elementary sample, used if `header` is not given. Default: 2.

edv : int, optional

Should be given if `header` is not given and the payload is encoded as Mark 5 data (i.e., `edv=0xab`).

classmethod fromfile(*fh*, *header*)
Read payload from filehandle and decode it into data.

Parameters

fh : filehandle

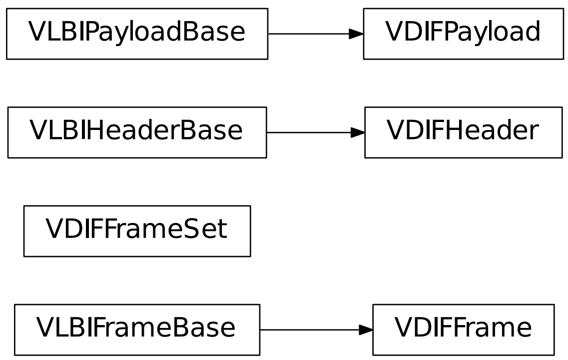
To read data from.

header : `VDIFHeader`

Used to infer the payload size, number of channels, bits per sample, and whether the data are complex.

tofile(*fh*)
Write payload to filehandle.

Class Inheritance Diagram



4.4.2 baseband.vdif.header Module

Definitions for VLBI VDIF Headers.

Implements a VDIFHeader class used to store header words, and decode/encode the information therein.

For the VDIF specification, see http://www.vlbi.org/vdif

Classes

`VDIFHeader`(words[, edv, verify])	VDIF Header, supporting different Extended Data Versions.
`VDIFBaseHeader`(words[, edv, verify])	Base for non-legacy VDIF headers that use 8 32-bit words.
`VDIFSampleRateHeader`(words[, edv, verify])	Base for VDIF headers that include the sample rate (EDV= 1, 3, 4).
`VDIFLegacyHeader`(words[, edv, verify])	Legacy VDIF header that uses only 4 32-bit words.
`VDIFHeader0`(words[, edv, verify])	VDIF Header for EDV=0.
`VDIFHeader1`(words[, edv, verify])	VDIF Header for EDV=1.
`VDIFHeader2`(words[, edv, verify])	VDIF Header for EDV=2.
`VDIFHeader3`(words[, edv, verify])	VDIF Header for EDV=3.
`VDIFMark5BHeader`(words[, edv, verify])	Mark 5B over VDIF (EDV=0xab).

VDIFHeader

class `baseband.vdif.header.`**`VDIFHeader`**(*words*, *edv=None*, *verify=True*, ***kwargs*)

Bases: `baseband.vlbi_base.header.VLBIHeaderBase`

VDIF Header, supporting different Extended Data Versions.

Will initialize a header instance appropriate for a given EDV. See http://www.vlbi.org/vdif/docs/VDIF_specification_Release_1.1.1.pdf

Parameters

words : tuple of int, or None

Eight (or four for legacy VDIF) 32-bit unsigned int header words. If `None`, set to a tuple of zeros for later initialisation.

edv : int, False, or None, optional

Extended data version. If `False`, a legacy header is used. If `None` (default), it is determined from the header. (Given it explicitly is mostly useful for a slight speed-up.)

verify : bool

Whether to do basic verification of integrity. Default: `True`.

Returns

header : `VDIFHeader` subclass

As appropriate for the extended data version.

Attributes Summary

`bps`	Bits per elementary sample.
`edv`	VDIF Extended Data Version (EDV).
`frame_nbytes`	Size of the frame in bytes.
`mutable`	Whether the header can be modified.
`nbytes`	Size of the header in bytes.
`nchan`	Number of channels in the frame.
`payload_nbytes`	Size of the payload in bytes.
`samples_per_frame`	Number of complete samples in the frame.
`station`	Station ID: two ASCII characters, or 16-bit int.
`time`	Converts ref_epoch, seconds, and frame_nr to Time object.

Methods Summary

`copy`()	Create a mutable and independent copy of the header.
`from_mark5b_header`(mark5b_header, bps, . . .)	Construct an Mark5B over VDIF header (EDV=0xab).
`fromfile`(fh[, edv, verify])	Read VDIF Header from file.
`fromkeys`(**kwargs)	Initialise a header from parsed values.
`fromvalues`([edv])	Initialise a header from parsed values.
`get_time`([frame_rate])	Converts ref_epoch, seconds, and frame_nr to Time object.
`keys`()	
`same_stream`(other)	Whether header is consistent with being from the same stream.
`set_time`(time[, frame_rate])	Converts Time object to ref_epoch, seconds, and frame_nr.
`tofile`(fh)	Write VLBI frame header to filehandle.
`update`(**kwargs)	Update the header by setting keywords or properties.
`verify`()	Verify that the length of the words is consistent.

Attributes Documentation

bps
Bits per elementary sample.

edv
VDIF Extended Data Version (EDV).

frame_nbytes
Size of the frame in bytes.

mutable
Whether the header can be modified.

nbytes
Size of the header in bytes.

nchan
Number of channels in the frame.

payload_nbytes
Size of the payload in bytes.

samples_per_frame
Number of complete samples in the frame.

station
Station ID: two ASCII characters, or 16-bit int.

time
Converts ref_epoch, seconds, and frame_nr to Time object.

Uses 'ref_epoch', which stores the number of half-years from 2000, and 'seconds'. By default, it also calculates the offset using the current frame number. For non-zero 'frame_nr', this requires the frame rate, which is calculated from the sample rate in the header.

Parameters
frame_rate : `Quantity`, optional

For non-zero 'frame_nr', this is required to calculate the corresponding offset.

Returns
time : `Time`

Methods Documentation

copy()
Create a mutable and independent copy of the header.

Keyword arguments can be passed on as needed by possible subclasses.

classmethod from_mark5b_header(*mark5b_header*, *bps*, *nchan*, ***kwargs*)
Construct an Mark5B over VDIF header (EDV=0xab).

See http://www.vlbi.org/vdif/docs/vdif_extension_0xab.pdf

Note that the Mark 5B header does not encode the bits-per-sample and the number of channels used in the payload, so these need to be given separately. A complete frame can be encapsulated with `from_mark5b_frame`.

Parameters
mark5b_header : `Mark5BHeader`

Used to set time, etc.

bps : int

Bits per elementary sample.

nchan : int

Number of channels carried in the Mark 5B payload.

****kwargs**

Any further arguments. Strictly, none are necessary to create a valid VDIF header, but this can be used to pass on, e.g., `invalid_data`.

classmethod fromfile(*fh*, *edv=None*, *verify=True*)

Read VDIF Header from file.

Parameters

fh : filehandle

To read data from.

edv : int, False, or None, optional

Extended data version. If `False`, a legacy header is used. If `None` (default), it is determined from the header. (Given it explicitly is mostly useful for a slight speed-up.)

verify : bool, optional

Whether to do basic verification of integrity. Default: `True`.

classmethod fromkeys(***kwargs*)

Initialise a header from parsed values.

Like `fromvalues()`, but without any interpretation of keywords.

Raises

KeyError : if not all keys required are pass in.

classmethod fromvalues(*edv=False*, ***kwargs*)

Initialise a header from parsed values.

Here, the parsed values must be given as keyword arguments, i.e., for any `header = cls(<data>)`, `cls.fromvalues(**header) == header`.

However, unlike for the `fromkeys()` class method, data can also be set using arguments named after methods, such as `bps` and `time`.

Given defaults:

invalid_data : `False` legacy_mode : `False` vdif_version : 1 thread_id : 0 frame_nr : 0 sync_pattern : 0xACABFEED for EDV 1 and 3, 0xa5ea5 for EDV 2

Values set by other keyword arguments (if present):

bits_per_sample : from `bps` frame_length : from `samples_per_frame` or `frame_nbytes` lg2_nchan : from `nchan` station_id : from `station` sampling_rate, sampling_unit : from `sample_rate` ref_epoch, seconds, frame_nr : from `time`

Note that to set `time` to non-integer seconds one also needs to pass in `frame_rate` or `sample_rate`.

get_time(*frame_rate=None*)

Converts ref_epoch, seconds, and frame_nr to Time object.

Uses 'ref_epoch', which stores the number of half-years from 2000, and 'seconds'. By default, it also calculates the offset using the current frame number. For non-zero 'frame_nr', this requires the frame rate, which is calculated from the sample rate in the header.

Parameters

frame_rate : `Quantity`, optional

For non-zero 'frame_nr', this is required to calculate the corresponding offset.

Returns

time : `Time`

keys()

same_stream(*other*)

Whether header is consistent with being from the same stream.

set_time(*time*, *frame_rate=None*)

Converts Time object to ref_epoch, seconds, and frame_nr.

For non-integer seconds, a frame rate is needed to calculate the 'frame_nr'.

Parameters

time : `Time`

The time to use for this header.

frame_rate : `Quantity`, optional

For calculating 'frame_nr' from the fractional seconds.

tofile(*fh*)

Write VLBI frame header to filehandle.

update(***kwargs*)

Update the header by setting keywords or properties.

Here, any keywords matching header keys are applied first, and any remaining ones are used to set header properties, in the order set by the class (in `_properties`).

Parameters

verify : bool, optional

If `True` (default), verify integrity after updating.

****kwargs**

Arguments used to set keywords and properties.

verify()

Verify that the length of the words is consistent.

Subclasses should override this to do more thorough checks.

VDIFBaseHeader

class baseband.vdif.header.**VDIFBaseHeader**(*words*, *edv=None*, *verify=True*, ***kwargs*)

Bases: `baseband.vdif.header.VDIFHeader`

Base for non-legacy VDIF headers that use 8 32-bit words.

Attributes Summary

`bps`	Bits per elementary sample.
`edv`	VDIF Extended Data Version (EDV).
`frame_nbytes`	Size of the frame in bytes.
`mutable`	Whether the header can be modified.
`nbytes`	Size of the header in bytes.
`nchan`	Number of channels in the frame.
`payload_nbytes`	Size of the payload in bytes.
`samples_per_frame`	Number of complete samples in the frame.
`station`	Station ID: two ASCII characters, or 16-bit int.
`time`	Converts ref_epoch, seconds, and frame_nr to Time object.

Methods Summary

`copy`()	Create a mutable and independent copy of the header.
`from_mark5b_header`(mark5b_header, bps, . . .)	Construct an Mark5B over VDIF header (EDV=0xab).
`fromfile`(fh[, edv, verify])	Read VDIF Header from file.
`fromkeys`(**kwargs)	Initialise a header from parsed values.
`fromvalues`([edv])	Initialise a header from parsed values.
`get_time`([frame_rate])	Converts ref_epoch, seconds, and frame_nr to Time object.
`keys`()	
`same_stream`(other)	Whether header is consistent with being from the same stream.
`set_time`(time[, frame_rate])	Converts Time object to ref_epoch, seconds, and frame_nr.
`tofile`(fh)	Write VLBI frame header to filehandle.
`update`(**kwargs)	Update the header by setting keywords or properties.
`verify`()	Basic checks of header integrity.

Attributes Documentation

bps
Bits per elementary sample.

edv
VDIF Extended Data Version (EDV).

frame_nbytes
Size of the frame in bytes.

mutable
Whether the header can be modified.

nbytes
Size of the header in bytes.

nchan
Number of channels in the frame.

payload_nbytes
Size of the payload in bytes.

samples_per_frame
Number of complete samples in the frame.

station
Station ID: two ASCII characters, or 16-bit int.

time
Converts ref_epoch, seconds, and frame_nr to Time object.

Uses 'ref_epoch', which stores the number of half-years from 2000, and 'seconds'. By default, it also calculates the offset using the current frame number. For non-zero 'frame_nr', this requires the frame rate, which is calculated from the sample rate in the header.

Parameters
frame_rate : `Quantity`, optional

For non-zero 'frame_nr', this is required to calculate the corresponding offset.

Returns
time : `Time`

Methods Documentation

copy()
Create a mutable and independent copy of the header.

Keyword arguments can be passed on as needed by possible subclasses.

classmethod from_mark5b_header(*mark5b_header*, *bps*, *nchan*, ***kwargs*)
Construct an Mark5B over VDIF header (EDV=0xab).

See http://www.vlbi.org/vdif/docs/vdif_extension_0xab.pdf

Note that the Mark 5B header does not encode the bits-per-sample and the number of channels used in the payload, so these need to be given separately. A complete frame can be encapsulated with `from_mark5b_frame`.

Parameters
mark5b_header : `Mark5BHeader`

Used to set time, etc.

bps : int

Bits per elementary sample.

nchan : int

Number of channels carried in the Mark 5B payload.

****kwargs**

Any further arguments. Strictly, none are necessary to create a valid VDIF header, but this can be used to pass on, e.g., `invalid_data`.

classmethod fromfile(*fh*, *edv=None*, *verify=True*)
Read VDIF Header from file.

Parameters
fh : filehandle

To read data from.

edv : int, False, or None, optional

Extended data version. If `False`, a legacy header is used. If `None` (default), it is determined from the header. (Given it explicitly is mostly useful for a slight speed-up.)

verify : bool, optional

Whether to do basic verification of integrity. Default: `True`.

classmethod fromkeys(***kwargs*)

Initialise a header from parsed values.

Like `fromvalues()`, but without any interpretation of keywords.

Raises

KeyError : if not all keys required are pass in.

classmethod fromvalues(*edv=False*, ***kwargs*)

Initialise a header from parsed values.

Here, the parsed values must be given as keyword arguments, i.e., for any `header = cls(<data>)`, `cls.fromvalues(**header) == header`.

However, unlike for the `fromkeys()` class method, data can also be set using arguments named after methods, such as `bps` and `time`.

Given defaults:

invalid_data : `False` legacy_mode : `False` vdif_version : 1 thread_id : 0 frame_nr : 0 sync_pattern : 0xACABFEED for EDV 1 and 3, 0xa5ea5 for EDV 2

Values set by other keyword arguments (if present):

bits_per_sample : from `bps` frame_length : from `samples_per_frame` or `frame_nbytes` lg2_nchan : from `nchan` station_id : from `station` sampling_rate, sampling_unit : from `sample_rate` ref_epoch, seconds, frame_nr : from `time`

Note that to set `time` to non-integer seconds one also needs to pass in `frame_rate` or `sample_rate`.

get_time(*frame_rate=None*)

Converts ref_epoch, seconds, and frame_nr to Time object.

Uses 'ref_epoch', which stores the number of half-years from 2000, and 'seconds'. By default, it also calculates the offset using the current frame number. For non-zero 'frame_nr', this requires the frame rate, which is calculated from the sample rate in the header.

Parameters

frame_rate : `Quantity`, optional

For non-zero 'frame_nr', this is required to calculate the corresponding offset.

Returns

time : `Time`

keys()

same_stream(*other*)

Whether header is consistent with being from the same stream.

set_time(*time*, *frame_rate=None*)

Converts Time object to ref_epoch, seconds, and frame_nr.

For non-integer seconds, a frame rate is needed to calculate the 'frame_nr'.

Parameters

time : Time

The time to use for this header.

frame_rate : Quantity, optional

For calculating 'frame_nr' from the fractional seconds.

tofile(*fh*)

Write VLBI frame header to filehandle.

update(***kwargs*)

Update the header by setting keywords or properties.

Here, any keywords matching header keys are applied first, and any remaining ones are used to set header properties, in the order set by the class (in `_properties`).

Parameters

verify : bool, optional

If True (default), verify integrity after updating.

****kwargs**

Arguments used to set keywords and properties.

verify()

Basic checks of header integrity.

VDIFSampleRateHeader

class baseband.vdif.header.**VDIFSampleRateHeader**(*words*, *edv=None*, *verify=True*, ***kwargs*)

Bases: baseband.vdif.header.VDIFBaseHeader

Base for VDIF headers that include the sample rate (EDV= 1, 3, 4).

Attributes Summary

bps	Bits per elementary sample.
edv	VDIF Extended Data Version (EDV).
frame_nbytes	Size of the frame in bytes.
frame_rate	Number of frames per second.
mutable	Whether the header can be modified.
nbytes	Size of the header in bytes.
nchan	Number of channels in the frame.
payload_nbytes	Size of the payload in bytes.
sample_rate	Number of complete samples per second.
samples_per_frame	Number of complete samples in the frame.
station	Station ID: two ASCII characters, or 16-bit int.
time	Converts ref_epoch, seconds, and frame_nr to Time object.

Methods Summary

copy()	Create a mutable and independent copy of the header.
from_mark5b_header(mark5b_header, bps, ...)	Construct an Mark5B over VDIF header (EDV=0xab).
fromfile(fh[, edv, verify])	Read VDIF Header from file.
fromkeys(**kwargs)	Initialise a header from parsed values.
fromvalues([edv])	Initialise a header from parsed values.
get_time([frame_rate])	Converts ref_epoch, seconds, and frame_nr to Time object.
keys()	
same_stream(other)	
set_time(time[, frame_rate])	Converts Time object to ref_epoch, seconds, and frame_nr.
tofile(fh)	Write VLBI frame header to filehandle.
update(**kwargs)	Update the header by setting keywords or properties.
verify()	Basic checks of header integrity.

Attributes Documentation

bps
Bits per elementary sample.

edv
VDIF Extended Data Version (EDV).

frame_nbytes
Size of the frame in bytes.

frame_rate
Number of frames per second.

Assumes the 'sampling_rate' header field represents a per-channel sample rate for complex samples, or half the sample rate for real ones.

mutable
Whether the header can be modified.

nbytes
Size of the header in bytes.

nchan
Number of channels in the frame.

payload_nbytes
Size of the payload in bytes.

sample_rate
Number of complete samples per second.

Assumes the 'sampling_rate' header field represents a per-channel sample rate for complex samples, or half the sample rate for real ones.

samples_per_frame
Number of complete samples in the frame.

station
Station ID: two ASCII characters, or 16-bit int.

time
Converts ref_epoch, seconds, and frame_nr to Time object.

Uses 'ref_epoch', which stores the number of half-years from 2000, and 'seconds'. By default, it also calculates the offset using the current frame number. For non-zero 'frame_nr', this requires the frame rate, which is calculated from the sample rate in the header. The latter can also be explicitly passed on.

Parameters

frame_rate : `Quantity`, optional

For non-zero 'frame_nr', this is used to calculate the corresponding offset. If not given, the frame rate from the header is used (if it is non-zero).

Returns

time : `Time`

Methods Documentation

`copy()`

Create a mutable and independent copy of the header.

Keyword arguments can be passed on as needed by possible subclasses.

classmethod from_mark5b_header(*mark5b_header*, *bps*, *nchan*, ***kwargs*)

Construct an Mark5B over VDIF header (EDV=0xab).

See http://www.vlbi.org/vdif/docs/vdif_extension_0xab.pdf

Note that the Mark 5B header does not encode the bits-per-sample and the number of channels used in the payload, so these need to be given separately. A complete frame can be encapsulated with `from_mark5b_frame`.

Parameters

mark5b_header : `Mark5BHeader`

Used to set time, etc.

bps : int

Bits per elementary sample.

nchan : int

Number of channels carried in the Mark 5B payload.

****kwargs**

Any further arguments. Strictly, none are necessary to create a valid VDIF header, but this can be used to pass on, e.g., `invalid_data`.

classmethod fromfile(*fh*, *edv=None*, *verify=True*)

Read VDIF Header from file.

Parameters

fh : filehandle

To read data from.

edv : int, False, or None, optional

Extended data version. If `False`, a legacy header is used. If `None` (default), it is determined from the header. (Given it explicitly is mostly useful for a slight speed-up.)

verify : bool, optional

Whether to do basic verification of integrity. Default: `True`.

classmethod fromkeys(***kwargs*)
Initialise a header from parsed values.

Like `fromvalues()`, but without any interpretation of keywords.

Raises
KeyError : if not all keys required are pass in.

classmethod fromvalues(*edv=False*, ***kwargs*)
Initialise a header from parsed values.

Here, the parsed values must be given as keyword arguments, i.e., for any `header = cls(<data>)`, `cls.fromvalues(**header) == header`.

However, unlike for the `fromkeys()` class method, data can also be set using arguments named after methods, such as `bps` and `time`.

Given defaults:

invalid_data : `False` legacy_mode : `False` vdif_version : 1 thread_id : 0 frame_nr : 0 sync_pattern : 0xACABFEED for EDV 1 and 3, 0xa5ea5 for EDV 2

Values set by other keyword arguments (if present):

bits_per_sample : from `bps` frame_length : from `samples_per_frame` or `frame_nbytes` lg2_nchan : from `nchan` station_id : from `station` sampling_rate, sampling_unit : from `sample_rate` ref_epoch, seconds, frame_nr : from `time`

Note that to set `time` to non-integer seconds one also needs to pass in `frame_rate` or `sample_rate`.

get_time(*frame_rate=None*)
Converts ref_epoch, seconds, and frame_nr to Time object.

Uses 'ref_epoch', which stores the number of half-years from 2000, and 'seconds'. By default, it also calculates the offset using the current frame number. For non-zero 'frame_nr', this requires the frame rate, which is calculated from the sample rate in the header. The latter can also be explicitly passed on.

Parameters
frame_rate : `Quantity`, optional

For non-zero 'frame_nr', this is used to calculate the corresponding offset. If not given, the frame rate from the header is used (if it is non-zero).

Returns
time : `Time`

keys()

same_stream(*other*)

set_time(*time*, *frame_rate=None*)
Converts Time object to ref_epoch, seconds, and frame_nr.

Parameters
time : `Time`

The time to use for this header.

frame_rate : `Quantity`, optional

For calculating 'frame_nr' from the fractional seconds. If not given, the frame rate from the header is used (if it is non-zero).

tofile(*fh*)
Write VLBI frame header to filehandle.

update(***kwargs*)
Update the header by setting keywords or properties.

Here, any keywords matching header keys are applied first, and any remaining ones are used to set header properties, in the order set by the class (in `_properties`).

Parameters

verify : bool, optional

If `True` (default), verify integrity after updating.

****kwargs**

Arguments used to set keywords and properties.

verify()
Basic checks of header integrity.

VDIFLegacyHeader

class baseband.vdif.header.**VDIFLegacyHeader**(*words*, *edv=None*, *verify=True*, ***kwargs*)
Bases: `baseband.vdif.header.VDIFHeader`

Legacy VDIF header that uses only 4 32-bit words.

See Section 6 of http://www.vlbi.org/vdif/docs/VDIF_specification_Release_1.1.1.pdf

Attributes Summary

`bps`	Bits per elementary sample.
`edv`	VDIF Extended Data Version (EDV).
`frame_nbytes`	Size of the frame in bytes.
`mutable`	Whether the header can be modified.
`nbytes`	Size of the header in bytes.
`nchan`	Number of channels in the frame.
`payload_nbytes`	Size of the payload in bytes.
`samples_per_frame`	Number of complete samples in the frame.
`station`	Station ID: two ASCII characters, or 16-bit int.
`time`	Converts ref_epoch, seconds, and frame_nr to Time object.

Methods Summary

`copy`()	Create a mutable and independent copy of the header.
`from_mark5b_header`(mark5b_header, bps, …)	Construct an Mark5B over VDIF header (EDV=0xab).
`fromfile`(fh[, edv, verify])	Read VDIF Header from file.
`fromkeys`(**kwargs)	Initialise a header from parsed values.
`fromvalues`([edv])	Initialise a header from parsed values.

Continued on next page

Table 19 – continued from previous page

get_time([frame_rate])	Converts ref_epoch, seconds, and frame_nr to Time object.
keys()	
same_stream(other)	Whether header is consistent with being from the same stream.
set_time(time[, frame_rate])	Converts Time object to ref_epoch, seconds, and frame_nr.
tofile(fh)	Write VLBI frame header to filehandle.
update(**kwargs)	Update the header by setting keywords or properties.
verify()	Basic checks of header integrity.

Attributes Documentation

bps
Bits per elementary sample.

edv
VDIF Extended Data Version (EDV).

frame_nbytes
Size of the frame in bytes.

mutable
Whether the header can be modified.

nbytes
Size of the header in bytes.

nchan
Number of channels in the frame.

payload_nbytes
Size of the payload in bytes.

samples_per_frame
Number of complete samples in the frame.

station
Station ID: two ASCII characters, or 16-bit int.

time
Converts ref_epoch, seconds, and frame_nr to Time object.

Uses 'ref_epoch', which stores the number of half-years from 2000, and 'seconds'. By default, it also calculates the offset using the current frame number. For non-zero 'frame_nr', this requires the frame rate, which is calculated from the sample rate in the header.

Parameters

frame_rate : Quantity, optional

For non-zero 'frame_nr', this is required to calculate the corresponding offset.

Returns

time : Time

Methods Documentation

copy()

Create a mutable and independent copy of the header.

Keyword arguments can be passed on as needed by possible subclasses.

classmethod **from_mark5b_header**(*mark5b_header*, *bps*, *nchan*, ***kwargs*)

Construct an Mark5B over VDIF header (EDV=0xab).

See http://www.vlbi.org/vdif/docs/vdif_extension_0xab.pdf

Note that the Mark 5B header does not encode the bits-per-sample and the number of channels used in the payload, so these need to be given separately. A complete frame can be encapsulated with `from_mark5b_frame`.

Parameters

mark5b_header : `Mark5BHeader`

Used to set time, etc.

bps : int

Bits per elementary sample.

nchan : int

Number of channels carried in the Mark 5B payload.

****kwargs**

Any further arguments. Strictly, none are necessary to create a valid VDIF header, but this can be used to pass on, e.g., `invalid_data`.

classmethod **fromfile**(*fh*, *edv=None*, *verify=True*)

Read VDIF Header from file.

Parameters

fh : filehandle

To read data from.

edv : int, False, or None, optional

Extended data version. If `False`, a legacy header is used. If `None` (default), it is determined from the header. (Given it explicitly is mostly useful for a slight speed-up.)

verify : bool, optional

Whether to do basic verification of integrity. Default: `True`.

classmethod **fromkeys**(***kwargs*)

Initialise a header from parsed values.

Like `fromvalues()`, but without any interpretation of keywords.

Raises

KeyError : if not all keys required are pass in.

classmethod **fromvalues**(*edv=False*, ***kwargs*)

Initialise a header from parsed values.

Here, the parsed values must be given as keyword arguments, i.e., for any `header = cls(<data>)`, `cls.fromvalues(**header) == header`.

However, unlike for the `fromkeys()` class method, data can also be set using arguments named after methods, such as `bps` and `time`.

Given defaults:

invalid_data : `False` legacy_mode : `False` vdif_version : 1 thread_id : 0 frame_nr : 0 sync_pattern : 0xACABFEED for EDV 1 and 3, 0xa5ea5 for EDV 2

Values set by other keyword arguments (if present):

bits_per_sample : from `bps` frame_length : from `samples_per_frame` or `frame_nbytes` lg2_nchan : from `nchan` station_id : from `station` sampling_rate, sampling_unit : from `sample_rate` ref_epoch, seconds, frame_nr : from `time`

Note that to set `time` to non-integer seconds one also needs to pass in `frame_rate` or `sample_rate`.

get_time(*frame_rate=None*)

Converts ref_epoch, seconds, and frame_nr to Time object.

Uses 'ref_epoch', which stores the number of half-years from 2000, and 'seconds'. By default, it also calculates the offset using the current frame number. For non-zero 'frame_nr', this requires the frame rate, which is calculated from the sample rate in the header.

Parameters

frame_rate : `Quantity`, optional

For non-zero 'frame_nr', this is required to calculate the corresponding offset.

Returns

time : `Time`

keys()

same_stream(*other*)

Whether header is consistent with being from the same stream.

set_time(*time*, *frame_rate=None*)

Converts Time object to ref_epoch, seconds, and frame_nr.

For non-integer seconds, a frame rate is needed to calculate the 'frame_nr'.

Parameters

time : `Time`

The time to use for this header.

frame_rate : `Quantity`, optional

For calculating 'frame_nr' from the fractional seconds.

tofile(*fh*)

Write VLBI frame header to filehandle.

update(***kwargs*)

Update the header by setting keywords or properties.

Here, any keywords matching header keys are applied first, and any remaining ones are used to set header properties, in the order set by the class (in `_properties`).

Parameters

verify : bool, optional

If `True` (default), verify integrity after updating.

****kwargs**

Arguments used to set keywords and properties.

verify()

Basic checks of header integrity.

VDIFHeader0

class baseband.vdif.header.**VDIFHeader0**(*words*, *edv=None*, *verify=True*, ***kwargs*)

Bases: baseband.vdif.header.VDIFBaseHeader

VDIF Header for EDV=0.

EDV=0 implies the extended user data fields are not used.

Attributes Summary

bps	Bits per elementary sample.
edv	VDIF Extended Data Version (EDV).
frame_nbytes	Size of the frame in bytes.
mutable	Whether the header can be modified.
nbytes	Size of the header in bytes.
nchan	Number of channels in the frame.
payload_nbytes	Size of the payload in bytes.
samples_per_frame	Number of complete samples in the frame.
station	Station ID: two ASCII characters, or 16-bit int.
time	Converts ref_epoch, seconds, and frame_nr to Time object.

Methods Summary

copy()	Create a mutable and independent copy of the header.
from_mark5b_header(mark5b_header, bps, ...)	Construct an Mark5B over VDIF header (EDV=0xab).
fromfile(fh[, edv, verify])	Read VDIF Header from file.
fromkeys(**kwargs)	Initialise a header from parsed values.
fromvalues([edv])	Initialise a header from parsed values.
get_time([frame_rate])	Converts ref_epoch, seconds, and frame_nr to Time object.
keys()	
same_stream(other)	Whether header is consistent with being from the same stream.
set_time(time[, frame_rate])	Converts Time object to ref_epoch, seconds, and frame_nr.
tofile(fh)	Write VLBI frame header to filehandle.
update(**kwargs)	Update the header by setting keywords or properties.
verify()	

Attributes Documentation

bps
Bits per elementary sample.

edv
VDIF Extended Data Version (EDV).

frame_nbytes
Size of the frame in bytes.

mutable
Whether the header can be modified.

nbytes
Size of the header in bytes.

nchan
Number of channels in the frame.

payload_nbytes
Size of the payload in bytes.

samples_per_frame
Number of complete samples in the frame.

station
Station ID: two ASCII characters, or 16-bit int.

time
Converts ref_epoch, seconds, and frame_nr to Time object.

Uses 'ref_epoch', which stores the number of half-years from 2000, and 'seconds'. By default, it also calculates the offset using the current frame number. For non-zero 'frame_nr', this requires the frame rate, which is calculated from the sample rate in the header.

Parameters
frame_rate : `Quantity`, optional

For non-zero 'frame_nr', this is required to calculate the corresponding offset.

Returns
time : `Time`

Methods Documentation

copy()
Create a mutable and independent copy of the header.

Keyword arguments can be passed on as needed by possible subclasses.

classmethod from_mark5b_header(*mark5b_header*, *bps*, *nchan*, ***kwargs*)
Construct an Mark5B over VDIF header (EDV=0xab).

See http://www.vlbi.org/vdif/docs/vdif_extension_0xab.pdf

Note that the Mark 5B header does not encode the bits-per-sample and the number of channels used in the payload, so these need to be given separately. A complete frame can be encapsulated with `from_mark5b_frame`.

Parameters
mark5b_header : `Mark5BHeader`

Used to set time, etc.

bps : int

Bits per elementary sample.

nchan : int

Number of channels carried in the Mark 5B payload.

****kwargs**

Any further arguments. Strictly, none are necessary to create a valid VDIF header, but this can be used to pass on, e.g., `invalid_data`.

classmethod fromfile(*fh*, *edv=None*, *verify=True*)

Read VDIF Header from file.

Parameters

fh : filehandle

To read data from.

edv : int, False, or None, optional

Extended data version. If `False`, a legacy header is used. If `None` (default), it is determined from the header. (Given it explicitly is mostly useful for a slight speed-up.)

verify : bool, optional

Whether to do basic verification of integrity. Default: `True`.

classmethod fromkeys(***kwargs*)

Initialise a header from parsed values.

Like `fromvalues()`, but without any interpretation of keywords.

Raises

KeyError : if not all keys required are pass in.

classmethod fromvalues(*edv=False*, ***kwargs*)

Initialise a header from parsed values.

Here, the parsed values must be given as keyword arguments, i.e., for any `header = cls(<data>)`, `cls.fromvalues(**header) == header`.

However, unlike for the `fromkeys()` class method, data can also be set using arguments named after methods, such as `bps` and `time`.

Given defaults:

invalid_data : `False` legacy_mode : `False` vdif_version : 1 thread_id : 0 frame_nr : 0 sync_pattern : 0xACABFEED for EDV 1 and 3, 0xa5ea5 for EDV 2

Values set by other keyword arguments (if present):

bits_per_sample : from `bps` frame_length : from `samples_per_frame` or `frame_nbytes` lg2_nchan : from `nchan` station_id : from `station` sampling_rate, sampling_unit : from `sample_rate` ref_epoch, seconds, frame_nr : from `time`

Note that to set `time` to non-integer seconds one also needs to pass in `frame_rate` or `sample_rate`.

get_time(*frame_rate=None*)

Converts ref_epoch, seconds, and frame_nr to Time object.

Uses 'ref_epoch', which stores the number of half-years from 2000, and 'seconds'. By default, it also calculates the offset using the current frame number. For non-zero 'frame_nr', this requires the frame rate, which is calculated from the sample rate in the header.

Parameters
frame_rate : `Quantity`, optional

For non-zero 'frame_nr', this is required to calculate the corresponding offset.

Returns
time : `Time`

keys()

same_stream(*other*)
Whether header is consistent with being from the same stream.

set_time(*time*, *frame_rate=None*)
Converts Time object to ref_epoch, seconds, and frame_nr.

For non-integer seconds, a frame rate is needed to calculate the 'frame_nr'.

Parameters
time : `Time`

The time to use for this header.

frame_rate : `Quantity`, optional

For calculating 'frame_nr' from the fractional seconds.

tofile(*fh*)
Write VLBI frame header to filehandle.

update(***kwargs*)
Update the header by setting keywords or properties.

Here, any keywords matching header keys are applied first, and any remaining ones are used to set header properties, in the order set by the class (in `_properties`).

Parameters
verify : bool, optional

If `True` (default), verify integrity after updating.

****kwargs**

Arguments used to set keywords and properties.

verify()

VDIFHeader1

class baseband.vdif.header.**VDIFHeader1**(*words*, *edv=None*, *verify=True*, ***kwargs*)
Bases: `baseband.vdif.header.VDIFSampleRateHeader`

VDIF Header for EDV=1.

See http://www.vlbi.org/vdif/docs/vdif_extension_0x01.pdf

Attributes Summary

bps	Bits per elementary sample.
edv	VDIF Extended Data Version (EDV).
frame_nbytes	Size of the frame in bytes.
frame_rate	Number of frames per second.
mutable	Whether the header can be modified.
nbytes	Size of the header in bytes.
nchan	Number of channels in the frame.
payload_nbytes	Size of the payload in bytes.
sample_rate	Number of complete samples per second.
samples_per_frame	Number of complete samples in the frame.
station	Station ID: two ASCII characters, or 16-bit int.
time	Converts ref_epoch, seconds, and frame_nr to Time object.

Methods Summary

copy()	Create a mutable and independent copy of the header.
from_mark5b_header(mark5b_header, bps, ...)	Construct an Mark5B over VDIF header (EDV=0xab).
fromfile(fh[, edv, verify])	Read VDIF Header from file.
fromkeys(**kwargs)	Initialise a header from parsed values.
fromvalues([edv])	Initialise a header from parsed values.
get_time([frame_rate])	Converts ref_epoch, seconds, and frame_nr to Time object.
keys()	
same_stream(other)	
set_time(time[, frame_rate])	Converts Time object to ref_epoch, seconds, and frame_nr.
tofile(fh)	Write VLBI frame header to filehandle.
update(**kwargs)	Update the header by setting keywords or properties.
verify()	Basic checks of header integrity.

Attributes Documentation

bps

Bits per elementary sample.

edv

VDIF Extended Data Version (EDV).

frame_nbytes

Size of the frame in bytes.

frame_rate

Number of frames per second.

Assumes the 'sampling_rate' header field represents a per-channel sample rate for complex samples, or half the sample rate for real ones.

mutable

Whether the header can be modified.

nbytes
Size of the header in bytes.

nchan
Number of channels in the frame.

payload_nbytes
Size of the payload in bytes.

sample_rate
Number of complete samples per second.

Assumes the 'sampling_rate' header field represents a per-channel sample rate for complex samples, or half the sample rate for real ones.

samples_per_frame
Number of complete samples in the frame.

station
Station ID: two ASCII characters, or 16-bit int.

time
Converts ref_epoch, seconds, and frame_nr to Time object.

Uses 'ref_epoch', which stores the number of half-years from 2000, and 'seconds'. By default, it also calculates the offset using the current frame number. For non-zero 'frame_nr', this requires the frame rate, which is calculated from the sample rate in the header. The latter can also be explicitly passed on.

Parameters
frame_rate : `Quantity`, optional

For non-zero 'frame_nr', this is used to calculate the corresponding offset. If not given, the frame rate from the header is used (if it is non-zero).

Returns
time : `Time`

Methods Documentation

copy()
Create a mutable and independent copy of the header.

Keyword arguments can be passed on as needed by possible subclasses.

classmethod from_mark5b_header(*mark5b_header*, *bps*, *nchan*, ***kwargs*)
Construct an Mark5B over VDIF header (EDV=0xab).

See http://www.vlbi.org/vdif/docs/vdif_extension_0xab.pdf

Note that the Mark 5B header does not encode the bits-per-sample and the number of channels used in the payload, so these need to be given separately. A complete frame can be encapsulated with `from_mark5b_frame`.

Parameters
mark5b_header : `Mark5BHeader`

Used to set time, etc.

bps : int

Bits per elementary sample.

nchan : int

Number of channels carried in the Mark 5B payload.

****kwargs**

Any further arguments. Strictly, none are necessary to create a valid VDIF header, but this can be used to pass on, e.g., `invalid_data`.

classmethod fromfile(*fh*, *edv=None*, *verify=True*)
Read VDIF Header from file.

Parameters
fh : filehandle

To read data from.

edv : int, False, or None, optional

Extended data version. If `False`, a legacy header is used. If `None` (default), it is determined from the header. (Given it explicitly is mostly useful for a slight speed-up.)

verify : bool, optional

Whether to do basic verification of integrity. Default: `True`.

classmethod fromkeys(***kwargs*)
Initialise a header from parsed values.

Like `fromvalues()`, but without any interpretation of keywords.

Raises
KeyError : if not all keys required are pass in.

classmethod fromvalues(*edv=False*, ***kwargs*)
Initialise a header from parsed values.

Here, the parsed values must be given as keyword arguments, i.e., for any `header = cls(<data>)`, `cls.fromvalues(**header) == header`.

However, unlike for the `fromkeys()` class method, data can also be set using arguments named after methods, such as `bps` and `time`.

Given defaults:

invalid_data : `False` legacy_mode : `False` vdif_version : 1 thread_id : 0 frame_nr : 0 sync_pattern : 0xACABFEED for EDV 1 and 3, 0xa5ea5 for EDV 2

Values set by other keyword arguments (if present):

bits_per_sample : from `bps` frame_length : from `samples_per_frame` or `frame_nbytes` lg2_nchan : from `nchan` station_id : from `station` sampling_rate, sampling_unit : from `sample_rate` ref_epoch, seconds, frame_nr : from `time`

Note that to set `time` to non-integer seconds one also needs to pass in `frame_rate` or `sample_rate`.

get_time(*frame_rate=None*)
Converts ref_epoch, seconds, and frame_nr to Time object.

Uses 'ref_epoch', which stores the number of half-years from 2000, and 'seconds'. By default, it also calculates the offset using the current frame number. For non-zero 'frame_nr', this requires the frame rate, which is calculated from the sample rate in the header. The latter can also be explicitly passed on.

Parameters
frame_rate : `Quantity`, optional

For non-zero 'frame_nr', this is used to calculate the corresponding offset. If not given, the frame rate from the header is used (if it is non-zero).

Returns
time : `Time`

`keys()`

`same_stream(other)`

`set_time(time, frame_rate=None)`
Converts Time object to ref_epoch, seconds, and frame_nr.

Parameters
time : `Time`

The time to use for this header.

frame_rate : `Quantity`, optional

For calculating 'frame_nr' from the fractional seconds. If not given, the frame rate from the header is used (if it is non-zero).

`tofile(fh)`
Write VLBI frame header to filehandle.

`update(**kwargs)`
Update the header by setting keywords or properties.

Here, any keywords matching header keys are applied first, and any remaining ones are used to set header properties, in the order set by the class (in `_properties`).

Parameters
verify : bool, optional

If `True` (default), verify integrity after updating.

****kwargs**

Arguments used to set keywords and properties.

`verify()`
Basic checks of header integrity.

VDIFHeader2

class `baseband.vdif.header.VDIFHeader2(words, edv=None, verify=True, **kwargs)`
Bases: `baseband.vdif.header.VDIFBaseHeader`

VDIF Header for EDV=2.

See http://www.vlbi.org/vdif/docs/alma-vdif-edv.pdf

Notes

This header is untested. It may need to have subclasses, based on possible different sync values.

Attributes Summary

bps	Bits per elementary sample.
edv	VDIF Extended Data Version (EDV).
frame_nbytes	Size of the frame in bytes.
mutable	Whether the header can be modified.
nbytes	Size of the header in bytes.
nchan	Number of channels in the frame.
payload_nbytes	Size of the payload in bytes.
samples_per_frame	Number of complete samples in the frame.
station	Station ID: two ASCII characters, or 16-bit int.
time	Converts ref_epoch, seconds, and frame_nr to Time object.

Methods Summary

copy()	Create a mutable and independent copy of the header.
from_mark5b_header(mark5b_header, bps, . . .)	Construct an Mark5B over VDIF header (EDV=0xab).
fromfile(fh[, edv, verify])	Read VDIF Header from file.
fromkeys(**kwargs)	Initialise a header from parsed values.
fromvalues([edv])	Initialise a header from parsed values.
get_time([frame_rate])	Converts ref_epoch, seconds, and frame_nr to Time object.
keys()	
same_stream(other)	Whether header is consistent with being from the same stream.
set_time(time[, frame_rate])	Converts Time object to ref_epoch, seconds, and frame_nr.
tofile(fh)	Write VLBI frame header to filehandle.
update(**kwargs)	Update the header by setting keywords or properties.
verify()	

Attributes Documentation

bps
Bits per elementary sample.

edv
VDIF Extended Data Version (EDV).

frame_nbytes
Size of the frame in bytes.

mutable
Whether the header can be modified.

nbytes
Size of the header in bytes.

nchan
Number of channels in the frame.

payload_nbytes
Size of the payload in bytes.

samples_per_frame
Number of complete samples in the frame.

station
Station ID: two ASCII characters, or 16-bit int.

time
Converts ref_epoch, seconds, and frame_nr to Time object.

Uses 'ref_epoch', which stores the number of half-years from 2000, and 'seconds'. By default, it also calculates the offset using the current frame number. For non-zero 'frame_nr', this requires the frame rate, which is calculated from the sample rate in the header.

Parameters
frame_rate : `Quantity`, optional

For non-zero 'frame_nr', this is required to calculate the corresponding offset.

Returns
time : `Time`

Methods Documentation

copy()
Create a mutable and independent copy of the header.

Keyword arguments can be passed on as needed by possible subclasses.

classmethod from_mark5b_header(*mark5b_header*, *bps*, *nchan*, ***kwargs*)
Construct an Mark5B over VDIF header (EDV=0xab).

See http://www.vlbi.org/vdif/docs/vdif_extension_0xab.pdf

Note that the Mark 5B header does not encode the bits-per-sample and the number of channels used in the payload, so these need to be given separately. A complete frame can be encapsulated with `from_mark5b_frame`.

Parameters
mark5b_header : `Mark5BHeader`

Used to set time, etc.

bps : int

Bits per elementary sample.

nchan : int

Number of channels carried in the Mark 5B payload.

****kwargs**

Any further arguments. Strictly, none are necessary to create a valid VDIF header, but this can be used to pass on, e.g., `invalid_data`.

classmethod fromfile(*fh*, *edv=None*, *verify=True*)
Read VDIF Header from file.

Parameters
fh : filehandle

To read data from.

edv : int, False, or None, optional

Extended data version. If `False`, a legacy header is used. If `None` (default), it is determined from the header. (Given it explicitly is mostly useful for a slight speed-up.)

verify : bool, optional

Whether to do basic verification of integrity. Default: `True`.

classmethod fromkeys(***kwargs*)

Initialise a header from parsed values.

Like `fromvalues()`, but without any interpretation of keywords.

Raises

KeyError : if not all keys required are pass in.

classmethod fromvalues(*edv=False*, ***kwargs*)

Initialise a header from parsed values.

Here, the parsed values must be given as keyword arguments, i.e., for any `header = cls(<data>)`, `cls.fromvalues(**header) == header`.

However, unlike for the `fromkeys()` class method, data can also be set using arguments named after methods, such as `bps` and `time`.

Given defaults:

invalid_data : `False` legacy_mode : `False` vdif_version : 1 thread_id : 0 frame_nr : 0 sync_pattern : 0xACABFEED for EDV 1 and 3, 0xa5ea5 for EDV 2

Values set by other keyword arguments (if present):

bits_per_sample : from `bps` frame_length : from `samples_per_frame` or `frame_nbytes` lg2_nchan : from `nchan` station_id : from `station` sampling_rate, sampling_unit : from `sample_rate` ref_epoch, seconds, frame_nr : from `time`

Note that to set `time` to non-integer seconds one also needs to pass in `frame_rate` or `sample_rate`.

get_time(*frame_rate=None*)

Converts ref_epoch, seconds, and frame_nr to Time object.

Uses 'ref_epoch', which stores the number of half-years from 2000, and 'seconds'. By default, it also calculates the offset using the current frame number. For non-zero 'frame_nr', this requires the frame rate, which is calculated from the sample rate in the header.

Parameters

frame_rate : `Quantity`, optional

For non-zero 'frame_nr', this is required to calculate the corresponding offset.

Returns

time : `Time`

keys()

same_stream(*other*)

Whether header is consistent with being from the same stream.

set_time(*time*, *frame_rate=None*)

Converts Time object to ref_epoch, seconds, and frame_nr.

For non-integer seconds, a frame rate is needed to calculate the 'frame_nr'.

Parameters

time : `Time`

The time to use for this header.

frame_rate : `Quantity`, optional

For calculating 'frame_nr' from the fractional seconds.

tofile(*fh*)

Write VLBI frame header to filehandle.

update(***kwargs*)

Update the header by setting keywords or properties.

Here, any keywords matching header keys are applied first, and any remaining ones are used to set header properties, in the order set by the class (in `_properties`).

Parameters

verify : bool, optional

If `True` (default), verify integrity after updating.

****kwargs**

Arguments used to set keywords and properties.

verify()

VDIFHeader3

class baseband.vdif.header.**VDIFHeader3**(*words, edv=None, verify=True, **kwargs*)

Bases: `baseband.vdif.header.VDIFSampleRateHeader`

VDIF Header for EDV=3.

See http://www.vlbi.org/vdif/docs/vdif_extension_0x03.pdf

Attributes Summary

`bps`	Bits per elementary sample.
`edv`	VDIF Extended Data Version (EDV).
`frame_nbytes`	Size of the frame in bytes.
`frame_rate`	Number of frames per second.
`mutable`	Whether the header can be modified.
`nbytes`	Size of the header in bytes.
`nchan`	Number of channels in the frame.
`payload_nbytes`	Size of the payload in bytes.
`sample_rate`	Number of complete samples per second.
`samples_per_frame`	Number of complete samples in the frame.
`station`	Station ID: two ASCII characters, or 16-bit int.
`time`	Converts ref_epoch, seconds, and frame_nr to Time object.

Methods Summary

copy()	Create a mutable and independent copy of the header.
from_mark5b_header(mark5b_header, bps, ...)	Construct an Mark5B over VDIF header (EDV=0xab).
fromfile(fh[, edv, verify])	Read VDIF Header from file.
fromkeys(**kwargs)	Initialise a header from parsed values.
fromvalues([edv])	Initialise a header from parsed values.
get_time([frame_rate])	Converts ref_epoch, seconds, and frame_nr to Time object.
keys()	
same_stream(other)	
set_time(time[, frame_rate])	Converts Time object to ref_epoch, seconds, and frame_nr.
tofile(fh)	Write VLBI frame header to filehandle.
update(**kwargs)	Update the header by setting keywords or properties.
verify()	

Attributes Documentation

bps
Bits per elementary sample.

edv
VDIF Extended Data Version (EDV).

frame_nbytes
Size of the frame in bytes.

frame_rate
Number of frames per second.

Assumes the 'sampling_rate' header field represents a per-channel sample rate for complex samples, or half the sample rate for real ones.

mutable
Whether the header can be modified.

nbytes
Size of the header in bytes.

nchan
Number of channels in the frame.

payload_nbytes
Size of the payload in bytes.

sample_rate
Number of complete samples per second.

Assumes the 'sampling_rate' header field represents a per-channel sample rate for complex samples, or half the sample rate for real ones.

samples_per_frame
Number of complete samples in the frame.

station
Station ID: two ASCII characters, or 16-bit int.

time
Converts ref_epoch, seconds, and frame_nr to Time object.

Uses 'ref_epoch', which stores the number of half-years from 2000, and 'seconds'. By default, it also calculates the offset using the current frame number. For non-zero 'frame_nr', this requires the frame rate, which is calculated from the sample rate in the header. The latter can also be explicitly passed on.

Parameters

frame_rate : `Quantity`, optional

For non-zero 'frame_nr', this is used to calculate the corresponding offset. If not given, the frame rate from the header is used (if it is non-zero).

Returns

time : `Time`

Methods Documentation

copy()

Create a mutable and independent copy of the header.

Keyword arguments can be passed on as needed by possible subclasses.

classmethod from_mark5b_header(*mark5b_header*, *bps*, *nchan*, ***kwargs*)

Construct an Mark5B over VDIF header (EDV=0xab).

See http://www.vlbi.org/vdif/docs/vdif_extension_0xab.pdf

Note that the Mark 5B header does not encode the bits-per-sample and the number of channels used in the payload, so these need to be given separately. A complete frame can be encapsulated with `from_mark5b_frame`.

Parameters

mark5b_header : `Mark5BHeader`

Used to set time, etc.

bps : int

Bits per elementary sample.

nchan : int

Number of channels carried in the Mark 5B payload.

****kwargs**

Any further arguments. Strictly, none are necessary to create a valid VDIF header, but this can be used to pass on, e.g., `invalid_data`.

classmethod fromfile(*fh*, *edv=None*, *verify=True*)

Read VDIF Header from file.

Parameters

fh : filehandle

To read data from.

edv : int, False, or None, optional

Extended data version. If `False`, a legacy header is used. If `None` (default), it is determined from the header. (Given it explicitly is mostly useful for a slight speed-up.)

verify : bool, optional

Whether to do basic verification of integrity. Default: `True`.

classmethod fromkeys(***kwargs*)

Initialise a header from parsed values.

Like `fromvalues()`, but without any interpretation of keywords.

Raises

KeyError : if not all keys required are pass in.

classmethod fromvalues(*edv=False*, ***kwargs*)

Initialise a header from parsed values.

Here, the parsed values must be given as keyword arguments, i.e., for any `header = cls(<data>)`, `cls.fromvalues(**header) == header`.

However, unlike for the `fromkeys()` class method, data can also be set using arguments named after methods, such as `bps` and `time`.

Given defaults:

invalid_data : `False` legacy_mode : `False` vdif_version : 1 thread_id : 0 frame_nr : 0 sync_pattern : 0xACABFEED for EDV 1 and 3, 0xa5ea5 for EDV 2

Values set by other keyword arguments (if present):

bits_per_sample : from `bps` frame_length : from `samples_per_frame` or `frame_nbytes` lg2_nchan : from `nchan` station_id : from `station` sampling_rate, sampling_unit : from `sample_rate` ref_epoch, seconds, frame_nr : from `time`

Note that to set `time` to non-integer seconds one also needs to pass in `frame_rate` or `sample_rate`.

get_time(*frame_rate=None*)

Converts ref_epoch, seconds, and frame_nr to Time object.

Uses 'ref_epoch', which stores the number of half-years from 2000, and 'seconds'. By default, it also calculates the offset using the current frame number. For non-zero 'frame_nr', this requires the frame rate, which is calculated from the sample rate in the header. The latter can also be explicitly passed on.

Parameters

frame_rate : `Quantity`, optional

For non-zero 'frame_nr', this is used to calculate the corresponding offset. If not given, the frame rate from the header is used (if it is non-zero).

Returns

time : `Time`

keys()

same_stream(*other*)

set_time(*time*, *frame_rate=None*)

Converts Time object to ref_epoch, seconds, and frame_nr.

Parameters

time : `Time`

The time to use for this header.

frame_rate : `Quantity`, optional

For calculating 'frame_nr' from the fractional seconds. If not given, the frame rate from the header is used (if it is non-zero).

tofile(*fh*)
Write VLBI frame header to filehandle.

update(***kwargs*)
Update the header by setting keywords or properties.

Here, any keywords matching header keys are applied first, and any remaining ones are used to set header properties, in the order set by the class (in `_properties`).

Parameters
verify : bool, optional

If `True` (default), verify integrity after updating.

****kwargs**

Arguments used to set keywords and properties.

verify()

VDIFMark5BHeader

class baseband.vdif.header.**VDIFMark5BHeader**(*words*, *edv=None*, *verify=True*, ***kwargs*)
Bases: `baseband.vdif.header.VDIFBaseHeader`, `baseband.mark5b.header.Mark5BHeader`

Mark 5B over VDIF (EDV=0xab).

See http://www.vlbi.org/vdif/docs/vdif_extension_0xab.pdf

Attributes Summary

`bps`	Bits per elementary sample.
`edv`	VDIF Extended Data Version (EDV).
`fraction`	Fractional seconds (decoded from 'bcd_fraction').
`frame_nbytes`	Size of the frame in bytes.
`jday`	Last three digits of MJD (decoded from 'bcd_jday').
`kday`	
`mutable`	Whether the header can be modified.
`nbytes`	Size of the header in bytes.
`nchan`	Number of channels in the frame.
`payload_nbytes`	Size of the payload in bytes.
`samples_per_frame`	Number of complete samples in the frame.
`seconds`	Integer seconds on day (decoded from 'bcd_seconds').
`station`	Station ID: two ASCII characters, or 16-bit int.
`time`	Convert ref_epoch, seconds, and fractional seconds to Time object.

Methods Summary

`copy`()	Create a mutable and independent copy of the header.

Continued on next page

Table 29 – continued from previous page

from_mark5b_header(mark5b_header, bps, ...)	Construct an Mark5B over VDIF header (EDV=0xab).
fromfile(fh[, edv, verify])	Read VDIF Header from file.
fromkeys(**kwargs)	Initialise a header from parsed values.
fromvalues([edv])	Initialise a header from parsed values.
get_time([frame_rate])	Convert ref_epoch, seconds, and fractional seconds to Time object.
infer_kday(ref_time)	Uses a reference time to set a header's kday.
keys()	
same_stream(other)	Whether header is consistent with being from the same stream.
set_time(time[, frame_rate])	
tofile(fh)	Write VLBI frame header to filehandle.
update(**kwargs)	Update the header by setting keywords or properties.
verify()	

Attributes Documentation

bps
Bits per elementary sample.

edv
VDIF Extended Data Version (EDV).

fraction
Fractional seconds (decoded from 'bcd_fraction').

The fraction is stored to 0.1 ms accuracy. Following mark5access, this is "unrounded" to give the exact time of the start of the frame for any total bit rate below 512 Mbps. For rates above this value, it is no longer guaranteed that subsequent frames have unique rates.

Note to the above: since a Mark5B frame contains 80000 bits, the total bit rate for which times can be unique would in principle be 800 Mbps. However, standard VLBI only uses bit rates that are powers of 2 in MHz.

frame_nbytes
Size of the frame in bytes.

jday
Last three digits of MJD (decoded from 'bcd_jday').

kday = None

mutable
Whether the header can be modified.

nbytes
Size of the header in bytes.

nchan
Number of channels in the frame.

payload_nbytes
Size of the payload in bytes.

samples_per_frame
Number of complete samples in the frame.

seconds
Integer seconds on day (decoded from 'bcd_seconds').

station
Station ID: two ASCII characters, or 16-bit int.

time
Convert ref_epoch, seconds, and fractional seconds to Time object.

Uses 'ref_epoch', which stores the number of half-years from 2000, and 'seconds', from the VDIF part of the header, and the fractional seconds from the Mark 5B part.

Since some Mark 5B headers do not store the fractional seconds, one can also calculates the offset using the current frame number by passing in a sample rate.

Furthermore, fractional seconds are stored only to 0.1 ms accuracy. In the code, this is "unrounded" to give the exact time of the start of the frame for any total bit rate below 512 Mbps. For rates above this value, it is no longer guaranteed that subsequent frames have unique rates, and one should pass in an explicit sample rate instead.

Parameters
frame_rate : `Quantity`, optional

For non-zero 'frame_nr', this is used to calculate the corresponding offset.

Returns
time : `Time`

Methods Documentation

copy()
Create a mutable and independent copy of the header.

Keyword arguments can be passed on as needed by possible subclasses.

classmethod from_mark5b_header(*mark5b_header*, *bps*, *nchan*, ***kwargs*)
Construct an Mark5B over VDIF header (EDV=0xab).

See http://www.vlbi.org/vdif/docs/vdif_extension_0xab.pdf

Note that the Mark 5B header does not encode the bits-per-sample and the number of channels used in the payload, so these need to be given separately. A complete frame can be encapsulated with `from_mark5b_frame`.

Parameters
mark5b_header : `Mark5BHeader`

Used to set time, etc.

bps : int

Bits per elementary sample.

nchan : int

Number of channels carried in the Mark 5B payload.

****kwargs**

Any further arguments. Strictly, none are necessary to create a valid VDIF header, but this can be used to pass on, e.g., `invalid_data`.

classmethod fromfile(*fh*, *edv=None*, *verify=True*)
Read VDIF Header from file.

Parameters
fh : filehandle

To read data from.

edv : int, False, or None, optional

Extended data version. If `False`, a legacy header is used. If `None` (default), it is determined from the header. (Given it explicitly is mostly useful for a slight speed-up.)

verify : bool, optional

Whether to do basic verification of integrity. Default: `True`.

classmethod fromkeys(***kwargs*)
Initialise a header from parsed values.

Like `fromvalues()`, but without any interpretation of keywords.

Raises
KeyError : if not all keys required are pass in.

classmethod fromvalues(*edv=False*, ***kwargs*)
Initialise a header from parsed values.

Here, the parsed values must be given as keyword arguments, i.e., for any `header = cls(<data>)`, `cls.fromvalues(**header) == header`.

However, unlike for the `fromkeys()` class method, data can also be set using arguments named after methods, such as `bps` and `time`.

Given defaults:

invalid_data : `False` legacy_mode : `False` vdif_version : 1 thread_id : 0 frame_nr : 0 sync_pattern : 0xACABFEED for EDV 1 and 3, 0xa5ea5 for EDV 2

Values set by other keyword arguments (if present):

bits_per_sample : from `bps` frame_length : from `samples_per_frame` or `frame_nbytes` lg2_nchan : from `nchan` station_id : from `station` sampling_rate, sampling_unit : from `sample_rate` ref_epoch, seconds, frame_nr : from `time`

Note that to set `time` to non-integer seconds one also needs to pass in `frame_rate` or `sample_rate`.

get_time(*frame_rate=None*)
Convert ref_epoch, seconds, and fractional seconds to Time object.

Uses 'ref_epoch', which stores the number of half-years from 2000, and 'seconds', from the VDIF part of the header, and the fractional seconds from the Mark 5B part.

Since some Mark 5B headers do not store the fractional seconds, one can also calculates the offset using the current frame number by passing in a sample rate.

Furthermore, fractional seconds are stored only to 0.1 ms accuracy. In the code, this is "unrounded" to give the exact time of the start of the frame for any total bit rate below 512 Mbps. For rates above this value, it is no longer guaranteed that subsequent frames have unique rates, and one should pass in an explicit sample rate instead.

Parameters
frame_rate : `Quantity`, optional

For non-zero 'frame_nr', this is used to calculate the corresponding offset.

Returns

time : Time

infer_kday(*ref_time*)

Uses a reference time to set a header's kday.

Parameters

ref_time : Time

Reference time within 500 days of the observation time.

keys()

same_stream(*other*)

Whether header is consistent with being from the same stream.

set_time(*time*, *frame_rate=None*)

tofile(*fh*)

Write VLBI frame header to filehandle.

update(***kwargs*)

Update the header by setting keywords or properties.

Here, any keywords matching header keys are applied first, and any remaining ones are used to set header properties, in the order set by the class (in `_properties`).

Parameters

crc : int or None, optional

If None (default), recalculate the CRC after updating.

verify : bool, optional

If True (default), verify integrity after updating.

****kwargs**

Arguments used to set keywords and properties.

verify()

Variables

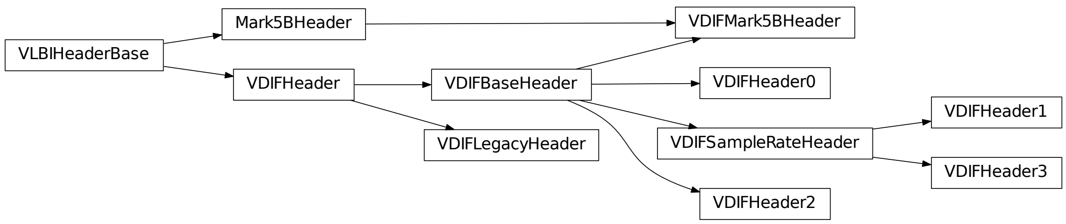
VDIF_HEADER_CLASSES	Dict for storing VDIF header class definitions, indexed by their EDV.

VDIF_HEADER_CLASSES

`baseband.vdif.header.VDIF_HEADER_CLASSES = {-1: <class 'baseband.vdif.header.VDIFLegacyHeader'>, 0: <class 'b`

Dict for storing VDIF header class definitions, indexed by their EDV.

Class Inheritance Diagram



4.4.3 baseband.vdif.payload Module

Definitions for VLBI VDIF payloads.

Implements a VDIFPayload class used to store payload words, and decode to or encode from a data array.

See the VDIF specification page for payload specifications.

Functions

`init_luts`()	Sets up the look-up tables for levels as a function of input byte.
`decode_2bit`(words)	Decodes data stored using 2 bits per sample.
`decode_4bit`(words)	Decodes data stored using 4 bits per sample.
`encode_2bit`(values)	Encodes values using 2 bits per sample, packing the result into bytes.
`encode_4bit`(values)	Encodes values using 4 bits per sample, packing the result into bytes.

init_luts

`baseband.vdif.payload.`**`init_luts`**`()`

Sets up the look-up tables for levels as a function of input byte.

Returns

lut1bit : `ndarray`

Look-up table for decoding bytes to 1-bit samples.

lut2bit : `ndarray`

As `lut1bit1`, but for 2-bit samples.

lut4bit : `ndarray`

As `lut1bit1`, but for 4-bit samples.

Notes

Look-up tables are two-dimensional arrays whose first axis is indexed by byte value (in uint8 form) and whose second axis represents sample temporal order. Table values are decoded sample values. Sec. 10 in the VDIF Specification states that samples are encoded by offset-binary, such that all 0 bits is lowest and all 1 bits is highest. I.e., for 2-bit sampling, the order is 00, 01, 10, 11. These are decoded using `decoder_levels`.

For example, the 2-bit sample sequence `-1, -1, 1, 1` is encoded as `0b10100101` (or `165` in uint8 form). To translate this back to sample values, access `lut2bit` using the byte as the key:

```
>>> lut2bit[0b10100101]
array([-1., -1.,  1.,  1.], dtype=float32)
```

decode_2bit

`baseband.vdif.payload.`**`decode_2bit`**`(`*words*`)`
Decodes data stored using 2 bits per sample.

decode_4bit

`baseband.vdif.payload.`**`decode_4bit`**`(`*words*`)`
Decodes data stored using 4 bits per sample.

encode_2bit

`baseband.vdif.payload.`**`encode_2bit`**`(`*values*`)`
Encodes values using 2 bits per sample, packing the result into bytes.

encode_4bit

`baseband.vdif.payload.`**`encode_4bit`**`(`*values*`)`
Encodes values using 4 bits per sample, packing the result into bytes.

Classes

`VDIFPayload`(words[, header, nchan, bps, . . .])	Container for decoding and encoding VDIF payloads.

VDIFPayload

class `baseband.vdif.payload.`**`VDIFPayload`**`(`*words*, *header=None*, *nchan=1*, *bps=2*, *complex_data=False*`)`

Bases: `baseband.vlbi_base.payload.VLBIPayloadBase`

Container for decoding and encoding VDIF payloads.

Parameters

words : `ndarray`

Array containg LSB unsigned words (with the right size) that encode the payload.

header : VDIFHeader

If given, used to infer the number of channels, bps, and whether the data are complex.

nchan : int, optional

Number of channels, used if header is not given. Default: 1.

bps : int, optional

Bits per elementary sample, used if header is not given. Default: 2.

complex_data : bool, optional

Whether the data are complex, used if header is not given. Default: False.

Attributes Summary

data	Full decoded payload.
dtype	Numeric type of the decoded data array.
nbytes	Size of the payload in bytes.
ndim	Number of dimensions of the decoded data array.
shape	Shape of the decoded data array.
size	Total number of component samples in the decoded data array.

Methods Summary

fromdata(data[, header, bps, edv])	Encode data as payload, using header information.
fromfile(fh, header)	Read payload from filehandle and decode it into data.
tofile(fh)	Write payload to filehandle.

Attributes Documentation

data
Full decoded payload.

dtype
Numeric type of the decoded data array.

nbytes
Size of the payload in bytes.

ndim
Number of dimensions of the decoded data array.

shape
Shape of the decoded data array.

size
Total number of component samples in the decoded data array.

Methods Documentation

classmethod fromdata(*data*, *header=None*, *bps=2*, *edv=None*)
Encode data as payload, using header information.

Parameters
data : `ndarray`

Values to be encoded.

header : `VDIFHeader`, optional

If given, used to infer the encoding, and to verify the number of channels and whether the data are complex.

bps : int, optional

Bits per elementary sample, used if `header` is not given. Default: 2.

edv : int, optional

Should be given if `header` is not given and the payload is encoded as Mark 5 data (i.e., `edv=0xab`).

classmethod fromfile(*fh*, *header*)
Read payload from filehandle and decode it into data.

Parameters
fh : filehandle

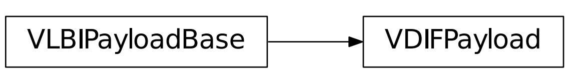
To read data from.

header : `VDIFHeader`

Used to infer the payload size, number of channels, bits per sample, and whether the data are complex.

tofile(*fh*)
Write payload to filehandle.

Class Inheritance Diagram



4.4.4 baseband.vdif.frame Module

Definitions for VLBI VDIF frames and frame sets.

Implements a VDIFFrame class that can be used to hold a header and a payload, providing access to the values encoded in both. Also, define a VDIFFrameSet class that combines a set of frames from different threads.

For the VDIF specification, see http://www.vlbi.org/vdif

Classes

VDIFFrame(header, payload[, valid, verify])	Representation of a VDIF data frame, consisting of a header and payload.
VDIFFrameSet(frames[, header0])	Representation of a set of VDIF frames, combining different threads.

VDIFFrame

class baseband.vdif.frame.**VDIFFrame**(*header*, *payload*, *valid=None*, *verify=True*)

Bases: baseband.vlbi_base.frame.VLBIFrameBase

Representation of a VDIF data frame, consisting of a header and payload.

Parameters

header : VDIFHeader

Wrapper around the encoded header words, providing access to the header information.

payload : VDIFPayload

Wrapper around the payload, provding mechanisms to decode it.

valid : bool or None

Whether the data are valid. If None (default), is inferred from header. Note that header is changed in-place if True or False.

verify : bool

Whether or not to do basic assertions that check the integrity (e.g., that channel information and whether or not data are complex are consistent between header and data). Default: True.

Notes

The Frame can also be instantiated using class methods:

fromfile : read header and payload from a filehandle

fromdata : encode data as payload

Of course, one can also do the opposite:

tofile : method to write header and payload to filehandle

data : property that yields full decoded payload

One can decode part of the payload by indexing or slicing the frame. If the frame does not contain valid data, all values returned are set to self.fill_value.

A number of properties are defined: shape, dtype and size are the shape, type and number of complete samples of the data array, and nbytes the frame size in bytes. Furthermore, the frame acts as a dictionary, with keys those of the header. Any attribute that is not defined on the frame itself, such as .time will be looked up on the header as well.

Attributes Summary

data	Full decoded frame.
dtype	Numeric type of the frame data.
fill_value	Value to replace invalid data in the frame.
nbytes	Size of the encoded frame in bytes.
ndim	Number of dimensions of the frame data.
sample_shape	Shape of a sample in the frame (nchan,).
shape	Shape of the frame data.
size	Total number of component samples in the frame data.
valid	Whether frame contains valid data.

Methods Summary

from_mark5b_frame(mark5b_frame[, verify])	Construct an Mark5B over VDIF frame (EDV=0xab).
fromdata(data[, header, verify])	Construct frame from data and header.
fromfile(fh[, edv, verify])	Read a frame from a filehandle.
keys()	
tofile(fh)	Write encoded frame to filehandle.
verify()	Verify integrity.

Attributes Documentation

data
Full decoded frame.

dtype
Numeric type of the frame data.

fill_value
Value to replace invalid data in the frame.

nbytes
Size of the encoded frame in bytes.

ndim
Number of dimensions of the frame data.

sample_shape
Shape of a sample in the frame (nchan,).

shape
Shape of the frame data.

size
Total number of component samples in the frame data.

valid
Whether frame contains valid data.

This is just the opposite of the `invalid_data` item in the header. If set, that header item is adjusted correspondingly.

Methods Documentation

classmethod from_mark5b_frame(*mark5b_frame*, *verify=True*, ***kwargs*)

Construct an Mark5B over VDIF frame (EDV=0xab).

Any additional keywords can be used to set VDIF header properties not found in the Mark 5B header (such as station).

See http://www.vlbi.org/vdif/docs/vdif_extension_0xab.pdf

classmethod fromdata(*data*, *header=None*, *verify=True*, ***kwargs*)

Construct frame from data and header.

Parameters

data : `ndarray`

Array holding complex or real data to be encoded.

header : `VDIFHeader` or None

If not given, will attempt to generate one using the keywords.

verify : bool, optional

Whether or not to do basic assertions that check the integrity (e.g., that channel information and whether or not data are complex are consistent between header and data). Default: `True`.

****kwargs**

If `header` is not given, these are used to initialize one.

classmethod fromfile(*fh*, *edv=None*, *verify=True*)

Read a frame from a filehandle.

Parameters

fh : filehandle

From which the header and payload are read.

edv : int, False, or None, optional

Extended Data Version. `False` is for legacy headers. If `None` (default), it will be determined from the words themselves.

verify : bool, optional

Whether or not to do basic assertions that check the integrity (e.g., that channel information and whether or not data are complex are consistent between header and data). Default: `True`.

keys()

tofile(*fh*)

Write encoded frame to filehandle.

verify()

Verify integrity.

Checks consistency between the header information and payload data shape and type.

VDIFFrameSet

class baseband.vdif.frame.**VDIFFrameSet**(*frames*, *header0=None*)

Bases: object

Representation of a set of VDIF frames, combining different threads.

Parameters

frames : list of VDIFFrame

Should all cover the same time span.

header0 : VDIFHeader

First header of the frame set. If None (default), is extracted from frames[0].

Notes

The FrameSet can also be read instantiated using class methods:

fromfile : read frames from a filehandle, optionally selecting threads

fromdata : encode data as a set of frames

Of course, one can also do the opposite:

tofile : write frames to filehandle

data : property that yields full decoded frame payloads

One can decode part of the payload by indexing or slicing the frame. If the frame does not contain valid data, all values returned are set to self.fill_value.

A number of properties are defined: shape, dtype and size are the shape, type and number of complete samples of the data array, and nbytes the frame size in bytes. Like a VDIFFrame, the frame set acts as a dictionary, with keys those of the header of the first frame (available via .header0). Any attribute that is not defined on the frame set itself, such as .time will also be looked up on the header.

Attributes Summary

data	Full decoded frame.
dtype	Numeric type of the frameset data.
fill_value	Value to replace invalid data in the frameset.
nbytes	Size of the encoded frame in bytes.
ndim	Number of dimensions of the frameset data.
sample_shape	Shape of a sample in the frameset (nthread, nchan).
shape	Shape of the frameset data.
size	Total number of component samples in the frameset data.
valid	Whether frameset contains valid data.

Methods Summary

fromdata(data[, headers, verify])	Construct a set of frames from data and headers.

Continued on next page

Table 39 – continued from previous page

fromfile(fh[, thread_ids, edv, verify])	Read a frame set from a file, starting at the current location.
keys()	
tofile(fh)	Write all encoded frames to filehandle.

Attributes Documentation

data
Full decoded frame.

dtype
Numeric type of the frameset data.

fill_value
Value to replace invalid data in the frameset.

nbytes
Size of the encoded frame in bytes.

ndim
Number of dimensions of the frameset data.

sample_shape
Shape of a sample in the frameset (nthread, nchan).

shape
Shape of the frameset data.

size
Total number of component samples in the frameset data.

valid
Whether frameset contains valid data.

Methods Documentation

classmethod fromdata(*data*, *headers=None*, *verify=True*, ***kwargs*)
Construct a set of frames from data and headers.

Parameters

data : ndarray

Array holding complex or real data to be encoded. Dimensions should be (samples_per_frame, nthread, nchan).

headers : VDIFHeader, list of same, or None

If a single header, a list with increasing thread_id is generated. If not given, will attempt to generate a header from the keyword arguments.

verify : bool

Whether or not to do basic assertions that check the integrety (e.g., that channel information and whether or not data are complex are consistent between header and data). Default: True.

****kwargs**

If header is not given, these are used to initialize one.

Returns
frameset : `VDIFFrameSet`

`classmethod fromfile`(*fh*, *thread_ids=None*, *edv=None*, *verify=True*)
Read a frame set from a file, starting at the current location.

Parameters
fh : filehandle

Handle to the VDIF file. Should be at the location where the frames are read from.

thread_ids : list or None, optional

The thread ids that should be read. If `None` (default), continue reading threads as long as the frame number does not increase.

edv : int or None, optional

The expected extended data version for the VDIF Header. If `None` (default), use that of the first frame. (Passing it in slightly improves file integrity checking.)

verify : bool, optional

Whether to do (light) sanity checks on the header. Default: `True`.

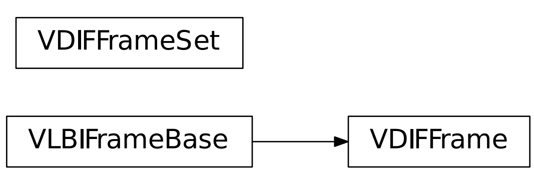
Returns
frameset : `VDIFFrameSet`

Its `frames` property holds a list of frames (in order of either their `thread_id` or following the input `thread_ids` list). Use the `data` attribute to convert to an array.

`keys`()

`tofile`(*fh*)
Write all encoded frames to filehandle.

Class Inheritance Diagram



4.4.5 baseband.vdif.base Module

Functions

`open`(name[, mode])	Open VDIF file for reading or writing.

open

`baseband.vdif.base.open(name, mode=u'rs', **kwargs)`

Open VDIF file for reading or writing.

Opened as a binary file, one gets a wrapped filehandle that adds methods to read/write a frame. Opened as a stream, the handle is wrapped further, with methods such as reading and writing to the file as if it were a stream of samples.

Parameters

name : str or filehandle

File name or handle.

mode : {'rb', 'wb', 'rs', or 'ws'}, optional

Whether to open for reading or writing, and as a regular binary file or as a stream. Default: 'rs', for reading a stream.

****kwargs**

Additional arguments when opening the file as a stream.

— For reading a stream : (see `VDIFStreamReader`)

sample_rate : `Quantity`, optional

Number of complete samples per second, i.e. the rate at which each channel in each thread is sampled. If `None` (default), will be inferred from the header or by scanning one second of the file.

squeeze : bool, optional

If `True` (default), remove any dimensions of length unity from decoded data.

subset : indexing object or tuple of objects, optional

Specific components of the complete sample to decode (after possible squeezing). If a single indexing object is passed, it selects threads. If a tuple is passed, the first selects threads and the second selects channels. If the tuple is empty (default), all components are read.

fill_value : float or complex, optional

Value to use for invalid or missing data. Default: 0.

verify : bool, optional

Whether to do basic checks of frame integrity when reading. The first frameset of the stream is always checked. Default: `True`.

— For writing a stream : (see `VDIFStreamWriter`)

header0 : `VDIFHeader`

Header for the first frame, holding time information, etc. Can instead give keyword arguments to construct a header (see `**kwargs`).

sample_rate : `Quantity`

Number of complete samples per second, i.e. the rate at which each channel in each thread is sampled. For EDV 1 and 3, can alternatively set `sample_rate` within the header.

nthread : int, optional

Number of threads (e.g., 2 for 2 polarisations). Default: 1.

squeeze : bool, optional

If True (default), writer accepts squeezed arrays as input, and adds any dimensions of length unity.

****kwargs**

If the header is not given, an attempt will be made to construct one with any further keyword arguments. See VDIFStreamWriter.

Returns

Filehandle

VDIFFileReader or VDIFFileWriter (binary), or VDIFStreamReader or VDIFStreamWriter (stream).

Classes

VDIFFileReader(fh_raw)	Simple reader for VDIF files.
VDIFFileWriter(fh_raw)	Simple writer for VDIF files.
VDIFStreamBase(fh_raw, header0[, ...])	Base for VDIF streams.
VDIFStreamReader(fh_raw[, sample_rate, ...])	VLBI VDIF format reader.
VDIFStreamWriter(fh_raw[, header0, ...])	VLBI VDIF format writer.

VDIFFileReader

class baseband.vdif.base.**VDIFFileReader**(*fh_raw*)

Bases: baseband.vlbi_base.base.VLBIFileReaderBase

Simple reader for VDIF files.

Wraps a binary filehandle, providing methods to help interpret the data, such as read_frame, read_frameset and get_frame_rate.

Parameters

fh_raw : filehandle

Filehandle of the raw binary data file.

Attributes Summary

info

Methods Summary

close()	
find_header([template_header, frame_nbytes, ...])	Find the nearest header from the current position.
get_frame_rate()	Determine the number of frames per second.
read_frame()	Read a single frame (header plus payload).
read_frameset([thread_ids, edv, verify])	Read a single frame (header plus payload).

Continued on next page

Table 43 – continued from previous page

read_header()	Read a single header from the file.

Attributes Documentation

info

Methods Documentation

close()

find_header(*template_header=None*, *frame_nbytes=None*, *edv=None*, *maximum=None*, *forward=True*)

Find the nearest header from the current position.

Search for a valid header at a given position which is consistent with `template_header` or with a header a frame size ahead. Note that the latter turns out to be an unexpectedly weak check on real data!

If successful, the file pointer is left at the start of the header.

Parameters

template_header : `VDIFHeader`

If given, used to infer the frame size and EDV.

frame_nbytes : int

Frame size in bytes, used if `template_header` is not given.

edv : int

EDV of the header, used if `template_header` is not given.

maximum : int, optional

Maximum number of bytes forward to search through. Default: twice the frame size.

forward : bool, optional

Seek forward if `True` (default), backward if `False`.

Returns

header : `VDIFHeader` or None

Retrieved VDIF header, or `None` if nothing found.

get_frame_rate()

Determine the number of frames per second.

This method first tries to determine the frame rate by looking for the highest frame number in the first second of data. If that fails, it attempts to extract the sample rate from the header.

Returns

frame_rate : `Quantity`

Frames per second.

read_frame()

Read a single frame (header plus payload).

Returns

frame : `VDIFFrame`

With `.header` and `.data` properties that return the `VDIFHeader` and data encoded in the frame, respectively.

read_frameset(*thread_ids=None*, *edv=None*, *verify=True*)

Read a single frame (header plus payload).

Parameters

thread_ids : list, optional

The thread ids that should be read. If `None` (default), read all threads.

edv : int, optional

The expected extended data version for the VDIF Header. If `None`, use that of the first frame. (Passing it in slightly improves file integrity checking.)

verify : bool, optional

Whether to do basic checks of frame integrity. Default: `True`.

Returns

frameset : `VDIFFrameSet`

With `.headers` and `.data` properties that return a list of `VDIFHeaders` and the data encoded in the frame set, respectively.

read_header()

Read a single header from the file.

Returns

header : `VDIFHeader`

VDIFFileWriter

class `baseband.vdif.base.`**VDIFFileWriter**(*fh_raw*)

Bases: `baseband.vlbi_base.base.VLBIFileBase`

Simple writer for VDIF files.

Adds `write_frame` and `write_frameset` methods to the basic VLBI binary file wrapper.

Methods Summary

`close`()	
`write_frame`(data[, header])	Write a single frame (header plus payload).
`write_frameset`(data[, header])	Write a single frame set (headers plus payloads).

Methods Documentation

close()

write_frame(*data*, *header=None*, ***kwargs*)

Write a single frame (header plus payload).

Parameters

data : `ndarray` or `VDIFFrame`

If an array, a `header` should be given, which will be used to get the information needed to encode the array, and to construct the VDIF frame.

header : `VDIFHeader`

Can instead give keyword arguments to construct a header. Ignored if `data` is a `VDIFFrame` instance.

****kwargs**

If `header` is not given, these are used to initialize one.

write_frameset(*data*, *header=None*, ***kwargs*)
Write a single frame set (headers plus payloads).

Parameters

data : `ndarray` or `VDIFFrameSet`

If an array, a header should be given, which will be used to get the information needed to encode the array, and to construct the VDIF frame set.

header : `VDIFHeader`, list of same

Can instead give keyword arguments to construct a header. Ignored if `data` is a `VDIFFrameSet` instance. If a list, should have a length matching the number of threads in `data`; if a single header, `thread_ids` corresponding to the number of threads are generated automatically.

****kwargs**

If `header` is not given, these are used to initialize one.

VDIFStreamBase

class baseband.vdif.base.**VDIFStreamBase**(*fh_raw*, *header0*, *sample_rate=None*, *nthread=1*, *squeeze=True*, *subset=()*, *fill_value=0.0*, *verify=True*)
Bases: `baseband.vlbi_base.base.VLBIStreamBase`

Base for VDIF streams.

Attributes Summary

`bps`	Bits per elementary sample.
`complex_data`	Whether the data are complex.
`header0`	First header of the file.
`sample_rate`	Number of complete samples per second.
`sample_shape`	Shape of a complete sample (possibly subset or squeezed).
`samples_per_frame`	Number of complete samples per frame.
`squeeze`	Whether data arrays have dimensions with length unity removed.
`start_time`	Start time of the file.
`subset`	Specific components of the complete sample to decode.
`time`	Time of the sample pointer's current offset in file.

Continued on next page

Table 45 – continued from previous page

verify	Whether to do consistency checks on frames being read.

Methods Summary

close()	
tell([unit])	Current offset in the file.

Attributes Documentation

bps
Bits per elementary sample.

complex_data
Whether the data are complex.

header0
First header of the file.

sample_rate
Number of complete samples per second.

sample_shape
Shape of a complete sample (possibly subset or squeezed).

samples_per_frame
Number of complete samples per frame.

squeeze
Whether data arrays have dimensions with length unity removed.

If True, data read out has such dimensions removed, and data passed in for writing has them inserted.

start_time
Start time of the file.

See also time for the time of the sample pointer's current offset, and (if available) stop_time for the time at the end of the file.

subset
Specific components of the complete sample to decode.

The order of dimensions is the same as for sample_shape. Set by the class initializer.

time
Time of the sample pointer's current offset in file.

See also start_time for the start time, and (if available) stop_time for the end time, of the file.

verify
Whether to do consistency checks on frames being read.

Methods Documentation

close()

tell(*unit=None*)
Current offset in the file.

Parameters
unit : `Unit` or str, optional

Time unit the offset should be returned in. By default, no unit is used, i.e., an integer enumerating samples is returned. For the special string 'time', the absolute time is calculated.

Returns
offset : int, `Quantity`, or `Time`

Offset in current file (or time at current position).

VDIFStreamReader

class baseband.vdif.base.**VDIFStreamReader**(*fh_raw*, *sample_rate=None*, *squeeze=True*, *subset=()*, *fill_value=0.0*, *verify=True*)
Bases: `baseband.vdif.base.VDIFStreamBase`, `baseband.vlbi_base.base.VLBIStreamReaderBase`

VLBI VDIF format reader.

Allows access to a VDIF file as a continuous series of samples.

Parameters
fh_raw : filehandle

Filehandle of the raw VDIF stream.

sample_rate : `Quantity`, optional

Number of complete samples per second, i.e. the rate at which each channel in each thread is sampled. If `None` (default), will be inferred from the header or by scanning one second of the file.

squeeze : bool, optional

If `True` (default), remove any dimensions of length unity from decoded data.

subset : indexing object or tuple of objects, optional

Specific components of the complete sample to decode (after possible squeezing). If a single indexing object is passed, it selects threads. If a tuple is passed, the first selects threads and the second selects channels. If the tuple is empty (default), all components are read.

fill_value : float or complex, optional

Value to use for invalid or missing data. Default: 0.

verify : bool, optional

Whether to do basic checks of frame integrity when reading. The first frameset of the stream is always checked. Default: `True`.

Attributes Summary

bps	Bits per elementary sample.

Continued on next page

Table 47 – continued from previous page

complex_data	Whether the data are complex.
dtype	
fill_value	Value to use for invalid or missing data.
header0	First header of the file.
info	Standardized information on stream readers.
ndim	Number of dimensions of the (squeezed/subset) stream data.
sample_rate	Number of complete samples per second.
sample_shape	Shape of a complete sample (possibly subset or squeezed).
samples_per_frame	Number of complete samples per frame.
shape	Shape of the (squeezed/subset) stream data.
size	Total number of component samples in the (squeezed/subset) stream data.
squeeze	Whether data arrays have dimensions with length unity removed.
start_time	Start time of the file.
stop_time	Time at the end of the file, just after the last sample.
subset	Specific components of the complete sample to decode.
time	Time of the sample pointer's current offset in file.
verify	Whether to do consistency checks on frames being read.

Methods Summary

close()	
read([count, out])	Read a number of complete (or subset) samples.
seek(offset[, whence])	Change the stream position.
tell([unit])	Current offset in the file.

Attributes Documentation

bps
Bits per elementary sample.

complex_data
Whether the data are complex.

dtype

fill_value
Value to use for invalid or missing data. Default: 0.

header0
First header of the file.

info
Standardized information on stream readers.

The `info` descriptor provides a few standard attributes, all of which can also be accessed directly on the stream filehandle. More detailed information on the underlying file is stored in its info, accessible via info.file_info.

Attributes

start_time	(Time) Time of the first complete sample.
stop_time	(Time) Time of the complete sample just beyond the end of the file.
sample_rate	(Quantity) Complete samples per unit of time.
shape	(tuple) Equivalent shape of the whole file, i.e., combining the number of complete samples and the shape of those samples.
bps	(int) Number of bits used to encode each elementary sample.
complex_data	(bool) Whether the data are complex.

ndim
Number of dimensions of the (squeezed/subset) stream data.

sample_rate
Number of complete samples per second.

sample_shape
Shape of a complete sample (possibly subset or squeezed).

samples_per_frame
Number of complete samples per frame.

shape
Shape of the (squeezed/subset) stream data.

size
Total number of component samples in the (squeezed/subset) stream data.

squeeze
Whether data arrays have dimensions with length unity removed.

If True, data read out has such dimensions removed, and data passed in for writing has them inserted.

start_time
Start time of the file.

See also time for the time of the sample pointer's current offset, and (if available) stop_time for the time at the end of the file.

stop_time
Time at the end of the file, just after the last sample.

See also start_time for the start time of the file, and time for the time of the sample pointer's current offset.

subset
Specific components of the complete sample to decode.

The order of dimensions is the same as for sample_shape. Set by the class initializer.

time
Time of the sample pointer's current offset in file.

See also start_time for the start time, and (if available) stop_time for the end time, of the file.

`verify`
Whether to do consistency checks on frames being read.

Methods Documentation

`close()`

`read(count=None, out=None)`
Read a number of complete (or subset) samples.

The range retrieved can span multiple frames.

Parameters
count : int or None, optional

Number of complete/subset samples to read. If `None` (default) or negative, the whole file is read. Ignored if `out` is given.

out : None or array, optional

Array to store the data in. If given, `count` will be inferred from the first dimension; the other dimension should equal `sample_shape`.

Returns
out : `ndarray` of float or complex

The first dimension is sample-time, and the remainder given by `sample_shape`.

`seek(offset, whence=0)`
Change the stream position.

This works like a normal filehandle seek, but the offset is in samples (or a relative or absolute time).

Parameters
offset : int, `Quantity`, or `Time`

Offset to move to. Can be an (integer) number of samples, an offset in time units, or an absolute time.

whence : {0, 1, 2, 'start', 'current', or 'end'}, optional

Like regular seek, the offset is taken to be from the start if `whence=0` (default), from the current position if 1, and from the end if 2. One can alternativey use 'start', 'current', or 'end' for 0, 1, or 2, respectively. Ignored if `offset` is a time.

`tell(unit=None)`
Current offset in the file.

Parameters
unit : `Unit` or str, optional

Time unit the offset should be returned in. By default, no unit is used, i.e., an integer enumerating samples is returned. For the special string 'time', the absolute time is calculated.

Returns
offset : int, `Quantity`, or `Time`

Offset in current file (or time at current position).

VDIFStreamWriter

class baseband.vdif.base.**VDIFStreamWriter**(*fh_raw*, *header0=None*, *sample_rate=None*, *nthread=1*, *squeeze=True*, ***kwargs*)

Bases: baseband.vdif.base.VDIFStreamBase, baseband.vlbi_base.base.VLBIStreamWriterBase

VLBI VDIF format writer.

Encodes and writes sequences of samples to file.

Parameters

fh_raw : filehandle

Which will write filled sets of frames to storage.

header0 : VDIFHeader

Header for the first frame, holding time information, etc. Can instead give keyword arguments to construct a header (see `**kwargs`).

sample_rate : Quantity

Number of complete samples per second, i.e. the rate at which each channel in each thread is sampled. For EDV 1 and 3, can alternatively set `sample_rate` within the header.

nthread : int, optional

Number of threads (e.g., 2 for 2 polarisations). Default: 1.

squeeze : bool, optional

If True (default), write accepts squeezed arrays as input, and adds any dimensions of length unity.

****kwargs**

If no header is given, an attempt is made to construct one from these. For a standard header, this would include the following.

— Header keywords : (see fromvalues())

time : Time

Start time of the file. Can instead pass on `ref_epoch` and `seconds`.

nchan : int, optional

Number of channels (default: 1). Note: different numbers of channels per thread is not supported.

complex_data : bool, optional

Whether data are complex. Default: False.

bps : int, optional

Bits per elementary sample, i.e. per real or imaginary component for complex data. Default: 1.

samples_per_frame : int

Number of complete samples per frame. Can alternatively use `frame_length`, the number of 8-byte words for header plus payload. For some EDV, this number is fixed (e.g., `frame_length=629` for `edv=3`, which corresponds to 20000 real 2-bit samples per frame).

station : 2 characters, optional

Station ID. Can also be an unsigned 2-byte integer. Default: 0.

edv : {False, 0, 1, 2, 3, 4, 0xab}

Extended Data Version.

Attributes Summary

bps	Bits per elementary sample.
complex_data	Whether the data are complex.
header0	First header of the file.
sample_rate	Number of complete samples per second.
sample_shape	Shape of a complete sample (possibly subset or squeezed).
samples_per_frame	Number of complete samples per frame.
squeeze	Whether data arrays have dimensions with length unity removed.
start_time	Start time of the file.
subset	Specific components of the complete sample to decode.
time	Time of the sample pointer's current offset in file.
verify	Whether to do consistency checks on frames being read.

Methods Summary

close()	
tell([unit])	Current offset in the file.
write(data[, valid])	Write data, buffering by frames as needed.

Attributes Documentation

bps
Bits per elementary sample.

complex_data
Whether the data are complex.

header0
First header of the file.

sample_rate
Number of complete samples per second.

sample_shape
Shape of a complete sample (possibly subset or squeezed).

samples_per_frame
Number of complete samples per frame.

squeeze
Whether data arrays have dimensions with length unity removed.

If True, data read out has such dimensions removed, and data passed in for writing has them inserted.

start_time
Start time of the file.

See also time for the time of the sample pointer's current offset, and (if available) stop_time for the time at the end of the file.

subset
Specific components of the complete sample to decode.

The order of dimensions is the same as for sample_shape. Set by the class initializer.

time
Time of the sample pointer's current offset in file.

See also start_time for the start time, and (if available) stop_time for the end time, of the file.

verify
Whether to do consistency checks on frames being read.

Methods Documentation

close()

tell(*unit=None*)
Current offset in the file.

Parameters
unit : Unit or str, optional

Time unit the offset should be returned in. By default, no unit is used, i.e., an integer enumerating samples is returned. For the special string 'time', the absolute time is calculated.

Returns
offset : int, Quantity, or Time

Offset in current file (or time at current position).

write(*data*, *valid=True*)
Write data, buffering by frames as needed.

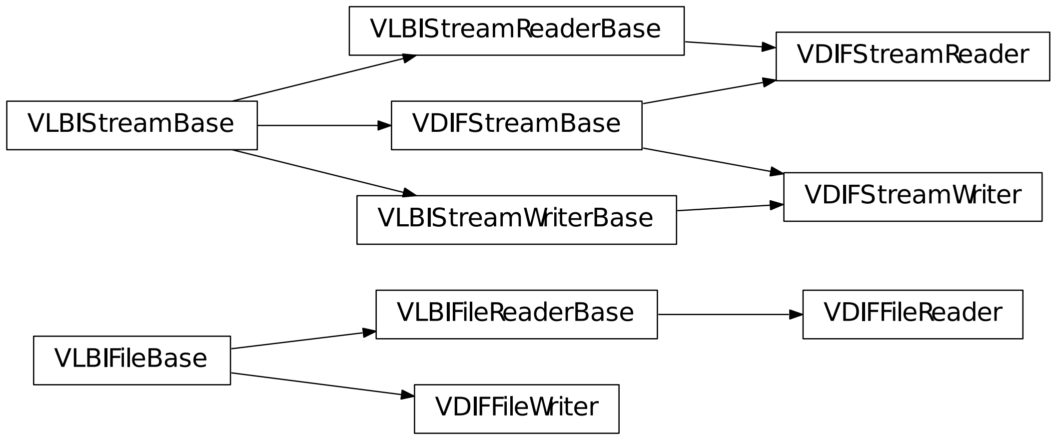
Parameters
data : ndarray

Piece of data to be written, with sample dimensions as given by sample_shape. This should be properly scaled to make best use of the dynamic range delivered by the encoding.

valid : bool, optional

Whether the current data are valid. Default: True.

Class Inheritance Diagram



CHAPTER 5

MARK 5B

The Mark 5B format is the output format of the Mark 5B disk-based VLBI data system. It is described in its design specifications.

5.1 File Structure

Each *data frame* consists of a *header* consisting of four 32-bit words (16 bytes) followed by a *payload* of 2500 32-bit words (10000 bytes). The header contains a sync word, frame number, and timestamp (accurate to 1 ms), as well as user-specified data; see Sec. 1 of the design specifications for details. The payload supports 2^n bit streams, for $0 \leq n \leq 5$, and the first sample of each stream corresponds precisely to the header time. *elementary samples* may be 1 or 2 bits in size, with the latter being stored in two successive bit streams. The number of *channels* is equal to the number of bit-streams divided by the number of bits per elementary sample (Baseband currently only supports files where all bit-streams are active). Files begin at a header (unlike for Mark 4), and an integer number of frames fit within 1 second.

The Mark 5B system also outputs files with the active bit-stream mask, number of frames per second, and observational metadata (Sec. 1.3 of the design specifications). Baseband does not yet use these files, and instead requires the user specify, for example, the *sample rate*.

5.2 Usage

This section covers reading and writing Mark 5B files with Baseband; general usage can be found under the *Getting Started* section. For situations in which one is unsure of a file's format, Baseband features the general `baseband.open` and `baseband.file_info` functions, which are also discussed in *Getting Started*. The examples below use the small sample file `baseband/data/sample.m5b`, and the `numpy`, `astropy.units`, `astropy.time.Time`, and `baseband.mark5b` modules:

```
>>> import numpy as np
>>> import astropy.units as u
>>> from astropy.time import Time
```

(continues on next page)

(continued from previous page)

```
>>> from baseband import mark5b
>>> from baseband.data import SAMPLE_MARK5B
```

Opening a Mark 5B file with `open` in binary mode provides a normal file reader extended with methods to read a `Mark5BFrame`. The number of channels, kiloday (thousands of MJD) and number of bits per sample must all be passed when using `read_frame`:

```
>>> fb = mark5b.open(SAMPLE_MARK5B, 'rb', kday=56000, nchan=8)
>>> frame = fb.read_frame()
>>> frame.shape
(5000, 8)
>>> fb.close()
```

Our sample file has 2-bit *component* samples, which is also the default for `read_frame`, so it does not need to be passed. Also, we may pass a reference `Time` object within 500 days of the observation start time to `ref_time`, rather than `kday`.

Opening as a stream wraps the low-level routines such that reading and writing is in units of samples. It also provides access to header information. Here, we also must provide `nchan`, `sample_rate`, and `ref_time` or `kday`:

```
>>> fh = mark5b.open(SAMPLE_MARK5B, 'rs', sample_rate=32*u.MHz, nchan=8,
...                  ref_time=Time('2014-06-13 12:00:00'))
>>> fh
<Mark5BStreamReader name=... offset=0
    sample_rate=32.0 MHz, samples_per_frame=5000,
    sample_shape=SampleShape(nchan=8), bps=2,
    start_time=2014-06-13T05:30:01.000000000>
>>> header0 = fh.header0    # To be used for writing, below.
>>> d = fh.read(10000)
>>> d.shape
(10000, 8)
>>> d[0, :3]
array([-3.316505, -1.       ,  1.       ], dtype=float32)
>>> fh.close()
```

When writing to file, we again need to pass in `sample_rate` and `nchan`, though time can either be passed explicitly or inferred from the header:

```
>>> fw = mark5b.open('test.m5b', 'ws', header0=header0,
...                  sample_rate=32*u.MHz, nchan=8)
>>> fw.write(d)
>>> fw.close()
>>> fh = mark5b.open('test.m5b', 'rs', sample_rate=32*u.MHz,
...                  kday=57000, nchan=8)
>>> np.all(fh.read() == d)
True
>>> fh.close()
```

5.3 Reference/API

5.3.1 baseband.mark5b Package

Mark5B VLBI data reader.

Code inspired by Walter Brisken's mark5access. See https://github.com/demorest/mark5access.

Also, for the Mark5B design, see http://www.haystack.mit.edu/tech/vlbi/mark5/mark5_memos/019.pdf

Functions

`open`(name[, mode])	Open Mark5B file for reading or writing.

open

`baseband.mark5b.open`(*name*, *mode=u'rs'*, ***kwargs*)

Open Mark5B file for reading or writing.

Opened as a binary file, one gets a wrapped filehandle that adds methods to read/write a frame. Opened as a stream, the handle is wrapped further, with methods such as reading and writing to the file as if it were a stream of samples.

Parameters

name : str or filehandle

File name or handle.

mode : {'rb', 'wb', 'rs', or 'ws'}, optional

Whether to open for reading or writing, and as a regular binary file or as a stream. Default: 'rs', for reading a stream.

****kwargs**

Additional arguments when opening the file as a stream.

— For reading a stream : (see `Mark5BStreamReader`)

sample_rate : `Quantity`, optional

Number of complete samples per second, i.e. the rate at which each channel is sampled. If `None` (default), will be inferred from scanning one second of the file or, failing that, using the time difference between two consecutive frames.

kday : int or None

Explicit thousands of MJD of the observation start time (eg. `57000` for MJD 57999), used to infer the full MJD from the header's time information. Can instead pass an approximate `ref_time`.

ref_time : `Time` or None

Reference time within 500 days of the observation start time, used to infer the full MJD. Only used if `kday` is not given.

nchan : int, optional

Number of channels. Default: 1.

bps : int, optional

Bits per elementary sample. Default: 2.

squeeze : bool, optional

If `True` (default), remove any dimensions of length unity from decoded data.

subset : indexing object, optional

Specific channels of the complete sample to decode (after possible squeezing). If an empty tuple (default), all channels are read.

fill_value : float or complex

Value to use for invalid or missing data. Default: 0.

verify : bool, optional

Whether to do basic checks of frame integrity when reading. The first frame of the stream is always checked. Default: `True`.

— For writing a stream : (see `Mark5BStreamWriter`)

header0 : `Mark5BHeader`

Header for the first frame, holding time information, etc. Can instead give keyword arguments to construct a header (see `**kwargs`).

sample_rate : `Quantity`

Number of complete samples per second, i.e. the rate at which each channel is sampled. Needed to calculate header timestamps.

nchan : int, optional

Number of channels. Default: 1.

bps : int, optional

Bits per elementary sample. Default: 2.

squeeze : bool, optional

If `True` (default), writer accepts squeezed arrays as input, and adds channel and thread dimensions if they have length unity.

****kwargs**

If no header is given, an attempt is made to construct one with any further keyword arguments. See `Mark5BStreamWriter`.

Returns

Filehandle

`Mark5BFileReader` or `Mark5BFileWriter` (binary), or `Mark5BStreamReader` or `Mark5BStreamWriter` (stream).

Classes

`Mark5BFrame`(header, payload[, valid, verify])	Representation of a Mark 5B frame, consisting of a header and payload.
`Mark5BHeader`(words[, kday, ref_time, verify])	Decoder/encoder of a Mark5B Frame Header.
`Mark5BPayload`(words[, nchan, bps, complex_data])	Container for decoding and encoding VDIF payloads.

Mark5BFrame

class baseband.mark5b.**Mark5BFrame**(*header*, *payload*, *valid=None*, *verify=True*)

Bases: `baseband.vlbi_base.frame.VLBIFrameBase`

Representation of a Mark 5B frame, consisting of a header and payload.

Parameters

header : Mark5BHeader

Wrapper around the encoded header words, providing access to the header information.

payload : Mark5BPayload

Wrapper around the payload, provding mechanisms to decode it.

valid : bool or None

Whether the data are valid. If None (default), the validity will be determined by checking whether the payload consists of the fill pattern 0x11223344.

verify : bool

Whether to do basic verification of integrity (default: True)

Notes

The Frame can also be read instantiated using class methods:

fromfile : read header and payload from a filehandle

fromdata : encode data as payload

Of course, one can also do the opposite:

tofile : method to write header and payload to filehandle

data : property that yields full decoded payload

A number of properties are defined: shape, dtype and size are the shape, type and number of complete samples of the data array, and nbytes the frame size in bytes. Furthermore, the frame acts as a dictionary, with keys those of the header. Any attribute that is not defined on the frame itself, such as .time will be looked up on the header as well.

Attributes Summary

data	Full decoded frame.
dtype	Numeric type of the frame data.
fill_value	Value to replace invalid data in the frame.
nbytes	Size of the encoded frame in bytes.
ndim	Number of dimensions of the frame data.
sample_shape	Shape of a sample in the frame (nchan,).
shape	Shape of the frame data.
size	Total number of component samples in the frame data.
valid	Whether frame contains valid data.

Methods Summary

fromdata(data[, header, bps, valid, verify])	Construct frame from data and header.
fromfile(fh[, kday, ref_time, nchan, bps, . . .])	Read a frame from a filehandle.
keys()	

Continued on next page

Table 4 – continued from previous page

tofile(fh)	Write encoded frame to filehandle.
verify()	Simple verification.

Attributes Documentation

data
Full decoded frame.

dtype
Numeric type of the frame data.

fill_value
Value to replace invalid data in the frame.

nbytes
Size of the encoded frame in bytes.

ndim
Number of dimensions of the frame data.

sample_shape
Shape of a sample in the frame (nchan,).

shape
Shape of the frame data.

size
Total number of component samples in the frame data.

valid
Whether frame contains valid data.

Methods Documentation

classmethod fromdata(*data*, *header=None*, *bps=2*, *valid=True*, *verify=True*, ***kwargs*)
Construct frame from data and header.

Parameters

data : ndarray

Array holding data to be encoded.

header : Mark5BHeader or None

If not given, will attempt to generate one using the keywords.

bps : int

Bits per elementary sample. Default: 2.

valid : bool

Whether the data are valid (default: True). If not, the payload will be set to a fill pattern.

verify : bool

Whether to do basic checks of frame integrity (default: True).

classmethod fromfile(*fh*, *kday=None*, *ref_time=None*, *nchan=1*, *bps=3*, *valid=None*, *verify=True*)
Read a frame from a filehandle.

Parameters

fh : filehandle

To read the header and payload from.

kday : int or None

Explicit thousands of MJD of the observation time. Can instead pass an approximate `ref_time`.

ref_time : `Time` or None

Reference time within 500 days of the observation time, used to infer the full MJD. Used only if `kday` is not given.

nchan : int, optional

Number of channels. Default: 1.

bps : int, optional

Bits per elementary sample. Default: 2.

verify : bool

Whether to do basic checks of frame integrity (default: `True`).

keys()

tofile(*fh*)

Write encoded frame to filehandle.

verify()

Simple verification. To be added to by subclasses.

Mark5BHeader

class baseband.mark5b.**Mark5BHeader**(*words*, *kday=None*, *ref_time=None*, *verify=True*, ***kwargs*)

Bases: `baseband.vlbi_base.header.VLBIHeaderBase`

Decoder/encoder of a Mark5B Frame Header.

See page 15 of http://www.haystack.edu/tech/vlbi/mark5/docs/Mark%205B%20users%20manual.pdf

Parameters

words : tuple of int, or None

Four 32-bit unsigned int header words. If `None`, set to a tuple of zeros for later initialisation.

kday : int or None

Explicit thousands of MJD of the observation time (needed to remove ambiguity in the Mark 5B time stamp). Can instead pass an approximate `ref_time`.

ref_time : `Time` or None

Reference time within 500 days of the observation time, used to infer the full MJD. Used only if `kday` is not given.

verify : bool, optional

Whether to do basic verification of integrity. Default: `True`.

Returns
header : Mark5BHeader

Attributes Summary

fraction	Fractional seconds (decoded from 'bcd_fraction').
frame_nbytes	Size of the frame in bytes.
jday	Last three digits of MJD (decoded from 'bcd_jday').
kday	
mutable	Whether the header can be modified.
nbytes	Size of the header in bytes.
payload_nbytes	Size of the payload in bytes.
seconds	Integer seconds on day (decoded from 'bcd_seconds').
time	Convert year, BCD time code to Time object.

Methods Summary

copy(**kwargs)	Create a mutable and independent copy of the header.
fromfile(fh, *args, **kwargs)	Read VLBI Header from file.
fromkeys(*args, **kwargs)	Initialise a header from parsed values.
fromvalues(**kwargs)	Initialise a header from parsed values.
get_time([frame_rate])	Convert year, BCD time code to Time object.
infer_kday(ref_time)	Uses a reference time to set a header's kday.
keys()	
set_time(time[, frame_rate])	Convert Time object to BCD timestamp elements and 'frame_nr'.
tofile(fh)	Write VLBI frame header to filehandle.
update(**kwargs)	Update the header by setting keywords or properties.
verify()	Verify header integrity.

Attributes Documentation

fraction
Fractional seconds (decoded from 'bcd_fraction').

The fraction is stored to 0.1 ms accuracy. Following mark5access, this is "unrounded" to give the exact time of the start of the frame for any total bit rate below 512 Mbps. For rates above this value, it is no longer guaranteed that subsequent frames have unique rates.

Note to the above: since a Mark5B frame contains 80000 bits, the total bit rate for which times can be unique would in principle be 800 Mbps. However, standard VLBI only uses bit rates that are powers of 2 in MHz.

frame_nbytes
Size of the frame in bytes.

jday
Last three digits of MJD (decoded from 'bcd_jday').

kday = None

mutable
Whether the header can be modified.

nbytes
Size of the header in bytes.

payload_nbytes
Size of the payload in bytes.

seconds
Integer seconds on day (decoded from 'bcd_seconds').

time
Convert year, BCD time code to Time object.

Calculate time using `jday`, `seconds`, and `fraction` properties (which reflect the bcd-encoded 'bcd_jday', 'bcd_seconds' and 'bcd_fraction' header items), plus `kday` from the initialisation. See http://www.haystack.edu/tech/vlbi/mark5/docs/Mark%205B%20users%20manual.pdf

Note that some non-compliant files do not have 'bcd_fraction' set. For those, the time can still be calculated using the header's 'frame_nr' by passing in a frame rate.

Furthermore, fractional seconds are stored only to 0.1 ms accuracy. In the code, this is "unrounded" to give the exact time of the start of the frame for any total bit rate below 512 Mbps. For higher rates, it is no longer guaranteed that subsequent frames have unique `fraction`, and one should pass in an explicit frame rate instead.

Parameters
frame_rate : `Quantity`, optional

Used to calculate the fractional second from the frame number instead of from the header's `fraction`.

Returns
`Time`

Methods Documentation

copy(***kwargs*)
Create a mutable and independent copy of the header.

Keyword arguments can be passed on as needed by possible subclasses.

classmethod fromfile(*fh*, **args*, ***kwargs*)
Read VLBI Header from file.

Arguments are the same as for class initialisation. The header constructed will be immutable.

classmethod fromkeys(**args*, ***kwargs*)
Initialise a header from parsed values.

Like fromvalues, but without any interpretation of keywords.

Raises
KeyError : if not all keys required are present in `kwargs`

classmethod fromvalues(***kwargs*)
Initialise a header from parsed values.

Here, the parsed values must be given as keyword arguments, i.e., for any `header = cls(<data>)`, `cls.fromvalues(**header) == header`.

However, unlike for the `Mark5BHeader.fromkeys()` class method, data can also be set using arguments named after methods, such as `jday` and `seconds`.

Given defaults:

sync_pattern : 0xABADDEED

Values set by other keyword arguments (if present):

bcd_jday : from `jday` or `time` bcd_seconds : from `seconds` or `time` bcd_fraction : from `fraction` or `time` (may need `frame_rate`) frame_nr : from `time` (may need `frame_rate`)

get_time(*frame_rate=None*)

Convert year, BCD time code to Time object.

Calculate time using `jday`, `seconds`, and `fraction` properties (which reflect the bcd-encoded 'bcd_jday', 'bcd_seconds' and 'bcd_fraction' header items), plus `kday` from the initialisation. See http://www.haystack.edu/tech/vlbi/mark5/docs/Mark%205B%20users%20manual.pdf

Note that some non-compliant files do not have 'bcd_fraction' set. For those, the time can still be calculated using the header's 'frame_nr' by passing in a frame rate.

Furthermore, fractional seconds are stored only to 0.1 ms accuracy. In the code, this is "unrounded" to give the exact time of the start of the frame for any total bit rate below 512 Mbps. For higher rates, it is no longer guaranteed that subsequent frames have unique `fraction`, and one should pass in an explicit frame rate instead.

Parameters

frame_rate : `Quantity`, optional

Used to calculate the fractional second from the frame number instead of from the header's `fraction`.

Returns

`Time`

infer_kday(*ref_time*)

Uses a reference time to set a header's `kday`.

Parameters

ref_time : `Time`

Reference time within 500 days of the observation time.

keys()

set_time(*time*, *frame_rate=None*)

Convert Time object to BCD timestamp elements and 'frame_nr'.

For non-integer seconds, the frame number will be calculated if not given explicitly. Doing so requires the frame rate.

Parameters

time : `Time`

The time to use for this header.

frame_rate : `Quantity`, optional

For calculating 'frame_nr' from the fractional seconds.

tofile(*fh*)

Write VLBI frame header to filehandle.

update(***kwargs*)
Update the header by setting keywords or properties.

Here, any keywords matching header keys are applied first, and any remaining ones are used to set header properties, in the order set by the class (in `_properties`).

Parameters
crc : int or None, optional

If `None` (default), recalculate the CRC after updating.

verify : bool, optional

If `True` (default), verify integrity after updating.

****kwargs**

Arguments used to set keywords and properties.

verify()
Verify header integrity.

Mark5BPayload

class baseband.mark5b.**Mark5BPayload**(*words*, *nchan=1*, *bps=2*, *complex_data=False*)
Bases: `baseband.vlbi_base.payload.VLBIPayloadBase`

Container for decoding and encoding VDIF payloads.

Parameters
words : `ndarray`

Array containg LSB unsigned words (with the right size) that encode the payload.

nchan : int, optional

Number of channels. Default: 1.

bps : int, optional

Bits per elementary sample. Default: 2.

Attributes Summary

`data`	Full decoded payload.
`dtype`	Numeric type of the decoded data array.
`nbytes`	Size of the payload in bytes.
`ndim`	Number of dimensions of the decoded data array.
`shape`	Shape of the decoded data array.
`size`	Total number of component samples in the decoded data array.

Methods Summary

`fromdata`(data[, bps])	Encode data as payload, using a given number of bits per sample.

Continued on next page

Table 8 – continued from previous page

fromfile(fh, *args, **kwargs)	Read payload from filehandle and decode it into data.
tofile(fh)	Write payload to filehandle.

Attributes Documentation

data
Full decoded payload.

dtype
Numeric type of the decoded data array.

nbytes
Size of the payload in bytes.

ndim
Number of dimensions of the decoded data array.

shape
Shape of the decoded data array.

size
Total number of component samples in the decoded data array.

Methods Documentation

classmethod fromdata(*data*, *bps=2*)
Encode data as payload, using a given number of bits per sample.

It is assumed that the last dimension is the number of channels.

classmethod fromfile(*fh*, **args*, ***kwargs*)
Read payload from filehandle and decode it into data.

Parameters
fh : filehandle

From which data is read.

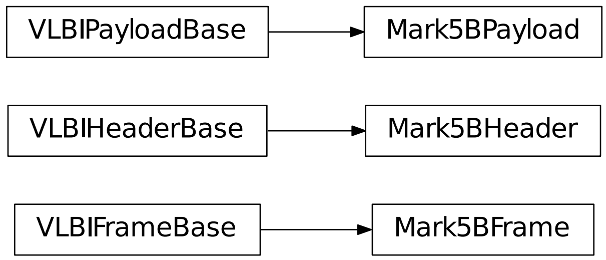
payload_nbytes : int

Number of bytes to read (default: as given in `cls._nbytes`).

Any other (keyword) arguments are passed on to the class initialiser.

tofile(*fh*)
Write payload to filehandle.

Class Inheritance Diagram



5.3.2 baseband.mark5b.header Module

Definitions for VLBI Mark5B Headers.

Implements a Mark5BHeader class used to store header words, and decode/encode the information therein.

For the specification, see http://www.haystack.edu/tech/vlbi/mark5/docs/Mark%205B%20users%20manual.pdf

Classes

`Mark5BHeader`(words[, kday, ref_time, verify])	Decoder/encoder of a Mark5B Frame Header.

Mark5BHeader

class `baseband.mark5b.header.`**`Mark5BHeader`**(*words*, *kday=None*, *ref_time=None*, *verify=True*, ***kwargs*)

Bases: `baseband.vlbi_base.header.VLBIHeaderBase`

Decoder/encoder of a Mark5B Frame Header.

See page 15 of http://www.haystack.edu/tech/vlbi/mark5/docs/Mark%205B%20users%20manual.pdf

Parameters

words : tuple of int, or None

Four 32-bit unsigned int header words. If `None`, set to a tuple of zeros for later initialisation.

kday : int or None

Explicit thousands of MJD of the observation time (needed to remove ambiguity in the Mark 5B time stamp). Can instead pass an approximate `ref_time`.

ref_time : `Time` or None

Reference time within 500 days of the observation time, used to infer the full MJD. Used only if `kday` is not given.

verify : bool, optional

Whether to do basic verification of integrity. Default: `True`.

Returns

header : `Mark5BHeader`

Attributes Summary

`fraction`	Fractional seconds (decoded from 'bcd_fraction').
`frame_nbytes`	Size of the frame in bytes.
`jday`	Last three digits of MJD (decoded from 'bcd_jday').
`kday`	
`mutable`	Whether the header can be modified.
`nbytes`	Size of the header in bytes.
`payload_nbytes`	Size of the payload in bytes.
`seconds`	Integer seconds on day (decoded from 'bcd_seconds').
`time`	Convert year, BCD time code to Time object.

Methods Summary

`copy`(**kwargs)	Create a mutable and independent copy of the header.
`fromfile`(fh, *args, **kwargs)	Read VLBI Header from file.
`fromkeys`(*args, **kwargs)	Initialise a header from parsed values.
`fromvalues`(**kwargs)	Initialise a header from parsed values.
`get_time`([frame_rate])	Convert year, BCD time code to Time object.
`infer_kday`(ref_time)	Uses a reference time to set a header's `kday`.
`keys`()	
`set_time`(time[, frame_rate])	Convert Time object to BCD timestamp elements and 'frame_nr'.
`tofile`(fh)	Write VLBI frame header to filehandle.
`update`(**kwargs)	Update the header by setting keywords or properties.
`verify`()	Verify header integrity.

Attributes Documentation

fraction

Fractional seconds (decoded from 'bcd_fraction').

The fraction is stored to 0.1 ms accuracy. Following mark5access, this is "unrounded" to give the exact time of the start of the frame for any total bit rate below 512 Mbps. For rates above this value, it is no longer guaranteed that subsequent frames have unique rates.

Note to the above: since a Mark5B frame contains 80000 bits, the total bit rate for which times can be unique would in principle be 800 Mbps. However, standard VLBI only uses bit rates that are powers of 2 in MHz.

frame_nbytes

Size of the frame in bytes.

jday

Last three digits of MJD (decoded from 'bcd_jday').

`kday = None`

`mutable`
Whether the header can be modified.

`nbytes`
Size of the header in bytes.

`payload_nbytes`
Size of the payload in bytes.

`seconds`
Integer seconds on day (decoded from 'bcd_seconds').

`time`
Convert year, BCD time code to Time object.

Calculate time using `jday`, `seconds`, and `fraction` properties (which reflect the bcd-encoded 'bcd_jday', 'bcd_seconds' and 'bcd_fraction' header items), plus `kday` from the initialisation. See http://www.haystack.edu/tech/vlbi/mark5/docs/Mark%205B%20users%20manual.pdf

Note that some non-compliant files do not have 'bcd_fraction' set. For those, the time can still be calculated using the header's 'frame_nr' by passing in a frame rate.

Furthermore, fractional seconds are stored only to 0.1 ms accuracy. In the code, this is "unrounded" to give the exact time of the start of the frame for any total bit rate below 512 Mbps. For higher rates, it is no longer guaranteed that subsequent frames have unique `fraction`, and one should pass in an explicit frame rate instead.

Parameters
frame_rate : `Quantity`, optional
Used to calculate the fractional second from the frame number instead of from the header's `fraction`.

Returns
`Time`

Methods Documentation

`copy`(***kwargs*)
Create a mutable and independent copy of the header.

Keyword arguments can be passed on as needed by possible subclasses.

`classmethod fromfile`(*fh*, **args*, ***kwargs*)
Read VLBI Header from file.

Arguments are the same as for class initialisation. The header constructed will be immutable.

`classmethod fromkeys`(**args*, ***kwargs*)
Initialise a header from parsed values.

Like fromvalues, but without any interpretation of keywords.

Raises
KeyError : if not all keys required are present in `kwargs`

`classmethod fromvalues`(***kwargs*)
Initialise a header from parsed values.

Here, the parsed values must be given as keyword arguments, i.e., for any `header = cls(<data>)`, `cls.fromvalues(**header) == header`.

However, unlike for the `Mark5BHeader.fromkeys()` class method, data can also be set using arguments named after methods, such as `jday` and `seconds`.

Given defaults:

sync_pattern : 0xABADDEED

Values set by other keyword arguments (if present):

bcd_jday : from `jday` or `time` bcd_seconds : from `seconds` or `time` bcd_fraction : from `fraction` or `time` (may need `frame_rate`) frame_nr : from `time` (may need `frame_rate`)

get_time(*frame_rate=None*)

Convert year, BCD time code to Time object.

Calculate time using `jday`, `seconds`, and `fraction` properties (which reflect the bcd-encoded 'bcd_jday', 'bcd_seconds' and 'bcd_fraction' header items), plus `kday` from the initialisation. See http://www.haystack.edu/tech/vlbi/mark5/docs/Mark%205B%20users%20manual.pdf

Note that some non-compliant files do not have 'bcd_fraction' set. For those, the time can still be calculated using the header's 'frame_nr' by passing in a frame rate.

Furthermore, fractional seconds are stored only to 0.1 ms accuracy. In the code, this is "unrounded" to give the exact time of the start of the frame for any total bit rate below 512 Mbps. For higher rates, it is no longer guaranteed that subsequent frames have unique `fraction`, and one should pass in an explicit frame rate instead.

Parameters

frame_rate : `Quantity`, optional

Used to calculate the fractional second from the frame number instead of from the header's `fraction`.

Returns

`Time`

infer_kday(*ref_time*)

Uses a reference time to set a header's `kday`.

Parameters

ref_time : `Time`

Reference time within 500 days of the observation time.

keys()

set_time(*time*, *frame_rate=None*)

Convert Time object to BCD timestamp elements and 'frame_nr'.

For non-integer seconds, the frame number will be calculated if not given explicitly. Doing so requires the frame rate.

Parameters

time : `Time`

The time to use for this header.

frame_rate : `Quantity`, optional

For calculating 'frame_nr' from the fractional seconds.

tofile(*fh*)
Write VLBI frame header to filehandle.

update(***kwargs*)
Update the header by setting keywords or properties.

Here, any keywords matching header keys are applied first, and any remaining ones are used to set header properties, in the order set by the class (in `_properties`).

Parameters
crc : int or None, optional

If `None` (default), recalculate the CRC after updating.

verify : bool, optional

If `True` (default), verify integrity after updating.

****kwargs**

Arguments used to set keywords and properties.

verify()
Verify header integrity.

Variables

`CRC16`	CRC polynomial used for Mark 5B Headers, as a check on the time code.
`crc16`	Cyclic Redundancy Check for a bitstream.

CRC16

baseband.mark5b.header.**CRC16 = 98309**
CRC polynomial used for Mark 5B Headers, as a check on the time code.

x^16 + x^15 + x^2 + 1, i.e., 0x18005. See page 11 of http://www.haystack.mit.edu/tech/vlbi/mark5/docs/230.3.pdf (defined there for VLBA headers).

This is also CRC-16-IBM mentioned in https://en.wikipedia.org/wiki/Cyclic_redundancy_check

crc16

baseband.mark5b.header.**crc16 = <baseband.vlbi_base.utils.CRC object>**
Cyclic Redundancy Check for a bitstream.

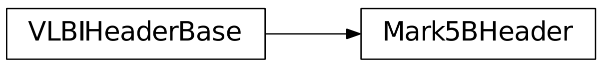
See https://en.wikipedia.org/wiki/Cyclic_redundancy_check

Once initialised, the instance can be used as a function that calculates the CRC, or one can use the check method to check that the CRC at the end of a stream is correct.

Parameters
polynomial : int

Binary encoded CRC divisor. For instance, that used by Mark 4 headers is 0x180f, or x^12 + x^11 + x^3 + x^2 + x + 1.

Class Inheritance Diagram



5.3.3 baseband.mark5b.payload Module

Definitions for VLBI Mark 5B payloads.

Implements a Mark5BPayload class used to store payload words, and decode to or encode from a data array.

For the specification, see http://www.haystack.edu/tech/vlbi/mark5/docs/Mark%205B%20users%20manual.pdf

Functions

`init_luts`()	Set up the look-up tables for levels as a function of input byte.
`decode_2bit`(words)	
`encode_2bit`(values)	Generic encoder for data stored using two bits.

init_luts

`baseband.mark5b.payload.`**`init_luts`**`()`

Set up the look-up tables for levels as a function of input byte.

For 1-bit mode, one has just the sign bit:

s	value
0	-1
1	+1

For 2-bit mode, there is a sign and a magnitude, which encode:

m	s	value	s*2+m
0	0	-Hi	0
0	1	+1	2
1	0	-1	1
1	1	+Hi	3

See Table 13 in https://science.nrao.edu/facilities/vlba/publications/memos/upgrade/sensimemo13.pdf and http://www.haystack.edu/tech/vlbi/mark5/docs/Mark%205B%20users%20manual.pdf Appendix A: sign always on even bit stream (0, 2, 4, . . .), and magnitude on adjacent odd stream (1, 3, 5, . . .).

In the above table, the last column is the index in the linearly increasing table of levels (decoder_levels[2]).

decode_2bit

baseband.mark5b.payload.**decode_2bit**(*words*)

encode_2bit

baseband.mark5b.payload.**encode_2bit**(*values*)

Generic encoder for data stored using two bits.

This returns an unsigned integer array containing encoded sample values that range from 0 to 3. The conversion from floating point sample value to unsigned int is given below, with `lv = TWO_BIT_1_SIGMA = 2.1745`:

Input range	Output
value < -lv	0
-lv < value < 0.	2
0. < value < lv	1
lv < value	3

This does not pack the samples into bytes.

Classes

Mark5BPayload(words[, nchan, bps, complex_data])	Container for decoding and encoding VDIF payloads.

Mark5BPayload

class baseband.mark5b.payload.**Mark5BPayload**(*words*, *nchan=1*, *bps=2*, *complex_data=False*)

Bases: baseband.vlbi_base.payload.VLBIPayloadBase

Container for decoding and encoding VDIF payloads.

Parameters

words : ndarray

Array containing LSB unsigned words (with the right size) that encode the payload.

nchan : int, optional

Number of channels. Default: 1.

bps : int, optional

Bits per elementary sample. Default: 2.

Attributes Summary

data	Full decoded payload.
dtype	Numeric type of the decoded data array.
nbytes	Size of the payload in bytes.
ndim	Number of dimensions of the decoded data array.
shape	Shape of the decoded data array.
size	Total number of component samples in the decoded data array.

Methods Summary

fromdata(data[, bps])	Encode data as payload, using a given number of bits per sample.
fromfile(fh, *args, **kwargs)	Read payload from filehandle and decode it into data.
tofile(fh)	Write payload to filehandle.

Attributes Documentation

data
Full decoded payload.

dtype
Numeric type of the decoded data array.

nbytes
Size of the payload in bytes.

ndim
Number of dimensions of the decoded data array.

shape
Shape of the decoded data array.

size
Total number of component samples in the decoded data array.

Methods Documentation

classmethod fromdata(*data*, *bps=2*)
Encode data as payload, using a given number of bits per sample.

It is assumed that the last dimension is the number of channels.

classmethod fromfile(*fh*, **args*, ***kwargs*)
Read payload from filehandle and decode it into data.

Parameters

fh : filehandle

From which data is read.

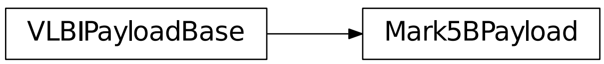
payload_nbytes : int

Number of bytes to read (default: as given in `cls._nbytes`).

Any other (keyword) arguments are passed on to the class initialiser.

tofile(*fh*)
Write payload to filehandle.

Class Inheritance Diagram



5.3.4 baseband.mark5b.frame Module

Definitions for VLBI Mark 5B frames.

Implements a Mark5BFrame class that can be used to hold a header and a payload, providing access to the values encoded in both.

For the specification, see http://www.haystack.edu/tech/vlbi/mark5/docs/Mark%205B%20users%20manual.pdf

Classes

`Mark5BFrame`(header, payload[, valid, verify])	Representation of a Mark 5B frame, consisting of a header and payload.

Mark5BFrame

class `baseband.mark5b.frame.`**`Mark5BFrame`**(*header*, *payload*, *valid=None*, *verify=True*)
Bases: `baseband.vlbi_base.frame.VLBIFrameBase`

Representation of a Mark 5B frame, consisting of a header and payload.

Parameters

header : `Mark5BHeader`

Wrapper around the encoded header words, providing access to the header information.

payload : `Mark5BPayload`

Wrapper around the payload, provding mechanisms to decode it.

valid : bool or None

Whether the data are valid. If `None` (default), the validity will be determined by checking whether the payload consists of the fill pattern 0x11223344.

verify : bool

Whether to do basic verification of integrity (default: `True`)

Notes

The Frame can also be read instantiated using class methods:

fromfile : read header and payload from a filehandle

fromdata : encode data as payload

Of course, one can also do the opposite:

tofile : method to write header and payload to filehandle

data : property that yields full decoded payload

A number of properties are defined: `shape`, `dtype` and `size` are the shape, type and number of complete samples of the data array, and `nbytes` the frame size in bytes. Furthermore, the frame acts as a dictionary, with keys those of the header. Any attribute that is not defined on the frame itself, such as `.time` will be looked up on the header as well.

Attributes Summary

`data`	Full decoded frame.
`dtype`	Numeric type of the frame data.
`fill_value`	Value to replace invalid data in the frame.
`nbytes`	Size of the encoded frame in bytes.
`ndim`	Number of dimensions of the frame data.
`sample_shape`	Shape of a sample in the frame (nchan,).
`shape`	Shape of the frame data.
`size`	Total number of component samples in the frame data.
`valid`	Whether frame contains valid data.

Methods Summary

`fromdata`(data[, header, bps, valid, verify])	Construct frame from data and header.
`fromfile`(fh[, kday, ref_time, nchan, bps, . . .])	Read a frame from a filehandle.
`keys`()	
`tofile`(fh)	Write encoded frame to filehandle.
`verify`()	Simple verification.

Attributes Documentation

data
Full decoded frame.

dtype
Numeric type of the frame data.

fill_value
Value to replace invalid data in the frame.

nbytes
Size of the encoded frame in bytes.

ndim
Number of dimensions of the frame data.

sample_shape
Shape of a sample in the frame (nchan,).

shape
Shape of the frame data.

size
Total number of component samples in the frame data.

valid
Whether frame contains valid data.

Methods Documentation

classmethod fromdata(*data*, *header=None*, *bps=2*, *valid=True*, *verify=True*, ***kwargs*)
Construct frame from data and header.

Parameters

data : `ndarray`

Array holding data to be encoded.

header : `Mark5BHeader` or None

If not given, will attempt to generate one using the keywords.

bps : int

Bits per elementary sample. Default: 2.

valid : bool

Whether the data are valid (default: `True`). If not, the payload will be set to a fill pattern.

verify : bool

Whether to do basic checks of frame integrity (default: `True`).

classmethod fromfile(*fh*, *kday=None*, *ref_time=None*, *nchan=1*, *bps=3*, *valid=None*, *verify=True*)
Read a frame from a filehandle.

Parameters

fh : filehandle

To read the header and payload from.

kday : int or None

Explicit thousands of MJD of the observation time. Can instead pass an approximate `ref_time`.

ref_time : `Time` or None

Reference time within 500 days of the observation time, used to infer the full MJD. Used only if `kday` is not given.

nchan : int, optional

Number of channels. Default: 1.

bps : int, optional

Bits per elementary sample. Default: 2.

verify : bool

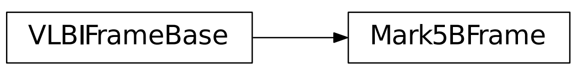
Whether to do basic checks of frame integrity (default: `True`).

keys()

tofile(*fh*)
Write encoded frame to filehandle.

verify()
Simple verification. To be added to by subclasses.

Class Inheritance Diagram



5.3.5 baseband.mark5b.base Module

Functions

`open`(name[, mode])	Open Mark5B file for reading or writing.

open

`baseband.mark5b.base.`**open**(*name*, *mode=u'rs'*, ***kwargs*)
Open Mark5B file for reading or writing.

Opened as a binary file, one gets a wrapped filehandle that adds methods to read/write a frame. Opened as a stream, the handle is wrapped further, with methods such as reading and writing to the file as if it were a stream of samples.

Parameters

name : str or filehandle

File name or handle.

mode : {'rb', 'wb', 'rs', or 'ws'}, optional

Whether to open for reading or writing, and as a regular binary file or as a stream. Default: 'rs', for reading a stream.

****kwargs**

Additional arguments when opening the file as a stream.

— For reading a stream : (see `Mark5BStreamReader`)

sample_rate : `Quantity`, optional

Number of complete samples per second, i.e. the rate at which each channel is sampled. If `None` (default), will be inferred from scanning one second of the file or, failing that, using the time difference between two consecutive frames.

kday : int or None

Explicit thousands of MJD of the observation start time (eg. `57000` for MJD 57999), used to infer the full MJD from the header's time information. Can instead pass an approximate `ref_time`.

ref_time : `Time` or None

Reference time within 500 days of the observation start time, used to infer the full MJD. Only used if `kday` is not given.

nchan : int, optional

Number of channels. Default: 1.

bps : int, optional

Bits per elementary sample. Default: 2.

squeeze : bool, optional

If `True` (default), remove any dimensions of length unity from decoded data.

subset : indexing object, optional

Specific channels of the complete sample to decode (after possible squeezing). If an empty tuple (default), all channels are read.

fill_value : float or complex

Value to use for invalid or missing data. Default: 0.

verify : bool, optional

Whether to do basic checks of frame integrity when reading. The first frame of the stream is always checked. Default: `True`.

— For writing a stream : (see `Mark5BStreamWriter`)

header0 : `Mark5BHeader`

Header for the first frame, holding time information, etc. Can instead give keyword arguments to construct a header (see `**kwargs`).

sample_rate : `Quantity`

Number of complete samples per second, i.e. the rate at which each channel is sampled. Needed to calculate header timestamps.

nchan : int, optional

Number of channels. Default: 1.

bps : int, optional

Bits per elementary sample. Default: 2.

squeeze : bool, optional

If `True` (default), writer accepts squeezed arrays as input, and adds channel and thread dimensions if they have length unity.

****kwargs**

If no header is given, an attempt is made to construct one with any further keyword arguments. See `Mark5BStreamWriter`.

Returns

Filehandle

`Mark5BFileReader` or `Mark5BFileWriter` (binary), or `Mark5BStreamReader` or `Mark5BStreamWriter` (stream).

Classes

`Mark5BFileReader`(fh_raw[, kday, ref_time, ...])	Simple reader for Mark 5B files.
`Mark5BFileWriter`(fh_raw)	Simple writer for Mark 5B files.
`Mark5BStreamReader`(fh_raw[, sample_rate, ...])	VLBI Mark 5B format reader.
`Mark5BStreamWriter`(fh_raw[, header0, ...])	VLBI Mark 5B format writer.

Mark5BFileReader

class baseband.mark5b.base.**Mark5BFileReader**(*fh_raw*, *kday=None*, *ref_time=None*, *nchan=None*, *bps=2*)

Bases: `baseband.vlbi_base.base.VLBIFileReaderBase`

Simple reader for Mark 5B files.

Wraps a binary filehandle, providing methods to help interpret the data, such as `read_frame` and `get_frame_rate`.

Parameters

fh_raw : filehandle

Filehandle of the raw binary data file.

kday : int or None

Explicit thousands of MJD of the observation time. Can instead pass an approximate `ref_time`.

ref_time : `Time` or None

Reference time within 500 days of the observation time, used to infer the full MJD. Used only if kday is not given.

nchan : int, optional

Number of channels. Default: 1.

bps : int, optional

Bits per elementary sample. Default: 2.

Attributes Summary

`info`

Methods Summary

close()	
find_header([forward, maximum])	Find the nearest header from the current position.
get_frame_rate()	Determine the number of frames per second.
read_frame([verify])	Read a single frame (header plus payload).
read_header()	Read a single header from the file.

Attributes Documentation

info

Methods Documentation

close()

find_header(*forward=True*, *maximum=None*)

Find the nearest header from the current position.

If successful, the file pointer is left at the start of the header.

Parameters

forward : bool, optional

Seek forward if `True` (default), backward if `False`.

maximum : int, optional

Maximum number of bytes to search through. Default: twice the frame size of 10016 bytes.

Returns

header : `Mark5BHeader` or None

Retrieved Mark 5B header, or `None` if nothing found.

get_frame_rate()

Determine the number of frames per second.

This method first tries to determine the frame rate by looking for the highest frame number in the first second of data. If that fails, it uses the time difference between two consecutive frames. This can fail if the headers do not store fractional seconds, or if the data rate is above 512 Mbps.

Returns

frame_rate : `Quantity`

Frames per second.

read_frame(*verify=True*)

Read a single frame (header plus payload).

Returns

frame : `Mark5BFrame`

With `header` and `data` properties that return the `Mark5BHeader` and data encoded in the frame, respectively.

verify : bool, optional

Whether to do basic checks of frame integrity. Default: `True`.

read_header()

Read a single header from the file.

Returns

header : `Mark5BHeader`

Mark5BFileWriter

class baseband.mark5b.base.**Mark5BFileWriter**(*fh_raw*)

Bases: `baseband.vlbi_base.base.VLBIFileBase`

Simple writer for Mark 5B files.

Adds `write_frame` method to the VLBI binary file wrapper.

Methods Summary

`close`()	
`write_frame`(data[, header, bps, valid])	Write a single frame (header plus payload).

Methods Documentation

close()

write_frame(*data*, *header=None*, *bps=2*, *valid=True*, ***kwargs*)

Write a single frame (header plus payload).

Parameters

data : `ndarray` or :`Mark5BFrame`

If an array, `header` should be given, which will be used to get the information needed to encode the array, and to construct the Mark 5B frame.

header : `Mark5BHeader`

Can instead give keyword arguments to construct a header. Ignored if `data` is a `Mark5BFrame` instance.

bps : int, optional

Bits per elementary sample, to use when encoding the payload. Ignored if `data` is a `Mark5BFrame` instance. Default: 2.

valid : bool, optional

Whether the data are valid; if `False`, a payload filled with an appropriate pattern will be crated. Ignored if `data` is a `Mark5BFrame` instance. Default: `True`.

****kwargs**

If `header` is not given, these are used to initialize one.

Mark5BStreamReader

class baseband.mark5b.base.**Mark5BStreamReader**(*fh_raw*, *sample_rate=None*, *kday=None*, *ref_time=None*, *nchan=None*, *bps=2*, *squeeze=True*, *subset=()*, *fill_value=0.0*, *verify=True*)

Bases: baseband.mark5b.base.Mark5BStreamBase, baseband.vlbi_base.base.VLBIStreamReaderBase

VLBI Mark 5B format reader.

Allows access a Mark 5B file as a continues series of samples.

Parameters

fh_raw : filehandle

Filehandle of the raw Mark 5B stream.

sample_rate : Quantity, optional

Number of complete samples per second, i.e. the rate at which each channel is sampled. If None (default), will be inferred from scanning one second of the file or, failing that, using the time difference between two consecutive frames.

kday : int or None

Explicit thousands of MJD of the observation start time (eg. 57000 for MJD 57999), used to infer the full MJD from the header's time information. Can instead pass an approximate ref_time.

ref_time : Time or None

Reference time within 500 days of the observation start time, used to infer the full MJD. Only used if kday is not given.

nchan : int

Number of channels. Needs to be explicitly passed in.

bps : int, optional

Bits per elementary sample. Default: 2.

squeeze : bool, optional

If True (default), remove any dimensions of length unity from decoded data.

subset : indexing object, optional

Specific channels of the complete sample to decode (after possible squeezing). If an empty tuple (default), all channels are read.

fill_value : float or complex

Value to use for invalid or missing data. Default: 0.

verify : bool, optional

Whether to do basic checks of frame integrity when reading. The first frame of the stream is always checked. Default: True.

Attributes Summary

bps	Bits per elementary sample.

Continued on next page

Table 25 – continued from previous page

complex_data	Whether the data are complex.
dtype	
fill_value	Value to use for invalid or missing data.
header0	First header of the file.
info	Standardized information on stream readers.
ndim	Number of dimensions of the (squeezed/subset) stream data.
sample_rate	Number of complete samples per second.
sample_shape	Shape of a complete sample (possibly subset or squeezed).
samples_per_frame	Number of complete samples per frame.
shape	Shape of the (squeezed/subset) stream data.
size	Total number of component samples in the (squeezed/subset) stream data.
squeeze	Whether data arrays have dimensions with length unity removed.
start_time	Start time of the file.
stop_time	Time at the end of the file, just after the last sample.
subset	Specific components of the complete sample to decode.
time	Time of the sample pointer's current offset in file.
verify	Whether to do consistency checks on frames being read.

Methods Summary

close()	
read([count, out])	Read a number of complete (or subset) samples.
seek(offset[, whence])	Change the stream position.
tell([unit])	Current offset in the file.

Attributes Documentation

bps
Bits per elementary sample.

complex_data
Whether the data are complex.

dtype

fill_value
Value to use for invalid or missing data. Default: 0.

header0
First header of the file.

info
Standardized information on stream readers.

The `info` descriptor provides a few standard attributes, all of which can also be accessed directly on the stream filehandle. More detailed information on the underlying file is stored in its info, accessible via info.file_info.

Attributes

start_time	(Time) Time of the first complete sample.
stop_time	(Time) Time of the complete sample just beyond the end of the file.
sample_rate	(Quantity) Complete samples per unit of time.
shape	(tuple) Equivalent shape of the whole file, i.e., combining the number of complete samples and the shape of those samples.
bps	(int) Number of bits used to encode each elementary sample.
complex_data	(bool) Whether the data are complex.

ndim
Number of dimensions of the (squeezed/subset) stream data.

sample_rate
Number of complete samples per second.

sample_shape
Shape of a complete sample (possibly subset or squeezed).

samples_per_frame
Number of complete samples per frame.

shape
Shape of the (squeezed/subset) stream data.

size
Total number of component samples in the (squeezed/subset) stream data.

squeeze
Whether data arrays have dimensions with length unity removed.

If True, data read out has such dimensions removed, and data passed in for writing has them inserted.

start_time
Start time of the file.

See also time for the time of the sample pointer's current offset, and (if available) stop_time for the time at the end of the file.

stop_time
Time at the end of the file, just after the last sample.

See also start_time for the start time of the file, and time for the time of the sample pointer's current offset.

subset
Specific components of the complete sample to decode.

The order of dimensions is the same as for sample_shape. Set by the class initializer.

time
Time of the sample pointer's current offset in file.

See also start_time for the start time, and (if available) stop_time for the end time, of the file.

verify
Whether to do consistency checks on frames being read.

Methods Documentation

close()

read(*count=None*, *out=None*)
Read a number of complete (or subset) samples.

The range retrieved can span multiple frames.

Parameters
count : int or None, optional

Number of complete/subset samples to read. If `None` (default) or negative, the whole file is read. Ignored if `out` is given.

out : None or array, optional

Array to store the data in. If given, `count` will be inferred from the first dimension; the other dimension should equal `sample_shape`.

Returns
out : `ndarray` of float or complex

The first dimension is sample-time, and the remainder given by `sample_shape`.

seek(*offset*, *whence=0*)
Change the stream position.

This works like a normal filehandle seek, but the offset is in samples (or a relative or absolute time).

Parameters
offset : int, `Quantity`, or `Time`

Offset to move to. Can be an (integer) number of samples, an offset in time units, or an absolute time.

whence : {0, 1, 2, 'start', 'current', or 'end'}, optional

Like regular seek, the offset is taken to be from the start if `whence=0` (default), from the current position if 1, and from the end if 2. One can alternativey use 'start', 'current', or 'end' for 0, 1, or 2, respectively. Ignored if `offset` is a time.

tell(*unit=None*)
Current offset in the file.

Parameters
unit : `Unit` or str, optional

Time unit the offset should be returned in. By default, no unit is used, i.e., an integer enumerating samples is returned. For the special string 'time', the absolute time is calculated.

Returns
offset : int, `Quantity`, or `Time`

Offset in current file (or time at current position).

Mark5BStreamWriter

class baseband.mark5b.base.**Mark5BStreamWriter**(*fh_raw*, *header0=None*, *sample_rate=None*, *nchan=1*, *bps=2*, *squeeze=True*, ***kwargs*)

Bases: baseband.mark5b.base.Mark5BStreamBase, baseband.vlbi_base.base.VLBIStreamWriterBase

VLBI Mark 5B format writer.

Encodes and writes sequences of samples to file.

Parameters

fh_raw : filehandle

For writing filled sets of frames to storage.

header0 : Mark5BHeader

Header for the first frame, holding time information, etc. Can instead give keyword arguments to construct a header (see `**kwargs`).

sample_rate : Quantity

Number of complete samples per second, i.e. the rate at which each channel is sampled. Needed to calculate header timestamps.

nchan : int, optional

Number of channels. Default: 1.

bps : int, optional

Bits per elementary sample. Default: 2.

squeeze : bool, optional

If True (default), write accepts squeezed arrays as input, and adds any dimensions of length unity.

****kwargs**

If no header is given, an attempt is made to construct one from these. For a standard header, the following suffices.

— Header kwargs : (see fromvalues())

time : Time

Start time of the file. Sets bcd-encoded unit day, hour, minute, second, and fraction, as well as the frame number, in the header.

Attributes Summary

bps	Bits per elementary sample.
complex_data	Whether the data are complex.
header0	First header of the file.
sample_rate	Number of complete samples per second.
sample_shape	Shape of a complete sample (possibly subset or squeezed).
samples_per_frame	Number of complete samples per frame.

Continued on next page

Table 27 – continued from previous page

squeeze	Whether data arrays have dimensions with length unity removed.
start_time	Start time of the file.
subset	Specific components of the complete sample to decode.
time	Time of the sample pointer's current offset in file.
verify	Whether to do consistency checks on frames being read.

Methods Summary

close()	
tell([unit])	Current offset in the file.
write(data[, valid])	Write data, buffering by frames as needed.

Attributes Documentation

bps
Bits per elementary sample.

complex_data
Whether the data are complex.

header0
First header of the file.

sample_rate
Number of complete samples per second.

sample_shape
Shape of a complete sample (possibly subset or squeezed).

samples_per_frame
Number of complete samples per frame.

squeeze
Whether data arrays have dimensions with length unity removed.

If `True`, data read out has such dimensions removed, and data passed in for writing has them inserted.

start_time
Start time of the file.

See also `time` for the time of the sample pointer's current offset, and (if available) `stop_time` for the time at the end of the file.

subset
Specific components of the complete sample to decode.

The order of dimensions is the same as for `sample_shape`. Set by the class initializer.

time
Time of the sample pointer's current offset in file.

See also `start_time` for the start time, and (if available) `stop_time` for the end time, of the file.

verify
Whether to do consistency checks on frames being read.

Methods Documentation

close()

tell(*unit=None*)

Current offset in the file.

Parameters

unit : `Unit` or str, optional

Time unit the offset should be returned in. By default, no unit is used, i.e., an integer enumerating samples is returned. For the special string 'time', the absolute time is calculated.

Returns

offset : int, `Quantity`, or `Time`

Offset in current file (or time at current position).

write(*data*, *valid=True*)

Write data, buffering by frames as needed.

Parameters

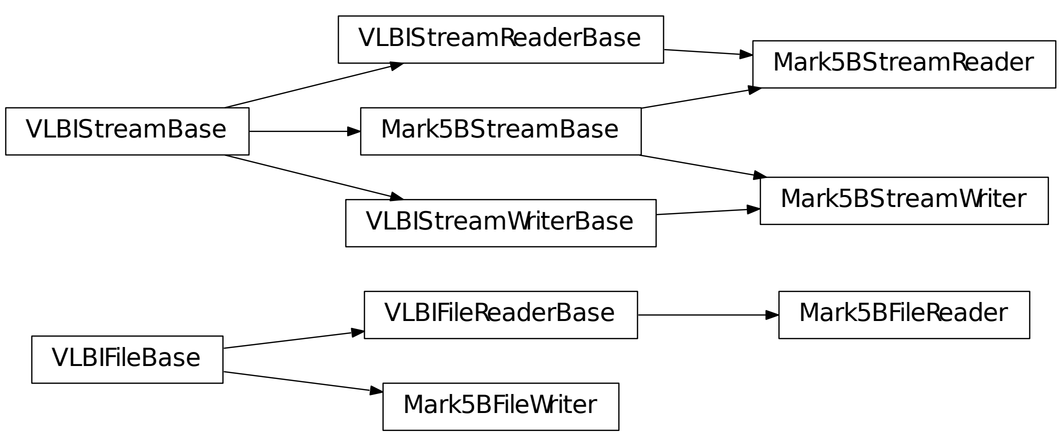
data : `ndarray`

Piece of data to be written, with sample dimensions as given by `sample_shape`. This should be properly scaled to make best use of the dynamic range delivered by the encoding.

valid : bool, optional

Whether the current data are valid. Default: `True`.

Class Inheritance Diagram



CHAPTER 6

MARK 4

The Mark 4 format is the output format of the MIT Haystack Observatory's Mark 4 VLBI magnetic tape-based data acquisition system, and one output format of its successor, the Mark 5A hard drive-based system. The format's specification is in the Mark IIIA/IV/VLBA design specifications.

Baseband currently only supports files that have been parity-stripped and corrected for barrel roll and data modulation.

6.1 File Structure

Mark 4 files contain up to 64 concurrent data "tracks". Tracks are divided into 22500-bit "tape frames", each of which consists of a 160-bit *header* followed by a 19840-bit *payload*. The header includes a timestamp (accurate to 1.25 ms), track ID, sideband, and fan-out/in factor (see below); the details of these can be found in 2.1.1 - 2.1.3 in the design specifications. The payload consists of a 1-bit *stream*. When recording 2-bit *elementary samples*, the data is split into two tracks, with one carrying the sign bit, and the other the magnitude bit.

The header takes the place of the first 160 bits of payload data, so that the first sample occurs `fanout * 160` sample times after the header time. This means that a Mark 4 stream is not contiguous in time. The length of one frame ranges from 1.25 ms to 160 ms in octave steps (which ensures an integer number of frames falls within 1 minute), setting the maximum sample rate per track to 18 megabits/track/s.

Data from a single *channel* may be distributed to multiple tracks - "fan-out" - or multiple channels fed to one track - "fan-in". Fan-out is used when sampling at rates higher than 18 megabits/track/s. Baseband currently only supports tracks using fan-out ("longitudinal data format").

Baseband reconstructs the tracks into channels (reconstituting 2-bit data from two tracks into a single channel if necessary) and combines tape frame headers into a single *data frame* header.

6.2 Usage

This section covers reading and writing Mark 4 files with Baseband; general usage can be found under the *Getting Started* section. For situations in which one is unsure of a file's format, Baseband features the general `baseband.open` and `baseband.file_info` functions, which are also discussed in *Getting Started*. The examples below use the small sample file baseband/data/sample.m4, and the numpy, astropy.units, astropy.time.Time, and baseband.mark4 modules:

```
>>> import numpy as np
>>> import astropy.units as u
>>> from astropy.time import Time
>>> from baseband import mark4
>>> from baseband.data import SAMPLE_MARK4
```

Opening a Mark 4 file with open in binary mode provides a normal file reader but extended with methods to read a Mark4Frame. Mark 4 files generally **do not start (or end) at a frame boundary**, so in binary mode one has to seek the first frame using locate_frame (which will also determine the number of Mark 4 tracks, if not given explicitly). Since Mark 4 files do not store the full time information, one must pass either the the decade the data was taken, or an equivalent reference Time object:

```
>>> fb = mark4.open(SAMPLE_MARK4, 'rb', decade=2010)
>>> fb.locate_frame()  # Locate first frame.
2696
>>> frame = fb.read_frame()
>>> frame.shape
(80000, 8)
>>> fb.close()
```

Opening in stream mode automatically seeks for the first frame, and wraps the low-level routines such that reading and writing is in units of samples. It also provides access to header information. Here we pass a reference Time object within 4 years of the observation start time to ref_time, rather than a decade:

```
>>> fh = mark4.open(SAMPLE_MARK4, 'rs', ref_time=Time('2013:100:23:00:00'))
>>> fh
<Mark4StreamReader name=... offset=0
    sample_rate=32.0 MHz, samples_per_frame=80000,
    sample_shape=SampleShape(nchan=8), bps=2,
    start_time=2014-06-16T07:38:12.47500>
>>> d = fh.read(6400)
>>> d.shape
(6400, 8)
>>> d[635:645, 0].astype(int)  # first channel
array([ 0,  0,  0,  0,  0, -1,  1,  3,  1, -1])
>>> fh.close()
```

As mentioned in the *File Structure* section, because the header takes the place of the first 160 samples of each track, the first payload sample occurs fanout * 160 sample times after the header time. The stream reader includes these overwritten samples as invalid data (zeros, by default):

```
>>> np.array_equal(d[:640], np.zeros((640,) + d.shape[1:]))
True
```

When writing to file, we need to pass in the sample rate in addition to decade. The number of tracks can be inferred from the header:

```
>>> fw = mark4.open('sample_mark4_segment.m4', 'ws', header0=frame.header,
...                 sample_rate=32*u.MHz, decade=2010)
>>> fw.write(frame.data)
>>> fw.close()
>>> fh = mark4.open('sample_mark4_segment.m4', 'rs',
...                 sample_rate=32.*u.MHz, decade=2010)
>>> np.all(fh.read(80000) == frame.data)
```

(continues on next page)

(continued from previous page)

```
True
>>> fh.close()
```

Note that above we had to pass in the sample rate even when opening the file for reading; this is because there is only a single frame in the file, and hence the sample rate cannot be inferred automatically.

6.3 Reference/API

6.3.1 baseband.mark4 Package

Mark 4 VLBI data reader.

Code inspired by Walter Brisken's mark5access. See https://github.com/demorest/mark5access.

The format itself is described in detail in http://www.haystack.mit.edu/tech/vlbi/mark5/docs/230.3.pdf

Functions

`open`(name[, mode])	Open Mark4 file for reading or writing.

open

`baseband.mark4.`**`open`**(*name*, *mode=u'rs'*, ***kwargs*)

Open Mark4 file for reading or writing.

Opened as a binary file, one gets a wrapped filehandle that adds methods to read/write a frame. Opened as a stream, the handle is wrapped further, with methods such as reading and writing to the file as if it were a stream of samples.

Parameters

name : str or filehandle

File name or handle.

mode : {'rb', 'wb', 'rs', or 'ws'}, optional

Whether to open for reading or writing, and as a regular binary file or as a stream. Default: 'rs', for reading a stream.

****kwargs**

Additional arguments when opening the file as a stream.

— For reading a stream : (see `Mark4StreamReader`)

sample_rate : `Quantity`, optional

Number of complete samples per second, i.e. the rate at which each channel is sampled. If not given, will be inferred from scanning two frames of the file.

ntrack : int, optional

Number of Mark 4 bitstreams. If `None` (default), will attempt to automatically detect it by scanning the file.

decade : int or None

Decade of the observation start time (eg. `2010` for 2018), needed to remove ambiguity in the Mark 4 time stamp (default: `None`). Can instead pass an approximate `ref_time`.

ref_time : `Time` or None

Reference time within 4 years of the start time of the observations. Used only if `decade` is not given.

squeeze : bool, optional

If `True` (default), remove any dimensions of length unity from decoded data.

subset : indexing object, optional

Specific channels of the complete sample to decode (after possible squeezing). If an empty tuple (default), all channels are read.

fill_value : float or complex, optional

Value to use for invalid or missing data. Default: 0.

verify : bool, optional

Whether to do basic checks of frame integrity when reading. The first frame of the stream is always checked. Default: `True`.

— For writing a stream : (see `Mark4StreamWriter`)

header0 : `Mark4Header`

Header for the first frame, holding time information, etc. Can instead give keyword arguments to construct a header (see `**kwargs`).

sample_rate : `Quantity`

Number of complete samples per second, i.e. the rate at which each channel is sampled. Needed to calculate header timestamps.

squeeze : bool, optional

If `True` (default), writer accepts squeezed arrays as input, and adds any dimensions of length unity.

****kwargs**

If the header is not given, an attempt will be made to construct one with any further keyword arguments. See `Mark4StreamWriter`.

Returns

Filehandle

`Mark4FileReader` or `Mark4FileWriter` (binary), or `Mark4StreamReader` or `Mark4StreamWriter` (stream)

Classes

`Mark4Frame`(header, payload[, valid, verify])	Representation of a Mark 4 frame, consisting of a header and payload.
`Mark4Header`(words[, ntrack, decade, . . .])	Decoder/encoder of a Mark 4 Header, containing all streams.
`Mark4Payload`(words[, header, nchan, bps, fanout])	Container for decoding and encoding Mark 4 payloads.

Mark4Frame

class baseband.mark4.**Mark4Frame**(*header*, *payload*, *valid=None*, *verify=True*)
Bases: baseband.vlbi_base.frame.VLBIFrameBase

Representation of a Mark 4 frame, consisting of a header and payload.

Parameters

header : Mark4Header

Wrapper around the encoded header words, providing access to the header information.

payload : Mark4Payload

Wrapper around the payload, provding mechanisms to decode it.

valid : bool or None, optional

Whether the data are valid. If None (default), inferred from header. Note that header is updated in-place if True or False.

verify : bool, optional

Whether or not to do basic assertions that check the integrity (e.g., that channel information and number of tracks are consistent between header and data). Default: True.

Notes

The Frame can also be read instantiated using class methods:

fromfile : read header and payload from a filehandle

fromdata : encode data as payload

Of course, one can also do the opposite:

tofile : method to write header and payload to filehandle

data : property that yields full decoded payload

One can decode part of the payload by indexing or slicing the frame. If the frame does not contain valid data, all values returned are set to self.fill_value.

A number of properties are defined: shape, dtype and size are the shape, type and number of complete samples of the data array, and nbytes the frame size in bytes. Furthermore, the frame acts as a dictionary, with keys those of the header. Any attribute that is not defined on the frame itself, such as .time will be looked up on the header as well.

Attributes Summary

data	Full decoded frame, with header part filled in.
dtype	Numeric type of the frame data.
fill_value	Value to replace invalid data in the frame.
nbytes	Size of the encoded frame in bytes.
ndim	Number of dimensions of the frame data.
sample_shape	Shape of a sample in the frame (nchan,).
shape	Shape of the frame data.

Continued on next page

Table 3 – continued from previous page

size	Total number of component samples in the frame data.
valid	Whether frame contains valid data.

Methods Summary

fromdata(data[, header, verify])	Construct frame from data and header.
fromfile(fh, ntrack[, decade, ref_time, verify])	Read a frame from a filehandle.
keys()	
tofile(fh)	Write encoded frame to filehandle.
verify()	Simple verification.

Attributes Documentation

data
Full decoded frame, with header part filled in.

dtype
Numeric type of the frame data.

fill_value
Value to replace invalid data in the frame.

nbytes
Size of the encoded frame in bytes.

ndim
Number of dimensions of the frame data.

sample_shape
Shape of a sample in the frame (nchan,).

shape
Shape of the frame data.

size
Total number of component samples in the frame data.

valid
Whether frame contains valid data.

None of the error flags are set.

Methods Documentation

classmethod fromdata(*data*, *header=None*, *verify=True*, ***kwargs*)
Construct frame from data and header.

Parameters

data : ndarray

Array holding complex or real data to be encoded. This should have the full size of a data frame, even though the part covered by the header will be ignored.

header : Mark4Header or None

If not given, will attempt to generate one using the keywords.

verify : bool, optional

Whether to do basic checks of frame integrity (default: `True`).

classmethod fromfile(*fh*, *ntrack*, *decade=None*, *ref_time=None*, *verify=True*)

Read a frame from a filehandle.

Parameters

fh : filehandle

To read header from.

ntrack : int

Number of Mark 4 bitstreams.

decade : int or None

Decade in which the observations were taken. Can instead pass an approximate `ref_time`.

ref_time : `Time` or None

Reference time within 4 years of the observation time. Used only if `decade` is not given.

verify : bool, optional

Whether to do basic verification of integrity. Default: `True`.

keys()

tofile(*fh*)

Write encoded frame to filehandle.

verify()

Simple verification. To be added to by subclasses.

Mark4Header

class baseband.mark4.**Mark4Header**(*words*, *ntrack=None*, *decade=None*, *ref_time=None*, *verify=True*)

Bases: `baseband.mark4.header.Mark4TrackHeader`

Decoder/encoder of a Mark 4 Header, containing all streams.

See http://www.haystack.mit.edu/tech/vlbi/mark5/docs/230.3.pdf

Parameters

words : `ndarray` of int, or None

Shape should be (5, number-of-tracks), and dtype np.uint32. If `None`, `ntrack` should be given and words will be initialized to 0.

ntrack : None or int

Number of Mark 4 bitstreams, to help initialize `words` if needed.

decade : int or None

Decade in which the observations were taken (needed to remove ambiguity in the Mark 4 time stamp). Can instead pass an approximate `ref_time`.

ref_time : `Time` or None

Reference time within 4 years of the observation time, used to infer the full Mark 4 timestamp. Used only if decade is not given.

verify : bool, optional

Whether to do basic verification of integrity. Default: `True`.

Returns

header : `Mark4Header`

Attributes Summary

`bps`	Bits per elementary sample (either 1 or 2).
`converters`	Converted ID and sideband used for each channel.
`decade`	
`fanout`	Number of samples stored in one payload item of size ntrack.
`fraction`	Fractional seconds (decoded from 'bcd_fraction').
`frame_nbytes`	Size of the frame in bytes.
`mutable`	Whether the header can be modified.
`nbytes`	Size of the header in bytes.
`nchan`	Number of channels (`ntrack * fanout`) in the frame.
`nsb`	Number of side bands used.
`ntrack`	Number of Mark 4 bitstreams.
`payload_nbytes`	Size of the payload in bytes.
`samples_per_frame`	Number of complete samples in the frame.
`stream_dtype`	Stream dtype required to hold this header's number of tracks.
`time`	Convert BCD time code to Time object for all tracks.
`track_assignment`	Assignments of tracks to channels and fanout items.
`track_id`	

Methods Summary

`copy`(**kwargs)	Create a mutable and independent copy of the header.
`fromfile`(fh, ntrack[, decade, ref_time, verify])	Read Mark 4 header from file.
`fromkeys`(*args, **kwargs)	Initialise a header from parsed values.
`fromvalues`(ntrack[, decade, ref_time])	Initialise a header from parsed values.
`get_time`()	Convert BCD time code to Time object for all tracks.
`infer_decade`(ref_time)	Uses a reference time to set a header's decade.
`keys`()	
`set_time`(time)	
`tofile`(fh)	Write VLBI frame header to filehandle.
`update`([crc, verify])	Update the header by setting keywords or properties.
`verify`()	Verify header integrity.

Attributes Documentation

bps

Bits per elementary sample (either 1 or 2).

If set, combined with `fanout` and `ntrack` to update 'magnitude_bit' for all tracks.

converters
Converted ID and sideband used for each channel.

Returns a structured array with numerical 'converter' and boolean 'lsb' entries (where `True` means lower sideband).

Can be set with a similar structured array or a `dict`; if just an an array is passed in, it will be assumed that the sideband has been set beforehand (e.g., by setting `nsb`) and that the array holds the converter IDs.

decade = None

fanout
Number of samples stored in one payload item of size ntrack.

If set, will update 'fan_out' for each track.

fraction
Fractional seconds (decoded from 'bcd_fraction').

frame_nbytes
Size of the frame in bytes.

mutable
Whether the header can be modified.

nbytes
Size of the header in bytes.

nchan
Number of channels (`ntrack * fanout`) in the frame.

If set, it is combined with `ntrack` and `fanout` to infer `bps`.

nsb
Number of side bands used.

If set, assumes all converters are upper sideband for 1, and that converter IDs alternate between upper and lower sideband for 2.

ntrack
Number of Mark 4 bitstreams.

payload_nbytes
Size of the payload in bytes.

Note that the payloads miss pieces overwritten by the header.

samples_per_frame
Number of complete samples in the frame.

If set, this uses the number of tracks to infer and set `fanout`.

stream_dtype
Stream dtype required to hold this header's number of tracks.

time
Convert BCD time code to Time object for all tracks.

If all tracks have the same fractional seconds, only a single Time instance is returned.

Uses bcd-encoded 'unit_year', 'day', 'hour', 'minute', 'second' and 'frac_sec', plus `decade` from the initialisation to calculate the time. See http://www.haystack.mit.edu/tech/vlbi/mark5/docs/230.3.pdf

track_assignment
Assignments of tracks to channels and fanout items.

The assignments are inferred from tables 10-14 in http://www.haystack.mit.edu/tech/vlbi/mark5/docs/230.3.pdf except that 2 has been subtracted so that tracks start at 0, and that for 64 tracks the arrays are suitably enlarged by adding another set of channels.

The returned array has shape `(fanout, nchan, bps)`.

track_id

Methods Documentation

copy(***kwargs*)
Create a mutable and independent copy of the header.

Keyword arguments can be passed on as needed by possible subclasses.

classmethod fromfile(*fh*, *ntrack*, *decade=None*, *ref_time=None*, *verify=True*)
Read Mark 4 header from file.

Parameters

fh : filehandle

To read header from.

ntrack : int

Number of Mark 4 bitstreams.

decade : int or None

Decade in which the observations were taken. Can instead pass an approximate `ref_time`.

ref_time : `Time` or None

Reference time within 4 years of the observation time. Used only if `decade` is not given.

verify : bool, optional

Whether to do basic verification of integrity. Default: `True`.

classmethod fromkeys(**args*, ***kwargs*)
Initialise a header from parsed values.

Like fromvalues, but without any interpretation of keywords.

Raises

KeyError : if not all keys required are present in `kwargs`

classmethod fromvalues(*ntrack*, *decade=None*, *ref_time=None*, ***kwargs*)
Initialise a header from parsed values.

Here, the parsed values must be given as keyword arguments, i.e., for any `header = cls(<words>)`, `cls.fromvalues(**header) == header`.

However, unlike for the `fromkeys` class method, data can also be set using arguments named after header methods, such as `time`.

Parameters

ntrack : int

Number of Mark 4 bitstreams.

decade : int or None, optional

Decade in which the observations were taken. Can instead pass an approximate `ref_time`. Not needed if `time` is given.

ref_time : `Time` or None, optional

Reference time within 4 years of the observation time. Used only if `decade` is not given, and not needed if `time` is given.

****kwargs :**

Values used to initialize header keys or methods.

— Header keywords : (minimum for a complete header)

time : `Time` instance

Time of the first sample.

bps : int

Bits per elementary sample.

fanout : int

Number of tracks over which a given channel is spread out.

get_time()

Convert BCD time code to Time object for all tracks.

If all tracks have the same fractional seconds, only a single Time instance is returned.

Uses bcd-encoded 'unit_year', 'day', 'hour', 'minute', 'second' and 'frac_sec', plus `decade` from the initialisation to calculate the time. See http://www.haystack.mit.edu/tech/vlbi/mark5/docs/230.3.pdf

infer_decade(*ref_time*)

Uses a reference time to set a header's `decade`.

Parameters

ref_time : `Time`

Reference time within 5 years of the observation time.

keys()

set_time(*time*)

tofile(*fh*)

Write VLBI frame header to filehandle.

update(*crc=None*, *verify=True*, ***kwargs*)

Update the header by setting keywords or properties.

Here, any keywords matching header keys are applied first, and any remaining ones are used to set header properties, in the order set by the class (in `_properties`).

Parameters

crc : int or None, optional

If `None` (default), recalculate the CRC after updating.

verify : bool, optional

If True (default), verify integrity after updating.

****kwargs**

Arguments used to set keywords and properties.

verify()

Verify header integrity.

Mark4Payload

class baseband.mark4.**Mark4Payload**(*words*, *header=None*, *nchan=1*, *bps=2*, *fanout=1*)

Bases: baseband.vlbi_base.payload.VLBIPayloadBase

Container for decoding and encoding Mark 4 payloads.

Parameters

words : ndarray

Array containg LSB unsigned words (with the right size) that encode the payload.

header : Mark4Header, optional

If given, used to infer the number of channels, bps, and fanout.

nchan : int, optional

Number of channels, used if header is not given. Default: 1.

bps : int, optional

Number of bits per sample, used if header is not given. Default: 2.

fanout : int, optional

Number of tracks every bit stream is spread over, used if header is not given. Default: 1.

Notes

The total number of tracks is nchan * bps * fanout.

Attributes Summary

data	Full decoded payload.
dtype	Numeric type of the decoded data array.
nbytes	Size of the payload in bytes.
ndim	Number of dimensions of the decoded data array.
shape	Shape of the decoded data array.
size	Total number of component samples in the decoded data array.

Methods Summary

fromdata(data, header)	Encode data as payload, using header information.
fromfile(fh, header)	Read payload from filehandle and decode it into data.
tofile(fh)	Write payload to filehandle.

Attributes Documentation

data
Full decoded payload.

dtype
Numeric type of the decoded data array.

nbytes
Size of the payload in bytes.

ndim
Number of dimensions of the decoded data array.

shape
Shape of the decoded data array.

size
Total number of component samples in the decoded data array.

Methods Documentation

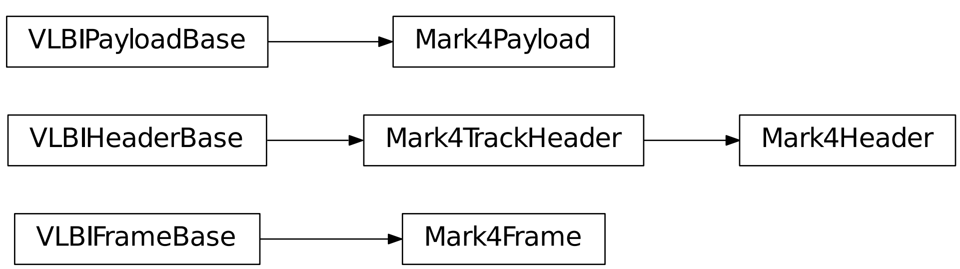
classmethod fromdata(*data*, *header*)
Encode data as payload, using header information.

classmethod fromfile(*fh*, *header*)
Read payload from filehandle and decode it into data.

The payload_nbytes, number of channels, bits per sample, and fanout ratio are all taken from the header.

tofile(*fh*)
Write payload to filehandle.

Class Inheritance Diagram



6.3.2 baseband.mark4.header Module

Definitions for VLBI Mark 4 Headers.

Implements a Mark4Header class used to store header words, and decode/encode the information therein.

For the specification of tape Mark 4 format, see http://www.haystack.mit.edu/tech/vlbi/mark5/docs/230.3.pdf

A little bit on the disk representation is at http://adsabs.harvard.edu/abs/2003ASPC..306..123W

Functions

`stream2words`(stream[, track])	Convert a stream of integers to uint32 header words.
`words2stream`(words)	Convert a set of uint32 header words to a stream of integers.

stream2words

baseband.mark4.header.**stream2words**(*stream*, *track=None*)

Convert a stream of integers to uint32 header words.

Parameters

stream : `array` of int

For each int, every bit corresponds to a particular track.

track : int, array, or None, optional

The track to extract. If `None` (default), extract all tracks that the type of int in the stream can hold.

words2stream

baseband.mark4.header.**words2stream**(*words*)

Convert a set of uint32 header words to a stream of integers.

Parameters

words : `array` of uint32

Returns

stream : `array` of int

For each int, every bit corresponds to a particular track.

Classes

`Mark4TrackHeader`(words[, decade, ref_time, ...])	Decoder/encoder of a Mark 4 Track Header.
`Mark4Header`(words[, ntrack, decade, ...])	Decoder/encoder of a Mark 4 Header, containing all streams.

Mark4TrackHeader

class baseband.mark4.header.**Mark4TrackHeader**(*words*, *decade=None*, *ref_time=None*, *verify=True*)

Bases: `baseband.vlbi_base.header.VLBIHeaderBase`

Decoder/encoder of a Mark 4 Track Header.

See http://www.haystack.mit.edu/tech/vlbi/mark5/docs/230.3.pdf

Parameters

words : tuple of int, or None

Five 32-bit unsigned int header words. If `None`, set to a list of zeros for later initialisation.

decade : int or None

Decade in which the observations were taken (needed to remove ambiguity in the Mark 4 time stamp). Can instead pass an approximate `ref_time`.

ref_time : `Time` or None

Reference time within 4 years of the observation time, used to infer the full Mark 4 timestamp. Used only if `decade` is not given.

verify : bool, optional

Whether to do basic verification of integrity. Default: `True`.

Returns

header : `Mark4TrackHeader`

Attributes Summary

`decade`	
`fraction`	Fractional seconds (decoded from 'bcd_fraction').
`mutable`	Whether the header can be modified.
`nbytes`	Size of the header in bytes.
`time`	Convert BCD time code to Time object.
`track_id`	

Methods Summary

`copy`(**kwargs)	Create a mutable and independent copy of the header.
`fromfile`(fh, *args, **kwargs)	Read VLBI Header from file.
`fromkeys`(*args, **kwargs)	Initialise a header from parsed values.
`fromvalues`(*args, **kwargs)	Initialise a header from parsed values.
`get_time`()	Convert BCD time code to Time object.
`infer_decade`(ref_time)	Uses a reference time to set a header's `decade`.
`keys`()	
`set_time`(time)	
`tofile`(fh)	Write VLBI frame header to filehandle.
`update`(**kwargs)	Update the header by setting keywords or properties.
`verify`()	Verify header integrity.

Attributes Documentation

decade = None

fraction
Fractional seconds (decoded from 'bcd_fraction').

mutable
Whether the header can be modified.

nbytes
Size of the header in bytes.

time
Convert BCD time code to Time object.

Calculate time using bcd-encoded 'bcd_unit_year', 'bcd_day', 'bcd_hour', 'bcd_minute', 'bcd_second' header items, as well as the `fraction` property (inferred from 'bcd_fraction') and `decade` from the initialisation. See See http://www.haystack.mit.edu/tech/vlbi/mark5/docs/230.3.pdf

track_id

Methods Documentation

copy(***kwargs*)
Create a mutable and independent copy of the header.

Keyword arguments can be passed on as needed by possible subclasses.

classmethod fromfile(*fh*, **args*, ***kwargs*)
Read VLBI Header from file.

Arguments are the same as for class initialisation. The header constructed will be immutable.

classmethod fromkeys(**args*, ***kwargs*)
Initialise a header from parsed values.

Like fromvalues, but without any interpretation of keywords.

Raises
KeyError : if not all keys required are present in `kwargs`

classmethod fromvalues(**args*, ***kwargs*)
Initialise a header from parsed values.

Here, the parsed values must be given as keyword arguments, i.e., for any `header = cls(<words>)`, `cls.fromvalues(**header) == header`.

However, unlike for the `fromkeys` class method, data can also be set using arguments named after header methods, such as `time`.

Parameters
***args**

Possible arguments required to initialize an empty header.

****kwargs**

Values used to initialize header keys or methods.

get_time()
Convert BCD time code to Time object.

Calculate time using bcd-encoded 'bcd_unit_year', 'bcd_day', 'bcd_hour', 'bcd_minute', 'bcd_second' header items, as well as the `fraction` property (inferred from 'bcd_fraction') and `decade` from the initialisation. See See http://www.haystack.mit.edu/tech/vlbi/mark5/docs/230.3.pdf

infer_decade(*ref_time*)
Uses a reference time to set a header's decade.

Parameters
ref_time : Time

Reference time within 5 years of the observation time.

keys()

set_time(*time*)

tofile(*fh*)
Write VLBI frame header to filehandle.

update(***kwargs*)
Update the header by setting keywords or properties.

Here, any keywords matching header keys are applied first, and any remaining ones are used to set header properties, in the order set by the class (in `_properties`).

Parameters
verify : bool, optional

If True (default), verify integrity after updating.

****kwargs**

Arguments used to set keywords and properties.

verify()
Verify header integrity.

Mark4Header

class baseband.mark4.header.**Mark4Header**(*words*, *ntrack=None*, *decade=None*, *ref_time=None*, *verify=True*)
Bases: `baseband.mark4.header.Mark4TrackHeader`

Decoder/encoder of a Mark 4 Header, containing all streams.

See http://www.haystack.mit.edu/tech/vlbi/mark5/docs/230.3.pdf

Parameters
words : ndarray of int, or None

Shape should be (5, number-of-tracks), and dtype np.uint32. If None, `ntrack` should be given and words will be initialized to 0.

ntrack : None or int

Number of Mark 4 bitstreams, to help initialize `words` if needed.

decade : int or None

Decade in which the observations were taken (needed to remove ambiguity in the Mark 4 time stamp). Can instead pass an approximate `ref_time`.

ref_time : Time or None

Reference time within 4 years of the observation time, used to infer the full Mark 4 timestamp. Used only if decade is not given.

verify : bool, optional

Whether to do basic verification of integrity. Default: `True`.

Returns

header : `Mark4Header`

Attributes Summary

`bps`	Bits per elementary sample (either 1 or 2).
`converters`	Converted ID and sideband used for each channel.
`decade`	
`fanout`	Number of samples stored in one payload item of size ntrack.
`fraction`	Fractional seconds (decoded from 'bcd_fraction').
`frame_nbytes`	Size of the frame in bytes.
`mutable`	Whether the header can be modified.
`nbytes`	Size of the header in bytes.
`nchan`	Number of channels (`ntrack * fanout`) in the frame.
`nsb`	Number of side bands used.
`ntrack`	Number of Mark 4 bitstreams.
`payload_nbytes`	Size of the payload in bytes.
`samples_per_frame`	Number of complete samples in the frame.
`stream_dtype`	Stream dtype required to hold this header's number of tracks.
`time`	Convert BCD time code to Time object for all tracks.
`track_assignment`	Assignments of tracks to channels and fanout items.
`track_id`	

Methods Summary

`copy`(**kwargs)	Create a mutable and independent copy of the header.
`fromfile`(fh, ntrack[, decade, ref_time, verify])	Read Mark 4 header from file.
`fromkeys`(*args, **kwargs)	Initialise a header from parsed values.
`fromvalues`(ntrack[, decade, ref_time])	Initialise a header from parsed values.
`get_time`()	Convert BCD time code to Time object for all tracks.
`infer_decade`(ref_time)	Uses a reference time to set a header's decade.
`keys`()	
`set_time`(time)	
`tofile`(fh)	Write VLBI frame header to filehandle.
`update`([crc, verify])	Update the header by setting keywords or properties.
`verify`()	Verify header integrity.

Attributes Documentation

bps

Bits per elementary sample (either 1 or 2).

If set, combined with `fanout` and `ntrack` to update 'magnitude_bit' for all tracks.

converters
Converted ID and sideband used for each channel.

Returns a structured array with numerical 'converter' and boolean 'lsb' entries (where `True` means lower sideband).

Can be set with a similar structured array or a `dict`; if just an an array is passed in, it will be assumed that the sideband has been set beforehand (e.g., by setting `nsb`) and that the array holds the converter IDs.

decade = None

fanout
Number of samples stored in one payload item of size ntrack.

If set, will update 'fan_out' for each track.

fraction
Fractional seconds (decoded from 'bcd_fraction').

frame_nbytes
Size of the frame in bytes.

mutable
Whether the header can be modified.

nbytes
Size of the header in bytes.

nchan
Number of channels (`ntrack * fanout`) in the frame.

If set, it is combined with `ntrack` and `fanout` to infer `bps`.

nsb
Number of side bands used.

If set, assumes all converters are upper sideband for 1, and that converter IDs alternate between upper and lower sideband for 2.

ntrack
Number of Mark 4 bitstreams.

payload_nbytes
Size of the payload in bytes.

Note that the payloads miss pieces overwritten by the header.

samples_per_frame
Number of complete samples in the frame.

If set, this uses the number of tracks to infer and set `fanout`.

stream_dtype
Stream dtype required to hold this header's number of tracks.

time
Convert BCD time code to Time object for all tracks.

If all tracks have the same fractional seconds, only a single Time instance is returned.

Uses bcd-encoded 'unit_year', 'day', 'hour', 'minute', 'second' and 'frac_sec', plus `decade` from the initialisation to calculate the time. See http://www.haystack.mit.edu/tech/vlbi/mark5/docs/230.3.pdf

track_assignment
Assignments of tracks to channels and fanout items.

The assignments are inferred from tables 10-14 in http://www.haystack.mit.edu/tech/vlbi/mark5/docs/230.3.pdf except that 2 has been subtracted so that tracks start at 0, and that for 64 tracks the arrays are suitably enlarged by adding another set of channels.

The returned array has shape `(fanout, nchan, bps)`.

track_id

Methods Documentation

copy(***kwargs*)
Create a mutable and independent copy of the header.

Keyword arguments can be passed on as needed by possible subclasses.

classmethod fromfile(*fh*, *ntrack*, *decade=None*, *ref_time=None*, *verify=True*)
Read Mark 4 header from file.

Parameters

fh : filehandle

To read header from.

ntrack : int

Number of Mark 4 bitstreams.

decade : int or None

Decade in which the observations were taken. Can instead pass an approximate `ref_time`.

ref_time : `Time` or None

Reference time within 4 years of the observation time. Used only if `decade` is not given.

verify : bool, optional

Whether to do basic verification of integrity. Default: `True`.

classmethod fromkeys(**args*, ***kwargs*)
Initialise a header from parsed values.

Like fromvalues, but without any interpretation of keywords.

Raises

KeyError : if not all keys required are present in `kwargs`

classmethod fromvalues(*ntrack*, *decade=None*, *ref_time=None*, ***kwargs*)
Initialise a header from parsed values.

Here, the parsed values must be given as keyword arguments, i.e., for any `header = cls(<words>)`, `cls.fromvalues(**header) == header`.

However, unlike for the `fromkeys` class method, data can also be set using arguments named after header methods, such as `time`.

Parameters

ntrack : int

Number of Mark 4 bitstreams.

decade : int or None, optional

Decade in which the observations were taken. Can instead pass an approximate `ref_time`. Not needed if `time` is given.

ref_time : `Time` or None, optional

Reference time within 4 years of the observation time. Used only if `decade` is not given, and not needed if `time` is given.

****kwargs :**

Values used to initialize header keys or methods.

— Header keywords : (minimum for a complete header)

time : `Time` instance

Time of the first sample.

bps : int

Bits per elementary sample.

fanout : int

Number of tracks over which a given channel is spread out.

get_time()

Convert BCD time code to Time object for all tracks.

If all tracks have the same fractional seconds, only a single Time instance is returned.

Uses bcd-encoded 'unit_year', 'day', 'hour', 'minute', 'second' and 'frac_sec', plus `decade` from the initialisation to calculate the time. See http://www.haystack.mit.edu/tech/vlbi/mark5/docs/230.3.pdf

infer_decade(*ref_time*)

Uses a reference time to set a header's `decade`.

Parameters

ref_time : `Time`

Reference time within 5 years of the observation time.

keys()

set_time(*time*)

tofile(*fh*)

Write VLBI frame header to filehandle.

update(*crc=None*, *verify=True*, ***kwargs*)

Update the header by setting keywords or properties.

Here, any keywords matching header keys are applied first, and any remaining ones are used to set header properties, in the order set by the class (in `_properties`).

Parameters

crc : int or None, optional

If `None` (default), recalculate the CRC after updating.

verify : bool, optional

If `True` (default), verify integrity after updating.

****kwargs**

Arguments used to set keywords and properties.

verify()
Verify header integrity.

Variables

CRC12	CRC polynomial used for Mark 4 Headers.
crc12	Cyclic Redundancy Check for a bitstream.

CRC12

baseband.mark4.header.**CRC12 = 6159**
CRC polynomial used for Mark 4 Headers.

x^12 + x^11 + x^3 + x^2 + x + 1, i.e., 0x180f. See page 4 of http://www.haystack.mit.edu/tech/vlbi/mark5/docs/230.3.pdf

This is also a 'standard' CRC-12 mentioned in https://en.wikipedia.org/wiki/Cyclic_redundancy_check

crc12

baseband.mark4.header.**crc12 = <baseband.vlbi_base.utils.CRC object>**
Cyclic Redundancy Check for a bitstream.

See https://en.wikipedia.org/wiki/Cyclic_redundancy_check

Once initialised, the instance can be used as a function that calculates the CRC, or one can use the check method to check that the CRC at the end of a stream is correct.

Parameters
polynomial : int

Binary encoded CRC divisor. For instance, that used by Mark 4 headers is 0x180f, or x^12 + x^11 + x^3 + x^2 + x + 1.

Class Inheritance Diagram

6.3.3 baseband.mark4.payload Module

Definitions for VLBI Mark 4 payloads.

Implements a Mark4Payload class used to store payload words, and decode to or encode from a data array.

For the specification, see http://www.haystack.mit.edu/tech/vlbi/mark5/docs/230.3.pdf

Functions

`reorder32`(x)	Reorder 32-track bits to bring signs & magnitudes together.
`reorder64`(x)	Reorder 64-track bits to bring signs & magnitudes together.
`init_luts`()	Set up the look-up tables for levels as a function of input byte.
`decode_8chan_2bit_fanout4`(frame)	Decode payload for 8 channels using 2 bits, fan-out 4 (64 tracks).
`encode_8chan_2bit_fanout4`(values)	Encode payload for 8 channels using 2 bits, fan-out 4 (64 tracks).

reorder32

`baseband.mark4.payload.`**`reorder32`**(*x*)

Reorder 32-track bits to bring signs & magnitudes together.

reorder64

`baseband.mark4.payload.`**`reorder64`**(*x*)

Reorder 64-track bits to bring signs & magnitudes together.

init_luts

`baseband.mark4.payload.`**`init_luts`**()

Set up the look-up tables for levels as a function of input byte.

decode_8chan_2bit_fanout4

`baseband.mark4.payload.`**`decode_8chan_2bit_fanout4`**(*frame*)

Decode payload for 8 channels using 2 bits, fan-out 4 (64 tracks).

encode_8chan_2bit_fanout4

`baseband.mark4.payload.`**`encode_8chan_2bit_fanout4`**(*values*)

Encode payload for 8 channels using 2 bits, fan-out 4 (64 tracks).

Classes

Mark4Payload(words[, header, nchan, bps, fanout])	Container for decoding and encoding Mark 4 payloads.

Mark4Payload

class baseband.mark4.payload.**Mark4Payload**(*words*, *header=None*, *nchan=1*, *bps=2*, *fanout=1*)

Bases: baseband.vlbi_base.payload.VLBIPayloadBase

Container for decoding and encoding Mark 4 payloads.

Parameters

words : ndarray

Array containg LSB unsigned words (with the right size) that encode the payload.

header : Mark4Header, optional

If given, used to infer the number of channels, bps, and fanout.

nchan : int, optional

Number of channels, used if header is not given. Default: 1.

bps : int, optional

Number of bits per sample, used if header is not given. Default: 2.

fanout : int, optional

Number of tracks every bit stream is spread over, used if header is not given. Default: 1.

Notes

The total number of tracks is `nchan * bps * fanout`.

Attributes Summary

data	Full decoded payload.
dtype	Numeric type of the decoded data array.
nbytes	Size of the payload in bytes.
ndim	Number of dimensions of the decoded data array.
shape	Shape of the decoded data array.
size	Total number of component samples in the decoded data array.

Methods Summary

fromdata(data, header)	Encode data as payload, using header information.
fromfile(fh, header)	Read payload from filehandle and decode it into data.
tofile(fh)	Write payload to filehandle.

Attributes Documentation

data
Full decoded payload.

dtype
Numeric type of the decoded data array.

nbytes
Size of the payload in bytes.

ndim
Number of dimensions of the decoded data array.

shape
Shape of the decoded data array.

size
Total number of component samples in the decoded data array.

Methods Documentation

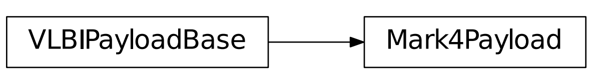
classmethod fromdata(*data*, *header*)
Encode data as payload, using header information.

classmethod fromfile(*fh*, *header*)
Read payload from filehandle and decode it into data.

The payload_nbytes, number of channels, bits per sample, and fanout ratio are all taken from the header.

tofile(*fh*)
Write payload to filehandle.

Class Inheritance Diagram



6.3.4 baseband.mark4.frame Module

Definitions for VLBI Mark 4 payloads.

Implements a Mark4Payload class used to store payload words, and decode to or encode from a data array.

For the specification, see http://www.haystack.mit.edu/tech/vlbi/mark5/docs/230.3.pdf

Classes

Mark4Frame(header, payload[, valid, verify])	Representation of a Mark 4 frame, consisting of a header and payload.

Mark4Frame

class baseband.mark4.frame.**Mark4Frame**(*header*, *payload*, *valid=None*, *verify=True*)
Bases: baseband.vlbi_base.frame.VLBIFrameBase

Representation of a Mark 4 frame, consisting of a header and payload.

Parameters

header : Mark4Header

Wrapper around the encoded header words, providing access to the header information.

payload : Mark4Payload

Wrapper around the payload, provding mechanisms to decode it.

valid : bool or None, optional

Whether the data are valid. If None (default), inferred from header. Note that header is updated in-place if True or False.

verify : bool, optional

Whether or not to do basic assertions that check the integrity (e.g., that channel information and number of tracks are consistent between header and data). Default: True.

Notes

The Frame can also be read instantiated using class methods:

fromfile : read header and payload from a filehandle

fromdata : encode data as payload

Of course, one can also do the opposite:

tofile : method to write header and payload to filehandle

data : property that yields full decoded payload

One can decode part of the payload by indexing or slicing the frame. If the frame does not contain valid data, all values returned are set to self.fill_value.

A number of properties are defined: shape, dtype and size are the shape, type and number of complete samples of the data array, and nbytes the frame size in bytes. Furthermore, the frame acts as a dictionary, with keys those of the header. Any attribute that is not defined on the frame itself, such as .time will be looked up on the header as well.

Attributes Summary

data	Full decoded frame, with header part filled in.
dtype	Numeric type of the frame data.
fill_value	Value to replace invalid data in the frame.
nbytes	Size of the encoded frame in bytes.

Continued on next page

Table 21 – continued from previous page

ndim	Number of dimensions of the frame data.
sample_shape	Shape of a sample in the frame (nchan,).
shape	Shape of the frame data.
size	Total number of component samples in the frame data.
valid	Whether frame contains valid data.

Methods Summary

fromdata(data[, header, verify])	Construct frame from data and header.
fromfile(fh, ntrack[, decade, ref_time, verify])	Read a frame from a filehandle.
keys()	
tofile(fh)	Write encoded frame to filehandle.
verify()	Simple verification.

Attributes Documentation

data
Full decoded frame, with header part filled in.

dtype
Numeric type of the frame data.

fill_value
Value to replace invalid data in the frame.

nbytes
Size of the encoded frame in bytes.

ndim
Number of dimensions of the frame data.

sample_shape
Shape of a sample in the frame (nchan,).

shape
Shape of the frame data.

size
Total number of component samples in the frame data.

valid
Whether frame contains valid data.

None of the error flags are set.

Methods Documentation

classmethod fromdata(*data*, *header=None*, *verify=True*, ***kwargs*)
Construct frame from data and header.

Parameters
data : ndarray

Array holding complex or real data to be encoded. This should have the full size of a data frame, even though the part covered by the header will be ignored.

header : `Mark4Header` or None

If not given, will attempt to generate one using the keywords.

verify : bool, optional

Whether to do basic checks of frame integrity (default: `True`).

classmethod fromfile(*fh*, *ntrack*, *decade=None*, *ref_time=None*, *verify=True*)

Read a frame from a filehandle.

Parameters

fh : filehandle

To read header from.

ntrack : int

Number of Mark 4 bitstreams.

decade : int or None

Decade in which the observations were taken. Can instead pass an approximate `ref_time`.

ref_time : `Time` or None

Reference time within 4 years of the observation time. Used only if `decade` is not given.

verify : bool, optional

Whether to do basic verification of integrity. Default: `True`.

keys()

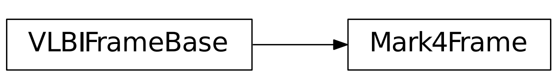
tofile(*fh*)

Write encoded frame to filehandle.

verify()

Simple verification. To be added to by subclasses.

Class Inheritance Diagram



6.3.5 baseband.mark4.base Module

Functions

open(name[, mode])	Open Mark4 file for reading or writing.

open

baseband.mark4.base.**open**(*name*, *mode=u'rs'*, ***kwargs*)

Open Mark4 file for reading or writing.

Opened as a binary file, one gets a wrapped filehandle that adds methods to read/write a frame. Opened as a stream, the handle is wrapped further, with methods such as reading and writing to the file as if it were a stream of samples.

Parameters

name : str or filehandle

File name or handle.

mode : {'rb', 'wb', 'rs', or 'ws'}, optional

Whether to open for reading or writing, and as a regular binary file or as a stream. Default: 'rs', for reading a stream.

****kwargs**

Additional arguments when opening the file as a stream.

— For reading a stream : (see Mark4StreamReader)

sample_rate : Quantity, optional

Number of complete samples per second, i.e. the rate at which each channel is sampled. If not given, will be inferred from scanning two frames of the file.

ntrack : int, optional

Number of Mark 4 bitstreams. If None (default), will attempt to automatically detect it by scanning the file.

decade : int or None

Decade of the observation start time (eg. 2010 for 2018), needed to remove ambiguity in the Mark 4 time stamp (default: None). Can instead pass an approximate ref_time.

ref_time : Time or None

Reference time within 4 years of the start time of the observations. Used only if decade is not given.

squeeze : bool, optional

If True (default), remove any dimensions of length unity from decoded data.

subset : indexing object, optional

Specific channels of the complete sample to decode (after possible squeezing). If an empty tuple (default), all channels are read.

fill_value : float or complex, optional

Value to use for invalid or missing data. Default: 0.

verify : bool, optional

Whether to do basic checks of frame integrity when reading. The first frame of the stream is always checked. Default: True.

— For writing a stream : (see Mark4StreamWriter)

header0 : Mark4Header

Header for the first frame, holding time information, etc. Can instead give keyword arguments to construct a header (see **kwargs).

sample_rate : Quantity

Number of complete samples per second, i.e. the rate at which each channel is sampled. Needed to calculate header timestamps.

squeeze : bool, optional

If True (default), writer accepts squeezed arrays as input, and adds any dimensions of length unity.

****kwargs**

If the header is not given, an attempt will be made to construct one with any further keyword arguments. See Mark4StreamWriter.

Returns

Filehandle

Mark4FileReader or Mark4FileWriter (binary), or Mark4StreamReader or Mark4StreamWriter (stream)

Classes

Mark4FileReader(fh_raw[, ntrack, decade, . . .])	Simple reader for Mark 4 files.
Mark4FileWriter(fh_raw)	Simple writer for Mark 4 files.
Mark4StreamReader(fh_raw[, sample_rate, . . .])	VLBI Mark 4 format reader.
Mark4StreamWriter(fh_raw[, header0, . . .])	VLBI Mark 4 format writer.

Mark4FileReader

class baseband.mark4.base.**Mark4FileReader**(*fh_raw*, *ntrack=None*, *decade=None*, *ref_time=None*)

Bases: baseband.vlbi_base.base.VLBIFileReaderBase

Simple reader for Mark 4 files.

Wraps a binary filehandle, providing methods to help interpret the data, such as `locate_frame, `read_frame` and get_frame_rate.

Parameters

fh_raw : filehandle

Filehandle of the raw binary data file.

ntrack : int or None, optional.

Number of Mark 4 bitstreams. Can be determined automatically as part of locating the first frame.

decade : int or None

Decade in which the observations were taken. Can instead pass an approximate ref_time.

ref_time : Time or None

Reference time within 4 years of the observation time. Used only if decade is not given.

Attributes Summary

info	Standardized information on Mark 4 file readers.

Methods Summary

close()	
determine_ntrack([maximum])	Determines the number of tracks, by seeking the next frame.
find_header([forward, maximum])	Find the nearest header from the current position.
get_frame_rate()	Determine the number of frames per second.
locate_frame([forward, maximum])	Locate the frame nearest the current position.
read_frame([verify])	Read a single frame (header plus payload).
read_header()	Read a single header from the file.

Attributes Documentation

info

Standardized information on Mark 4 file readers.

The `info` descriptor has a number of standard attributes, which are determined from arguments passed in opening the file, from the first header (`info.header0`) and from possibly scanning the file to determine the duration of frames. `Mark4FileReaderInfo` has two additional attributes specific to Mark 4 files (`ntrack` and `offset0`, see below).

Examples

The most common use is simply to print information:

```
>>> from baseband.data import SAMPLE_MARK4
>>> from baseband import mark4
>>> fh = mark4.open(SAMPLE_MARK4, 'rb')
>>> fh.info
File information:
format = mark4
frame_rate = 400.0 Hz
sample_rate = 32.0 MHz
samples_per_frame = 80000
sample_shape = (8,)
bps = 2
complex_data = False
offset0 = 2696

missing:  decade, ref_time: needed to infer full times.

>>> fh.close()

>>> fh = mark4.open(SAMPLE_MARK4, 'rb', decade=2010)
>>> fh.info
```

(continues on next page)

(continued from previous page)

```
File information:
format = mark4
frame_rate = 400.0 Hz
sample_rate = 32.0 MHz
samples_per_frame = 80000
sample_shape = (8,)
bps = 2
complex_data = False
start_time = 2014-06-16T07:38:12.475000000
offset0 = 2696
>>> fh.close()
```

Attributes

format	(str or None) File format, or None if the underlying file cannot be parsed.
frame_rate	(Quantity) Number of data frames per unit of time.
sample_rate	(Quantity) Complete samples per unit of time.
samples_per_frame	(int) Number of complete samples in each frame.
sample_shape	(tuple) Dimensions of each complete sample (e.g., `(nchan,)`).
bps	(int) Number of bits used to encode each elementary sample.
complex_data	(bool) Whether the data are complex.
start_time	(Time) Time of the first complete sample.
ntrack	(int) Number of "tape tracks" simulated in the disk file.
offset0	(int) Offset in bytes from the start of the file to the location of the first header.
missing	(dict) Entries are keyed by names of arguments that should be passed to the file reader to obtain full information. The associated entries explain why these arguments are needed. For Mark 4, the possible entries are `decade` and `ref_time`.

Methods Documentation

close()

determine_ntrack(*maximum=None*)

Determines the number of tracks, by seeking the next frame.

Uses `find_frame` to look for the first occurrence of a frame from the current position for all supported `ntrack` values. Returns the first `ntrack` for which `find_frame` is successful, setting the file's `ntrack` property appropriately, and leaving the file pointer at the start of the frame.

Parameters

maximum : int, optional

Maximum number of bytes forward to search through. Default: twice the frame size (`20000 * ntrack // 8`).

Returns

ntrack : int or None

Number of Mark 4 bitstreams. `None` if no frame was found.

find_header(*forward=True*, *maximum=None*)
Find the nearest header from the current position.

If successful, the file pointer is left at the start of the header.

Parameters
forward : bool, optional

Seek forward if `True` (default), backward if `False`.

maximum : int, optional

Maximum number of bytes forward to search through. Default: twice the frame size (`20000 * ntrack // 8`).

Returns
header : `Mark4Header` or None

Retrieved Mark 4 header, or `None` if nothing found.

get_frame_rate()
Determine the number of frames per second.

The frame rate is calculated from the time elapsed between the first two frames, as inferred from their time stamps.

Returns
frame_rate : `Quantity`

Frames per second.

locate_frame(*forward=True*, *maximum=None*)
Locate the frame nearest the current position.

The search is for the following pattern:

- 32*tracks bits set at offset bytes
- 1*tracks bits unset before offset
- 32*tracks bits set at offset+2500*tracks bytes

This reflects 'sync_pattern' of 0xffffffff for a given header and one a frame ahead, which is in word 2, plus the lsb of word 1, which is 'system_id'.

If the file does not have ntrack is set, it will be auto-determined.

Parameters
forward : bool, optional

Whether to search forwards or backwards. Default: `True`.

maximum : int, optional

Maximum number of bytes forward to search through. Default: twice the frame size (`20000 * ntrack // 8`).

Returns
offset : int or `None`

Byte offset of the next frame. `None` if the search was not successful.

read_frame(*verify=True*)
Read a single frame (header plus payload).

Returns

frame : Mark4Frame

With .header and .data properties that return the Mark4Header and data encoded in the frame, respectively.

verify : bool, optional

Whether to do basic checks of frame integrity. Default: True.

read_header()

Read a single header from the file.

Returns

header : Mark4Header

Mark4FileWriter

class baseband.mark4.base.**Mark4FileWriter**(*fh_raw*)

Bases: baseband.vlbi_base.base.VLBIFileBase

Simple writer for Mark 4 files.

Adds write_frame method to the VLBI binary file wrapper.

Methods Summary

close()	
write_frame(data[, header])	Write a single frame (header plus payload).

Methods Documentation

close()

write_frame(*data*, *header=None*, ***kwargs*)

Write a single frame (header plus payload).

Parameters

data : ndarray or Mark4Frame

If an array, a header should be given, which will be used to get the information needed to encode the array, and to construct the Mark 4 frame.

header : Mark4Header

Can instead give keyword arguments to construct a header. Ignored if payload is a Mark4Frame instance.

****kwargs :**

If header is not given, these are used to initialize one.

Mark4StreamReader

class baseband.mark4.base.**Mark4StreamReader**(*fh_raw*, *sample_rate=None*, *ntrack=None*, *decade=None*, *ref_time=None*, *squeeze=True*, *subset=()*, *fill_value=0.0*, *verify=True*)

Bases: baseband.mark4.base.Mark4StreamBase, baseband.vlbi_base.base.VLBIStreamReaderBase

VLBI Mark 4 format reader.

Allows access to a Mark 4 file as a continuous series of samples. Parts of the data stream replaced by header values are filled in.

Parameters

fh_raw : filehandle

Filehandle of the raw Mark 4 stream.

sample_rate : Quantity, optional

Number of complete samples per second, i.e. the rate at which each channel is sampled. If None, will be inferred from scanning two frames of the file.

ntrack : int or None, optional

Number of Mark 4 bitstreams. If None (default), will attempt to automatically detect it by scanning the file.

decade : int or None

Decade of the observation start time (eg. 2010 for 2018), needed to remove ambiguity in the Mark 4 time stamp. Can instead pass an approximate ref_time.

ref_time : Time or None

Reference time within 4 years of the start time of the observations. Used only if decade is not given.

squeeze : bool, optional

If True (default), remove any dimensions of length unity from decoded data.

subset : indexing object, optional

Specific channels of the complete sample to decode (after possible squeezing). If an empty tuple (default), all channels are read.

fill_value : float or complex, optional

Value to use for invalid or missing data. Default: 0.

verify : bool, optional

Whether to do basic checks of frame integrity when reading. The first frame of the stream is always checked. Default: True.

Attributes Summary

bps	Bits per elementary sample.
complex_data	Whether the data are complex.
dtype	
fill_value	Value to use for invalid or missing data.

Continued on next page

Table 28 – continued from previous page

header0	First header of the file.
info	Standardized information on stream readers.
ndim	Number of dimensions of the (squeezed/subset) stream data.
sample_rate	Number of complete samples per second.
sample_shape	Shape of a complete sample (possibly subset or squeezed).
samples_per_frame	Number of complete samples per frame.
shape	Shape of the (squeezed/subset) stream data.
size	Total number of component samples in the (squeezed/subset) stream data.
squeeze	Whether data arrays have dimensions with length unity removed.
start_time	Start time of the file.
stop_time	Time at the end of the file, just after the last sample.
subset	Specific components of the complete sample to decode.
time	Time of the sample pointer's current offset in file.
verify	Whether to do consistency checks on frames being read.

Methods Summary

close()	
read([count, out])	Read a number of complete (or subset) samples.
seek(offset[, whence])	Change the stream position.
tell([unit])	Current offset in the file.

Attributes Documentation

bps
Bits per elementary sample.

complex_data
Whether the data are complex.

dtype

fill_value
Value to use for invalid or missing data. Default: 0.

header0
First header of the file.

info
Standardized information on stream readers.

The `info` descriptor provides a few standard attributes, all of which can also be accessed directly on the stream filehandle. More detailed information on the underlying file is stored in its info, accessible via `info.file_info`.

Attributes

start_time	(`Time`) Time of the first complete sample.
stop_time	(`Time`) Time of the complete sample just beyond the end of the file.
sample_rate	(`Quantity`) Complete samples per unit of time.
shape	(tuple) Equivalent shape of the whole file, i.e., combining the number of complete samples and the shape of those samples.
bps	(int) Number of bits used to encode each elementary sample.
complex_data	(bool) Whether the data are complex.

ndim
Number of dimensions of the (squeezed/subset) stream data.

sample_rate
Number of complete samples per second.

sample_shape
Shape of a complete sample (possibly subset or squeezed).

samples_per_frame
Number of complete samples per frame.

shape
Shape of the (squeezed/subset) stream data.

size
Total number of component samples in the (squeezed/subset) stream data.

squeeze
Whether data arrays have dimensions with length unity removed.

If `True`, data read out has such dimensions removed, and data passed in for writing has them inserted.

start_time
Start time of the file.

See also `time` for the time of the sample pointer's current offset, and (if available) `stop_time` for the time at the end of the file.

stop_time
Time at the end of the file, just after the last sample.

See also `start_time` for the start time of the file, and `time` for the time of the sample pointer's current offset.

subset
Specific components of the complete sample to decode.

The order of dimensions is the same as for `sample_shape`. Set by the class initializer.

time
Time of the sample pointer's current offset in file.

See also `start_time` for the start time, and (if available) `stop_time` for the end time, of the file.

verify
Whether to do consistency checks on frames being read.

Methods Documentation

close()

read(*count=None*, *out=None*)

Read a number of complete (or subset) samples.

The range retrieved can span multiple frames.

Parameters

count : int or None, optional

Number of complete/subset samples to read. If `None` (default) or negative, the whole file is read. Ignored if `out` is given.

out : None or array, optional

Array to store the data in. If given, `count` will be inferred from the first dimension; the other dimension should equal `sample_shape`.

Returns

out : `ndarray` of float or complex

The first dimension is sample-time, and the remainder given by `sample_shape`.

seek(*offset*, *whence=0*)

Change the stream position.

This works like a normal filehandle seek, but the offset is in samples (or a relative or absolute time).

Parameters

offset : int, `Quantity`, or `Time`

Offset to move to. Can be an (integer) number of samples, an offset in time units, or an absolute time.

whence : {0, 1, 2, 'start', 'current', or 'end'}, optional

Like regular seek, the offset is taken to be from the start if `whence=0` (default), from the current position if 1, and from the end if 2. One can alternativey use 'start', 'current', or 'end' for 0, 1, or 2, respectively. Ignored if `offset` is a time.

tell(*unit=None*)

Current offset in the file.

Parameters

unit : `Unit` or str, optional

Time unit the offset should be returned in. By default, no unit is used, i.e., an integer enumerating samples is returned. For the special string 'time', the absolute time is calculated.

Returns

offset : int, `Quantity`, or `Time`

Offset in current file (or time at current position).

Mark4StreamWriter

class baseband.mark4.base.**Mark4StreamWriter**(*fh_raw*, *header0=None*, *sample_rate=None*, *squeeze=True*, ***kwargs*)

Bases: baseband.mark4.base.Mark4StreamBase, baseband.vlbi_base.base.VLBIStreamWriterBase

VLBI Mark 4 format writer.

Encodes and writes sequences of samples to file.

Parameters

raw : filehandle

Which will write filled sets of frames to storage.

header0 : Mark4Header

Header for the first frame, holding time information, etc. Can instead give keyword arguments to construct a header (see `**kwargs`).

sample_rate : Quantity

Number of complete samples per second, i.e. the rate at which each channel is sampled. Needed to calculate header timestamps.

squeeze : bool, optional

If `True` (default), `write` accepts squeezed arrays as input, and adds any dimensions of length unity.

****kwargs**

If no header is given, an attempt is made to construct one from these. For a standard header, this would include the following.

— Header keywords : (see `fromvalues()`)

time : Time

Start time of the file. Sets bcd-encoded unit year, day, hour, minute, second in the header.

ntrack : int

Number of Mark 4 bitstreams (equal to number of channels times `fanout` times `bps`)

bps : int

Bits per elementary sample.

fanout : int

Number of tracks over which a given channel is spread out.

Attributes Summary

bps	Bits per elementary sample.
complex_data	Whether the data are complex.
header0	First header of the file.
sample_rate	Number of complete samples per second.

Continued on next page

Table 30 – continued from previous page

sample_shape	Shape of a complete sample (possibly subset or squeezed).
samples_per_frame	Number of complete samples per frame.
squeeze	Whether data arrays have dimensions with length unity removed.
start_time	Start time of the file.
subset	Specific components of the complete sample to decode.
time	Time of the sample pointer's current offset in file.
verify	Whether to do consistency checks on frames being read.

Methods Summary

close()	
tell([unit])	Current offset in the file.
write(data[, valid])	Write data, buffering by frames as needed.

Attributes Documentation

bps
Bits per elementary sample.

complex_data
Whether the data are complex.

header0
First header of the file.

sample_rate
Number of complete samples per second.

sample_shape
Shape of a complete sample (possibly subset or squeezed).

samples_per_frame
Number of complete samples per frame.

squeeze
Whether data arrays have dimensions with length unity removed.

If `True`, data read out has such dimensions removed, and data passed in for writing has them inserted.

start_time
Start time of the file.

See also `time` for the time of the sample pointer's current offset, and (if available) `stop_time` for the time at the end of the file.

subset
Specific components of the complete sample to decode.

The order of dimensions is the same as for `sample_shape`. Set by the class initializer.

time
Time of the sample pointer's current offset in file.

See also `start_time` for the start time, and (if available) `stop_time` for the end time, of the file.

verify
Whether to do consistency checks on frames being read.

Methods Documentation

close()

tell(*unit=None*)
Current offset in the file.

Parameters
unit : `Unit` or str, optional

Time unit the offset should be returned in. By default, no unit is used, i.e., an integer enumerating samples is returned. For the special string 'time', the absolute time is calculated.

Returns
offset : int, `Quantity`, or `Time`

Offset in current file (or time at current position).

write(*data*, *valid=True*)
Write data, buffering by frames as needed.

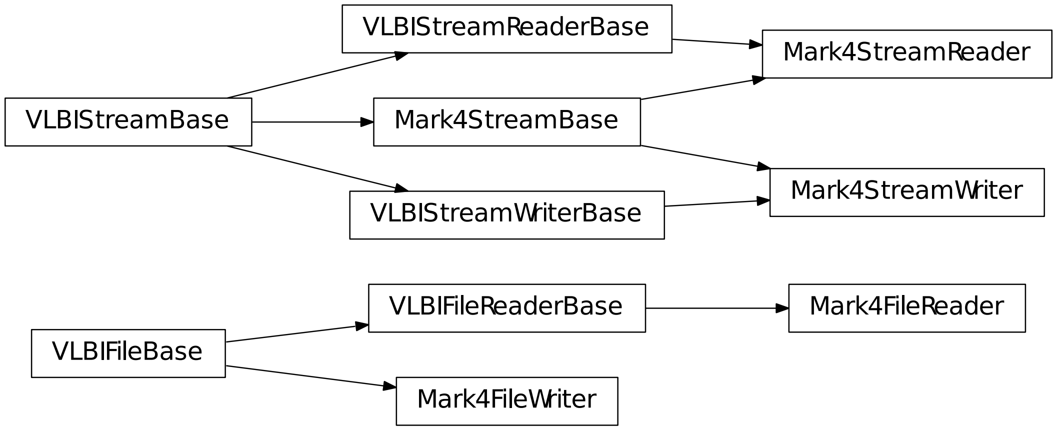
Parameters
data : `ndarray`

Piece of data to be written, with sample dimensions as given by `sample_shape`. This should be properly scaled to make best use of the dynamic range delivered by the encoding.

valid : bool, optional

Whether the current data are valid. Default: `True`.

Class Inheritance Diagram



CHAPTER 7

DADA

Distributed Acquisition and Data Analysis (DADA) format data files contain a single *data frame* consisting of an ASCII *header* of typically 4096 bytes followed by a *payload*.

7.1 Usage

This section covers reading and writing DADA files with Baseband; general usage is covered in the *Getting Started* section. For situations in which one is unsure of a file's format, Baseband features the general `baseband.open` and `baseband.file_info` functions, which are also discussed in *Getting Started*. The examples below use the sample file `baseband/data/sample.dada`, and the the `astropy.units` and `baseband.dada` modules:

```
>>> from baseband import dada
>>> import astropy.units as u
>>> from baseband.data import SAMPLE_DADA
```

Single files can be opened with `open` in binary mode. DADA files typically consist of just a single header and payload, and can be read into a single `DADAFrame`.

```
>>> fb = dada.open(SAMPLE_DADA, 'rb')
>>> frame = fb.read_frame()
>>> frame.shape
(16000, 2, 1)
>>> frame[:3].squeeze()
array([[ -38.-38.j,  -38.-38.j],
       [ -38.-38.j,  -40. +0.j],
       [-105.+60.j,   85.-15.j]], dtype=complex64)
>>> fb.close()
```

Since the files can be quite large, the payload is mapped (with `numpy.memmap`), so that if one accesses part of the data, only the corresponding parts of the encoded payload are loaded into memory (since the sample file is encoded using 8 bits, the above example thus loads 12 bytes into memory).

Opening in stream mode wraps the low-level routines such that reading and writing is in units of samples, and provides access to header information:

```
>>> fh = dada.open(SAMPLE_DADA, 'rs')
>>> fh
<DADAStreamReader name=... offset=0
    sample_rate=16.0 MHz, samples_per_frame=16000,
    sample_shape=SampleShape(npol=2), bps=8,
    start_time=2013-07-02T01:39:20.000>
>>> d = fh.read(10000)
>>> d.shape
(10000, 2)
>>> d[:3]
array([[ -38.-38.j,  -38.-38.j],
       [ -38.-38.j,  -40. +0.j],
       [-105.+60.j,   85.-15.j]], dtype=complex64)
>>> fh.close()
```

To set up a file for writing as a stream is possible as well:

```
>>> from astropy.time import Time
>>> fw = dada.open('{utc_start}.{obs_offset:016d}.000000.dada', 'ws',
...                sample_rate=16*u.MHz, samples_per_frame=5000,
...                npol=2, nchan=1, bps=8, complex_data=True,
...                time=Time('2013-07-02T01:39:20.000'))
>>> fw.write(d)
>>> fw.close()
>>> import os
>>> [f for f in sorted(os.listdir('.')) if f.startswith('2013')]
['2013-07-02-01:39:20.0000000000000000.000000.dada',
 '2013-07-02-01:39:20.0000000000020000.000000.dada']
>>> fr = dada.open('2013-07-02-01:39:20.{obs_offset:016d}.000000.dada', 'rs')
>>> d2 = fr.read()
>>> (d == d2).all()
True
>>> fr.close()
```

Here, we have used an even smaller size of the payload, to show how one can define multiple files. DADA data are typically stored in sequences of files. If, in place of a single filename, one passes a time-ordered list or tuple of filenames to `open`, it uses `sequentialfile.open` to read or write to them as a single contiguous file. If, as above, one passes a template string, `open` uses `DADAFileNameSequencer` to create a subscriptable filename generator, which is then passed to `sequentialfile.open`. (See API links for further details.)

7.2 Reference/API

7.2.1 baseband.dada Package

Distributed Acquisition and Data Analysis (DADA) format reader/writer.

Functions

open(name[, mode])	Open DADA file for reading or writing.

open

baseband.dada.**open**(*name*, *mode=u'rs'*, ***kwargs*)

Open DADA file for reading or writing.

Opened as a binary file, one gets a wrapped filehandle that adds methods to read/write a frame. Opened as a stream, the handle is wrapped further, with methods such as reading and writing to the file as if it were a stream of samples.

Parameters

name : str or filehandle

File name or handle.

mode : {'rb', 'wb', 'rs', or 'ws'}, optional

Whether to open for reading or writing, and as a regular binary file or as a stream. Default: 'rs', for reading a stream.

****kwargs**

Additional arguments when opening the file as a stream.

— For reading a stream : (see `DADAStreamReader`)

squeeze : bool, optional

If `True` (default), remove any dimensions of length unity from decoded data.

subset : indexing object or tuple of objects, optional

Specific components of the complete sample to decode (after possibly squeezing). If a single indexing object is passed, it selects polarizations. With a tuple, the first selects polarizations and the second selects channels. If the tuple is empty (default), all components are read.

— For writing a stream : (see `DADAStreamWriter`)

header0 : `DADAHeader`

Header for the first frame, holding time information, etc. Can instead give keyword arguments to construct a header (see `**kwargs`).

squeeze : bool, optional

If `True` (default), writer accepts squeezed arrays as input, and adds any dimensions of length unity.

****kwargs**

If the header is not given, an attempt will be made to construct one with any further keyword arguments.

— Header keywords : (see `fromvalues()`)

time : `Time`

Start time of the file.

samples_per_frame : int,

Number of complete samples per frame.

sample_rate : `Quantity`

Number of complete samples per second, i.e. the rate at which each channel of each polarization is sampled.

offset : Quantity or TimeDelta, optional

Time offset from the start of the whole observation (default: 0).

npol : int, optional

Number of polarizations (default: 1).

nchan : int, optional

Number of channels (default: 1).

complex_data : bool, optional

Whether data are complex (default: False).

bps : int, optional

Bits per elementary sample, i.e. per real or imaginary component for complex data (default: 8).

Returns

Filehandle

DADAFileReader or DADAFileWriter (binary), or DADAStreamReader or DADAStreamWriter (stream).

Notes

For streams, one can also pass in a list of files, or a template string that can be formatted using 'frame_nr', 'obs_offset', and other header keywords (by DADAFileNameSequencer).

For writing, one can mimic what is done at quite a few telescopes by using the template '{utc_start}_{obs_offset:016d}.000000.dada'.

For reading, to read series such as the above, use something like '2013-07-02-01:37:40_{obs_offset:016d}.000000.dada'. Note that here we have to pass in the date explicitly, since the template is used to get the first file name, before any header is read, and therefore the only keywords available are 'frame_nr', 'file_nr', and 'obs_offset', all of which are assumed to be zero for the first file. To avoid this restriction, pass in keyword arguments with values appropriate for the first file.

One may also pass in a sequentialfile object (opened in 'rb' mode for reading or 'w+b' for writing), though for typical use cases it is practically identical to passing in a list or template.

Classes

DADAFrame(header, payload[, valid, verify])	Representation of a DADA file, consisting of a header and payload.
DADAHeader(*args, **kwargs)	DADA baseband file format header.
DADAPayload(words[, header, sample_shape, …])	Container for decoding and encoding DADA payloads.

DADAFrame

class baseband.dada.**DADAFrame**(*header*, *payload*, *valid=True*, *verify=True*)

Bases: baseband.vlbi_base.frame.VLBIFrameBase

Representation of a DADA file, consisting of a header and payload.

Parameters

header : DADAHeader

Wrapper around the header lines, providing access to the values.

payload : DADAPayload

Wrapper around the payload, provding mechanisms to decode it.

valid : bool, optional

Whether the data are valid. Default: True.

verify : bool, optional

Whether to do basic verification of integrity. Default: True.

Notes

DADA files do not support storing whether data are valid or not on disk. Hence, this has to be determined independently. If `valid=False`, any decoded data are set to `cls.fill_value` (by default, 0).

The Frame can also be instantiated using class methods:

fromfile : read header and map or read payload from a filehandle

fromdata : encode data as payload

Of course, one can also do the opposite:

tofile : method to write header and payload to filehandle

data : property that yields full decoded payload

One can decode part of the payload by indexing or slicing the frame.

A number of properties are defined: `shape`, `dtype` and `size` are the shape, type and number of complete samples of the data array, and `nbytes` the frame size in bytes. Furthermore, the frame acts as a dictionary, with keys those of the header. Any attribute that is not defined on the frame itself, such as `.time` will be looked up on the header as well.

Attributes Summary

`data`	Full decoded frame.
`dtype`	Numeric type of the frame data.
`fill_value`	Value to replace invalid data in the frame.
`nbytes`	Size of the encoded frame in bytes.
`ndim`	Number of dimensions of the frame data.
`sample_shape`	Shape of a sample in the frame (nchan,).
`shape`	Shape of the frame data.
`size`	Total number of component samples in the frame data.
`valid`	Whether frame contains valid data.

Methods Summary

`fromdata`(data[, header, valid, verify])	Construct frame from data and header.

Continued on next page

Table 4 – continued from previous page

fromfile(fh[, memmap, valid, verify])	Read a frame from a filehandle, possible mapping the payload.
keys()	
tofile(fh)	Write encoded frame to filehandle.
verify()	Simple verification.

Attributes Documentation

data
Full decoded frame.

dtype
Numeric type of the frame data.

fill_value
Value to replace invalid data in the frame.

nbytes
Size of the encoded frame in bytes.

ndim
Number of dimensions of the frame data.

sample_shape
Shape of a sample in the frame (nchan,).

shape
Shape of the frame data.

size
Total number of component samples in the frame data.

valid
Whether frame contains valid data.

Methods Documentation

classmethod fromdata(*data*, *header=None*, *valid=True*, *verify=True*, ***kwargs*)
Construct frame from data and header.

Note that since DADA files are generally very large, one would normally map the file, and then set pieces of it by assigning to slices of the frame. See `memmap_frame`.

Parameters

data : `ndarray`

Array holding complex or real data to be encoded.

header : `DADAHeader` or None

If not given, will attempt to generate one using the keywords.

valid : bool, optional

Whether the data are valid (default: `True`). Note that this information cannot be written to disk.

verify : bool, optional

Whether or not to do basic assertions that check the integrity. Default: `True`.

****kwargs**

If `header` is not given, these are used to initialize one.

classmethod fromfile(*fh*, *memmap=True*, *valid=True*, *verify=True*)

Read a frame from a filehandle, possible mapping the payload.

Parameters

fh : filehandle

To read header from.

memmap : bool, optional

If `True` (default), use `memmap` to map the payload. If `False`, just read it from disk.

valid : bool, optional

Whether the data are valid (default: `True`). Note that this cannot be inferred from the header or payload itself. If `False`, any data read will be set to `cls.fill_value`.

verify : bool, optional

Whether to do basic verification of integrity. Default: `True`.

keys()

tofile(*fh*)

Write encoded frame to filehandle.

verify()

Simple verification. To be added to by subclasses.

DADAHeader

class baseband.dada.**DADAHeader**(**args*, ***kwargs*)

Bases: `collections.OrderedDict`

DADA baseband file format header.

Defines a number of routines common to all baseband format headers.

Parameters

***args** : str or iterable

If a string, parsed as a DADA header from a file, otherwise as for the OrderedDict baseclass.

verify : bool, optional

Whether to do minimal verification that the header is consistent with the DADA standard. Default: `True`.

mutable : bool, optional

Whether to allow the header to be changed after initialisation. Default: `True`.

****kwargs**

Any further header keywords to be set. If any value is a 2-item tuple, the second one will be considered a comment.

Notes

Like OrderedDict, in order to ensure keywords are kept in the right order, one should pass on values as a tuple, not as a dict. E.g., to copy a header, one should not do `DADAHeader(**header)`, but rather:

```
DADAHeader(((key, header[key]) for key in header))
```

or, to also keep the comments:

```
DADAHeader(((key, (header[key], header.comments[key]))
           for key in header))
```

Attributes Summary

bps	Bits per elementary sample.
complex_data	Whether the data are complex.
frame_nbytes	Size of the frame in bytes.
nbytes	Size of the header in bytes.
offset	Offset from start of observation in units of time.
payload_nbytes	Size of the payload in bytes.
sample_rate	Number of complete samples per second.
sample_shape	Shape of a sample in the payload (npol, nchan).
samples_per_frame	Number of complete samples in the frame.
sideband	True if upper sideband.
start_time	Start time of the observation.
time	Start time of the part of the observation covered by this header.

Methods Summary

clear()	
copy()	Create a mutable and independent copy of the header.
fromfile(fh[, verify])	Reads in DADA header block from a file.
fromkeys(*args, **kwargs)	Initialise a header from keyword values.
fromvalues(**kwargs)	Initialise a header from parsed values.
get(k[,d])	
has_key(k)	
items()	
iteritems()	od.iteritems -> an iterator over the (key, value) pairs in od
iterkeys()	
itervalues()	od.itervalues -> an iterator over the values in od
keys()	
pop(k[,d])	value.
popitem()	Pairs are returned in LIFO order if last is true or FIFO order if false.
setdefault(k[,d])	
tofile(fh)	Write DADA file header to filehandle.
update(**kwargs)	Update the header with new values.

Continued on next page

Table 6 – continued from previous page

values()	
verify()	Basic check of integrity.
viewitems()	
viewkeys()	
viewvalues()	

Attributes Documentation

bps
Bits per elementary sample.

complex_data
Whether the data are complex.

frame_nbytes
Size of the frame in bytes.

nbytes
Size of the header in bytes.

offset
Offset from start of observation in units of time.

payload_nbytes
Size of the payload in bytes.

sample_rate
Number of complete samples per second.

Can be set with a negative quantity to set sideband.

sample_shape
Shape of a sample in the payload (npol, nchan).

samples_per_frame
Number of complete samples in the frame.

sideband
True if upper sideband.

start_time
Start time of the observation.

time
Start time of the part of the observation covered by this header.

Methods Documentation

clear() → None. Remove all items from od.

copy()
Create a mutable and independent copy of the header.

classmethod fromfile(*fh*, *verify=True*)
Reads in DADA header block from a file.

The file pointer should be at the start.

Parameters

fh : filehandle

To read data from.

verify: bool, optional

Whether to do basic checks on whether the header is valid. Default: `True`.

classmethod fromkeys(**args*, ***kwargs*)

Initialise a header from keyword values.

Like fromvalues, but without any interpretation of keywords.

This just calls the class initializer; it is present for compatibility with other header classes only.

classmethod fromvalues(***kwargs*)

Initialise a header from parsed values.

Here, the parsed values must be given as keyword arguments, i.e., for any `header`, `cls.fromvalues(**header) == header`.

However, unlike for the `fromkeys` class method, data can also be set using arguments named after header methods, such as `time`.

Furthermore, some header defaults are set in `DADAHeader._defaults`.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

has_key(*k*) → True if D has a key k, else False

items() → list of (key, value) pairs in od

iteritems()

od.iteritems -> an iterator over the (key, value) pairs in od

iterkeys() → an iterator over the keys in od

itervalues()

od.itervalues -> an iterator over the values in od

keys() → list of keys in od

pop(*k*[, *d*]) → v, remove specified key and return the corresponding

value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), return and remove a (key, value) pair.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault(*k*[, *d*]) → od.get(k,d), also set od[k]=d if k not in od

tofile(*fh*)

Write DADA file header to filehandle.

Parts of the header beyond the ascii lines are filled with 0x00. Note that file should in principle be at the start, but we don't check for that since that would break SequentialFileWriter.

update(***kwargs*)

Update the header with new values.

Here, any keywords matching properties are processed as well, in the order set by the class (in `_properties`), and after all other keywords have been processed.

Parameters

verify : bool, optional

If `True` (default), verify integrity after updating.

****kwargs**

Arguments used to set keywords and properties.

values() → list of values in od

verify()

Basic check of integrity.

viewitems() → a set-like object providing a view on od's items

viewkeys() → a set-like object providing a view on od's keys

viewvalues() → an object providing a view on od's values

DADAPayload

class baseband.dada.**DADAPayload**(*words*, *header=None*, *sample_shape=()*, *bps=8*, *complex_data=False*)

Bases: `baseband.vlbi_base.payload.VLBIPayloadBase`

Container for decoding and encoding DADA payloads.

Parameters

words : `ndarray`

Array containg LSB unsigned words (with the right size) that encode the payload.

header : `DADAHeader`

Header that provides information about how the payload is encoded. If not given, the following arguments have to be passed in.

bps : int, optional

Number of bits per sample part (i.e., per channel and per real or imaginary component). Default: 8.

sample_shape : tuple, optional

Shape of the samples; e.g., (nchan,). Default: ().

complex_data : bool, optional

Whether data are complex. Default: `False`.

Attributes Summary

`data`	Full decoded payload.

Continued on next page

Table 7 – continued from previous page

dtype	Numeric type of the decoded data array.
nbytes	Size of the payload in bytes.
ndim	Number of dimensions of the decoded data array.
shape	Shape of the decoded data array.
size	Total number of component samples in the decoded data array.

Methods Summary

fromdata(data[, header, bps])	Encode data as a payload.
fromfile(fh[, header, memmap, payload_nbytes])	Read or map encoded data in file.
tofile(fh)	Write payload to filehandle.

Attributes Documentation

data
Full decoded payload.

dtype
Numeric type of the decoded data array.

nbytes
Size of the payload in bytes.

ndim
Number of dimensions of the decoded data array.

shape
Shape of the decoded data array.

size
Total number of component samples in the decoded data array.

Methods Documentation

classmethod fromdata(*data*, *header=None*, *bps=2*)
Encode data as a payload.

Parameters

data : ndarray

Data to be encoded. The last dimension is taken as the number of channels.

header : header instance, optional

If given, used to infer the bps.

bps : int, optional

Bits per elementary sample, i.e., per channel and per real or imaginary component, used if header is not given. Default: 2.

classmethod fromfile(*fh*, *header=None*, *memmap=False*, *payload_nbytes=None*, ***kwargs*)
Read or map encoded data in file.

Parameters

fh : filehandle

Handle to the file which will be read or mapped.

header : DADAHeader, optional

If given, used to infer payload_nbytes, bps, sample_shape, and complex_data. If not given, those have to be passed in.

memmap : bool, optional

If False (default), read from file. Otherwise, map the file in memory (see memmap).

payload_nbytes : int, optional

Number of bytes to read (default: as given in header, cls._nbytes, or, for mapping, to the end of the file).

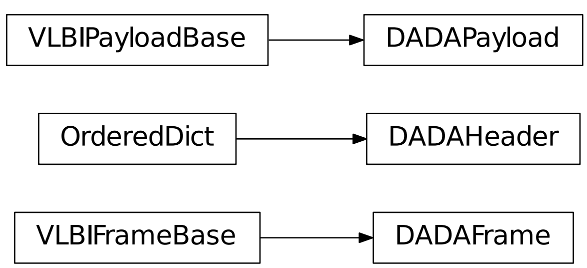
****kwargs**

Additional arguments are passed on to the class initializer. These are only needed if header is not given.

tofile(*fh*)

Write payload to filehandle.

Class Inheritance Diagram



7.2.2 baseband.dada.header Module

Definitions for DADA pulsar baseband headers.

Implements a DADAHeader class used to store header definitions in a FITS header, and read & write these from files.

Classes

DADAHeader(*args, **kwargs)	DADA baseband file format header.

DADAHeader

class baseband.dada.header.**DADAHeader**(**args*, ***kwargs*)

Bases: `collections.OrderedDict`

DADA baseband file format header.

Defines a number of routines common to all baseband format headers.

Parameters

***args** : str or iterable

If a string, parsed as a DADA header from a file, otherwise as for the OrderedDict baseclass.

verify : bool, optional

Whether to do minimal verification that the header is consistent with the DADA standard. Default: `True`.

mutable : bool, optional

Whether to allow the header to be changed after initialisation. Default: `True`.

****kwargs**

Any further header keywords to be set. If any value is a 2-item tuple, the second one will be considered a comment.

Notes

Like `OrderedDict`, in order to ensure keywords are kept in the right order, one should pass on values as a tuple, not as a dict. E.g., to copy a header, one should not do `DADAHeader(**header)`, but rather:

```
DADAHeader(((key, header[key]) for key in header))
```

or, to also keep the comments:

```
DADAHeader(((key, (header[key], header.comments[key]))
          for key in header))
```

Attributes Summary

`bps`	Bits per elementary sample.
`complex_data`	Whether the data are complex.
`frame_nbytes`	Size of the frame in bytes.
`nbytes`	Size of the header in bytes.
`offset`	Offset from start of observation in units of time.
`payload_nbytes`	Size of the payload in bytes.
`sample_rate`	Number of complete samples per second.
`sample_shape`	Shape of a sample in the payload (npol, nchan).
`samples_per_frame`	Number of complete samples in the frame.
`sideband`	True if upper sideband.
`start_time`	Start time of the observation.

Continued on next page

Table 10 – continued from previous page

time	Start time of the part of the observation covered by this header.

Methods Summary

clear()	
copy()	Create a mutable and independent copy of the header.
fromfile(fh[, verify])	Reads in DADA header block from a file.
fromkeys(*args, **kwargs)	Initialise a header from keyword values.
fromvalues(**kwargs)	Initialise a header from parsed values.
get(k[,d])	
has_key(k)	
items()	
iteritems()	od.iteritems -> an iterator over the (key, value) pairs in od
iterkeys()	
itervalues()	od.itervalues -> an iterator over the values in od
keys()	
pop(k[,d])	value.
popitem()	Pairs are returned in LIFO order if last is true or FIFO order if false.
setdefault(k[,d])	
tofile(fh)	Write DADA file header to filehandle.
update(**kwargs)	Update the header with new values.
values()	
verify()	Basic check of integrity.
viewitems()	
viewkeys()	
viewvalues()	

Attributes Documentation

bps
Bits per elementary sample.

complex_data
Whether the data are complex.

frame_nbytes
Size of the frame in bytes.

nbytes
Size of the header in bytes.

offset
Offset from start of observation in units of time.

payload_nbytes
Size of the payload in bytes.

sample_rate
Number of complete samples per second.

Can be set with a negative quantity to set sideband.

sample_shape
Shape of a sample in the payload (npol, nchan).

samples_per_frame
Number of complete samples in the frame.

sideband
True if upper sideband.

start_time
Start time of the observation.

time
Start time of the part of the observation covered by this header.

Methods Documentation

clear() → None. Remove all items from od.

copy()
Create a mutable and independent copy of the header.

classmethod **fromfile**(*fh*, *verify=True*)
Reads in DADA header block from a file.

The file pointer should be at the start.

Parameters
fh : filehandle

To read data from.

verify: bool, optional

Whether to do basic checks on whether the header is valid. Default: `True`.

classmethod **fromkeys**(**args*, ***kwargs*)
Initialise a header from keyword values.

Like fromvalues, but without any interpretation of keywords.

This just calls the class initializer; it is present for compatibility with other header classes only.

classmethod **fromvalues**(***kwargs*)
Initialise a header from parsed values.

Here, the parsed values must be given as keyword arguments, i.e., for any `header`, `cls.fromvalues(**header) == header`.

However, unlike for the `fromkeys` class method, data can also be set using arguments named after header methods, such as `time`.

Furthermore, some header defaults are set in `DADAHeader._defaults`.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

has_key(*k*) → True if D has a key k, else False

items() → list of (key, value) pairs in od

iteritems()
od.iteritems -> an iterator over the (key, value) pairs in od

iterkeys() → an iterator over the keys in od

itervalues()
od.itervalues -> an iterator over the values in od

keys() → list of keys in od

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), return and remove a (key, value) pair.
Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault(*k*[, *d*]) → od.get(k,d), also set od[k]=d if k not in od

tofile(*fh*)
Write DADA file header to filehandle.

Parts of the header beyond the ascii lines are filled with 0x00. Note that file should in principle be at the start, but we don't check for that since that would break SequentialFileWriter.

update(***kwargs*)
Update the header with new values.

Here, any keywords matching properties are processed as well, in the order set by the class (in `_properties`), and after all other keywords have been processed.

Parameters
verify : bool, optional

If `True` (default), verify integrity after updating.

****kwargs**

Arguments used to set keywords and properties.

values() → list of values in od

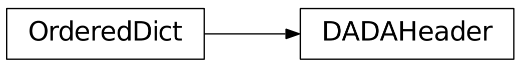
verify()
Basic check of integrity.

viewitems() → a set-like object providing a view on od's items

viewkeys() → a set-like object providing a view on od's keys

viewvalues() → an object providing a view on od's values

Class Inheritance Diagram



7.2.3 baseband.dada.payload Module

Payload for DADA format.

Classes

`DADAPayload`(words[, header, sample_shape, . . .])	Container for decoding and encoding DADA payloads.

DADAPayload

class baseband.dada.payload.**DADAPayload**(*words*, *header=None*, *sample_shape=()*, *bps=8*, *complex_data=False*)

Bases: `baseband.vlbi_base.payload.VLBIPayloadBase`

Container for decoding and encoding DADA payloads.

Parameters

words : `ndarray`

Array containg LSB unsigned words (with the right size) that encode the payload.

header : `DADAHeader`

Header that provides information about how the payload is encoded. If not given, the following arguments have to be passed in.

bps : int, optional

Number of bits per sample part (i.e., per channel and per real or imaginary component). Default: 8.

sample_shape : tuple, optional

Shape of the samples; e.g., (nchan,). Default: ().

complex_data : bool, optional

Whether data are complex. Default: `False`.

Attributes Summary

`data`	Full decoded payload.

Continued on next page

Table 13 – continued from previous page

dtype	Numeric type of the decoded data array.
nbytes	Size of the payload in bytes.
ndim	Number of dimensions of the decoded data array.
shape	Shape of the decoded data array.
size	Total number of component samples in the decoded data array.

Methods Summary

fromdata(data[, header, bps])	Encode data as a payload.
fromfile(fh[, header, memmap, payload_nbytes])	Read or map encoded data in file.
tofile(fh)	Write payload to filehandle.

Attributes Documentation

data
Full decoded payload.

dtype
Numeric type of the decoded data array.

nbytes
Size of the payload in bytes.

ndim
Number of dimensions of the decoded data array.

shape
Shape of the decoded data array.

size
Total number of component samples in the decoded data array.

Methods Documentation

classmethod fromdata(*data*, *header=None*, *bps=2*)
Encode data as a payload.

Parameters

data : ndarray

Data to be encoded. The last dimension is taken as the number of channels.

header : header instance, optional

If given, used to infer the bps.

bps : int, optional

Bits per elementary sample, i.e., per channel and per real or imaginary component, used if header is not given. Default: 2.

classmethod fromfile(*fh*, *header=None*, *memmap=False*, *payload_nbytes=None*, ***kwargs*)
Read or map encoded data in file.

Parameters

fh : filehandle

Handle to the file which will be read or mapped.

header : DADAHeader, optional

If given, used to infer payload_nbytes, bps, sample_shape, and complex_data. If not given, those have to be passed in.

memmap : bool, optional

If False (default), read from file. Otherwise, map the file in memory (see memmap).

payload_nbytes : int, optional

Number of bytes to read (default: as given in header, cls._nbytes, or, for mapping, to the end of the file).

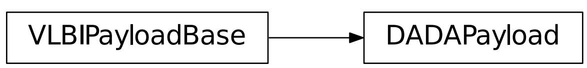
****kwargs**

Additional arguments are passed on to the class initializer. These are only needed if header is not given.

tofile(*fh*)

Write payload to filehandle.

Class Inheritance Diagram



7.2.4 baseband.dada.frame Module

Classes

DADAFrame(header, payload[, valid, verify])	Representation of a DADA file, consisting of a header and payload.

DADAFrame

class baseband.dada.frame.**DADAFrame**(*header*, *payload*, *valid=True*, *verify=True*)

Bases: baseband.vlbi_base.frame.VLBIFrameBase

Representation of a DADA file, consisting of a header and payload.

Parameters

header : DADAHeader

Wrapper around the header lines, providing access to the values.

payload : DADAPayload

Wrapper around the payload, provding mechanisms to decode it.

valid : bool, optional

Whether the data are valid. Default: True.

verify : bool, optional

Whether to do basic verification of integrity. Default: True.

Notes

DADA files do not support storing whether data are valid or not on disk. Hence, this has to be determined independently. If `valid=False`, any decoded data are set to `cls.fill_value` (by default, 0).

The Frame can also be instantiated using class methods:

fromfile : read header and map or read payload from a filehandle

fromdata : encode data as payload

Of course, one can also do the opposite:

tofile : method to write header and payload to filehandle

data : property that yields full decoded payload

One can decode part of the payload by indexing or slicing the frame.

A number of properties are defined: `shape`, `dtype` and `size` are the shape, type and number of complete samples of the data array, and `nbytes` the frame size in bytes. Furthermore, the frame acts as a dictionary, with keys those of the header. Any attribute that is not defined on the frame itself, such as `.time` will be looked up on the header as well.

Attributes Summary

data	Full decoded frame.
dtype	Numeric type of the frame data.
fill_value	Value to replace invalid data in the frame.
nbytes	Size of the encoded frame in bytes.
ndim	Number of dimensions of the frame data.
sample_shape	Shape of a sample in the frame (nchan,).
shape	Shape of the frame data.
size	Total number of component samples in the frame data.
valid	Whether frame contains valid data.

Methods Summary

fromdata(data[, header, valid, verify])	Construct frame from data and header.
fromfile(fh[, memmap, valid, verify])	Read a frame from a filehandle, possible mapping the payload.
keys()	

Continued on next page

Table 17 – continued from previous page

tofile(fh)	Write encoded frame to filehandle.
verify()	Simple verification.

Attributes Documentation

data
Full decoded frame.

dtype
Numeric type of the frame data.

fill_value
Value to replace invalid data in the frame.

nbytes
Size of the encoded frame in bytes.

ndim
Number of dimensions of the frame data.

sample_shape
Shape of a sample in the frame (nchan,).

shape
Shape of the frame data.

size
Total number of component samples in the frame data.

valid
Whether frame contains valid data.

Methods Documentation

classmethod fromdata(*data*, *header=None*, *valid=True*, *verify=True*, ***kwargs*)
Construct frame from data and header.

Note that since DADA files are generally very large, one would normally map the file, and then set pieces of it by assigning to slices of the frame. See `memmap_frame`.

Parameters

data : `ndarray`

Array holding complex or real data to be encoded.

header : `DADAHeader` or None

If not given, will attempt to generate one using the keywords.

valid : bool, optional

Whether the data are valid (default: `True`). Note that this information cannot be written to disk.

verify : bool, optional

Whether or not to do basic assertions that check the integrity. Default: `True`.

****kwargs**

If `header` is not given, these are used to initialize one.

classmethod fromfile(*fh*, *memmap=True*, *valid=True*, *verify=True*)

Read a frame from a filehandle, possible mapping the payload.

Parameters

fh : filehandle

To read header from.

memmap : bool, optional

If `True` (default), use `memmap` to map the payload. If `False`, just read it from disk.

valid : bool, optional

Whether the data are valid (default: `True`). Note that this cannot be inferred from the header or payload itself. If `False`, any data read will be set to `cls.fill_value`.

verify : bool, optional

Whether to do basic verification of integrity. Default: `True`.

keys()

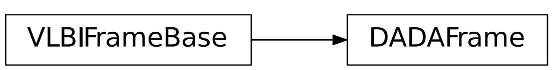
tofile(*fh*)

Write encoded frame to filehandle.

verify()

Simple verification. To be added to by subclasses.

Class Inheritance Diagram



7.2.5 baseband.dada.base Module

Functions

open(name[, mode])	Open DADA file for reading or writing.

open

baseband.dada.base.**open**(*name*, *mode=u'rs'*, ***kwargs*)

Open DADA file for reading or writing.

Opened as a binary file, one gets a wrapped filehandle that adds methods to read/write a frame. Opened as a stream, the handle is wrapped further, with methods such as reading and writing to the file as if it were a stream of samples.

Parameters

name : str or filehandle

File name or handle.

mode : {'rb', 'wb', 'rs', or 'ws'}, optional

Whether to open for reading or writing, and as a regular binary file or as a stream. Default: 'rs', for reading a stream.

****kwargs**

Additional arguments when opening the file as a stream.

— For reading a stream : (see `DADAStreamReader`)

squeeze : bool, optional

If `True` (default), remove any dimensions of length unity from decoded data.

subset : indexing object or tuple of objects, optional

Specific components of the complete sample to decode (after possibly squeezing). If a single indexing object is passed, it selects polarizations. With a tuple, the first selects polarizations and the second selects channels. If the tuple is empty (default), all components are read.

— For writing a stream : (see `DADAStreamWriter`)

header0 : `DADAHeader`

Header for the first frame, holding time information, etc. Can instead give keyword arguments to construct a header (see `**kwargs`).

squeeze : bool, optional

If `True` (default), writer accepts squeezed arrays as input, and adds any dimensions of length unity.

****kwargs**

If the header is not given, an attempt will be made to construct one with any further keyword arguments.

— Header keywords : (see `fromvalues()`)

time : `Time`

Start time of the file.

samples_per_frame : int,

Number of complete samples per frame.

sample_rate : `Quantity`

Number of complete samples per second, i.e. the rate at which each channel of each polarization is sampled.

offset : `Quantity` or `TimeDelta`, optional

Time offset from the start of the whole observation (default: 0).

npol : int, optional

Number of polarizations (default: 1).

nchan : int, optional

Number of channels (default: 1).

complex_data : bool, optional

Whether data are complex (default: `False`).

bps : int, optional

Bits per elementary sample, i.e. per real or imaginary component for complex data (default: 8).

Returns

Filehandle

`DADAFileReader` or `DADAFileWriter` (binary), or `DADAStreamReader` or `DADAStreamWriter` (stream).

Notes

For streams, one can also pass in a list of files, or a template string that can be formatted using 'frame_nr', 'obs_offset', and other header keywords (by `DADAFileNameSequencer`).

For writing, one can mimic what is done at quite a few telescopes by using the template '{utc_start}_{obs_offset:016d}.000000.dada'.

For reading, to read series such as the above, use something like '2013-07-02-01:37:40_{obs_offset:016d}.000000.dada'. Note that here we have to pass in the date explicitly, since the template is used to get the first file name, before any header is read, and therefore the only keywords available are 'frame_nr', 'file_nr', and 'obs_offset', all of which are assumed to be zero for the first file. To avoid this restriction, pass in keyword arguments with values appropriate for the first file.

One may also pass in a `sequentialfile` object (opened in 'rb' mode for reading or 'w+b' for writing), though for typical use cases it is practically identical to passing in a list or template.

Classes

`DADAFileNameSequencer`(template, header)	List-like generator of filenames using a template.
`DADAFileReader`(fh_raw)	Simple reader for DADA files.
`DADAFileWriter`(fh_raw)	Simple writer/mapper for DADA files.
`DADAStreamBase`(fh_raw, header0[, squeeze, . . .])	Base for DADA streams.
`DADAStreamReader`(fh_raw[, squeeze, subset, . . .])	DADA format reader.
`DADAStreamWriter`(fh_raw, header0[, squeeze])	DADA format writer.

DADAFileNameSequencer

class baseband.dada.base.**DADAFileNameSequencer**(*template*, *header*)

List-like generator of filenames using a template.

The template is formatted, filling in any items in curly brackets with values from the header, as well as possibly a file number equal to the indexing value, indicated with '{file_nr}'. The value '{obs_offset}' is treated specially, in being calculated using `header['OBS_OFFSET'] + file_nr * header['FILE_SIZE']`, where `header['FILE_SIZE']` is the file size in bytes.

The length of the instance will be the number of files that exist that match the template for increasing values of the file number.

Parameters

template : str

Template to format to get specific filenames.

header : dict-like

Structure holding key'd values that are used to fill in the format. Keys must be in all caps (eg. DATE), as with DADA header keys.

Examples

```
>>> from baseband import dada
>>> dfs = dada.base.DADAFileNameSequencer('a{file_nr:03d}.dada', {})
>>> dfs[10]
'a010.dada'
>>> dfs = dada.base.DADAFileNameSequencer(
...     '{date}_{file_nr:03d}.dada', {'DATE': "2018-01-01"})
>>> dfs[10]
'2018-01-01_010.dada'
>>> from baseband.data import SAMPLE_DADA
>>> with open(SAMPLE_DADA, 'rb') as fh:
...     header = dada.DADAHeader.fromfile(fh)
>>> template = '{utc_start}.{obs_offset:016d}.000000.dada'
>>> dfs = DADAFileNameSequencer(template, header)
>>> dfs[0]
'2013-07-02-01:37:40.0000006400000000.000000.dada'
>>> dfs[1]
'2013-07-02-01:37:40.0000006400064000.000000.dada'
>>> dfs[10]
'2013-07-02-01:37:40.0000006400640000.000000.dada'
```

DADAFileReader

class baseband.dada.base.**DADAFileReader**(*fh_raw*)

Bases: baseband.vlbi_base.base.VLBIFileReaderBase

Simple reader for DADA files.

Wraps a binary filehandle, providing methods to help interpret the data, such as read_frame and get_frame_rate. By default, frame payloads are mapped rather than fully read into physical memory.

Parameters

fh_raw : filehandle

Filehandle of the raw binary data file.

Attributes Summary

info	Standardized information on file readers.

Methods Summary

close()	
get_frame_rate()	Determine the number of frames per second.

Continued on next page

Table 21 – continued from previous page

read_frame([memmap, verify])	Read the frame header and read or map the corresponding payload.
read_header()	Read a single header from the file.

Attributes Documentation

info

Standardized information on file readers.

The info descriptor has a number of standard attributes, which are determined from arguments passed in opening the file, from the first header (info.header0) and from possibly scanning the file to determine the duration of frames.

Examples

The most common use is simply to print information:

```
>>> from baseband.data import SAMPLE_MARK5B
>>> from baseband import mark5b
>>> fh = mark5b.open(SAMPLE_MARK5B, 'rb')
>>> fh.info
File information:
format = mark5b
frame_rate = 6400.0 Hz
bps = 2
complex_data = False

missing:  nchan: needed to determine sample shape and rate.
          kday, ref_time: needed to infer full times.

>>> fh.close()

>>> fh = mark5b.open(SAMPLE_MARK5B, 'rb', kday=56000, nchan=8)
>>> fh.info
File information:
format = mark5b
frame_rate = 6400.0 Hz
sample_rate = 32.0 MHz
samples_per_frame = 5000
sample_shape = (8,)
bps = 2
complex_data = False
start_time = 2014-06-13T05:30:01.000000000
>>> fh.close()
```

Attributes

format	(str or None) File format, or None if the underlying file cannot be parsed.
frame_rate	(Quantity) Number of data frames per unit of time.
sample_rate	(Quantity) Complete samples per unit of time.
samples_per_frame	(int) Number of complete samples in each frame.
sample_shape	(tuple) Dimensions of each complete sample (e.g., (nchan,)).
bps	(int) Number of bits used to encode each elementary sample.
complex_data	(bool) Whether the data are complex.
start_time	(Time) Time of the first complete sample.
missing	(dict) Entries are keyed by names of arguments that should be passed to the file reader to obtain full information. The associated entries explain why these arguments are needed.

Methods Documentation

close()

get_frame_rate()

Determine the number of frames per second.

The routine uses the sample rate and number of samples per frame from the first header in the file.

Returns

frame_rate : Quantity

Frames per second.

read_frame(*memmap=True*, *verify=True*)

Read the frame header and read or map the corresponding payload.

Parameters

memmap : bool, optional

If True (default), map the payload using memmap, so that parts are only loaded into memory as needed to access data.

verify : bool, optional

Whether to do basic checks of frame integrity. Default: True.

Returns

frame : DADAFrame

With .header and .payload properties. The .data property returns all data encoded in the frame. Since this may be too large to fit in memory, it may be better to access the parts of interest by slicing the frame.

read_header()

Read a single header from the file.

Returns

header : DADAHeader

DADAFileWriter

class baseband.dada.base.**DADAFileWriter**(*fh_raw*)

Bases: baseband.vlbi_base.base.VLBIFileBase

Simple writer/mapper for DADA files.

Adds write_frame and memmap_frame methods to the VLBI binary file wrapper. The latter allows one to encode data in pieces, writing to disk as needed.

Methods Summary

close()	
memmap_frame([header])	Get frame by writing the header to disk and mapping its payload.
write_frame(data[, header])	Write a single frame (header plus payload).

Methods Documentation

close()

memmap_frame(*header=None*, ***kwargs*)

Get frame by writing the header to disk and mapping its payload.

The header is written to disk immediately, but the payload is mapped, so that it can be filled in pieces, by setting slices of the frame.

Parameters

header : DADAHeader

Written to disk immediately. Can instead give keyword arguments to construct a header.

****kwargs**

If header is not given, these are used to initialize one.

Returns

frame: DADAFrame

By assigning slices to data, the payload can be encoded piecewise.

write_frame(*data*, *header=None*, ***kwargs*)

Write a single frame (header plus payload).

Parameters

data : ndarray or DADAFrame

If an array, a header should be given, which will be used to get the information needed to encode the array, and to construct the DADA frame.

header : DADAHeader

Can instead give keyword arguments to construct a header. Ignored if data is a DADAFrame instance.

****kwargs**

If header is not given, these are used to initialize one.

DADAStreamBase

class baseband.dada.base.**DADAStreamBase**(*fh_raw*, *header0*, *squeeze=True*, *subset=()*, *verify=True*)

Bases: baseband.vlbi_base.base.VLBIStreamBase

Base for DADA streams.

Attributes Summary

bps	Bits per elementary sample.
complex_data	Whether the data are complex.
header0	First header of the file.
sample_rate	Number of complete samples per second.
sample_shape	Shape of a complete sample (possibly subset or squeezed).
samples_per_frame	Number of complete samples per frame.
squeeze	Whether data arrays have dimensions with length unity removed.
start_time	Start time of the file.
subset	Specific components of the complete sample to decode.
time	Time of the sample pointer's current offset in file.
verify	Whether to do consistency checks on frames being read.

Methods Summary

close()	
tell([unit])	Current offset in the file.

Attributes Documentation

bps
Bits per elementary sample.

complex_data
Whether the data are complex.

header0
First header of the file.

sample_rate
Number of complete samples per second.

sample_shape
Shape of a complete sample (possibly subset or squeezed).

samples_per_frame
Number of complete samples per frame.

squeeze
Whether data arrays have dimensions with length unity removed.

If `True`, data read out has such dimensions removed, and data passed in for writing has them inserted.

start_time
Start time of the file.

See also `time` for the time of the sample pointer's current offset, and (if available) `stop_time` for the time at the end of the file.

subset
Specific components of the complete sample to decode.

The order of dimensions is the same as for `sample_shape`. Set by the class initializer.

time
Time of the sample pointer's current offset in file.

See also `start_time` for the start time, and (if available) `stop_time` for the end time, of the file.

verify
Whether to do consistency checks on frames being read.

Methods Documentation

close()

tell(*unit=None*)
Current offset in the file.

Parameters
unit : `Unit` or str, optional

Time unit the offset should be returned in. By default, no unit is used, i.e., an integer enumerating samples is returned. For the special string 'time', the absolute time is calculated.

Returns
offset : int, `Quantity`, or `Time`

Offset in current file (or time at current position).

DADAStreamReader

class baseband.dada.base.**DADAStreamReader**(*fh_raw*, *squeeze=True*, *subset=()*, *verify=True*)
Bases: `baseband.dada.base.DADAStreamBase`, `baseband.vlbi_base.base.VLBIStreamReaderBase`

DADA format reader.

Allows access to DADA files as a continuous series of samples.

Parameters
fh_raw : filehandle

Filehandle of the raw DADA stream.

squeeze : bool, optional

If `True` (default), remove any dimensions of length unity from decoded data.

subset : indexing object or tuple of objects, optional

Specific components of the complete sample to decode (after possibly squeezing). If a single indexing object is passed, it selects polarizations. With a tuple, the first selects polarizations and the second selects channels. If the tuple is empty (default), all components are read.

verify : bool, optional

Whether to do basic checks of frame integrity when reading. The first frame of the stream is always checked, so `verify` is effective only when reading sequences of files. Default: `True`.

Attributes Summary

`bps`	Bits per elementary sample.
`complex_data`	Whether the data are complex.
`dtype`	
`fill_value`	Value to use for invalid or missing data.
`header0`	First header of the file.
`info`	Standardized information on stream readers.
`ndim`	Number of dimensions of the (squeezed/subset) stream data.
`sample_rate`	Number of complete samples per second.
`sample_shape`	Shape of a complete sample (possibly subset or squeezed).
`samples_per_frame`	Number of complete samples per frame.
`shape`	Shape of the (squeezed/subset) stream data.
`size`	Total number of component samples in the (squeezed/subset) stream data.
`squeeze`	Whether data arrays have dimensions with length unity removed.
`start_time`	Start time of the file.
`stop_time`	Time at the end of the file, just after the last sample.
`subset`	Specific components of the complete sample to decode.
`time`	Time of the sample pointer's current offset in file.
`verify`	Whether to do consistency checks on frames being read.

Methods Summary

`close`()	
`read`([count, out])	Read a number of complete (or subset) samples.
`seek`(offset[, whence])	Change the stream position.
`tell`([unit])	Current offset in the file.

Attributes Documentation

bps

Bits per elementary sample.

complex_data

Whether the data are complex.

dtype

fill_value
Value to use for invalid or missing data. Default: 0.

header0
First header of the file.

info
Standardized information on stream readers.

The `info` descriptor provides a few standard attributes, all of which can also be accessed directly on the stream filehandle. More detailed information on the underlying file is stored in its info, accessible via `info.file_info`.

Attributes

start_time	(`Time`) Time of the first complete sample.
stop_time	(`Time`) Time of the complete sample just beyond the end of the file.
sample_rate	(`Quantity`) Complete samples per unit of time.
shape	(tuple) Equivalent shape of the whole file, i.e., combining the number of complete samples and the shape of those samples.
bps	(int) Number of bits used to encode each elementary sample.
complex_data	(bool) Whether the data are complex.

ndim
Number of dimensions of the (squeezed/subset) stream data.

sample_rate
Number of complete samples per second.

sample_shape
Shape of a complete sample (possibly subset or squeezed).

samples_per_frame
Number of complete samples per frame.

shape
Shape of the (squeezed/subset) stream data.

size
Total number of component samples in the (squeezed/subset) stream data.

squeeze
Whether data arrays have dimensions with length unity removed.

If `True`, data read out has such dimensions removed, and data passed in for writing has them inserted.

start_time
Start time of the file.

See also `time` for the time of the sample pointer's current offset, and (if available) `stop_time` for the time at the end of the file.

stop_time
Time at the end of the file, just after the last sample.

See also `start_time` for the start time of the file, and `time` for the time of the sample pointer's current offset.

subset
Specific components of the complete sample to decode.

The order of dimensions is the same as for `sample_shape`. Set by the class initializer.

time
Time of the sample pointer's current offset in file.

See also `start_time` for the start time, and (if available) `stop_time` for the end time, of the file.

verify
Whether to do consistency checks on frames being read.

Methods Documentation

close()

read(*count=None*, *out=None*)
Read a number of complete (or subset) samples.

The range retrieved can span multiple frames.

Parameters

count : int or None, optional

Number of complete/subset samples to read. If `None` (default) or negative, the whole file is read. Ignored if `out` is given.

out : None or array, optional

Array to store the data in. If given, `count` will be inferred from the first dimension; the other dimension should equal `sample_shape`.

Returns

out : `ndarray` of float or complex

The first dimension is sample-time, and the remainder given by `sample_shape`.

seek(*offset*, *whence=0*)
Change the stream position.

This works like a normal filehandle seek, but the offset is in samples (or a relative or absolute time).

Parameters

offset : int, `Quantity`, or `Time`

Offset to move to. Can be an (integer) number of samples, an offset in time units, or an absolute time.

whence : {0, 1, 2, 'start', 'current', or 'end'}, optional

Like regular seek, the offset is taken to be from the start if `whence=0` (default), from the current position if 1, and from the end if 2. One can alternativey use 'start', 'current', or 'end' for 0, 1, or 2, respectively. Ignored if `offset` is a time.

tell(*unit=None*)
Current offset in the file.

Parameters
unit : Unit or str, optional

Time unit the offset should be returned in. By default, no unit is used, i.e., an integer enumerating samples is returned. For the special string 'time', the absolute time is calculated.

Returns
offset : int, Quantity, or Time

Offset in current file (or time at current position).

DADAStreamWriter

class baseband.dada.base.**DADAStreamWriter**(*fh_raw*, *header0*, *squeeze=True*)
Bases: baseband.dada.base.DADAStreamBase, baseband.vlbi_base.base.VLBIStreamWriterBase

DADA format writer.

Encodes and writes sequences of samples to file.

Parameters
raw : filehandle

For writing the header and raw data to storage.

header0 : DADAHeader

Header for the first frame, holding time information, etc.

squeeze : bool, optional

If True (default), write accepts squeezed arrays as input, and adds any dimensions of length unity.

Attributes Summary

bps	Bits per elementary sample.
complex_data	Whether the data are complex.
header0	First header of the file.
sample_rate	Number of complete samples per second.
sample_shape	Shape of a complete sample (possibly subset or squeezed).
samples_per_frame	Number of complete samples per frame.
squeeze	Whether data arrays have dimensions with length unity removed.
start_time	Start time of the file.
subset	Specific components of the complete sample to decode.
time	Time of the sample pointer's current offset in file.
verify	Whether to do consistency checks on frames being read.

Methods Summary

close()	
tell([unit])	Current offset in the file.
write(data[, valid])	Write data, buffering by frames as needed.

Attributes Documentation

bps
Bits per elementary sample.

complex_data
Whether the data are complex.

header0
First header of the file.

sample_rate
Number of complete samples per second.

sample_shape
Shape of a complete sample (possibly subset or squeezed).

samples_per_frame
Number of complete samples per frame.

squeeze
Whether data arrays have dimensions with length unity removed.

If `True`, data read out has such dimensions removed, and data passed in for writing has them inserted.

start_time
Start time of the file.

See also `time` for the time of the sample pointer's current offset, and (if available) `stop_time` for the time at the end of the file.

subset
Specific components of the complete sample to decode.

The order of dimensions is the same as for `sample_shape`. Set by the class initializer.

time
Time of the sample pointer's current offset in file.

See also `start_time` for the start time, and (if available) `stop_time` for the end time, of the file.

verify
Whether to do consistency checks on frames being read.

Methods Documentation

close()

tell(*unit=None*)
Current offset in the file.

Parameters

unit : `Unit` or str, optional

Time unit the offset should be returned in. By default, no unit is used, i.e., an integer enumerating samples is returned. For the special string 'time', the absolute time is calculated.

Returns

offset : int, `Quantity`, or `Time`

Offset in current file (or time at current position).

write(*data*, *valid=True*)

Write data, buffering by frames as needed.

Parameters

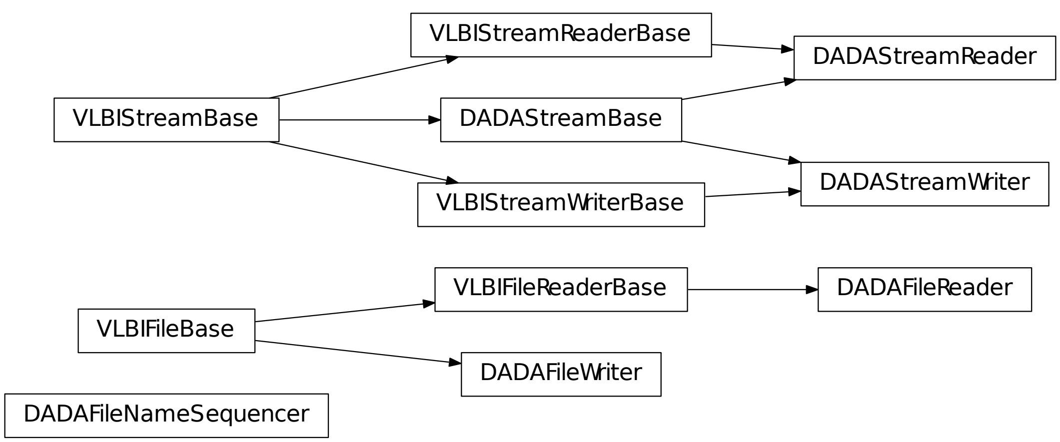
data : `ndarray`

Piece of data to be written, with sample dimensions as given by `sample_shape`. This should be properly scaled to make best use of the dynamic range delivered by the encoding.

valid : bool, optional

Whether the current data are valid. Default: `True`.

Class Inheritance Diagram



CHAPTER 8

GUPPI

The GUPPI format is the output of the Green Bank Ultimate Pulsar Processing Instrument and any clones operating at other telescopes, such as PUPPI at the Arecibo Observatory. Baseband specifically supports GUPPI data **taken in baseband mode**, and is based off of DSPSR's implementation. While general format specifications can be found at the SERA Project and on Paul Demorest's site, some of the header information could be invalid or not applicable, particularly with older files.

Baseband currently only supports 8-bit *elementary samples*.

8.1 File Structure

Each GUPPI file contains multiple (typically 128) *frames*, with each frame consisting of an ASCII *header* composed of 80-character entries, followed by a binary *payload* (or "block"). The header's length is variable, but always ends with "END" followed by 77 spaces.

How samples are stored in the payload depends on whether or not it is **channels-first**. A channels-first payload stores each channel's *stream* in a contiguous data block, while a non-channels-first one groups the *components* of a *complete sample* together (like with other formats). In either case, for each channel polarization samples from the same point in time are stored adjacent to one another. At the end of each channel's data is a section of **overlap samples** identical to the first samples in the next payload. Baseband retains these redundant samples when reading individual GUPPI frames, but removes them when reading files as a stream.

8.2 Usage

This section covers reading and writing GUPPI files with Baseband; general usage is covered in the *Getting Started* section. For situations in which one is unsure of a file's format, Baseband features the general `baseband.open` and `baseband.file_info` functions, which are also discussed in *Getting Started*. The examples below use the sample PUPPI file `baseband/data/sample_puppi.raw`, and the the `astropy.units` and `baseband.guppi` modules:

```
>>> from baseband import guppi
>>> import astropy.units as u
>>> from baseband.data import SAMPLE_PUPPI
```

Single files can be opened with `open` in binary mode, which provides a normal file reader, but extended with methods to read a `GUPPIFrame`:

```
>>> fb = guppi.open(SAMPLE_PUPPI, 'rb')
>>> frame = fb.read_frame()
>>> frame.shape
(1024, 2, 4)
>>> frame[:3, 0, 1]
array([-32.-10.j, -15.-14.j,   9.-13.j], dtype=complex64)
>>> fb.close()
```

Since the files can be quite large, the payload is mapped (with `numpy.memmap`), so that if one accesses part of the data, only the corresponding parts of the encoded payload are loaded into memory (since the sample file is encoded using 8 bits, the above example thus loads 6 bytes into memory).

Opening in stream mode wraps the low-level routines such that reading and writing is in units of samples, and provides access to header information:

```
>>> fh = guppi.open(SAMPLE_PUPPI, 'rs')
>>> fh
<GUPPIStreamReader name=... offset=0
    sample_rate=250.0 Hz, samples_per_frame=960,
    sample_shape=SampleShape(npol=2, nchan=4), bps=8,
    start_time=2018-01-14T14:11:33.000>
>>> d = fh.read()
>>> d.shape
(3840, 2, 4)
>>> d[:3, 0, 1]
array([-32.-10.j, -15.-14.j,   9.-13.j], dtype=complex64)
>>> fh.close()
```

Note that `fh.samples_per_frame` represents the number of samples per frame **excluding overlap samples**, since the stream reader works on a linearly increasing sequence of samples. Frames themselves have access to the overlap, and `fh.header0.samples_per_frame` returns the number of samples per frame including overlap.

To set up a file for writing as a stream is possible as well. Overlap must be zero when writing (so we set `samples_per_frame` to its stream reader value from above):

```
>>> from astropy.time import Time
>>> files = ['puppi_test.000{i}.raw'.format(i=i) for i in range(2)]
>>> fw = guppi.open(files, 'ws', frames_per_file=2, sample_rate=250*u.Hz,
...                 samples_per_frame=960, pktsize=1024,
...                 time=Time(58132.59135416667, format='mjd'),
...                 npol=2, nchan=4)
>>> fw.write(d)
>>> fw.close()
>>> fr = guppi.open(files, 'rs')
>>> d2 = fr.read()
>>> (d == d2).all()
True
>>> fr.close()
```

Here we show how we can write to a sequence of files. One may pass a time-ordered list or tuple of filenames to `open`, which then uses `sequentialfile.open` to read or write to them as a single contiguous file. Unlike when writing

DADA files, which have one frame per file, we must specify the number of frames in one file. Note that typically one does not have to pass `PKTSIZE`, the UDP data packet size (set by the observing mode), but the sample file has small enough frames that the default of 8192 bytes is too large. Baseband only uses `PKTSIZE` to double-check the sample offset of the frame, so `PKTSIZE` must be set to a value such each payload, excluding overlap samples, contains an integer number of packets.

8.3 Reference/API

8.3.1 baseband.guppi Package

Green Bank Ultimate Pulsar Processing Instrument (GUPPI) format reader/writer.

Functions

`open`(name[, mode])	Open GUPPI file for reading or writing.

open

`baseband.guppi.open`(*name*, *mode=u'rs'*, ***kwargs*)

Open GUPPI file for reading or writing.

Opened as a binary file, one gets a wrapped filehandle that adds methods to read/write a frame. Opened as a stream, the handle is wrapped further, with methods such as reading and writing to the file as if it were a stream of samples.

Parameters

name : str or filehandle

File name or handle.

mode : {'rb', 'wb', 'rs', or 'ws'}, optional

Whether to open for reading or writing, and as a regular binary file or as a stream. Default: 'rs', for reading a stream.

****kwargs**

Additional arguments when opening the file as a stream.

— For reading a stream : (see `GUPPIStreamReader`)

squeeze : bool, optional

If `True` (default), remove any dimensions of length unity from decoded data.

subset : indexing object or tuple of objects, optional

Specific components of the complete sample to decode (after possibly squeezing). If a single indexing object is passed, it selects polarizations. With a tuple, the first selects polarizations and the second selects channels. If the tuple is empty (default), all components are read.

— For writing a stream : (see `GUPPIStreamWriter`)

header0 : `GUPPIHeader`

Header for the first frame, holding time information, etc. Can instead give keyword arguments to construct a header (see **kwargs).

squeeze : bool, optional

If True (default), writer accepts squeezed arrays as input, and adds any dimensions of length unity.

frames_per_file : int, optional

When writing to a sequence of files, sets the number of frames within each file. Default: 128.

****kwargs**

If the header is not given, an attempt will be made to construct one with any further keyword arguments.

— Header keywords : (see fromvalues())

time : Time

Start time of the file. Must have an integer number of seconds.

sample_rate : Quantity

Number of complete samples per second, i.e. the rate at which each channel of each polarization is sampled.

samples_per_frame : int

Number of complete samples per frame. Can alternatively give payload_nbytes.

payload_nbytes : int

Number of bytes per payload. Can alternatively give samples_per_frame.

offset : Quantity or TimeDelta, optional

Time offset from the start of the whole observation (default: 0).

npol : int, optional

Number of polarizations (default: 1).

nchan : int, optional

Number of channels (default: 1). For GUPPI, complex data is only allowed when nchan > 1.

bps : int, optional

Bits per elementary sample, i.e. per real or imaginary component for complex data (default: 8).

Returns

Filehandle

GUPPIFileReader or GUPPIFileWriter (binary), or GUPPIStreamReader or GUPPIStreamWriter (stream).

Notes

For streams, one can also pass in a list of files, or equivalently a sequentialfile object (opened in 'rb' mode for reading or 'w+b' for writing).

Classes

GUPPIFrame(header, payload[, valid, verify])	Representation of a GUPPI file, consisting of a header and payload.
GUPPIHeader(*args, **kwargs)	GUPPI baseband file format header.
GUPPIPayload(words[, header, sample_shape, ...])	Container for decoding and encoding GUPPI payloads.

GUPPIFrame

class baseband.guppi.**GUPPIFrame**(*header*, *payload*, *valid=True*, *verify=True*)

Bases: baseband.vlbi_base.frame.VLBIFrameBase

Representation of a GUPPI file, consisting of a header and payload.

Parameters

header : GUPPIHeader

Wrapper around the header lines, providing access to the values.

payload : GUPPIPayload

Wrapper around the payload, provding mechanisms to decode it.

valid : bool, optional

Whether the data are valid. Default: True.

verify : bool, optional

Whether to do basic verification of integrity. Default: True.

Notes

GUPPI files do not support storing whether data are valid or not on disk. Hence, this has to be determined independently. If `valid=False`, any decoded data are set to `cls.fill_value` (by default, 0).

The Frame can also be instantiated using class methods:

fromfile : read header and and map or read payload from a filehandle

fromdata : encode data as payload

Of course, one can also do the opposite:

tofile : method to write header and payload to filehandle

data : property that yields full decoded payload

One can decode part of the payload by indexing or slicing the frame.

A number of properties are defined: shape, dtype and size are the shape, type and number of complete samples of the data array, and nbytes the frame size in bytes. Furthermore, the frame acts as a dictionary, with keys those of the header. Any attribute that is not defined on the frame itself, such as `.time` will be looked up on the header as well.

Attributes Summary

data	Full decoded frame.
dtype	Numeric type of the frame data.
fill_value	Value to replace invalid data in the frame.
nbytes	Size of the encoded frame in bytes.
ndim	Number of dimensions of the frame data.
sample_shape	Shape of a sample in the frame (nchan,).
shape	Shape of the frame data.
size	Total number of component samples in the frame data.
valid	Whether frame contains valid data.

Methods Summary

fromdata(data[, header, valid, verify])	Construct frame from data and header.
fromfile(fh[, memmap, valid, verify])	Read a frame from a filehandle, possible mapping the payload.
keys()	
tofile(fh)	Write encoded frame to filehandle.
verify()	Simple verification.

Attributes Documentation

data
Full decoded frame.

dtype
Numeric type of the frame data.

fill_value
Value to replace invalid data in the frame.

nbytes
Size of the encoded frame in bytes.

ndim
Number of dimensions of the frame data.

sample_shape
Shape of a sample in the frame (nchan,).

shape
Shape of the frame data.

size
Total number of component samples in the frame data.

valid
Whether frame contains valid data.

Methods Documentation

classmethod fromdata(*data*, *header=None*, *valid=True*, *verify=True*, ***kwargs*)
Construct frame from data and header.

Note that since GUPPI files are generally very large, one would normally map the file, and then set pieces of it by assigning to slices of the frame. See `memmap_frame`.

Parameters

data : `ndarray`

Array holding complex or real data to be encoded.

header : `GUPPIHeader` or None, optional

If not given, will attempt to generate one using the keywords.

valid : bool, optional

Whether the data are valid (default: `True`). Note that this information cannot be written to disk.

verify : bool, optional

Whether or not to do basic assertions that check the integrity. Default: `True`.

****kwargs**

If `header` is not given, these are used to initialize one.

classmethod fromfile(*fh*, *memmap=True*, *valid=True*, *verify=True*)

Read a frame from a filehandle, possible mapping the payload.

Parameters

fh : filehandle

To read header from.

memmap : bool, optional

If `True` (default), use `memmap` to map the payload. If `False`, just read it from disk.

valid : bool, optional

Whether the data are valid (default: `True`). Note that this cannot be inferred from the header or payload itself. If `False`, any data read will be set to `cls.fill_value`.

verify : bool, optional

Whether to do basic verification of integrity. Default: `True`.

keys()

tofile(*fh*)

Write encoded frame to filehandle.

verify()

Simple verification. To be added to by subclasses.

GUPPIHeader

class baseband.guppi.**GUPPIHeader**(**args*, ***kwargs*)

Bases: `astropy.io.fits.Header`

GUPPI baseband file format header.

Parameters

***args** : str or iterable

If a string, parsed as a GUPPI header from a file, otherwise as for the `astropy.io.fits.Header` baseclass.

verify : bool, optional

Whether to do minimal verification that the header is consistent with the GUPPI standard. Default: `True`.

mutable : bool, optional

Whether to allow the header to be changed after initialisation. Default: `True`.

****kwargs**

Any further header keywords to be set.

Notes

Like `Header`, the initialiser does not accept keyword arguments to populate an array - instead, one must pass an iterable. In order to ensure keywords are kept in the right order, one should pass on values as a tuple, not as a dict. E.g., to copy a header, one should not do `GUPPIHeader({key: header[key] for key in header})`, but rather:

```
GUPPIHeader(((key, header[key]) for key in header))
```

or, to also keep the comments:

```
GUPPIHeader(((key, (header[key], header.comments[key]))
             for key in header))
```

Attributes Summary

`bps`	Bits per elementary sample.
`cards`	The underlying physical cards that make up this Header; it can be looked at, but it should not be modified directly.
`channels_first`	True if encoded payload ordering is (nchan, nsample, npol).
`comments`	View the comments associated with each keyword, if any.
`complex_data`	Whether the data are complex.
`frame_nbytes`	Size of the frame in bytes.
`nbytes`	Size of the header in bytes.
`nchan`	Number of channels.
`npol`	Number of polarisations.
`offset`	Offset from start of observation in units of time.
`overlap`	Number of complete samples that overlap with the next frame.
`payload_nbytes`	Size of the payload in bytes.
`sample_rate`	Number of complete samples per second.
`sample_shape`	Shape of a sample in the payload (npol, nchan).
`samples_per_frame`	Number of complete samples in the frame, including overlap.

Continued on next page

Table 5 – continued from previous page

sideband	True if upper sideband.
start_time	Start time of the observation.
time	Start time of the part of the observation covered by this header.

Methods Summary

add_blank([value, before, after])	Add a blank card.
add_comment(value[, before, after])	Add a COMMENT card.
add_history(value[, before, after])	Add a HISTORY card.
append([card, useblanks, bottom, end])	Appends a new keyword+value card to the end of the Header, similar to list.append.
clear()	Remove all cards from the header.
copy()	Create a mutable and independent copy of the header.
count(keyword)	Returns the count of the given keyword in the header, similar to list.count if the Header object is treated as a list of keywords.
extend(cards[, strip, unique, update, ...])	Appends multiple keyword+value cards to the end of the header, similar to list.extend.
fromfile(fh[, verify])	Reads in GUPPI header block from a file.
fromkeys(*args, **kwargs)	Initialise a header from keyword values.
fromstring(data[, sep])	Creates an HDU header from a byte string containing the entire header data.
fromtextfile(fileobj[, endcard])	Read a header from a simple text file or file-like object.
fromvalues(**kwargs)	Initialise a header from parsed values.
get(key[, default])	Similar to dict.get()–returns the value associated with keyword in the header, or a default value if the keyword is not found.
index(keyword[, start, stop])	Returns the index if the first instance of the given keyword in the header, similar to list.index if the Header object is treated as a list of keywords.
insert(key, card[, useblanks, after])	Inserts a new keyword+value card into the Header at a given location, similar to list.insert.
items()	Like dict.items().
iteritems()	Like dict.iteritems().
iterkeys()	Like dict.iterkeys()–iterating directly over the Header instance has the same behavior.
itervalues()	Like dict.itervalues().
keys()	Return a list of keywords in the header in the order they appear–like dict.keys() but ordered.
pop(*args)	Works like list.pop() if no arguments or an index argument are supplied; otherwise works like dict.pop().
popitem()	Similar to dict.popitem().
remove(keyword[, ignore_missing, remove_all])	Removes the first instance of the given keyword from the header similar to list.remove if the Header object is treated as a list of keywords.
rename_keyword(oldkeyword, newkeyword[, force])	Rename a card's keyword in the header.

Continued on next page

Table 6 – continued from previous page

set(keyword[, value, comment, before, after])	Set the value and/or comment and/or position of a specified keyword.
setdefault(key[, default])	Similar to dict.setdefault().
tofile(fh)	Write GUPPI file header to filehandle.
tostring([sep, endcard, padding])	Returns a string representation of the header.
totextfile(**kwargs)	Write the header as text to a file or a file-like object.
update(**kwargs)	Update the header with new values.
values()	Returns a list of the values of all cards in the header.
verify()	Basic check of integrity.

Attributes Documentation

bps
Bits per elementary sample.

cards
The underlying physical cards that make up this Header; it can be looked at, but it should not be modified directly.

channels_first
True if encoded payload ordering is (nchan, nsample, npol).

comments
View the comments associated with each keyword, if any.

For example, to see the comment on the NAXIS keyword:

```
>>> header.comments['NAXIS']
number of data axes
```

Comments can also be updated through this interface:

```
>>> header.comments['NAXIS'] = 'Number of data axes'
```

complex_data
Whether the data are complex.

frame_nbytes
Size of the frame in bytes.

nbytes
Size of the header in bytes.

nchan
Number of channels.

npol
Number of polarisations.

offset
Offset from start of observation in units of time.

overlap
Number of complete samples that overlap with the next frame.

payload_nbytes
Size of the payload in bytes.

sample_rate
Number of complete samples per second.

Can be set with a negative quantity to set sideband. Overlap samples are not included in the rate.

sample_shape
Shape of a sample in the payload (npol, nchan).

samples_per_frame
Number of complete samples in the frame, including overlap.

sideband
True if upper sideband.

start_time
Start time of the observation.

time
Start time of the part of the observation covered by this header.

Methods Documentation

add_blank(*value=", before=None, after=None*)
Add a blank card.

Parameters
value : str, optional

Text to be added.

before : str or int, optional

Same as in `Header.update`

after : str or int, optional

Same as in `Header.update`

add_comment(*value, before=None, after=None*)
Add a `COMMENT` card.

Parameters
value : str

Text to be added.

before : str or int, optional

Same as in `Header.update`

after : str or int, optional

Same as in `Header.update`

add_history(*value, before=None, after=None*)
Add a `HISTORY` card.

Parameters
value : str

History text to be added.

before : str or int, optional

Same as in `Header.update`

after : str or int, optional

Same as in `Header.update`

append(*card=None*, *useblanks=True*, *bottom=False*, *end=False*)

Appends a new keyword+value card to the end of the Header, similar to `list.append`.

By default if the last cards in the Header have commentary keywords, this will append the new keyword before the commentary (unless the new keyword is also commentary).

Also differs from `list.append` in that it can be called with no arguments: In this case a blank card is appended to the end of the Header. In the case all the keyword arguments are ignored.

Parameters

card : str, tuple

A keyword or a (keyword, value, [comment]) tuple representing a single header card; the comment is optional in which case a 2-tuple may be used

useblanks : bool, optional

If there are blank cards at the end of the Header, replace the first blank card so that the total number of cards in the Header does not increase. Otherwise preserve the number of blank cards.

bottom : bool, optional

If True, instead of appending after the last non-commentary card, append after the last non-blank card.

end : bool, optional

If True, ignore the useblanks and bottom options, and append at the very end of the Header.

clear()

Remove all cards from the header.

copy()

Create a mutable and independent copy of the header.

count(*keyword*)

Returns the count of the given keyword in the header, similar to `list.count` if the Header object is treated as a list of keywords.

Parameters

keyword : str

The keyword to count instances of in the header

extend(*cards*, *strip=True*, *unique=False*, *update=False*, *update_first=False*, *useblanks=True*, *bottom=False*, *end=False*)

Appends multiple keyword+value cards to the end of the header, similar to `list.extend`.

Parameters

cards : iterable

An iterable of (keyword, value, [comment]) tuples; see `Header.append`.

strip : bool, optional

Remove any keywords that have meaning only to specific types of HDUs, so that only more general keywords are added from extension Header or Card list (default: `True`).

unique : bool, optional

If `True`, ensures that no duplicate keywords are appended; keywords already in this header are simply discarded. The exception is commentary keywords (COMMENT, HISTORY, etc.): they are only treated as duplicates if their values match.

update : bool, optional

If `True`, update the current header with the values and comments from duplicate keywords in the input header. This supercedes the `unique` argument. Commentary keywords are treated the same as if `unique=True`.

update_first : bool, optional

If the first keyword in the header is 'SIMPLE', and the first keyword in the input header is 'XTENSION', the 'SIMPLE' keyword is replaced by the 'XTENSION' keyword. Likewise if the first keyword in the header is 'XTENSION' and the first keyword in the input header is 'SIMPLE', the 'XTENSION' keyword is replaced by the 'SIMPLE' keyword. This behavior is otherwise dumb as to whether or not the resulting header is a valid primary or extension header. This is mostly provided to support backwards compatibility with the old `Header.fromTxtFile` method, and only applies if `update=True`.

useblanks, bottom, end : bool, optional

These arguments are passed to `Header.append()` while appending new cards to the header.

classmethod fromfile(*fh*, *verify=True*)

Reads in GUPPI header block from a file.

Parameters

fh : filehandle

To read data from.

verify: bool, optional

Whether to do basic checks on whether the header is valid. Verify is automatically called by `fromstring`, so this flag exists only to standardize the API.

classmethod fromkeys(**args*, ***kwargs*)

Initialise a header from keyword values.

Like fromvalues, but without any interpretation of keywords.

This extracts 'verify' and 'mutable', then passes the remaining kwargs to the class initializer as a dict (for compatibility with fits.Header). It is present for compatibility with other header classes only.

classmethod fromstring(*data*, *sep=''*)

Creates an HDU header from a byte string containing the entire header data.

Parameters

data : str

String containing the entire header.

sep : str, optional

The string separating cards from each other, such as a newline. By default there is no card separator (as is the case in a raw FITS file).

Returns

header

A new `Header` instance.

classmethod fromtextfile(*fileobj*, *endcard=False*)
Read a header from a simple text file or file-like object.

Equivalent to:

```
>>> Header.fromfile(fileobj, sep='\n', endcard=False,
...                 padding=False)
```

See also:

fromfile

classmethod fromvalues(***kwargs*)
Initialise a header from parsed values.

Here, the parsed values must be given as keyword arguments, i.e., for any header, `cls.fromvalues(**header) == header`.

However, unlike for the `fromkeys` class method, data can also be set using arguments named after header methods, such as `time`.

Furthermore, some header defaults are set in `GUPPIHeader._defaults`.

get(*key*, *default=None*)
Similar to `dict.get()`–returns the value associated with keyword in the header, or a default value if the keyword is not found.

Parameters
key : str

A keyword that may or may not be in the header.

default : optional

A default value to return if the keyword is not found in the header.

Returns
value

The value associated with the given keyword, or the default value if the keyword is not in the header.

index(*keyword*, *start=None*, *stop=None*)
Returns the index if the first instance of the given keyword in the header, similar to `list.index` if the Header object is treated as a list of keywords.

Parameters
keyword : str

The keyword to look up in the list of all keywords in the header

start : int, optional

The lower bound for the index

stop : int, optional

The upper bound for the index

insert(*key*, *card*, *useblanks=True*, *after=False*)
Inserts a new keyword+value card into the Header at a given location, similar to `list.insert`.

Parameters
key : int, str, or tuple

The index into the list of header keywords before which the new keyword should be inserted, or the name of a keyword before which the new keyword should be inserted. Can also accept a (keyword, index) tuple for inserting around duplicate keywords.

card : str, tuple

A keyword or a (keyword, value, [comment]) tuple; see `Header.append`

useblanks : bool, optional

If there are blank cards at the end of the Header, replace the first blank card so that the total number of cards in the Header does not increase. Otherwise preserve the number of blank cards.

after : bool, optional

If set to `True`, insert *after* the specified index or keyword, rather than before it. Defaults to `False`.

items()
Like `dict.items()`.

iteritems()
Like `dict.iteritems()`.

iterkeys()
Like `dict.iterkeys()`–iterating directly over the `Header` instance has the same behavior.

itervalues()
Like `dict.itervalues()`.

keys()
Return a list of keywords in the header in the order they appear–like `dict.keys()` but ordered.

pop(**args*)
Works like `list.pop()` if no arguments or an index argument are supplied; otherwise works like `dict.pop()`.

popitem()
Similar to `dict.popitem()`.

remove(*keyword*, *ignore_missing=False*, *remove_all=False*)
Removes the first instance of the given keyword from the header similar to `list.remove` if the Header object is treated as a list of keywords.

Parameters

keyword : str

The keyword of which to remove the first instance in the header.

ignore_missing : bool, optional

When True, ignores missing keywords. Otherwise, if the keyword is not present in the header a KeyError is raised.

remove_all : bool, optional

When True, all instances of keyword will be removed. Otherwise only the first instance of the given keyword is removed.

rename_keyword(*oldkeyword*, *newkeyword*, *force=False*)
Rename a card's keyword in the header.

Parameters

oldkeyword : str or int

Old keyword or card index

newkeyword : str

New keyword

force : bool, optional

When `True`, if the new keyword already exists in the header, force the creation of a duplicate keyword. Otherwise a `ValueError` is raised.

set(*keyword*, *value=None*, *comment=None*, *before=None*, *after=None*)

Set the value and/or comment and/or position of a specified keyword.

If the keyword does not already exist in the header, a new keyword is created in the specified position, or appended to the end of the header if no position is specified.

This method is similar to `Header.update()` prior to Astropy v0.1.

Note: It should be noted that `header.set(keyword, value)` and `header.set(keyword, value, comment)` are equivalent to `header[keyword] = value` and `header[keyword] = (value, comment)` respectively.

New keywords can also be inserted relative to existing keywords using, for example:

```
>>> header.insert('NAXIS1', ('NAXIS', 2, 'Number of axes'))
```

to insert before an existing keyword, or:

```
>>> header.insert('NAXIS', ('NAXIS1', 4096), after=True)
```

to insert after an existing keyword.

The only advantage of using `Header.set()` is that it easily replaces the old usage of `Header.update()` both conceptually and in terms of function signature.

Parameters

keyword : str

A header keyword

value : str, optional

The value to set for the given keyword; if None the existing value is kept, but '' may be used to set a blank value

comment : str, optional

The comment to set for the given keyword; if None the existing comment is kept, but `''` may be used to set a blank comment

before : str, int, optional

Name of the keyword, or index of the `Card` before which this card should be located in the header. The argument `before` takes precedence over `after` if both specified.

after : str, int, optional

Name of the keyword, or index of the `Card` after which this card should be located in the header.

setdefault(*key*, *default=None*)
Similar to dict.setdefault().

tofile(*fh*)
Write GUPPI file header to filehandle.

Uses tostring.

tostring(*sep="*, *endcard=True*, *padding=True*)
Returns a string representation of the header.

By default this uses no separator between cards, adds the END card, and pads the string with spaces to the next multiple of 2880 bytes. That is, it returns the header exactly as it would appear in a FITS file.

Parameters

sep : str, optional

The character or string with which to separate cards. By default there is no separator, but one could use `'\\n'`, for example, to separate each card with a new line

endcard : bool, optional

If True (default) adds the END card to the end of the header string

padding : bool, optional

If True (default) pads the string with spaces out to the next multiple of 2880 characters

Returns

s : str

A string representing a FITS header.

totextfile(***kwargs*)
Write the header as text to a file or a file-like object.

Equivalent to:

```
>>> Header.tofile(fileobj, sep='\n', endcard=False,
...               padding=False, overwrite=overwrite)
```

Changed in version 1.3: `overwrite` replaces the deprecated `clobber` argument.

See also:

tofile

update(***kwargs*)
Update the header with new values.

Here, any keywords matching properties are processed as well, in the order set by the class (in `_properties`), and after all other keywords have been processed.

Parameters

verify : bool, optional

If True (default), verify integrity after updating.

****kwargs**

Arguments used to set keywords and properties.

values()
Returns a list of the values of all cards in the header.

`verify()`
Basic check of integrity.

GUPPIPayload

class `baseband.guppi.GUPPIPayload`(*words*, *header=None*, *sample_shape=()*, *bps=8*, *complex_data=False*, *channels_first=True*)

Bases: `baseband.vlbi_base.payload.VLBIPayloadBase`

Container for decoding and encoding GUPPI payloads.

Parameters

words : `ndarray`

Array containg LSB unsigned words (with the right size) that encode the payload.

header : `GUPPIHeader`

Header that provides information about how the payload is encoded. If not given, the following arguments have to be passed in.

bps : int, optional

Number of bits per sample part (i.e., per channel and per real or imaginary component). Default: 8.

sample_shape : tuple, optional

Shape of the samples; e.g., (nchan,). Default: ().

complex_data : bool, optional

Whether data are complex. Default: `False`.

channels_first : bool, optional

Whether the encoded payload is stored as (nchan, nsample, npol), rather than (nsample, nchan, npol). Default: `True`.

Attributes Summary

`data`	Full decoded payload.
`dtype`	Numeric type of the decoded data array.
`nbytes`	Size of the payload in bytes.
`ndim`	Number of dimensions of the decoded data array.
`shape`	Shape of the decoded data array.
`size`	Total number of component samples in the decoded data array.

Methods Summary

`fromdata`(data[, header, bps, channels_first])	Encode data as a payload.
`fromfile`(fh[, header, memmap, payload_nbytes])	Read or map encoded data in file.
`tofile`(fh)	Write payload to filehandle.

Attributes Documentation

data
Full decoded payload.

dtype
Numeric type of the decoded data array.

nbytes
Size of the payload in bytes.

ndim
Number of dimensions of the decoded data array.

shape
Shape of the decoded data array.

size
Total number of component samples in the decoded data array.

Methods Documentation

classmethod fromdata(*data*, *header=None*, *bps=8*, *channels_first=True*)
Encode data as a payload.

Parameters

data : `ndarray`

Data to be encoded. The last dimension is taken as the number of channels.

header : `GUPPIHeader`, optional

If given, used to infer the `bps` and `channels_first`.

bps : int, optional

Bits per elementary sample, used if `header` is `None`. Default: 8.

channels_first : bool, optional

Whether encoded data should be ordered as (nchan, nsample, npol), used if `header` is `None`. Default: `True`.

classmethod fromfile(*fh*, *header=None*, *memmap=False*, *payload_nbytes=None*, ***kwargs*)
Read or map encoded data in file.

Parameters

fh : filehandle

Handle to the file which will be read or mapped.

header : `GUPPIHeader`, optional

If given, used to infer `payload_nbytes`, `bps`, `sample_shape`, `complex_data` and `channels_first`. If not given, those have to be passed in.

memmap : bool, optional

If `False` (default), read from file. Otherwise, map the file in memory (see `memmap`).

payload_nbytes : int, optional

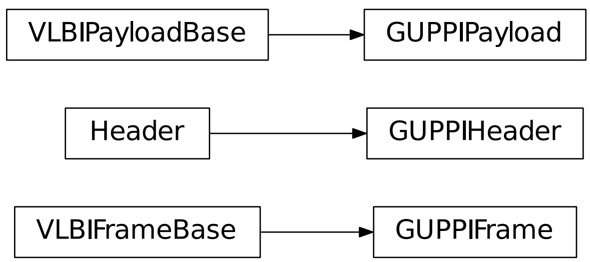
Number of bytes to read (default: as given in `header`, `cls._nbytes`, or, for mapping, to the end of the file).

****kwargs**

Additional arguments are passed on to the class initializer. These are only needed if `header` is not given.

`tofile`(*fh*)
Write payload to filehandle.

Class Inheritance Diagram



8.3.2 baseband.guppi.header Module

Definitions for GUPPI headers.

Implements a GUPPIHeader class that reads & writes FITS-like headers from file.

Classes

`GUPPIHeader`(*args, **kwargs)	GUPPI baseband file format header.

GUPPIHeader

class `baseband.guppi.header.`**`GUPPIHeader`**(**args*, ***kwargs*)
Bases: `astropy.io.fits.Header`

GUPPI baseband file format header.

Parameters

***args** : str or iterable

If a string, parsed as a GUPPI header from a file, otherwise as for the `astropy.io.fits.Header` baseclass.

verify : bool, optional

Whether to do minimal verification that the header is consistent with the GUPPI standard. Default: `True`.

mutable : bool, optional

Whether to allow the header to be changed after initialisation. Default: `True`.

****kwargs**

Any further header keywords to be set.

Notes

Like `Header`, the initialiser does not accept keyword arguments to populate an array - instead, one must pass an iterable. In order to ensure keywords are kept in the right order, one should pass on values as a tuple, not as a dict. E.g., to copy a header, one should not do `GUPPIHeader({key: header[key] for key in header})`, but rather:

```
GUPPIHeader(((key, header[key]) for key in header))
```

or, to also keep the comments:

```
GUPPIHeader(((key, (header[key], header.comments[key]))
             for key in header))
```

Attributes Summary

`bps`	Bits per elementary sample.
`cards`	The underlying physical cards that make up this Header; it can be looked at, but it should not be modified directly.
`channels_first`	True if encoded payload ordering is (nchan, nsample, npol).
`comments`	View the comments associated with each keyword, if any.
`complex_data`	Whether the data are complex.
`frame_nbytes`	Size of the frame in bytes.
`nbytes`	Size of the header in bytes.
`nchan`	Number of channels.
`npol`	Number of polarisations.
`offset`	Offset from start of observation in units of time.
`overlap`	Number of complete samples that overlap with the next frame.
`payload_nbytes`	Size of the payload in bytes.
`sample_rate`	Number of complete samples per second.
`sample_shape`	Shape of a sample in the payload (npol, nchan).
`samples_per_frame`	Number of complete samples in the frame, including overlap.
`sideband`	True if upper sideband.
`start_time`	Start time of the observation.
`time`	Start time of the part of the observation covered by this header.

Methods Summary

add_blank([value, before, after])	Add a blank card.
add_comment(value[, before, after])	Add a COMMENT card.
add_history(value[, before, after])	Add a HISTORY card.
append([card, useblanks, bottom, end])	Appends a new keyword+value card to the end of the Header, similar to list.append.
clear()	Remove all cards from the header.
copy()	Create a mutable and independent copy of the header.
count(keyword)	Returns the count of the given keyword in the header, similar to list.count if the Header object is treated as a list of keywords.
extend(cards[, strip, unique, update, . . .])	Appends multiple keyword+value cards to the end of the header, similar to list.extend.
fromfile(fh[, verify])	Reads in GUPPI header block from a file.
fromkeys(*args, **kwargs)	Initialise a header from keyword values.
fromstring(data[, sep])	Creates an HDU header from a byte string containing the entire header data.
fromtextfile(fileobj[, endcard])	Read a header from a simple text file or file-like object.
fromvalues(**kwargs)	Initialise a header from parsed values.
get(key[, default])	Similar to dict.get()–returns the value associated with keyword in the header, or a default value if the keyword is not found.
index(keyword[, start, stop])	Returns the index if the first instance of the given keyword in the header, similar to list.index if the Header object is treated as a list of keywords.
insert(key, card[, useblanks, after])	Inserts a new keyword+value card into the Header at a given location, similar to list.insert.
items()	Like dict.items().
iteritems()	Like dict.iteritems().
iterkeys()	Like dict.iterkeys()–iterating directly over the Header instance has the same behavior.
itervalues()	Like dict.itervalues().
keys()	Return a list of keywords in the header in the order they appear–like dict.keys() but ordered.
pop(*args)	Works like list.pop() if no arguments or an index argument are supplied; otherwise works like dict.pop().
popitem()	Similar to dict.popitem().
remove(keyword[, ignore_missing, remove_all])	Removes the first instance of the given keyword from the header similar to list.remove if the Header object is treated as a list of keywords.
rename_keyword(oldkeyword, newkeyword[, force])	Rename a card's keyword in the header.
set(keyword[, value, comment, before, after])	Set the value and/or comment and/or position of a specified keyword.
setdefault(key[, default])	Similar to dict.setdefault().
tofile(fh)	Write GUPPI file header to filehandle.
tostring([sep, endcard, padding])	Returns a string representation of the header.
totextfile(**kwargs)	Write the header as text to a file or a file-like object.
update(**kwargs)	Update the header with new values.
values()	Returns a list of the values of all cards in the header.

Continued on next page

Table 11 – continued from previous page

verify()	Basic check of integrity.

Attributes Documentation

bps
Bits per elementary sample.

cards
The underlying physical cards that make up this Header; it can be looked at, but it should not be modified directly.

channels_first
True if encoded payload ordering is (nchan, nsample, npol).

comments
View the comments associated with each keyword, if any.

For example, to see the comment on the NAXIS keyword:

```
>>> header.comments['NAXIS']
number of data axes
```

Comments can also be updated through this interface:

```
>>> header.comments['NAXIS'] = 'Number of data axes'
```

complex_data
Whether the data are complex.

frame_nbytes
Size of the frame in bytes.

nbytes
Size of the header in bytes.

nchan
Number of channels.

npol
Number of polarisations.

offset
Offset from start of observation in units of time.

overlap
Number of complete samples that overlap with the next frame.

payload_nbytes
Size of the payload in bytes.

sample_rate
Number of complete samples per second.

Can be set with a negative quantity to set sideband. Overlap samples are not included in the rate.

sample_shape
Shape of a sample in the payload (npol, nchan).

samples_per_frame
Number of complete samples in the frame, including overlap.

sideband
True if upper sideband.

start_time
Start time of the observation.

time
Start time of the part of the observation covered by this header.

Methods Documentation

add_blank(*value=''*, *before=None*, *after=None*)
Add a blank card.

Parameters
value : str, optional

Text to be added.

before : str or int, optional

Same as in `Header.update`

after : str or int, optional

Same as in `Header.update`

add_comment(*value*, *before=None*, *after=None*)
Add a `COMMENT` card.

Parameters
value : str

Text to be added.

before : str or int, optional

Same as in `Header.update`

after : str or int, optional

Same as in `Header.update`

add_history(*value*, *before=None*, *after=None*)
Add a `HISTORY` card.

Parameters
value : str

History text to be added.

before : str or int, optional

Same as in `Header.update`

after : str or int, optional

Same as in `Header.update`

append(*card=None*, *useblanks=True*, *bottom=False*, *end=False*)
Appends a new keyword+value card to the end of the Header, similar to `list.append`.

By default if the last cards in the Header have commentary keywords, this will append the new keyword before the commentary (unless the new keyword is also commentary).

Also differs from `list.append` in that it can be called with no arguments: In this case a blank card is appended to the end of the Header. In the case all the keyword arguments are ignored.

Parameters

card : str, tuple

A keyword or a (keyword, value, [comment]) tuple representing a single header card; the comment is optional in which case a 2-tuple may be used

useblanks : bool, optional

If there are blank cards at the end of the Header, replace the first blank card so that the total number of cards in the Header does not increase. Otherwise preserve the number of blank cards.

bottom : bool, optional

If True, instead of appending after the last non-commentary card, append after the last non-blank card.

end : bool, optional

If True, ignore the useblanks and bottom options, and append at the very end of the Header.

clear()

Remove all cards from the header.

copy()

Create a mutable and independent copy of the header.

count(*keyword*)

Returns the count of the given keyword in the header, similar to `list.count` if the Header object is treated as a list of keywords.

Parameters

keyword : str

The keyword to count instances of in the header

extend(*cards*, *strip=True*, *unique=False*, *update=False*, *update_first=False*, *useblanks=True*, *bottom=False*, *end=False*)

Appends multiple keyword+value cards to the end of the header, similar to `list.extend`.

Parameters

cards : iterable

An iterable of (keyword, value, [comment]) tuples; see `Header.append`.

strip : bool, optional

Remove any keywords that have meaning only to specific types of HDUs, so that only more general keywords are added from extension Header or Card list (default: `True`).

unique : bool, optional

If `True`, ensures that no duplicate keywords are appended; keywords already in this header are simply discarded. The exception is commentary keywords (COMMENT, HISTORY, etc.): they are only treated as duplicates if their values match.

update : bool, optional

If `True`, update the current header with the values and comments from duplicate keywords in the input header. This supercedes the `unique` argument. Commentary keywords are treated the same as if `unique=True`.

update_first : bool, optional

If the first keyword in the header is 'SIMPLE', and the first keyword in the input header is 'XTENSION', the 'SIMPLE' keyword is replaced by the 'XTENSION' keyword. Likewise if the first keyword in the header is 'XTENSION' and the first keyword in the input header is 'SIMPLE', the 'XTENSION' keyword is replaced by the 'SIMPLE' keyword. This behavior is otherwise dumb as to whether or not the resulting header is a valid primary or extension header. This is mostly provided to support backwards compatibility with the old `Header.fromTxtFile` method, and only applies if `update=True`.

useblanks, bottom, end : bool, optional

These arguments are passed to `Header.append()` while appending new cards to the header.

classmethod fromfile(*fh*, *verify=True*)

Reads in GUPPI header block from a file.

Parameters

fh : filehandle

To read data from.

verify: bool, optional

Whether to do basic checks on whether the header is valid. Verify is automatically called by `fromstring`, so this flag exists only to standardize the API.

classmethod fromkeys(**args*, ***kwargs*)

Initialise a header from keyword values.

Like fromvalues, but without any interpretation of keywords.

This extracts 'verify' and 'mutable', then passes the remaining kwargs to the class initializer as a dict (for compatibility with fits.Header). It is present for compatibility with other header classes only.

classmethod fromstring(*data*, *sep="*)

Creates an HDU header from a byte string containing the entire header data.

Parameters

data : str

String containing the entire header.

sep : str, optional

The string separating cards from each other, such as a newline. By default there is no card separator (as is the case in a raw FITS file).

Returns

header

A new `Header` instance.

classmethod fromtextfile(*fileobj*, *endcard=False*)

Read a header from a simple text file or file-like object.

Equivalent to:

```
>>> Header.fromfile(fileobj, sep='\n', endcard=False,
...                 padding=False)
```

See also:

`fromfile`

classmethod fromvalues(***kwargs*)

Initialise a header from parsed values.

Here, the parsed values must be given as keyword arguments, i.e., for any `header`, `cls.fromvalues(**header) == header`.

However, unlike for the `fromkeys` class method, data can also be set using arguments named after header methods, such as `time`.

Furthermore, some header defaults are set in `GUPPIHeader._defaults`.

get(*key*, *default=None*)

Similar to `dict.get()`–returns the value associated with keyword in the header, or a default value if the keyword is not found.

Parameters

key : str

A keyword that may or may not be in the header.

default : optional

A default value to return if the keyword is not found in the header.

Returns

value

The value associated with the given keyword, or the default value if the keyword is not in the header.

index(*keyword*, *start=None*, *stop=None*)

Returns the index if the first instance of the given keyword in the header, similar to `list.index` if the Header object is treated as a list of keywords.

Parameters

keyword : str

The keyword to look up in the list of all keywords in the header

start : int, optional

The lower bound for the index

stop : int, optional

The upper bound for the index

insert(*key*, *card*, *useblanks=True*, *after=False*)

Inserts a new keyword+value card into the Header at a given location, similar to `list.insert`.

Parameters

key : int, str, or tuple

The index into the list of header keywords before which the new keyword should be inserted, or the name of a keyword before which the new keyword should be inserted. Can also accept a (keyword, index) tuple for inserting around duplicate keywords.

card : str, tuple

A keyword or a (keyword, value, [comment]) tuple; see `Header.append`

useblanks : bool, optional

If there are blank cards at the end of the Header, replace the first blank card so that the total number of cards in the Header does not increase. Otherwise preserve the number of blank cards.

after : bool, optional

If set to `True`, insert *after* the specified index or keyword, rather than before it. Defaults to `False`.

items()
Like `dict.items()`.

iteritems()
Like `dict.iteritems()`.

iterkeys()
Like `dict.iterkeys()`–iterating directly over the `Header` instance has the same behavior.

itervalues()
Like `dict.itervalues()`.

keys()
Return a list of keywords in the header in the order they appear–like `dict.keys()` but ordered.

pop(**args*)
Works like `list.pop()` if no arguments or an index argument are supplied; otherwise works like `dict.pop()`.

popitem()
Similar to `dict.popitem()`.

remove(*keyword*, *ignore_missing=False*, *remove_all=False*)
Removes the first instance of the given keyword from the header similar to `list.remove` if the Header object is treated as a list of keywords.

Parameters

keyword : str

The keyword of which to remove the first instance in the header.

ignore_missing : bool, optional

When True, ignores missing keywords. Otherwise, if the keyword is not present in the header a KeyError is raised.

remove_all : bool, optional

When True, all instances of keyword will be removed. Otherwise only the first instance of the given keyword is removed.

rename_keyword(*oldkeyword*, *newkeyword*, *force=False*)
Rename a card's keyword in the header.

Parameters

oldkeyword : str or int

Old keyword or card index

newkeyword : str

New keyword

force : bool, optional

When `True`, if the new keyword already exists in the header, force the creation of a duplicate keyword. Otherwise a `ValueError` is raised.

set(*keyword*, *value=None*, *comment=None*, *before=None*, *after=None*)
Set the value and/or comment and/or position of a specified keyword.

If the keyword does not already exist in the header, a new keyword is created in the specified position, or appended to the end of the header if no position is specified.

This method is similar to `Header.update()` prior to Astropy v0.1.

Note: It should be noted that `header.set(keyword, value)` and `header.set(keyword, value, comment)` are equivalent to `header[keyword] = value` and `header[keyword] = (value, comment)` respectively.

New keywords can also be inserted relative to existing keywords using, for example:

```
>>> header.insert('NAXIS1', ('NAXIS', 2, 'Number of axes'))
```

to insert before an existing keyword, or:

```
>>> header.insert('NAXIS', ('NAXIS1', 4096), after=True)
```

to insert after an existing keyword.

The only advantage of using `Header.set()` is that it easily replaces the old usage of `Header.update()` both conceptually and in terms of function signature.

Parameters

keyword : str

A header keyword

value : str, optional

The value to set for the given keyword; if None the existing value is kept, but '' may be used to set a blank value

comment : str, optional

The comment to set for the given keyword; if None the existing comment is kept, but `''` may be used to set a blank comment

before : str, int, optional

Name of the keyword, or index of the `Card` before which this card should be located in the header. The argument `before` takes precedence over `after` if both specified.

after : str, int, optional

Name of the keyword, or index of the `Card` after which this card should be located in the header.

setdefault(*key*, *default=None*)

Similar to `dict.setdefault()`.

tofile(*fh*)

Write GUPPI file header to filehandle.

Uses `tostring`.

tostring(*sep=''*, *endcard=True*, *padding=True*)

Returns a string representation of the header.

By default this uses no separator between cards, adds the END card, and pads the string with spaces to the next multiple of 2880 bytes. That is, it returns the header exactly as it would appear in a FITS file.

Parameters

sep : str, optional

The character or string with which to separate cards. By default there is no separator, but one could use `'\\n'`, for example, to separate each card with a new line

endcard : bool, optional

If True (default) adds the END card to the end of the header string

padding : bool, optional

If True (default) pads the string with spaces out to the next multiple of 2880 characters

Returns

s : str

A string representing a FITS header.

totextfile(***kwargs*)

Write the header as text to a file or a file-like object.

Equivalent to:

```
>>> Header.tofile(fileobj, sep='\n', endcard=False,
...               padding=False, overwrite=overwrite)
```

Changed in version 1.3: `overwrite` replaces the deprecated `clobber` argument.

See also:

`tofile`

update(***kwargs*)

Update the header with new values.

Here, any keywords matching properties are processed as well, in the order set by the class (in `_properties`), and after all other keywords have been processed.

Parameters

verify : bool, optional

If `True` (default), verify integrity after updating.

****kwargs**

Arguments used to set keywords and properties.

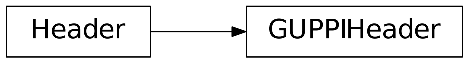
values()

Returns a list of the values of all cards in the header.

verify()

Basic check of integrity.

Class Inheritance Diagram



8.3.3 baseband.guppi.payload Module

Payload for GUPPI format.

Classes

GUPPIPayload(words[, header, sample_shape, . . .])	Container for decoding and encoding GUPPI payloads.

GUPPIPayload

class baseband.guppi.payload.**GUPPIPayload**(*words*, *header=None*, *sample_shape=()*, *bps=8*, *complex_data=False*, *channels_first=True*)

Bases: baseband.vlbi_base.payload.VLBIPayloadBase

Container for decoding and encoding GUPPI payloads.

Parameters

words : ndarray

Array containg LSB unsigned words (with the right size) that encode the payload.

header : GUPPIHeader

Header that provides information about how the payload is encoded. If not given, the following arguments have to be passed in.

bps : int, optional

Number of bits per sample part (i.e., per channel and per real or imaginary component). Default: 8.

sample_shape : tuple, optional

Shape of the samples; e.g., (nchan,). Default: ().

complex_data : bool, optional

Whether data are complex. Default: False.

channels_first : bool, optional

Whether the encoded payload is stored as (nchan, nsample, npol), rather than (nsample, nchan, npol). Default: True.

Attributes Summary

`data`	Full decoded payload.
`dtype`	Numeric type of the decoded data array.
`nbytes`	Size of the payload in bytes.
`ndim`	Number of dimensions of the decoded data array.
`shape`	Shape of the decoded data array.
`size`	Total number of component samples in the decoded data array.

Methods Summary

`fromdata`(data[, header, bps, channels_first])	Encode data as a payload.
`fromfile`(fh[, header, memmap, payload_nbytes])	Read or map encoded data in file.
`tofile`(fh)	Write payload to filehandle.

Attributes Documentation

data
Full decoded payload.

dtype
Numeric type of the decoded data array.

nbytes
Size of the payload in bytes.

ndim
Number of dimensions of the decoded data array.

shape
Shape of the decoded data array.

size
Total number of component samples in the decoded data array.

Methods Documentation

classmethod fromdata(*data*, *header=None*, *bps=8*, *channels_first=True*)
Encode data as a payload.

Parameters

data : `ndarray`

Data to be encoded. The last dimension is taken as the number of channels.

header : `GUPPIHeader`, optional

If given, used to infer the bps and `channels_first`.

bps : int, optional

Bits per elementary sample, used if `header` is `None`. Default: 8.

channels_first : bool, optional

Whether encoded data should be ordered as (nchan, nsample, npol), used if header is None. Default: True.

classmethod fromfile(*fh*, *header=None*, *memmap=False*, *payload_nbytes=None*, ***kwargs*)
Read or map encoded data in file.

Parameters

fh : filehandle

Handle to the file which will be read or mapped.

header : GUPPIHeader, optional

If given, used to infer payload_nbytes, bps, sample_shape, complex_data and channels_first. If not given, those have to be passed in.

memmap : bool, optional

If False (default), read from file. Otherwise, map the file in memory (see memmap).

payload_nbytes : int, optional

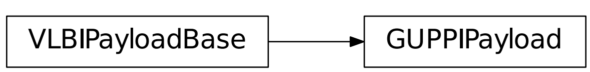
Number of bytes to read (default: as given in header, cls._nbytes, or, for mapping, to the end of the file).

****kwargs**

Additional arguments are passed on to the class initializer. These are only needed if header is not given.

tofile(*fh*)
Write payload to filehandle.

Class Inheritance Diagram



8.3.4 baseband.guppi.frame Module

Classes

GUPPIFrame(header, payload[, valid, verify])	Representation of a GUPPI file, consisting of a header and payload.

GUPPIFrame

class baseband.guppi.frame.**GUPPIFrame**(*header*, *payload*, *valid=True*, *verify=True*)
Bases: baseband.vlbi_base.frame.VLBIFrameBase

Representation of a GUPPI file, consisting of a header and payload.

Parameters

header : GUPPIHeader

Wrapper around the header lines, providing access to the values.

payload : GUPPIPayload

Wrapper around the payload, provding mechanisms to decode it.

valid : bool, optional

Whether the data are valid. Default: True.

verify : bool, optional

Whether to do basic verification of integrity. Default: True.

Notes

GUPPI files do not support storing whether data are valid or not on disk. Hence, this has to be determined independently. If `valid=False`, any decoded data are set to `cls.fill_value` (by default, 0).

The Frame can also be instantiated using class methods:

fromfile : read header and and map or read payload from a filehandle

fromdata : encode data as payload

Of course, one can also do the opposite:

tofile : method to write header and payload to filehandle

data : property that yields full decoded payload

One can decode part of the payload by indexing or slicing the frame.

A number of properties are defined: shape, dtype and size are the shape, type and number of complete samples of the data array, and nbytes the frame size in bytes. Furthermore, the frame acts as a dictionary, with keys those of the header. Any attribute that is not defined on the frame itself, such as `.time` will be looked up on the header as well.

Attributes Summary

data	Full decoded frame.
dtype	Numeric type of the frame data.
fill_value	Value to replace invalid data in the frame.
nbytes	Size of the encoded frame in bytes.
ndim	Number of dimensions of the frame data.
sample_shape	Shape of a sample in the frame (nchan,).
shape	Shape of the frame data.
size	Total number of component samples in the frame data.
valid	Whether frame contains valid data.

Methods Summary

fromdata(data[, header, valid, verify])	Construct frame from data and header.
fromfile(fh[, memmap, valid, verify])	Read a frame from a filehandle, possible mapping the payload.
keys()	
tofile(fh)	Write encoded frame to filehandle.
verify()	Simple verification.

Attributes Documentation

data
Full decoded frame.

dtype
Numeric type of the frame data.

fill_value
Value to replace invalid data in the frame.

nbytes
Size of the encoded frame in bytes.

ndim
Number of dimensions of the frame data.

sample_shape
Shape of a sample in the frame (nchan,).

shape
Shape of the frame data.

size
Total number of component samples in the frame data.

valid
Whether frame contains valid data.

Methods Documentation

classmethod fromdata(*data*, *header=None*, *valid=True*, *verify=True*, ***kwargs*)
Construct frame from data and header.

Note that since GUPPI files are generally very large, one would normally map the file, and then set pieces of it by assigning to slices of the frame. See memmap_frame.

Parameters

data : ndarray

Array holding complex or real data to be encoded.

header : GUPPIHeader or None, optional

If not given, will attempt to generate one using the keywords.

valid : bool, optional

Whether the data are valid (default: True). Note that this information cannot be written to disk.

verify : bool, optional

Whether or not to do basic assertions that check the integrity. Default: `True`.

****kwargs**

If `header` is not given, these are used to initialize one.

classmethod fromfile(*fh*, *memmap=True*, *valid=True*, *verify=True*)
Read a frame from a filehandle, possible mapping the payload.

Parameters

fh : filehandle

To read header from.

memmap : bool, optional

If `True` (default), use `memmap` to map the payload. If `False`, just read it from disk.

valid : bool, optional

Whether the data are valid (default: `True`). Note that this cannot be inferred from the header or payload itself. If `False`, any data read will be set to `cls.fill_value`.

verify : bool, optional

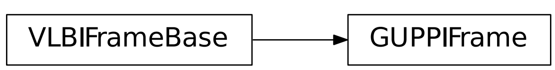
Whether to do basic verification of integrity. Default: `True`.

keys()

tofile(*fh*)
Write encoded frame to filehandle.

verify()
Simple verification. To be added to by subclasses.

Class Inheritance Diagram



8.3.5 baseband.guppi.base Module

Functions

`open`(name[, mode])	Open GUPPI file for reading or writing.

open

baseband.guppi.base.**open**(*name*, *mode=u'rs'*, ***kwargs*)
Open GUPPI file for reading or writing.

Opened as a binary file, one gets a wrapped filehandle that adds methods to read/write a frame. Opened as a stream, the handle is wrapped further, with methods such as reading and writing to the file as if it were a stream of samples.

Parameters

name : str or filehandle

File name or handle.

mode : {'rb', 'wb', 'rs', or 'ws'}, optional

Whether to open for reading or writing, and as a regular binary file or as a stream. Default: 'rs', for reading a stream.

****kwargs**

Additional arguments when opening the file as a stream.

— For reading a stream : (see `GUPPIStreamReader`)

squeeze : bool, optional

If `True` (default), remove any dimensions of length unity from decoded data.

subset : indexing object or tuple of objects, optional

Specific components of the complete sample to decode (after possibly squeezing). If a single indexing object is passed, it selects polarizations. With a tuple, the first selects polarizations and the second selects channels. If the tuple is empty (default), all components are read.

— For writing a stream : (see `GUPPIStreamWriter`)

header0 : `GUPPIHeader`

Header for the first frame, holding time information, etc. Can instead give keyword arguments to construct a header (see `**kwargs`).

squeeze : bool, optional

If `True` (default), writer accepts squeezed arrays as input, and adds any dimensions of length unity.

frames_per_file : int, optional

When writing to a sequence of files, sets the number of frames within each file. Default: 128.

****kwargs**

If the header is not given, an attempt will be made to construct one with any further keyword arguments.

— Header keywords : (see `fromvalues()`)

time : `Time`

Start time of the file. Must have an integer number of seconds.

sample_rate : `Quantity`

Number of complete samples per second, i.e. the rate at which each channel of each polarization is sampled.

samples_per_frame : int

Number of complete samples per frame. Can alternatively give `payload_nbytes`.

payload_nbytes : int

Number of bytes per payload. Can alternatively give `samples_per_frame`.

offset : `Quantity` or `TimeDelta`, optional

Time offset from the start of the whole observation (default: 0).

npol : int, optional

Number of polarizations (default: 1).

nchan : int, optional

Number of channels (default: 1). For GUPPI, complex data is only allowed when nchan > 1.

bps : int, optional

Bits per elementary sample, i.e. per real or imaginary component for complex data (default: 8).

Returns

Filehandle

`GUPPIFileReader` or `GUPPIFileWriter` (binary), or `GUPPIStreamReader` or `GUPPIStreamWriter` (stream).

Notes

For streams, one can also pass in a list of files, or equivalently a `sequentialfile` object (opened in 'rb' mode for reading or 'w+b' for writing).

Classes

`GUPPIFileReader`(fh_raw)	Simple reader for GUPPI files.
`GUPPIFileWriter`(fh_raw)	Simple writer/mapper for GUPPI files.
`GUPPIStreamBase`(fh_raw, header0[, squeeze, ...])	Base for GUPPI streams.
`GUPPIStreamReader`(fh_raw[, squeeze, subset, ...])	GUPPI format reader.
`GUPPIStreamWriter`(fh_raw, header0[, squeeze])	GUPPI format writer.

GUPPIFileReader

class `baseband.guppi.base.`**`GUPPIFileReader`**(*fh_raw*)

Bases: `baseband.vlbi_base.base.VLBIFileReaderBase`

Simple reader for GUPPI files.

Wraps a binary filehandle, providing methods to help interpret the data, such as `read_frame` and `get_frame_rate`. By default, frame payloads are mapped rather than fully read into physical memory.

Parameters

fh_raw : filehandle

Filehandle of the raw binary data file.

Attributes Summary

info	Standardized information on file readers.

Methods Summary

close()	
get_frame_rate()	Determine the number of frames per second.
read_frame([memmap, verify])	Read the frame header and read or map the corresponding payload.
read_header()	Read a single header from the file.

Attributes Documentation

info

Standardized information on file readers.

The info descriptor has a number of standard attributes, which are determined from arguments passed in opening the file, from the first header (info.header0) and from possibly scanning the file to determine the duration of frames.

Examples

The most common use is simply to print information:

```
>>> from baseband.data import SAMPLE_MARK5B
>>> from baseband import mark5b
>>> fh = mark5b.open(SAMPLE_MARK5B, 'rb')
>>> fh.info
File information:
format = mark5b
frame_rate = 6400.0 Hz
bps = 2
complex_data = False

missing:  nchan: needed to determine sample shape and rate.
          kday, ref_time: needed to infer full times.

>>> fh.close()

>>> fh = mark5b.open(SAMPLE_MARK5B, 'rb', kday=56000, nchan=8)
>>> fh.info
File information:
format = mark5b
frame_rate = 6400.0 Hz
sample_rate = 32.0 MHz
samples_per_frame = 5000
sample_shape = (8,)
bps = 2
complex_data = False
start_time = 2014-06-13T05:30:01.000000000
>>> fh.close()
```

Attributes

format	(str or None) File format, or None if the underlying file cannot be parsed.
frame_rate	(Quantity) Number of data frames per unit of time.
sample_rate	(Quantity) Complete samples per unit of time.
samples_per_frame	(int) Number of complete samples in each frame.
sample_shape	(tuple) Dimensions of each complete sample (e.g., (nchan,)).
bps	(int) Number of bits used to encode each elementary sample.
complex_data	(bool) Whether the data are complex.
start_time	(Time) Time of the first complete sample.
missing	(dict) Entries are keyed by names of arguments that should be passed to the file reader to obtain full information. The associated entries explain why these arguments are needed.

Methods Documentation

close()

get_frame_rate()

Determine the number of frames per second.

The routine uses the sample rate and number of samples per frame (excluding overlap) from the first header in the file.

Returns

frame_rate : Quantity

Frames per second.

read_frame(*memmap=True*, *verify=True*)

Read the frame header and read or map the corresponding payload.

Parameters

memmap : bool, optional

If True (default), map the payload using memmap, so that parts are only loaded into memory as needed to access data.

verify : bool, optional

Whether to do basic checks of frame integrity. Default: True.

Returns

frame : GUPPIFrame

With .header and .payload properties. The .data property returns all data encoded in the frame. Since this may be too large to fit in memory, it may be better to access the parts of interest by slicing the frame.

read_header()

Read a single header from the file.

Returns

header : GUPPIHeader

GUPPIFileWriter

class baseband.guppi.base.**GUPPIFileWriter**(*fh_raw*)

Bases: `baseband.vlbi_base.base.VLBIFileBase`

Simple writer/mapper for GUPPI files.

Adds `write_frame` and `memmap_frame` methods to the VLBI binary file wrapper. The latter allows one to encode data in pieces, writing to disk as needed.

Methods Summary

`close`()	
`memmap_frame`([header])	Get frame by writing the header to disk and mapping its payload.
`write_frame`(data[, header])	Write a single frame (header plus payload).

Methods Documentation

close()

memmap_frame(*header=None*, ***kwargs*)

Get frame by writing the header to disk and mapping its payload.

The header is written to disk immediately, but the payload is mapped, so that it can be filled in pieces, by setting slices of the frame.

Parameters

header : `GUPPIHeader`

Written to disk immediately. Can instead give keyword arguments to construct a header.

****kwargs**

If `header` is not given, these are used to initialize one.

Returns

frame: `GUPPIFrame`

By assigning slices to data, the payload can be encoded piecewise.

write_frame(*data*, *header=None*, ***kwargs*)

Write a single frame (header plus payload).

Parameters

data : `ndarray` or `GUPPIFrame`

If an array, a `header` should be given, which will be used to get the information needed to encode the array, and to construct the GUPPI frame.

header : `GUPPIHeader`

Can instead give keyword arguments to construct a header. Ignored if `data` is a `GUPPIFrame` instance.

****kwargs**

If `header` is not given, these are used to initialize one.

GUPPIStreamBase

class baseband.guppi.base.**GUPPIStreamBase**(*fh_raw*, *header0*, *squeeze=True*, *subset=()*, *verify=True*)

Bases: baseband.vlbi_base.base.VLBIStreamBase

Base for GUPPI streams.

Attributes Summary

bps	Bits per elementary sample.
complex_data	Whether the data are complex.
header0	First header of the file.
sample_rate	Number of complete samples per second.
sample_shape	Shape of a complete sample (possibly subset or squeezed).
samples_per_frame	Number of complete samples per frame, excluding overlap.
squeeze	Whether data arrays have dimensions with length unity removed.
start_time	Start time of the file.
subset	Specific components of the complete sample to decode.
time	Time of the sample pointer's current offset in file.
verify	Whether to do consistency checks on frames being read.

Methods Summary

close()	
tell([unit])	Current offset in the file.

Attributes Documentation

bps
Bits per elementary sample.

complex_data
Whether the data are complex.

header0
First header of the file.

sample_rate
Number of complete samples per second.

sample_shape
Shape of a complete sample (possibly subset or squeezed).

samples_per_frame
Number of complete samples per frame, excluding overlap.

squeeze
Whether data arrays have dimensions with length unity removed.

If `True`, data read out has such dimensions removed, and data passed in for writing has them inserted.

start_time
Start time of the file.

See also `time` for the time of the sample pointer's current offset, and (if available) `stop_time` for the time at the end of the file.

subset
Specific components of the complete sample to decode.

The order of dimensions is the same as for `sample_shape`. Set by the class initializer.

time
Time of the sample pointer's current offset in file.

See also `start_time` for the start time, and (if available) `stop_time` for the end time, of the file.

verify
Whether to do consistency checks on frames being read.

Methods Documentation

close()

tell(*unit=None*)
Current offset in the file.

Parameters
unit : `Unit` or str, optional

Time unit the offset should be returned in. By default, no unit is used, i.e., an integer enumerating samples is returned. For the special string 'time', the absolute time is calculated.

Returns
offset : int, `Quantity`, or `Time`

Offset in current file (or time at current position).

GUPPIStreamReader

class baseband.guppi.base.**GUPPIStreamReader**(*fh_raw*, *squeeze=True*, *subset=()*, *verify=True*)
Bases: `baseband.guppi.base.GUPPIStreamBase`, `baseband.vlbi_base.base.VLBIStreamReaderBase`

GUPPI format reader.

Allows access to GUPPI files as a continuous series of samples.

Parameters
fh_raw : filehandle

Filehandle of the raw GUPPI stream.

squeeze : bool, optional

If `True` (default), remove any dimensions of length unity from decoded data.

subset : indexing object or tuple of objects, optional

Specific components of the complete sample to decode (after possibly squeezing). If a single indexing object is passed, it selects polarizations. With a tuple, the first selects polarizations and the second selects channels. If the tuple is empty (default), all components are read.

verify : bool, optional

Whether to do basic checks of frame integrity when reading. The first frame of the stream is always checked, so `verify` is effective only when reading sequences of files. Default: `True`.

Attributes Summary

`bps`	Bits per elementary sample.
`complex_data`	Whether the data are complex.
`dtype`	
`fill_value`	Value to use for invalid or missing data.
`header0`	First header of the file.
`info`	Standardized information on stream readers.
`ndim`	Number of dimensions of the (squeezed/subset) stream data.
`sample_rate`	Number of complete samples per second.
`sample_shape`	Shape of a complete sample (possibly subset or squeezed).
`samples_per_frame`	Number of complete samples per frame, excluding overlap.
`shape`	Shape of the (squeezed/subset) stream data.
`size`	Total number of component samples in the (squeezed/subset) stream data.
`squeeze`	Whether data arrays have dimensions with length unity removed.
`start_time`	Start time of the file.
`stop_time`	Time at the end of the file, just after the last sample.
`subset`	Specific components of the complete sample to decode.
`time`	Time of the sample pointer's current offset in file.
`verify`	Whether to do consistency checks on frames being read.

Methods Summary

`close`()	
`read`([count, out])	Read a number of complete (or subset) samples.
`seek`(offset[, whence])	Change the stream position.
`tell`([unit])	Current offset in the file.

Attributes Documentation

bps

Bits per elementary sample.

complex_data
Whether the data are complex.

dtype

fill_value
Value to use for invalid or missing data. Default: 0.

header0
First header of the file.

info
Standardized information on stream readers.

The `info` descriptor provides a few standard attributes, all of which can also be accessed directly on the stream filehandle. More detailed information on the underlying file is stored in its info, accessible via `info.file_info`.

Attributes

start_time	(`Time`) Time of the first complete sample.
stop_time	(`Time`) Time of the complete sample just beyond the end of the file.
sample_rate	(`Quantity`) Complete samples per unit of time.
shape	(tuple) Equivalent shape of the whole file, i.e., combining the number of complete samples and the shape of those samples.
bps	(int) Number of bits used to encode each elementary sample.
complex_data	(bool) Whether the data are complex.

ndim
Number of dimensions of the (squeezed/subset) stream data.

sample_rate
Number of complete samples per second.

sample_shape
Shape of a complete sample (possibly subset or squeezed).

samples_per_frame
Number of complete samples per frame, excluding overlap.

shape
Shape of the (squeezed/subset) stream data.

size
Total number of component samples in the (squeezed/subset) stream data.

squeeze
Whether data arrays have dimensions with length unity removed.

If `True`, data read out has such dimensions removed, and data passed in for writing has them inserted.

start_time
Start time of the file.

See also `time` for the time of the sample pointer's current offset, and (if available) `stop_time` for the time at the end of the file.

stop_time
Time at the end of the file, just after the last sample.

See also start_time for the start time of the file, and time for the time of the sample pointer's current offset.

subset
Specific components of the complete sample to decode.

The order of dimensions is the same as for sample_shape. Set by the class initializer.

time
Time of the sample pointer's current offset in file.

See also start_time for the start time, and (if available) stop_time for the end time, of the file.

verify
Whether to do consistency checks on frames being read.

Methods Documentation

close()

read(*count=None*, *out=None*)
Read a number of complete (or subset) samples.

The range retrieved can span multiple frames.

Parameters

count : int or None, optional

Number of complete/subset samples to read. If None (default) or negative, the whole file is read. Ignored if out is given.

out : None or array, optional

Array to store the data in. If given, count will be inferred from the first dimension; the other dimension should equal sample_shape.

Returns

out : ndarray of float or complex

The first dimension is sample-time, and the remainder given by sample_shape.

seek(*offset*, *whence=0*)
Change the stream position.

This works like a normal filehandle seek, but the offset is in samples (or a relative or absolute time).

Parameters

offset : int, Quantity, or Time

Offset to move to. Can be an (integer) number of samples, an offset in time units, or an absolute time.

whence : {0, 1, 2, 'start', 'current', or 'end'}, optional

Like regular seek, the offset is taken to be from the start if whence=0 (default), from the current position if 1, and from the end if 2. One can alternativey use 'start', 'current', or 'end' for 0, 1, or 2, respectively. Ignored if offset is a time.

tell(*unit=None*)

Current offset in the file.

Parameters

unit : Unit or str, optional

Time unit the offset should be returned in. By default, no unit is used, i.e., an integer enumerating samples is returned. For the special string 'time', the absolute time is calculated.

Returns

offset : int, Quantity, or Time

Offset in current file (or time at current position).

GUPPIStreamWriter

class baseband.guppi.base.**GUPPIStreamWriter**(*fh_raw*, *header0*, *squeeze=True*)

Bases: baseband.guppi.base.GUPPIStreamBase, baseband.vlbi_base.base.VLBIStreamWriterBase

GUPPI format writer.

Encodes and writes sequences of samples to file.

Parameters

raw : filehandle

For writing the header and raw data to storage.

header0 : GUPPIHeader

Header for the first frame, holding time information, etc.

squeeze : bool, optional

If True (default), write accepts squeezed arrays as input, and adds any dimensions of length unity.

Attributes Summary

bps	Bits per elementary sample.
complex_data	Whether the data are complex.
header0	First header of the file.
sample_rate	Number of complete samples per second.
sample_shape	Shape of a complete sample (possibly subset or squeezed).
samples_per_frame	Number of complete samples per frame, excluding overlap.
squeeze	Whether data arrays have dimensions with length unity removed.
start_time	Start time of the file.
subset	Specific components of the complete sample to decode.
time	Time of the sample pointer's current offset in file.
verify	Whether to do consistency checks on frames being read.

Methods Summary

close()	
tell([unit])	Current offset in the file.
write(data[, valid])	Write data, buffering by frames as needed.

Attributes Documentation

bps
Bits per elementary sample.

complex_data
Whether the data are complex.

header0
First header of the file.

sample_rate
Number of complete samples per second.

sample_shape
Shape of a complete sample (possibly subset or squeezed).

samples_per_frame
Number of complete samples per frame, excluding overlap.

squeeze
Whether data arrays have dimensions with length unity removed.

If `True`, data read out has such dimensions removed, and data passed in for writing has them inserted.

start_time
Start time of the file.

See also `time` for the time of the sample pointer's current offset, and (if available) `stop_time` for the time at the end of the file.

subset
Specific components of the complete sample to decode.

The order of dimensions is the same as for `sample_shape`. Set by the class initializer.

time
Time of the sample pointer's current offset in file.

See also `start_time` for the start time, and (if available) `stop_time` for the end time, of the file.

verify
Whether to do consistency checks on frames being read.

Methods Documentation

close()

tell(*unit=None*)
Current offset in the file.

Parameters

unit : `Unit` or str, optional

Time unit the offset should be returned in. By default, no unit is used, i.e., an integer enumerating samples is returned. For the special string 'time', the absolute time is calculated.

Returns

offset : int, `Quantity`, or `Time`

Offset in current file (or time at current position).

write(*data*, *valid=True*)

Write data, buffering by frames as needed.

Parameters

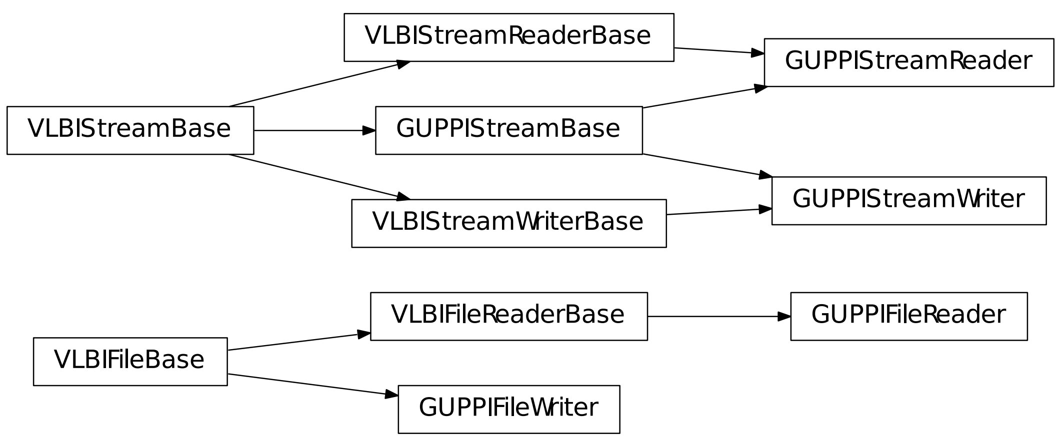
data : `ndarray`

Piece of data to be written, with sample dimensions as given by `sample_shape`. This should be properly scaled to make best use of the dynamic range delivered by the encoding.

valid : bool, optional

Whether the current data are valid. Default: `True`.

Class Inheritance Diagram



CHAPTER 9

GSB

The GMRT software backend (GSB) file format is the standard output of the initial correlator of the Giant Metrewave Radio Telescope (GMRT). The GSB design is described by Roy et al. (2010, Exper. Astron. 28:25-60) with further specifications and operating procedures given on the relevant GMRT/GSB pages.

9.1 File Structure

A GSB dataset consists of an ASCII file with a sequence of *headers*, and one or more accompanying binary data files. Each line in the header and its corresponding data comprise a *data frame*, though these do not have explicit divisions in the data files.

Baseband currently supports two forms of GSB data: **rawdump**, for storing real-valued raw voltage timestreams, and **phased**, for storing complex pre-channelized data from the GMRT in phased array baseband mode.

Data in rawdump format is stored in a binary file representing the voltage stream from one polarization of a single dish. Each such file is accompanied by a header file which contains GPS timestamps, in the form:

```
YYYY MM DD HH MM SS 0.SSSSSSSSS
```

In the default rawdump observing setup, samples are recorded at a rate of 33.3333… megasamples per second (Msps). Each sample is 4 bits in size, and two samples are grouped into bytes such that the oldest sample occupies the least significant bit. Each frame consists of **4 megabytes** of data, or 2^{23}, samples; as such, the timespan of one frame is exactly **0.25165824 s**.

Data in phased format is normally spread over four binary files and one accompanying header file. The binary files come in two pairs, one for each polarization, with the pair contain the first and second half of the data of each frame.

When recording GSB in phased array voltage beam (ie. baseband) mode, the "raw", or pre-channelized, *sample rate* is either 33.3333… Msps at 8 bits per sample or 66.6666… Msps at 4 bits per sample (in the latter case, sample bit-ordering is the same as for rawdump). Channelization via fast Fourier transform sets the channelized *complete sample* rate to the raw rate divided by $2N_{\mathrm{F}}$, where N_{F} is the number of Fourier channels (either 256 or 512). The timespan of one frame is **0.25165824 s**, and one frame is **8 megabytes** in size, for either raw sample rate.

The phased header's structure is:

```
<PC TIME> <GPS TIME> <SEQ NUMBER> <MEM BLOCK>
```

where <PC TIME> and <GPS TIME> are the less accurate computer-based and exact GPS-based timestamps, respectively, with the same format as the rawdump timestamp; <SEQ NUMBER> is the frame number; and <MEM BLOCK> a redundant modulo-8 shared memory block number.

9.2 Usage Notes

This section covers reading and writing GSB files with Baseband; general usage is covered in the *Getting Started* section. While Baseband features the general `baseband.open` and `baseband.file_info` functions, these cannot read GSB binary files without the accompanying timestamp file (at which point it is obvious the files are GSB). `baseband.file_info`, however, can be used on the timestamp file to determine if it is in rawdump or phased format.

The examples below use the samplefiles in the `baseband/data/gsb/` directory, and the `numpy`, `astropy.units` and `baseband.gsb` modules:

```
>>> import numpy as np
>>> import astropy.units as u
>>> from baseband import gsb
>>> from baseband.data import (
...     SAMPLE_GSB_RAWDUMP, SAMPLE_GSB_RAWDUMP_HEADER,
...     SAMPLE_GSB_PHASED, SAMPLE_GSB_PHASED_HEADER)
```

A single timestamp file can be opened with `open` in text mode:

```
>>> ft = gsb.open(SAMPLE_GSB_RAWDUMP_HEADER, 'rt')
>>> ft.read_timestamp()
<GSBRawdumpHeader gps: 2015 04 27 18 45 00 0.000000240>
>>> ft.close()
```

Reading payloads requires the samples per frame or sample rate. For phased the sample rate is:

```
sample_rate = raw_sample_rate / (2 * nchan)
```

where the raw sample rate is the pre-channelized one, and `nchan` the number of Fourier channels. The samples per frame for both rawdump and phased is:

```
samples_per_frame = timespan_of_frame * sample_rate
```

Note: Since the number of samples per frame is an integer number while both the frame timespan and sample rate are not, it is better to separately caculate `samples_per_frame` rather than multiplying `timespan_of_frame` with `sample_rate` in order to avoid rounding issues.

Alternatively, if the size of the frame buffer and the frame rate are known, the former can be used to determine `samples_per_frame`, and the latter used to determine `sample_rate` by inverting the above equation.

If `samples_per_frame` is not given, Baseband assumes it is the equivalent of 4 megabytes of data for rawdump, or 8 megabytes if phased. If `sample_rate` is not given, it is calculated from `samples_per_frame` assuming `timespan_of_frame = 0.25165824` (see *File Structure* above).

A single payload file can be opened with `open` in binary mode. Here, for our sample file, we have to take into account that in order to keep these files small, their sample size has been reduced to only **4 or 8 kilobytes** worth of samples per frame (for the default timespan). So, we define their sample rate here, and use that to calculate `payload_nbytes`, the size of one frame in bytes. Since rawdump samples are 4 bits, `payload_nbytes` is just `samples_per_frame / 2`:

```
>>> rawdump_samples_per_frame = 2**13
>>> payload_nbytes = rawdump_samples_per_frame // 2
>>> fb = gsb.open(SAMPLE_GSB_RAWDUMP, 'rb', payload_nbytes=payload_nbytes,
...               nchan=1, bps=4, complex_data=False)
>>> payload = fb.read_payload()
>>> payload[:4]
array([[ 0.],
       [-2.],
       [-2.],
       [ 0.]], dtype=float32)
>>> fb.close()
```

(`payload_nbytes` for phased data is the size of one frame *divided by the number of binary files*.)

Opening in stream mode allows timestamp and binary files to be read in concert to create data frames, and also wraps the low-level routines such that reading and writing is in units of samples, and provides access to header information.

When opening a rawdump file in stream mode, we pass the timestamp file as the first argument, and the binary file to the `raw` keyword. As per above, we also pass `samples_per_frame`:

```
>>> fh_rd = gsb.open(SAMPLE_GSB_RAWDUMP_HEADER, mode='rs',
...                  raw=SAMPLE_GSB_RAWDUMP,
...                  samples_per_frame=rawdump_samples_per_frame)
>>> fh_rd.header0
<GSBRawdumpHeader gps: 2015 04 27 18 45 00 0.000000240>
>>> dr = fh_rd.read()
>>> dr.shape
(81920,)
>>> dr[:3]
array([ 0., -2., -2.], dtype=float32)
>>> fh_rd.close()
```

To open a phased fileset in stream mode, we package the binary files into a nested tuple with the format:

```
((L pol stream 1, L pol stream 2), (R pol stream 1, R pol stream 2))
```

The nested tuple is passed to `raw` (note that we again have to pass a non-default sample rate):

```
>>> phased_samples_per_frame = 2**3
>>> fh_ph = gsb.open(SAMPLE_GSB_PHASED_HEADER, mode='rs',
...                  raw=SAMPLE_GSB_PHASED,
...                  samples_per_frame=phased_samples_per_frame)
>>> header0 = fh_ph.header0     # To be used for writing, below.
>>> dp = fh_ph.read()
>>> dp.shape
(80, 2, 512)
>>> dp[0, 0, :3]
array([30.+12.j, -1. +8.j,  7.+19.j], dtype=complex64)
>>> fh_ph.close()
```

To set up a file for writing, we need to pass names for both timestamp and raw files, as well as `sample_rate`, `samples_per_frame`, and either the first header or a `time` object. We first calculate `sample_rate`:

```
>>> timespan = 0.25165824 * u.s
>>> rawdump_sample_rate = (rawdump_samples_per_frame / timespan).to(u.MHz)
>>> phased_sample_rate = (phased_samples_per_frame / timespan).to(u.MHz)
```

To write a rawdump file:

```
>>> from astropy.time import Time
>>> fw_rd = gsb.open('test_rawdump.timestamp',
...                  mode='ws', raw='test_rawdump.dat',
...                  sample_rate=rawdump_sample_rate,
...                  samples_per_frame=rawdump_samples_per_frame,
...                  time=Time('2015-04-27T13:15:00'))
>>> fw_rd.write(dr)
>>> fw_rd.close()
>>> fh_rd = gsb.open('test_rawdump.timestamp', mode='rs',
...                  raw='test_rawdump.dat',
...                  sample_rate=rawdump_sample_rate,
...                  samples_per_frame=rawdump_samples_per_frame)
>>> np.all(dr == fh_rd.read())
True
>>> fh_rd.close()
```

To write a phased file, we need to pass a nested tuple of filenames or filehandles:

```
>>> test_phased_bin = (('test_phased_pL1.dat', 'test_phased_pL2.dat'),
...                    ('test_phased_pR1.dat', 'test_phased_pR2.dat'))
>>> fw_ph = gsb.open('test_phased.timestamp',
...                  mode='ws', raw=test_phased_bin,
...                  sample_rate=phased_sample_rate,
...                  samples_per_frame=phased_samples_per_frame,
...                  header0=header0)
>>> fw_ph.write(dp)
>>> fw_ph.close()
>>> fh_ph = gsb.open('test_phased.timestamp', mode='rs',
...                  raw=test_phased_bin,
...                  sample_rate=phased_sample_rate,
...                  samples_per_frame=phased_samples_per_frame)
>>> np.all(dp == fh_ph.read())
True
>>> fh_ph.close()
```

Baseband does not use the PC time in the phased header, and, when writing, simply uses the same time for both GPS and PC times. Since the PC time can drift from the GPS time by several tens of milliseconds, `test_phased.timestamp` will not be identical to `SAMPLE_GSB_PHASED`, even though we have written the exact same data to file.

9.3 Reference/API

9.3.1 baseband.gsb Package

GMRT Software Backend (GSB) data reader.

See http://gmrt.ncra.tifr.res.in/gmrt_hpage/sub_system/gmrt_gsb/index.htm

Functions

open(name[, mode])	Open GSB file(s) for reading or writing.

open

`baseband.gsb.open(name, mode=u'rs', **kwargs)`

Open GSB file(s) for reading or writing.

A GSB data set contains a text header file and one or more raw data files. When the file is opened as text, one gets a standard filehandle, but with methods to read/write timestamps. When it is opened as a binary, one similarly gets methods to read/write frames. Opened as a stream, the file is interpreted as a timestamp file, but raw files need to be given too. This allows access to the stream(s) as series of samples.

Parameters

name : str

Filename of timestamp or raw data file.

mode : {'rb', 'wb', 'rt', 'wt', 'rs', or 'ws'}, optional

Whether to open for reading or writing, and as a regular text or binary file (for timestamps and data, respectively) or as a stream. Default: 'rs', for reading a stream.

****kwargs**

Additional arguments when opening the file as a stream.

— For both reading and writing of streams :

raw : str or (tuple of) tuple of str

Name of files holding payload data. A single file is needed for rawdump, and a tuple for phased. For a nested tuple, the outer tuple determines the number of polarizations, and the inner tuple(s) the number of streams per polarization. E.g., `((polL1, polL2), (polR1, polR2))` for two streams per polarization. A single tuple is interpreted as streams of a single polarization.

sample_rate : `Quantity`, optional

Number of complete samples per second, i.e. the rate at which each channel of each polarization is sampled. If `None`, will be inferred assuming the frame rate is exactly 251.658240 ms.

samples_per_frame : int, optional

Number of complete samples per frame. Can give `payload_nbytes` instead.

payload_nbytes : int, optional

Number of bytes per payload, divided by the number of raw files. If both `samples_per_frame` and `payload_nbytes` are `None`, `payload_nbytes` is set to `2**22` (4 MB) for rawdump, and `2**23` (8 MB) divided by the number of streams per polarization for phased.

nchan : int, optional

Number of channels. Default: 1 for rawdump, 512 for phased.

bps : int, optional

Bits per elementary sample, i.e. per real or imaginary component for complex data. Default: 4 for rawdump, 8 for phased.

complex_data : bool, optional

Whether data are complex. Default: `False` for rawdump, `True` for phased.

squeeze : bool, optional

If True (default) and reading, remove any dimensions of length unity from decoded data. If True and writing, accept squeezed arrays as input, and adds any dimensions of length unity.

— For reading only : (see GSBStreamReader)

subset : indexing object or tuple of objects, optional

Specific components of the complete sample to decode (after possibly squeezing). If a single indexing object is passed, it selects (available) polarizations. If a tuple is passed, the first selects polarizations and the second selects channels. If the tuple is empty (default), all components are read.

verify : bool, optional

Whether to do basic checks of frame integrity when reading. The first frame of the stream is always checked. Default: True.

— For writing only : (see GSBStreamWriter)

header0 : GSBHeader

Header for the first frame, holding time information, etc. Can instead give keyword arguments to construct a header.

****kwargs**

If the header is not given, an attempt will be made to construct one with any further keyword arguments. If one requires to explicitly set the mode of the GSB stream, use `header_mode`. If not given, it will be 'rawdump' if only a single raw file is present, or 'phased' otherwise. See GSBStreamWriter.

Returns

Filehandle

GSBFileReader or GSBFileWriter (binary), or GSBStreamReader or GSBStreamWriter (stream)

Classes

GSBFrame(header, payload[, valid, verify])	Frame encapsulating GSB rawdump or phased data.
GSBHeader(words[, mode, nbytes, utc_offset, . . .])	GSB Header, based on a line from a timestamp file.
GSBPayload(words[, sample_shape, bps, . . .])	Container for decoding and encoding GSB payloads.

GSBFrame

class baseband.gsb.**GSBFrame**(*header*, *payload*, *valid=True*, *verify=True*)

Bases: `baseband.vlbi_base.frame.VLBIFrameBase`

Frame encapsulating GSB rawdump or phased data.

For rawdump data, lines in the timestamp file are associated with single blocks of raw data. For phased data, the lines are associated with one or two polarisations, each consisting of two blocks of raw data. Hence, the raw data come from two or four files.

Parameters

header : GSBHeader

Based on line from rawdump or phased timestamp file.

payload : GSBPayload

Based on a single block of rawdump data, or the combined blocks for phased data.

valid : bool, optional

Whether the data are valid. Default: True.

verify : bool, optional

Whether to verify consistency of the frame parts. Default: True.

Notes

GSB files do not support storing whether data are valid or not on disk. Hence, this has to be determined independently. If `valid=False`, any decoded data are set to `cls.fill_value` (by default, 0).

The Frame can also be read instantiated using class methods:

fromfile : read header and payload from their respective filehandles

fromdata : encode data as payload

Of course, one can also do the opposite:

tofile
[method to write header and payload to filehandles (splitting] payload in the appropriate files).

data : property that yields full decoded payload

A number of properties are defined: `shape`, `dtype` and `size` are the shape, type and number of complete samples of the data array, and `nbytes` the frame size in bytes. Furthermore, the frame acts as a dictionary, with keys those of the header. Any attribute that is not defined on the frame itself, such as `.time` will be looked up on the header as well.

Attributes Summary

`data`	Full decoded frame.
`dtype`	Numeric type of the frame data.
`fill_value`	Value to replace invalid data in the frame.
`nbytes`	Size of the encoded frame in the raw data file in bytes.
`ndim`	Number of dimensions of the frame data.
`sample_shape`	Shape of a sample in the frame (nchan,).
`shape`	Shape of the frame data.
`size`	Total number of component samples in the frame data.
`valid`	Whether frame contains valid data.

Methods Summary

`fromdata`(data, header, *args, **kwargs)	Construct frame from data and header.
`fromfile`(fh_ts, fh_raw[, payload_nbytes, …])	Read a frame from timestamp and raw data filehandles.
`keys`()	

Continued on next page

Table 4 – continued from previous page

tofile(fh_ts, fh_raw)	Write encoded frame to timestamp and raw data filehandles.
verify()	Simple verification.

Attributes Documentation

data
Full decoded frame.

dtype
Numeric type of the frame data.

fill_value
Value to replace invalid data in the frame.

nbytes
Size of the encoded frame in the raw data file in bytes.

ndim
Number of dimensions of the frame data.

sample_shape
Shape of a sample in the frame (nchan,).

shape
Shape of the frame data.

size
Total number of component samples in the frame data.

valid
Whether frame contains valid data.

Methods Documentation

classmethod fromdata(*data*, *header*, **args*, ***kwargs*)
Construct frame from data and header.

Parameters

data : ndarray

Array holding data to be encoded.

header : VLBIHeaderBase

Header for the frame.

***args, **kwargs :**

Any arguments beyond the filehandle are used to help initialize the payload, except for `valid` and `verify`, which are passed on to the header and class initializers.

classmethod fromfile(*fh_ts*, *fh_raw*, *payload_nbytes=16777216*, *nchan=1*, *bps=4*, *complex_data=False*, *valid=True*, *verify=True*)
Read a frame from timestamp and raw data filehandles.

Any arguments beyond the filehandle are used to help initialize the payload, except for `valid` and `verify`, which are passed on to the header and class initializers.

Parameters

fh_ts : filehandle

To the timestamp file. The next line will be read.

fh_raw : file_handle or tuple

Should be a single handle for a rawdump data frame, or a tuple containing tuples with pairs of handles for a phased one. E.g., `((L1, L2), (R1, R2))` for left and right polarisations.

payload_nbytes : int, optional

Size of the individual payloads in bytes. Default: `2**24` (16 MB).

nchan : int, optional

Number of channels. Default: 1.

bps : int, optional

Bits per elementary sample. Default: 4.

complex_data : bool, optional

Whether data are complex. Default: `False`.

valid : bool, optional

Whether the data are valid (default: `True`). Note that this cannot be inferred from the header or payload itself. If `False`, any data read will be set to `cls.fill_value`.

verify : bool, optional

Whether to verify consistency of the frame parts. Default: `True`.

keys()

tofile(*fh_ts*, *fh_raw*)

Write encoded frame to timestamp and raw data filehandles.

Parameters

fh_ts : filehandle

To the timestamp file. A line will be added to it.

fh_raw : file_handle or tuple

Should be a single handle for a rawdump data frame, or a tuple containing tuples with pairs of handles for a phased one. E.g., `((L1, L2), (R1, R2))` for left and right polarisations.

verify()

Simple verification. To be added to by subclasses.

GSBHeader

class baseband.gsb.**GSBHeader**(*words*, *mode=None*, *nbytes=None*, *utc_offset=<Quantity 5.5 h>*, *verify=True*)

Bases: `baseband.vlbi_base.header.VLBIHeaderBase`

GSB Header, based on a line from a timestamp file.

Parameters

words : list of str, or None

If `None`, set to a list of empty strings for later initialisation.

mode : str or None, optional

Mode in which data was taken: 'phased' or 'rawdump'. If `None`, it is determined from the words.

nbytes : int or None, optional

Number of characters in the header, including trailing blank spaces and carriage returns. If `None`, is determined from the words assuming one trailing blank space and one CR.

verify : bool, optional

Whether to do basic verification of integrity. Default: `True`.

Returns

header : `GSBHeader` subclass

As appropriate for the mode.

Attributes Summary

`mode`	Mode in which data was taken: 'phased' or 'rawdump'.
`mutable`	Whether the header can be modified.
`nbytes`	Size of the header in characters.

Methods Summary

`copy`(**kwargs)	Create a mutable and independent copy of the header.
`fromfile`(fh, *args, **kwargs)	Read GSB Header from a line from a timestamp file.
`fromkeys`([mode, nbytes])	Initialise a header from parsed values.
`fromvalues`([mode, nbytes])	Initialise a header from parsed values.
`keys`()	
`seek_offset`(n[, nbytes])	Offset in bytes needed to move a file pointer to another header.
`tofile`(fh)	Write GSB header as a line to the filehandle.
`update`(**kwargs)	Update the header by setting keywords or properties.
`verify`()	Verify that the length of the words is consistent.

Attributes Documentation

mode

Mode in which data was taken: 'phased' or 'rawdump'.

mutable

Whether the header can be modified.

nbytes

Size of the header in characters.

Assumes the string terminates in one blank space and one carriage return.

Methods Documentation

copy(***kwargs*)
Create a mutable and independent copy of the header.

Keyword arguments can be passed on as needed by possible subclasses.

classmethod fromfile(*fh*, **args*, ***kwargs*)
Read GSB Header from a line from a timestamp file.

Arguments are the same as for class initialisation. The header constructed will be immutable.

classmethod fromkeys(*mode=None*, *nbytes=None*, **args*, ***kwargs*)
Initialise a header from parsed values.

Like fromvalues, but without any interpretation of keywords.

Raises
KeyError : if not all keys required are present in `kwargs`

classmethod fromvalues(*mode=None*, *nbytes=None*, **args*, ***kwargs*)
Initialise a header from parsed values.

Here, the parsed values must be given as keyword arguments, i.e., for any `header = cls(<words>)`, `cls.fromvalues(**header) == header`.

However, unlike for the `fromkeys` class method, data can also be set using arguments named after header methods, such as `time`.

Parameters
***args**

Possible arguments required to initialize an empty header.

****kwargs**

Values used to initialize header keys or methods.

keys()

seek_offset(*n*, *nbytes=None*)
Offset in bytes needed to move a file pointer to another header.

Some GSB headers have variable size and hence one cannot trivially jump to another entry in a timestamp file. This routine allows one to calculate the offset required to move the file pointer `n` headers.

Parameters
n : int

The number of headers to move to, relative to the present header.

nbytes : int, optional

The size in bytes of the present header (if not given, will use the header's `nbytes` property).

tofile(*fh*)
Write GSB header as a line to the filehandle.

update(***kwargs*)
Update the header by setting keywords or properties.

Here, any keywords matching header keys are applied first, and any remaining ones are used to set header properties, in the order set by the class (in `_properties`).

Parameters

verify : bool, optional

If `True` (default), verify integrity after updating.

****kwargs**

Arguments used to set keywords and properties.

`verify()`

Verify that the length of the words is consistent.

Subclasses should override this to do more thorough checks.

GSBPayload

`class baseband.gsb.GSBPayload(words, sample_shape=(), bps=2, complex_data=False)`

Bases: `baseband.vlbi_base.payload.VLBIPayloadBase`

Container for decoding and encoding GSB payloads.

Parameters

words : `ndarray`

Array containg LSB unsigned words (with the right size) that encode the payload.

sample_shape : tuple, optional

Shape of the samples; e.g., (nchan,). Default: ().

bps : int, optional

Bits per elementary sample. Default: 2.

complex_data : bool, optional

Whether data are complex. Default: `False`.

Attributes Summary

`data`	Full decoded payload.
`dtype`	Numeric type of the decoded data array.
`nbytes`	Size of the payload in bytes.
`ndim`	Number of dimensions of the decoded data array.
`shape`	Shape of the decoded data array.
`size`	Total number of component samples in the decoded data array.

Methods Summary

`fromdata`(data[, header, bps])	Encode data as a payload.
`fromfile`(fh[, payload_nbytes, nchan, bps, . . .])	Read payloads from several threads.
`tofile`(fh)	

Attributes Documentation

data
Full decoded payload.

dtype
Numeric type of the decoded data array.

nbytes
Size of the payload in bytes.

ndim
Number of dimensions of the decoded data array.

shape
Shape of the decoded data array.

size
Total number of component samples in the decoded data array.

Methods Documentation

classmethod fromdata(*data*, *header=None*, *bps=2*)
Encode data as a payload.

Parameters

data : `ndarray`

Data to be encoded. The last dimension is taken as the number of channels.

header : header instance, optional

If given, used to infer the bps.

bps : int, optional

Bits per elementary sample, i.e., per channel and per real or imaginary component, used if header is not given. Default: 2.

classmethod fromfile(*fh*, *payload_nbytes=None*, *nchan=1*, *bps=4*, *complex_data=False*)
Read payloads from several threads.

Parameters

fh : filehandle or tuple of tuple of filehandle

Handles to the sets of files from which data are read. The outer tuple holds distinct threads, while the inner ones holds parts of those threads. Typically, these are the two polarisations and the two parts of each in which phased baseband data are stored.

payload_nbytes : int

Number of bytes to read from each part.

nchan : int, optional

Number of channels. Default: 1.

bps : int, optional

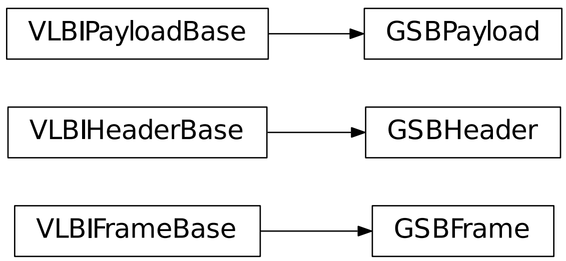
Bits per elementary sample. Default: 4.

complex_data : bool, optional

Whether data are complex. Default: `False`.

tofile(*fh*)

Class Inheritance Diagram



9.3.2 baseband.gsb.header Module

Definitions for GSB Headers, using the timestamp files.

Somewhat out of data description for phased data: http://gmrt.ncra.tifr.res.in/gmrt_hpage/sub_system/gmrt_gsb/GSB_beam_timestamp_note_v1.pdf and for rawdump data http://gmrt.ncra.tifr.res.in/gmrt_hpage/sub_system/gmrt_gsb/GSB_rawdump_data_format_v2.pdf

Classes

`TimeGSB`(val1, val2, scale, precision, …[, …])	GSB header date-time format `YYYY MM DD HH MM SS 0.SSSSSSSSS`.
`GSBHeader`(words[, mode, nbytes, utc_offset, …])	GSB Header, based on a line from a timestamp file.
`GSBRawdumpHeader`(words[, mode, nbytes, …])	GSB rawdump header.
`GSBPhasedHeader`(words[, mode, nbytes, …])	GSB phased header.

TimeGSB

class baseband.gsb.header.**TimeGSB**(*val1*, *val2*, *scale*, *precision*, *in_subfmt*, *out_subfmt*, *from_jd=False*)
Bases: `astropy.time.TimeString`

GSB header date-time format `YYYY MM DD HH MM SS 0.SSSSSSSSS`.

For example, `2000 01 01 00 00 00 0.000000000` is midnight on January 1, 2000.

Attributes Summary

`c`	

Continued on next page

Table 10 – continued from previous page

name	
scale	Time scale
value	

Methods Summary

format_string(str_fmt, **kwargs)	Write time to a string using a given format.
parse_string(timestr, subfmts)	Read time from a single string, using a set of possible formats.
set_jds(val1, val2)	Parse the time strings contained in val1 and set jd1, jd2
str_kwargs()	Generator that yields a dict of values corresponding to the calendar date and time for the internal JD values.
to_value([parent])	

Attributes Documentation

c = u'f'

name = u'gsb'

scale
Time scale

value

Methods Documentation

format_string(*str_fmt*, ***kwargs*)
Write time to a string using a given format.

By default, just interprets str_fmt as a format string, but subclasses can add to this.

parse_string(*timestr*, *subfmts*)
Read time from a single string, using a set of possible formats.

set_jds(*val1*, *val2*)
Parse the time strings contained in val1 and set jd1, jd2

str_kwargs()
Generator that yields a dict of values corresponding to the calendar date and time for the internal JD values.

to_value(*parent=None*)

GSBHeader

class baseband.gsb.header.**GSBHeader**(*words*, *mode=None*, *nbytes=None*, *utc_offset=<Quantity 5.5 h>*, *verify=True*)

Bases: `baseband.vlbi_base.header.VLBIHeaderBase`

GSB Header, based on a line from a timestamp file.

Parameters

words : list of str, or None

If `None`, set to a list of empty strings for later initialisation.

mode : str or None, optional

Mode in which data was taken: 'phased' or 'rawdump'. If `None`, it is determined from the words.

nbytes : int or None, optional

Number of characters in the header, including trailing blank spaces and carriage returns. If `None`, is determined from the words assuming one trailing blank space and one CR.

verify : bool, optional

Whether to do basic verification of integrity. Default: `True`.

Returns

header : `GSBHeader` subclass

As appropriate for the mode.

Attributes Summary

`mode`	Mode in which data was taken: 'phased' or 'rawdump'.
`mutable`	Whether the header can be modified.
`nbytes`	Size of the header in characters.

Methods Summary

`copy`(**kwargs)	Create a mutable and independent copy of the header.
`fromfile`(fh, *args, **kwargs)	Read GSB Header from a line from a timestamp file.
`fromkeys`([mode, nbytes])	Initialise a header from parsed values.
`fromvalues`([mode, nbytes])	Initialise a header from parsed values.
`keys`()	
`seek_offset`(n[, nbytes])	Offset in bytes needed to move a file pointer to another header.
`tofile`(fh)	Write GSB header as a line to the filehandle.
`update`(**kwargs)	Update the header by setting keywords or properties.
`verify`()	Verify that the length of the words is consistent.

Attributes Documentation

mode
Mode in which data was taken: 'phased' or 'rawdump'.

mutable
Whether the header can be modified.

nbytes
Size of the header in characters.

Assumes the string terminates in one blank space and one carriage return.

Methods Documentation

copy(***kwargs*)
Create a mutable and independent copy of the header.

Keyword arguments can be passed on as needed by possible subclasses.

classmethod fromfile(*fh*, **args*, ***kwargs*)
Read GSB Header from a line from a timestamp file.

Arguments are the same as for class initialisation. The header constructed will be immutable.

classmethod fromkeys(*mode=None*, *nbytes=None*, **args*, ***kwargs*)
Initialise a header from parsed values.

Like fromvalues, but without any interpretation of keywords.

Raises
KeyError : if not all keys required are present in `kwargs`

classmethod fromvalues(*mode=None*, *nbytes=None*, **args*, ***kwargs*)
Initialise a header from parsed values.

Here, the parsed values must be given as keyword arguments, i.e., for any `header = cls(<words>)`, `cls.fromvalues(**header) == header`.

However, unlike for the `fromkeys` class method, data can also be set using arguments named after header methods, such as `time`.

Parameters
***args**

Possible arguments required to initialize an empty header.

****kwargs**

Values used to initialize header keys or methods.

keys()

seek_offset(*n*, *nbytes=None*)
Offset in bytes needed to move a file pointer to another header.

Some GSB headers have variable size and hence one cannot trivially jump to another entry in a timestamp file. This routine allows one to calculate the offset required to move the file pointer `n` headers.

Parameters
n : int

The number of headers to move to, relative to the present header.

nbytes : int, optional

The size in bytes of the present header (if not given, will use the header's `nbytes` property).

tofile(*fh*)
Write GSB header as a line to the filehandle.

update(***kwargs*)
Update the header by setting keywords or properties.

Here, any keywords matching header keys are applied first, and any remaining ones are used to set header properties, in the order set by the class (in `_properties`).

Parameters

verify : bool, optional

If `True` (default), verify integrity after updating.

****kwargs**

Arguments used to set keywords and properties.

verify()
Verify that the length of the words is consistent.

Subclasses should override this to do more thorough checks.

GSBRawdumpHeader

class baseband.gsb.header.**GSBRawdumpHeader**(*words*, *mode=None*, *nbytes=None*, *utc_offset=<Quantity 5.5 h>*, *verify=True*)

Bases: `baseband.gsb.header.GSBHeader`

GSB rawdump header.

Attributes Summary

`gps_time`	
`mode`	Mode in which data was taken: 'phased' or 'rawdump'.
`mutable`	Whether the header can be modified.
`nbytes`	Size of the header in characters.
`time`	

Methods Summary

`copy`(**kwargs)	Create a mutable and independent copy of the header.
`fromfile`(fh, *args, **kwargs)	Read GSB Header from a line from a timestamp file.
`fromkeys`([mode, nbytes])	Initialise a header from parsed values.
`fromvalues`([mode, nbytes])	Initialise a header from parsed values.
`keys`()	

Continued on next page

Table 15 – continued from previous page

seek_offset(n[, nbytes])	Offset in bytes needed to move a file pointer to another header.
tofile(fh)	Write GSB header as a line to the filehandle.
update(**kwargs)	Update the header by setting keywords or properties.
verify()	Verify that the length of the words is consistent.

Attributes Documentation

gps_time

mode
Mode in which data was taken: 'phased' or 'rawdump'.

mutable
Whether the header can be modified.

nbytes
Size of the header in characters.

Assumes the string terminates in one blank space and one carriage return.

time

Methods Documentation

copy(***kwargs*)
Create a mutable and independent copy of the header.

Keyword arguments can be passed on as needed by possible subclasses.

classmethod fromfile(*fh*, **args*, ***kwargs*)
Read GSB Header from a line from a timestamp file.

Arguments are the same as for class initialisation. The header constructed will be immutable.

classmethod fromkeys(*mode=None*, *nbytes=None*, **args*, ***kwargs*)
Initialise a header from parsed values.

Like fromvalues, but without any interpretation of keywords.

Raises
KeyError : if not all keys required are present in `kwargs`

classmethod fromvalues(*mode=None*, *nbytes=None*, **args*, ***kwargs*)
Initialise a header from parsed values.

Here, the parsed values must be given as keyword arguments, i.e., for any `header = cls(<words>)`, `cls.fromvalues(**header) == header`.

However, unlike for the `fromkeys` class method, data can also be set using arguments named after header methods, such as `time`.

Parameters
***args**

Possible arguments required to initialize an empty header.

****kwargs**

Values used to initialize header keys or methods.

keys()

seek_offset(*n*, *nbytes=None*)
Offset in bytes needed to move a file pointer to another header.

Some GSB headers have variable size and hence one cannot trivially jump to another entry in a timestamp file. This routine allows one to calculate the offset required to move the file pointer n headers.

Parameters
n : int

The number of headers to move to, relative to the present header.

nbytes : int, optional

The size in bytes of the present header (if not given, will use the header's nbytes property).

tofile(*fh*)
Write GSB header as a line to the filehandle.

update(***kwargs*)
Update the header by setting keywords or properties.

Here, any keywords matching header keys are applied first, and any remaining ones are used to set header properties, in the order set by the class (in _properties).

Parameters
verify : bool, optional

If True (default), verify integrity after updating.

****kwargs**

Arguments used to set keywords and properties.

verify()
Verify that the length of the words is consistent.

Subclasses should override this to do more thorough checks.

GSBPhasedHeader

class baseband.gsb.header.**GSBPhasedHeader**(*words*, *mode=None*, *nbytes=None*, *utc_offset=<Quantity 5.5 h>*, *verify=True*)
Bases: baseband.gsb.header.GSBRawdumpHeader

GSB phased header.

Attributes Summary

gps_time	
mode	Mode in which data was taken: 'phased' or 'rawdump'.
mutable	Whether the header can be modified.
nbytes	Size of the header in characters.

Continued on next page

Table 16 – continued from previous page

pc_time	
time	

Methods Summary

copy(**kwargs)	Create a mutable and independent copy of the header.
fromfile(fh, *args, **kwargs)	Read GSB Header from a line from a timestamp file.
fromkeys([mode, nbytes])	Initialise a header from parsed values.
fromvalues([mode, nbytes])	Initialise a header from parsed values.
keys()	
seek_offset(n[, nbytes])	Offset in bytes needed to move a file pointer to another header.
tofile(fh)	Write GSB header as a line to the filehandle.
update(**kwargs)	Update the header by setting keywords or properties.
verify()	Verify that the length of the words is consistent.

Attributes Documentation

gps_time

mode
Mode in which data was taken: 'phased' or 'rawdump'.

mutable
Whether the header can be modified.

nbytes
Size of the header in characters.

Assumes the string terminates in one blank space and one carriage return.

pc_time

time

Methods Documentation

copy(***kwargs*)
Create a mutable and independent copy of the header.

Keyword arguments can be passed on as needed by possible subclasses.

classmethod fromfile(*fh*, **args*, ***kwargs*)
Read GSB Header from a line from a timestamp file.

Arguments are the same as for class initialisation. The header constructed will be immutable.

classmethod fromkeys(*mode=None*, *nbytes=None*, **args*, ***kwargs*)
Initialise a header from parsed values.

Like fromvalues, but without any interpretation of keywords.

Raises

KeyError : if not all keys required are present in `kwargs`

classmethod fromvalues(*mode=None*, *nbytes=None*, **args*, ***kwargs*)

Initialise a header from parsed values.

Here, the parsed values must be given as keyword arguments, i.e., for any `header = cls(<words>)`, `cls.fromvalues(**header) == header`.

However, unlike for the `fromkeys` class method, data can also be set using arguments named after header methods, such as `time`.

Parameters

***args**

Possible arguments required to initialize an empty header.

****kwargs**

Values used to initialize header keys or methods.

keys()

seek_offset(*n*, *nbytes=None*)

Offset in bytes needed to move a file pointer to another header.

GSB headers for phased data differ in size depending on the sequence number, making it impossible to trivially jump to another entry in a timestamp file. This routine allows one to calculate the offset required to move the file pointer `n` headers.

Parameters

n : int

The number of headers to move to, relative to the present header.

nbytes : int, optional

The size in bytes of the present header (if not given, will use the header's `nbytes` property).

tofile(*fh*)

Write GSB header as a line to the filehandle.

update(***kwargs*)

Update the header by setting keywords or properties.

Here, any keywords matching header keys are applied first, and any remaining ones are used to set header properties, in the order set by the class (in `_properties`).

Parameters

verify : bool, optional

If `True` (default), verify integrity after updating.

****kwargs**

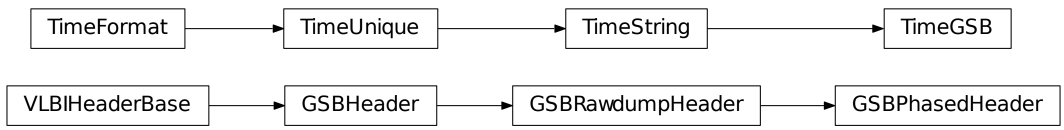
Arguments used to set keywords and properties.

verify()

Verify that the length of the words is consistent.

Subclasses should override this to do more thorough checks.

Class Inheritance Diagram



9.3.3 baseband.gsb.payload Module

Definitions for GSB payloads.

Implements a GSBPayload class used to store payload blocks, and decode to or encode from a data array.

See http://gmrt.ncra.tifr.res.in/gmrt_hpage/sub_system/gmrt_gsb/index.htm

Classes

GSBPayload(words[, sample_shape, bps, . . .])	Container for decoding and encoding GSB payloads.

GSBPayload

class baseband.gsb.payload.**GSBPayload**(*words*, *sample_shape=()*, *bps=2*, *complex_data=False*)

Bases: baseband.vlbi_base.payload.VLBIPayloadBase

Container for decoding and encoding GSB payloads.

Parameters

words : ndarray

Array containing LSB unsigned words (with the right size) that encode the payload.

sample_shape : tuple, optional

Shape of the samples; e.g., (nchan,). Default: ().

bps : int, optional

Bits per elementary sample. Default: 2.

complex_data : bool, optional

Whether data are complex. Default: False.

Attributes Summary

data	Full decoded payload.
dtype	Numeric type of the decoded data array.

Continued on next page

Table 19 – continued from previous page

nbytes	Size of the payload in bytes.
ndim	Number of dimensions of the decoded data array.
shape	Shape of the decoded data array.
size	Total number of component samples in the decoded data array.

Methods Summary

fromdata(data[, header, bps])	Encode data as a payload.
fromfile(fh[, payload_nbytes, nchan, bps, . . .])	Read payloads from several threads.
tofile(fh)	

Attributes Documentation

data
Full decoded payload.

dtype
Numeric type of the decoded data array.

nbytes
Size of the payload in bytes.

ndim
Number of dimensions of the decoded data array.

shape
Shape of the decoded data array.

size
Total number of component samples in the decoded data array.

Methods Documentation

classmethod fromdata(*data*, *header=None*, *bps=2*)
Encode data as a payload.

Parameters

data : ndarray

Data to be encoded. The last dimension is taken as the number of channels.

header : header instance, optional

If given, used to infer the bps.

bps : int, optional

Bits per elementary sample, i.e., per channel and per real or imaginary component, used if header is not given. Default: 2.

classmethod fromfile(*fh*, *payload_nbytes=None*, *nchan=1*, *bps=4*, *complex_data=False*)
Read payloads from several threads.

Parameters

fh : filehandle or tuple of tuple of filehandle

Handles to the sets of files from which data are read. The outer tuple holds distinct threads, while the inner ones holds parts of those threads. Typically, these are the two polarisations and the two parts of each in which phased baseband data are stored.

payload_nbytes : int

Number of bytes to read from each part.

nchan : int, optional

Number of channels. Default: 1.

bps : int, optional

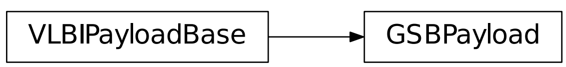
Bits per elementary sample. Default: 4.

complex_data : bool, optional

Whether data are complex. Default: `False`.

tofile(*fh*)

Class Inheritance Diagram



9.3.4 baseband.gsb.frame Module

Classes

`GSBFrame`(header, payload[, valid, verify])	Frame encapsulating GSB rawdump or phased data.

GSBFrame

class baseband.gsb.frame.**GSBFrame**(*header*, *payload*, *valid=True*, *verify=True*)

Bases: `baseband.vlbi_base.frame.VLBIFrameBase`

Frame encapsulating GSB rawdump or phased data.

For rawdump data, lines in the timestamp file are associated with single blocks of raw data. For phased data, the lines are associated with one or two polarisations, each consisting of two blocks of raw data. Hence, the raw data come from two or four files.

Parameters

header : `GSBHeader`

Based on line from rawdump or phased timestamp file.

payload : `GSBPayload`

Based on a single block of rawdump data, or the combined blocks for phased data.

valid : bool, optional

Whether the data are valid. Default: True.

verify : bool, optional

Whether to verify consistency of the frame parts. Default: True.

Notes

GSB files do not support storing whether data are valid or not on disk. Hence, this has to be determined independently. If valid=False, any decoded data are set to cls.fill_value (by default, 0).

The Frame can also be read instantiated using class methods:

fromfile : read header and payload from their respective filehandles

fromdata : encode data as payload

Of course, one can also do the opposite:

tofile
[method to write header and payload to filehandles (splitting] payload in the appropriate files).

data : property that yields full decoded payload

A number of properties are defined: shape, dtype and size are the shape, type and number of complete samples of the data array, and nbytes the frame size in bytes. Furthermore, the frame acts as a dictionary, with keys those of the header. Any attribute that is not defined on the frame itself, such as .time will be looked up on the header as well.

Attributes Summary

data	Full decoded frame.
dtype	Numeric type of the frame data.
fill_value	Value to replace invalid data in the frame.
nbytes	Size of the encoded frame in the raw data file in bytes.
ndim	Number of dimensions of the frame data.
sample_shape	Shape of a sample in the frame (nchan,).
shape	Shape of the frame data.
size	Total number of component samples in the frame data.
valid	Whether frame contains valid data.

Methods Summary

fromdata(data, header, *args, **kwargs)	Construct frame from data and header.
fromfile(fh_ts, fh_raw[, payload_nbytes, . . .])	Read a frame from timestamp and raw data filehandles.
keys()	

Continued on next page

Table 23 – continued from previous page

tofile(fh_ts, fh_raw)	Write encoded frame to timestamp and raw data filehandles.
verify()	Simple verification.

Attributes Documentation

data
Full decoded frame.

dtype
Numeric type of the frame data.

fill_value
Value to replace invalid data in the frame.

nbytes
Size of the encoded frame in the raw data file in bytes.

ndim
Number of dimensions of the frame data.

sample_shape
Shape of a sample in the frame (nchan,).

shape
Shape of the frame data.

size
Total number of component samples in the frame data.

valid
Whether frame contains valid data.

Methods Documentation

classmethod fromdata(*data*, *header*, **args*, ***kwargs*)
Construct frame from data and header.

Parameters

data : ndarray

Array holding data to be encoded.

header : VLBIHeaderBase

Header for the frame.

***args, **kwargs :**

Any arguments beyond the filehandle are used to help initialize the payload, except for `valid` and `verify`, which are passed on to the header and class initializers.

classmethod fromfile(*fh_ts*, *fh_raw*, *payload_nbytes=16777216*, *nchan=1*, *bps=4*, *complex_data=False*, *valid=True*, *verify=True*)
Read a frame from timestamp and raw data filehandles.

Any arguments beyond the filehandle are used to help initialize the payload, except for `valid` and `verify`, which are passed on to the header and class initializers.

Parameters

fh_ts : filehandle

To the timestamp file. The next line will be read.

fh_raw : file_handle or tuple

Should be a single handle for a rawdump data frame, or a tuple containing tuples with pairs of handles for a phased one. E.g., `((L1, L2), (R1, R2))` for left and right polarisations.

payload_nbytes : int, optional

Size of the individual payloads in bytes. Default: `2**24` (16 MB).

nchan : int, optional

Number of channels. Default: 1.

bps : int, optional

Bits per elementary sample. Default: 4.

complex_data : bool, optional

Whether data are complex. Default: `False`.

valid : bool, optional

Whether the data are valid (default: `True`). Note that this cannot be inferred from the header or payload itself. If `False`, any data read will be set to `cls.fill_value`.

verify : bool, optional

Whether to verify consistency of the frame parts. Default: `True`.

keys()

tofile(*fh_ts*, *fh_raw*)

Write encoded frame to timestamp and raw data filehandles.

Parameters

fh_ts : filehandle

To the timestamp file. A line will be added to it.

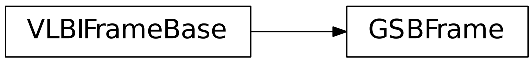
fh_raw : file_handle or tuple

Should be a single handle for a rawdump data frame, or a tuple containing tuples with pairs of handles for a phased one. E.g., `((L1, L2), (R1, R2))` for left and right polarisations.

verify()

Simple verification. To be added to by subclasses.

Class Inheritance Diagram



9.3.5 baseband.gsb.base Module

Functions

open(name[, mode])	Open GSB file(s) for reading or writing.

open

baseband.gsb.base.**open**(*name*, *mode=u'rs'*, ***kwargs*)

Open GSB file(s) for reading or writing.

A GSB data set contains a text header file and one or more raw data files. When the file is opened as text, one gets a standard filehandle, but with methods to read/write timestamps. When it is opened as a binary, one similarly gets methods to read/write frames. Opened as a stream, the file is interpreted as a timestamp file, but raw files need to be given too. This allows access to the stream(s) as series of samples.

Parameters

name : str

Filename of timestamp or raw data file.

mode : {'rb', 'wb', 'rt', 'wt', 'rs', or 'ws'}, optional

Whether to open for reading or writing, and as a regular text or binary file (for timestamps and data, respectively) or as a stream. Default: 'rs', for reading a stream.

****kwargs**

Additional arguments when opening the file as a stream.

— For both reading and writing of streams :

raw : str or (tuple of) tuple of str

Name of files holding payload data. A single file is needed for rawdump, and a tuple for phased. For a nested tuple, the outer tuple determines the number of polarizations, and the inner tuple(s) the number of streams per polarization. E.g., `((polL1, polL2), (polR1, polR2))` for two streams per polarization. A single tuple is interpreted as streams of a single polarization.

sample_rate : `Quantity`, optional

Number of complete samples per second, i.e. the rate at which each channel of each polarization is sampled. If `None`, will be inferred assuming the frame rate is exactly 251.658240 ms.

samples_per_frame : int, optional

Number of complete samples per frame. Can give `payload_nbytes` instead.

payload_nbytes : int, optional

Number of bytes per payload, divided by the number of raw files. If both `samples_per_frame` and `payload_nbytes` are `None`, `payload_nbytes` is set to `2**22` (4 MB) for rawdump, and `2**23` (8 MB) divided by the number of streams per polarization for phased.

nchan : int, optional

Number of channels. Default: 1 for rawdump, 512 for phased.

bps : int, optional

Bits per elementary sample, i.e. per real or imaginary component for complex data. Default: 4 for rawdump, 8 for phased.

complex_data : bool, optional

Whether data are complex. Default: `False` for rawdump, `True` for phased.

squeeze : bool, optional

If `True` (default) and reading, remove any dimensions of length unity from decoded data. If `True` and writing, accept squeezed arrays as input, and adds any dimensions of length unity.

— For reading only : (see `GSBStreamReader`)

subset : indexing object or tuple of objects, optional

Specific components of the complete sample to decode (after possibly squeezing). If a single indexing object is passed, it selects (available) polarizations. If a tuple is passed, the first selects polarizations and the second selects channels. If the tuple is empty (default), all components are read.

verify : bool, optional

Whether to do basic checks of frame integrity when reading. The first frame of the stream is always checked. Default: `True`.

— For writing only : (see `GSBStreamWriter`)

header0 : `GSBHeader`

Header for the first frame, holding time information, etc. Can instead give keyword arguments to construct a header.

****kwargs**

If the header is not given, an attempt will be made to construct one with any further keyword arguments. If one requires to explicitly set the mode of the GSB stream, use `header_mode`. If not given, it will be 'rawdump' if only a single raw file is present, or 'phased' otherwise. See `GSBStreamWriter`.

Returns

Filehandle

`GSBFileReader` or `GSBFileWriter` (binary), or `GSBStreamReader` or `GSBStreamWriter` (stream)

Classes

GSBFileReader(fh_raw, payload_nbytes[, …])	Simple reader for GSB data files.
GSBFileWriter(fh_raw)	Simple writer for GSB data files.
GSBStreamReader(fh_ts, fh_raw[, …])	GSB format reader.
GSBStreamWriter(fh_ts, fh_raw[, header0, …])	GSB format writer.

GSBFileReader

class baseband.gsb.base.**GSBFileReader**(*fh_raw*, *payload_nbytes*, *nchan=1*, *bps=4*, *complex_data=False*)

Bases: `baseband.vlbi_base.base.VLBIFileBase`

Simple reader for GSB data files.

Wraps a binary filehandle, providing a `read_payload` method to help interpret the data.

Parameters

payload_nbytes : int

Number of bytes to read.

nchan : int, optional

Number of channels. Default: 1.

bps : int, optional

Bits per elementary sample, i.e. per real or imaginary component for complex data. Default: 4.

complex_data : bool, optional

Whether data are complex. Default: `False`.

Methods Summary

close()	
read_payload()	Read a single block.

Methods Documentation

close()

read_payload()

Read a single block.

Returns

frame : `GSBPayload`

With a `.data` property that returns the data encoded.

GSBFileWriter

class baseband.gsb.base.**GSBFileWriter**(*fh_raw*)

Bases: `baseband.vlbi_base.base.VLBIFileBase`

Simple writer for GSB data files.

Adds `write_payload` method to the basic VLBI binary file wrapper.

Methods Summary

`close`()	
`write_payload`(data[, bps])	Write single data block.

Methods Documentation

close()

write_payload(*data*, *bps=4*)

Write single data block.

Parameters

data : `ndarray` or `GSBPayload`

If an array, bps needs to be passed in.

bps : int, optional

Bits per elementary sample, to use when encoding the payload. Ignored if `data` is a GSB payload. Default: 4.

GSBStreamReader

class baseband.gsb.base.**GSBStreamReader**(*fh_ts*, *fh_raw*, *sample_rate=None*, *samples_per_frame=None*, *payload_nbytes=None*, *nchan=None*, *bps=None*, *complex_data=None*, *squeeze=True*, *subset=()*, *verify=True*)

Bases: `baseband.gsb.base.GSBStreamBase`, `baseband.vlbi_base.base.VLBIStreamReaderBase`

GSB format reader.

Allows access to GSB files as a continuous series of samples. Requires both a timestamp and one or more corresponding raw data files.

Parameters

fh_ts : `GSBTimeStampIO`

Header filehandle.

fh_raw : filehandle, or nested tuple of filehandles

Raw binary data filehandle(s). A single file is needed for rawdump, and a tuple for phased. For a nested tuple, the outer tuple determines the number of polarizations, and the inner tuple(s) the number of streams per polarization. E.g., `((polL1, polL2), (polR1, polR2))` for two streams per polarization. A single tuple is interpreted as streams of a single polarization.

sample_rate : Quantity, optional

Number of complete samples per second, i.e. the rate at which each channel of each polarization is sampled. If None, will be inferred assuming the frame rate is exactly 0.25165824 s.

samples_per_frame : int, optional

Number of complete samples per frame. Can give payload_nbytes instead.

payload_nbytes : int, optional

Number of bytes per payload, divided by the number of raw files. If both samples_per_frame and payload_nbytes are None, payload_nbytes is set to 2**22 (4 MB) for rawdump, and 2**23 (8 MB) divided by the number of streams per polarization for phased.

nchan : int, optional

Number of channels. Default: 1 for rawdump, 512 for phased.

bps : int, optional

Bits per elementary sample, i.e. per real or imaginary component for complex data. Default: 4 for rawdump, 8 for phased.

complex_data : bool, optional

Whether data are complex. Default: False for rawdump, True for phased.

squeeze : bool, optional

If True (default), remove any dimensions of length unity from decoded data.

subset : indexing object or tuple of objects, optional

Specific components of the complete sample to decode (after possibly squeezing). If a single indexing object is passed, it selects (available) polarizations. If a tuple is passed, the first selects polarizations and the second selects channels. If the tuple is empty (default), all components are read.

verify : bool, optional

Whether to do basic checks of frame integrity when reading. The first frame of the stream is always checked. Default: True.

Attributes Summary

bps	Bits per elementary sample.
complex_data	Whether the data are complex.
dtype	
fill_value	Value to use for invalid or missing data.
header0	First header of the file.
info	
ndim	Number of dimensions of the (squeezed/subset) stream data.
sample_rate	Number of complete samples per second.
sample_shape	Shape of a complete sample (possibly subset or squeezed).

Continued on next page

Table 28 – continued from previous page

samples_per_frame	Number of complete samples per frame.
shape	Shape of the (squeezed/subset) stream data.
size	Total number of component samples in the (squeezed/subset) stream data.
squeeze	Whether data arrays have dimensions with length unity removed.
start_time	Start time of the file.
stop_time	Time at the end of the file, just after the last sample.
subset	Specific components of the complete sample to decode.
time	Time of the sample pointer's current offset in file.
verify	Whether to do consistency checks on frames being read.

Methods Summary

close()	
read([count, out])	Read a number of complete (or subset) samples.
seek(offset[, whence])	Change the stream position.
tell([unit])	Current offset in the file.

Attributes Documentation

bps
Bits per elementary sample.

complex_data
Whether the data are complex.

dtype

fill_value
Value to use for invalid or missing data. Default: 0.

header0
First header of the file.

info

ndim
Number of dimensions of the (squeezed/subset) stream data.

sample_rate
Number of complete samples per second.

sample_shape
Shape of a complete sample (possibly subset or squeezed).

samples_per_frame
Number of complete samples per frame.

shape
Shape of the (squeezed/subset) stream data.

size
Total number of component samples in the (squeezed/subset) stream data.

squeeze
Whether data arrays have dimensions with length unity removed.

If True, data read out has such dimensions removed, and data passed in for writing has them inserted.

start_time
Start time of the file.

See also time for the time of the sample pointer's current offset, and (if available) stop_time for the time at the end of the file.

stop_time
Time at the end of the file, just after the last sample.

See also start_time for the start time of the file, and time for the time of the sample pointer's current offset.

subset
Specific components of the complete sample to decode.

The order of dimensions is the same as for sample_shape. Set by the class initializer.

time
Time of the sample pointer's current offset in file.

See also start_time for the start time, and (if available) stop_time for the end time, of the file.

verify
Whether to do consistency checks on frames being read.

Methods Documentation

close()

read(*count=None*, *out=None*)
Read a number of complete (or subset) samples.

The range retrieved can span multiple frames.

Parameters
count : int or None, optional

Number of complete/subset samples to read. If None (default) or negative, the whole file is read. Ignored if out is given.

out : None or array, optional

Array to store the data in. If given, count will be inferred from the first dimension; the other dimension should equal sample_shape.

Returns
out : ndarray of float or complex

The first dimension is sample-time, and the remainder given by sample_shape.

seek(*offset*, *whence=0*)
Change the stream position.

This works like a normal filehandle seek, but the offset is in samples (or a relative or absolute time).

Parameters

offset : int, `Quantity`, or `Time`

Offset to move to. Can be an (integer) number of samples, an offset in time units, or an absolute time.

whence : {0, 1, 2, 'start', 'current', or 'end'}, optional

Like regular seek, the offset is taken to be from the start if `whence=0` (default), from the current position if 1, and from the end if 2. One can alternativey use 'start', 'current', or 'end' for 0, 1, or 2, respectively. Ignored if `offset` is a time.

tell(*unit=None*)

Current offset in the file.

Parameters

unit : `Unit` or str, optional

Time unit the offset should be returned in. By default, no unit is used, i.e., an integer enumerating samples is returned. For the special string 'time', the absolute time is calculated.

Returns

offset : int, `Quantity`, or `Time`

Offset in current file (or time at current position).

GSBStreamWriter

class baseband.gsb.base.**GSBStreamWriter**(*fh_ts*, *fh_raw*, *header0=None*, *sample_rate=None*, *samples_per_frame=None*, *payload_nbytes=None*, *nchan=None*, *bps=None*, *complex_data=None*, *squeeze=True*, ***kwargs*)

Bases: `baseband.gsb.base.GSBStreamBase`, `baseband.vlbi_base.base.VLBIStreamWriterBase`

GSB format writer.

Encodes and writes sequences of samples to file.

Parameters

fh_ts : `GSBTimeStampIO`

For writing headers to storage.

fh_raw : filehandle, or nested tuple of filehandles

For writing raw binary data to storage. A single file is needed for rawdump, and a tuple for phased. For a nested tuple, the outer tuple determines the number of polarizations, and the inner tuple(s) the number of streams per polarization. E.g., `((polL1, polL2), (polR1, polR2))` for two streams per polarization. A single tuple is interpreted as streams of a single polarization.

header0 : `GSBHeader`

Header for the first frame, holding time information, etc. Can instead give keyword arguments to construct a header (see `**kwargs`).

sample_rate : `Quantity`, optional

Number of complete samples per second, i.e. the rate at which each channel of each polarization is sampled. If not given, will be inferred assuming the frame rate is exactly 0.25165824 s.

samples_per_frame : int, optional

Number of complete samples per frame. Can give `payload_nbytes` instead.

payload_nbytes : int, optional

Number of bytes per payload, divided by the number of raw files. If both `samples_per_frame` and `payload_nbytes` are `None`, `payload_nbytes` is set to `2**22` (4 MB) for rawdump, and `2**23` (8 MB) divided by the number of streams per polarization for phased.

nchan : int, optional

Number of channels. Default: 1 for rawdump, 512 for phased.

bps : int, optional

Bits per elementary sample, i.e. per real or imaginary component for complex data. Default: 4 for rawdump, 8 for phased.

complex_data : bool, optional

Whether data are complex. Default: `False` for rawdump, `True` for phased.

squeeze : bool, optional

If `True` (default), `write` accepts squeezed arrays as input, and adds any dimensions of length unity.

****kwargs**

If no header is given, an attempt is made to construct one from these. For a standard header, this would include the following.

— Header keywords : (see `fromvalues()`)

time : `Time`

Start time of the file.

header_mode : 'rawdump' or 'phased', optional

Used to explicitly set the mode of the GSB stream. Default: 'rawdump' if only a single raw file is present, or 'phased' otherwise.

seq_nr : int, optional

Frame number, only used for phased (default: 0).

Attributes Summary

`bps`	Bits per elementary sample.
`complex_data`	Whether the data are complex.
`header0`	First header of the file.
`sample_rate`	Number of complete samples per second.
`sample_shape`	Shape of a complete sample (possibly subset or squeezed).
`samples_per_frame`	Number of complete samples per frame.
`squeeze`	Whether data arrays have dimensions with length unity removed.
`start_time`	Start time of the file.

Continued on next page

Table 30 – continued from previous page

subset	Specific components of the complete sample to decode.
time	Time of the sample pointer's current offset in file.
verify	Whether to do consistency checks on frames being read.

Methods Summary

close()	
flush()	
tell([unit])	Current offset in the file.
write(data[, valid])	Write data, buffering by frames as needed.

Attributes Documentation

bps
Bits per elementary sample.

complex_data
Whether the data are complex.

header0
First header of the file.

sample_rate
Number of complete samples per second.

sample_shape
Shape of a complete sample (possibly subset or squeezed).

samples_per_frame
Number of complete samples per frame.

squeeze
Whether data arrays have dimensions with length unity removed.

If `True`, data read out has such dimensions removed, and data passed in for writing has them inserted.

start_time
Start time of the file.

See also `time` for the time of the sample pointer's current offset, and (if available) `stop_time` for the time at the end of the file.

subset
Specific components of the complete sample to decode.

The order of dimensions is the same as for `sample_shape`. Set by the class initializer.

time
Time of the sample pointer's current offset in file.

See also `start_time` for the start time, and (if available) `stop_time` for the end time, of the file.

verify
Whether to do consistency checks on frames being read.

Methods Documentation

close()

flush()

tell(*unit=None*)
Current offset in the file.

Parameters
unit : `Unit` or str, optional

Time unit the offset should be returned in. By default, no unit is used, i.e., an integer enumerating samples is returned. For the special string 'time', the absolute time is calculated.

Returns
offset : int, `Quantity`, or `Time`

Offset in current file (or time at current position).

write(*data*, *valid=True*)
Write data, buffering by frames as needed.

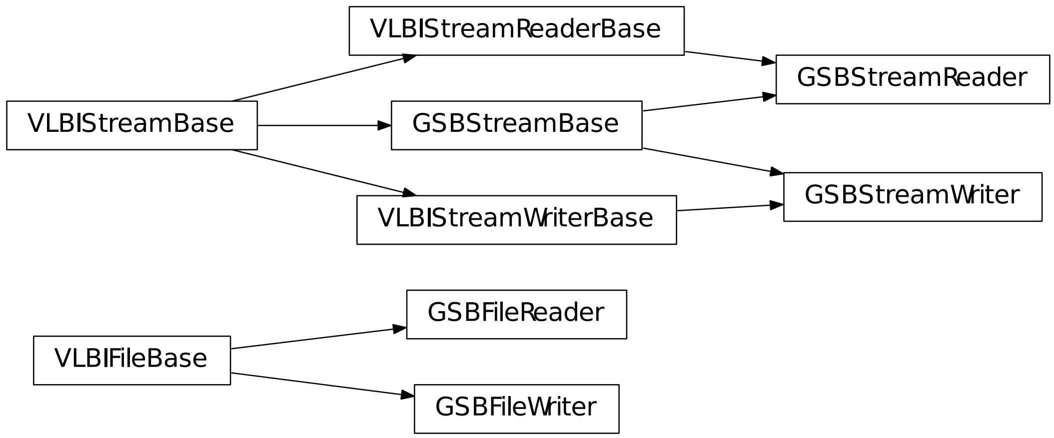
Parameters
data : `ndarray`

Piece of data to be written, with sample dimensions as given by `sample_shape`. This should be properly scaled to make best use of the dynamic range delivered by the encoding.

valid : bool, optional

Whether the current data are valid. Default: `True`.

Class Inheritance Diagram



Part III

Core framework and utilities

These sections contain APIs and usage notes for the sequential file opener, the API for the set of core utility functions and classes located in vlbi_base, and sample data that come with baseband (mostly used for testing).

CHAPTER 10

Baseband Helpers

Helpers assist with reading and writing all file formats. Currently, they only include the `sequentialfile` module for reading a sequence of files as a single one.

10.1 Sequential File

The `sequentialfile` module is for reading from and writing to a sequence of files as if they were a single, contiguous one. Like with file formats, there is a master `sequentialfile.open` function to open sequences either for reading or writing. It returns sequential file objects that have `read`, `write`, `seek`, `tell`, and `close` methods that work identically to their single file object counterparts. They additionally have `memmap` methods to read or write to files through `numpy.memmap`.

As an example of how to use `open`, we write the data from the sample VDIF file `baseband/data/sample.vdif` into a sequence of two files - as the sample file has two *framesets* - and then read the files back in. We first load the required data:

```
>>> from baseband import vdif
>>> from baseband.data import SAMPLE_VDIF
>>> import numpy as np
>>> fh = vdif.open(SAMPLE_VDIF, 'rs')
>>> d = fh.read()
```

We now open a sequential file object for writing:

```
>>> from baseband.helpers import sequentialfile as sf
>>> filenames = ["seqvdif_{0}".format(i) for i in range(2)]
>>> file_size = fh.fh_raw.seek(0, 2) // 2
>>> fwr = sf.open(filenames, mode='wb', file_size=file_size)
```

The first argument passed to `open` must be a **time-ordered sequence** of filenames in a list, tuple, or other subscriptable object that returns `IndexError` when the index is out of bounds. The read mode is 'wb', though note that writing using `numpy.memmap` (eg. required for the DADA stream writer) is only possible if `mode='w+b'`. `file_size` determines the largest size a file may reach before the next one in the sequence is opened for writing. We set `file_size` such that each file holds exactly one frameset.

Note: Setting `file_size` to a larger value than above will lead to the two files having different sizes. By default, `file_size=None`, meaning it can be arbitrarily large, in which case only one file will be created.

To write the data, we pass `fwr` to `vdif.open`:

```
>>> fw = vdif.open(fwr, 'ws', header0=fh.header0,
...                sample_rate=fh.sample_rate,
...                nthread=fh.sample_shape.nthread)
>>> fw.write(d)
>>> fw.close()    # This implicitly closes fwr.
```

To read the sequence and confirm their contents are identical to the sample file's, we may again use `open`:

```
>>> frr = sf.open(filenames, mode='rb')
>>> fr = vdif.open(frr, 'rs', sample_rate=fh.sample_rate)
>>> fr.header0.time == fh.header0.time
True
>>> np.all(fr.read() == d)
True
>>> fr.close()
```

We can also open the second file on its own and confirm it contains the second frameset of the sample file:

```
>>> fsf = vdif.open(filenames[1], mode='rs', sample_rate=fh.sample_rate)
>>> fh.seek(fh.shape[0] // 2)    # Seek to start of second frameset.
20000
>>> fsf.header0.time == fh.time
True
>>> np.all(fsf.read() == fh.read())
True
>>> fsf.close()
>>> fh.close()  # Close sample file.
```

While `sequentialfile` can be used for any format, since file sequences are common for DADA, it is implicitly used if a list of files or filename template is passed to `dada.open`. See the DADA *Usage* section for details.

10.2 Reference/API

10.2.1 baseband.helpers Package

10.2.2 baseband.helpers.sequentialfile Module

Functions

`open`(files[, mode, file_size, opener])	Read or write several files as if they were one contiguous one.

open

baseband.helpers.sequentialfile.**open**(*files*, *mode=u'rb'*, *file_size=None*, *opener=None*)
Read or write several files as if they were one contiguous one.

Parameters
files : list, tuple, or other iterable of str, filehandle

Contains the names of the underlying files that should be combined, ordered in time. If not a list or tuple, it should allow indexing with positive indices, and raise `IndexError` if these are out of range.

mode : str, optional

The mode with which the files should be opened (default: 'rb').

file_size : int, optional

For writing, the maximum size of a file, beyond which a new file should be opened. Default: `None`, which means it is unlimited and only a single file will be written.

opener : callable, optional

Function to open a single file (default: `io.open`).

Notes

The returned reader/writer will have a `memmap` method with which part of the files can be mapped to memory (like with `memmap`), as long as those parts do not span files (and the underlying files are regular ones). For writing, this requires opening in read-write mode (i.e., 'w+b').

Methods other than `read`, `write`, `seek`, `tell`, and `close` are tried on the underlying file. This implies, e.g., `readline` is possible, though the line cannot span multiple files.

The reader assumes the sequence of files is **contiguous in time**, ie. with no gaps in the data.

Classes

`SequentialFileReader`(files[, mode, opener])	Read several files as if they were one contiguous one.
`SequentialFileWriter`(files[, mode, ...])	Write several files as if they were one contiguous one.

SequentialFileReader

class baseband.helpers.sequentialfile.**SequentialFileReader**(*files*, *mode=u'rb'*, *opener=None*)
Bases: `baseband.helpers.sequentialfile.SequentialFileBase`

Read several files as if they were one contiguous one.

Parameters
files : list, tuple, or other iterable of str, filehandle

The contains the names of the underlying files that should be combined. If not a list or tuple, it should allow indexing with positive indices, and raise `IndexError` if these are out of range.

mode : str, optional

The mode with which the files should be opened (default: 'rb')

opener : callable, optional

Function to open a single file (default: `io.open`).

Attributes Summary

`file_size`	Size of the underlying file currently open for reading.
`size`	Size of all underlying files combined.

Methods Summary

`close`()	Close the currently open local file, and therewith the set.
`memmap`([dtype, mode, offset, shape, order])	Map part of the file in memory.
`read`([count])	Read and return up to n bytes.
`seek`(offset[, whence])	Change stream position.
`tell`()	Return the current stream position.

Attributes Documentation

file_size
Size of the underlying file currently open for reading.

size
Size of all underlying files combined.

Methods Documentation

close()
Close the currently open local file, and therewith the set.

memmap(*dtype=<type 'numpy.uint8'>*, *mode=None*, *offset=None*, *shape=None*, *order=u'C'*)
Map part of the file in memory.

Note that the map cannot span multiple underlying files. Parameters are as for `memmap`.

read(*count=None*)
Read and return up to n bytes.

If the argument is omitted, None, or negative, reads and returns all data until EOF.

If the argument is positive, and the underlying raw stream is not 'interactive', multiple raw reads may be issued to satisfy the byte count (unless EOF is reached first). But for interactive raw streams (as well as sockets and pipes), at most one raw read will be issued, and a short result does not imply that EOF is imminent.

Returns an empty bytes object on EOF.

Returns None if the underlying raw stream was open in non-blocking mode and no data is available at the moment.

seek(*offset*, *whence=0*)
Change stream position.

Change the stream position to the given byte offset. The offset is interpreted relative to the position indicated by whence. Values for whence are:

- 0 – start of stream (the default); offset should be zero or positive
- 1 – current stream position; offset may be negative
- 2 – end of stream; offset is usually negative

Return the new absolute position.

tell()

Return the current stream position.

SequentialFileWriter

class baseband.helpers.sequentialfile.**SequentialFileWriter**(*files*, *mode=u'w+b'*, *file_size=None*, *opener=None*)

Bases: baseband.helpers.sequentialfile.SequentialFileBase

Write several files as if they were one contiguous one.

Note that the file is not seekable and readable.

Parameters

files : list, tuple, or other iterable of str, filehandle

The contains the names of the underlying files that should be combined. If not a list or tuple, it should allow indexing with positive indices (e.g., returning a name as derived from a template). It should raise raise IndexError if the index is out of range.

mode : str, optional

The mode with which the files should be opened (default: 'w+b'). If this does not include '+' for reading, memory maps are not possibe.

file_size : int, optional

The maximum size a file is allowed to have. Default: None, which means it is unlimited and only a single file will be written (making using this class somewhat pointless).

opener : callable, optional

Function to open a single file (default: io.open).

Methods Summary

close()	Close the currently open local file, and therewith the set.
memmap([dtype, mode, offset, shape, order])	Map part of the file in memory.
tell()	Return the current stream position.
write(data)	Write the given buffer to the IO stream.

Methods Documentation

close()

Close the currently open local file, and therewith the set.

memmap(*dtype=<type 'numpy.uint8'>*, *mode=None*, *offset=None*, *shape=None*, *order=u'C'*)

Map part of the file in memory. Cannnot span file boundaries.

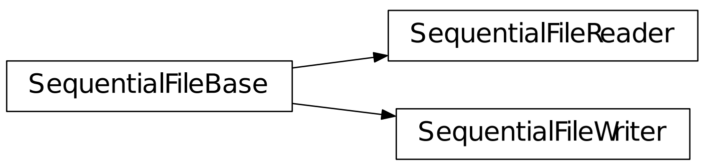
tell()
Return the current stream position.

write(*data*)
Write the given buffer to the IO stream.

Returns the number of bytes written, which is always len(b).

Raises BlockingIOError if the buffer is full and the underlying raw stream cannot accept more data at the moment.

Class Inheritance Diagram



CHAPTER 11

VLBI Base

Routines on which the readers and writers for specific VLBI formats are based.

11.1 Reference/API

11.1.1 baseband.vlbi_base Package

11.1.2 baseband.vlbi_base.header Module

Base definitions for VLBI Headers, used for VDIF and Mark 5B.

Defines a header class VLBIHeaderBase that can be used to hold the words corresponding to a frame header, providing access to the values encoded in via a dict-like interface. Definitions for headers are constructed using the HeaderParser class.

Functions

`make_parser`(word_index, bit_index, bit_length)	Construct a function that converts specific bits from a header.
`make_setter`(word_index, bit_index, bit_length)	Construct a function that uses a value to set specific bits in a header.

make_parser

`baseband.vlbi_base.header.`**`make_parser`**(*word_index*, *bit_index*, *bit_length*, *default=None*)

Construct a function that converts specific bits from a header.

The function acts on a tuple/array of 32-bit words, extracting given bits from a specific word and convert them to bool (for single bit) or integer.

The parameters are those that define header keywords, and all parsers do `(words[word_index] >> bit_index) & ((1 << bit_length) - 1)`, except that that they have been optimized for the specific cases of single bits, full words, and items starting at bit 0. As a special case, bit_length=64 allows one to extract two words as a single (long) integer.

Parameters

word_index : int

Index into the tuple of words passed to the function.

bit_index : int

Index to the starting bit of the part to be extracted.

bit_length : int

Number of bits to be extracted.

Returns

parser : function

To be used as `parser(words)`.

make_setter

`baseband.vlbi_base.header.`**`make_setter`**(*word_index*, *bit_index*, *bit_length*, *default=None*)

Construct a function that uses a value to set specific bits in a header.

The function will act on a tuple/array of words, setting given bits from a given word using a value.

The parameters are just those that define header keywords.

Parameters

word_index : int

Index into the tuple of words passed to the function.

bit_index : int

Index to the starting bit of the part to be extracted.

bit_length : int

Number of bits to be extracted.

default : int or bool or None

Possible default value to use in function if no default is passed on.

Returns

setter : function

To be used as `setter(words, value)`.

Classes

`HeaderProperty`(header_parser, getter[, doc])	Mimic a dictionary, calculating entries from header words.
`HeaderPropertyGetter`(getter[, doc])	Special property for attaching HeaderProperty.
`HeaderParser`(*args, **kwargs)	Parser & setter for VLBI header keywords.

Continued on next page

Table 2 – continued from previous page

VLBIHeaderBase(words[, verify])	Base class for all VLBI headers.

HeaderProperty

class baseband.vlbi_base.header.**HeaderProperty**(*header_parser*, *getter*, *doc=None*)

Bases: object

Mimic a dictionary, calculating entries from header words.

Used to calculate setter functions and extract default values.

Parameters

header_parser : HeaderParser

A dict with header encoding information.

getter : function

Function that uses the encoding information to calculate a result.

HeaderPropertyGetter

class baseband.vlbi_base.header.**HeaderPropertyGetter**(*getter*, *doc=None*)

Bases: object

Special property for attaching HeaderProperty.

HeaderParser

class baseband.vlbi_base.header.**HeaderParser**(**args*, ***kwargs*)

Bases: collections.OrderedDict

Parser & setter for VLBI header keywords.

An ordered dict of header keywords, with values that describe how they are encoded in a given VLBI header. Initialisation is as a normal OrderedDict, with a key, value pairs. The value should be a tuple containing:

word_index
[int] Index into the header words for this key.

bit_index
[int] Index to the starting bit of the part used for this key.

bit_length
[int] Number of bits.

default
[int or bool or None] Possible default value to use in initialisation (e.g., a sync pattern).

The class provides dict-like properties `parsers`, `setters`, and `defaults`, which return functions that get a given keyword from header words, set the corresponding part of the header words to a value, or return the default value (if defined).

Note that while in principle, parsers and setters could be calculated on the fly, we precalculate the parsers to speed up header keyword access.

Attributes Summary

defaults	Dict-like allowing access to default header values.
parsers	Dict with functions to get specific header values.
setters	Dict-like returning function to set specific header value.

Methods Summary

clear()	
copy()	Make an independent copy.
fromkeys(S[, v])	If not specified, the value defaults to None.
get(k[,d])	
has_key(k)	
items()	
iteritems()	od.iteritems -> an iterator over the (key, value) pairs in od
iterkeys()	
itervalues()	od.itervalues -> an iterator over the values in od
keys()	
pop(k[,d])	value.
popitem()	Pairs are returned in LIFO order if last is true or FIFO order if false.
setdefault(k[,d])	
update(other)	Update the parser with the information from another one.
values()	
viewitems()	
viewkeys()	
viewvalues()	

Attributes Documentation

defaults
Dict-like allowing access to default header values.

parsers
Dict with functions to get specific header values.

setters
Dict-like returning function to set specific header value.

Methods Documentation

clear() → None. Remove all items from od.

copy()
Make an independent copy.

classmethod fromkeys(*S*[, *v*]) → New ordered dictionary with keys from S.
If not specified, the value defaults to None.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

has_key(*k*) → True if D has a key k, else False

items() → list of (key, value) pairs in od

iteritems()
od.iteritems -> an iterator over the (key, value) pairs in od

iterkeys() → an iterator over the keys in od

itervalues()
od.itervalues -> an iterator over the values in od

keys() → list of keys in od

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), return and remove a (key, value) pair.
Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault(*k*[, *d*]) → od.get(k,d), also set od[k]=d if k not in od

update(*other*)
Update the parser with the information from another one.

values() → list of values in od

viewitems() → a set-like object providing a view on od's items

viewkeys() → a set-like object providing a view on od's keys

viewvalues() → an object providing a view on od's values

VLBIHeaderBase

class baseband.vlbi_base.header.**VLBIHeaderBase**(*words*, *verify=True*)
Bases: `object`

Base class for all VLBI headers.

Defines a number of common routines.

Generally, the actual class should define:

_struct: `Struct` instance that can pack/unpack header words.

_header_parser: `HeaderParser` instance corresponding to this class.

_properties: tuple of properties accessible/usable in initialisation

It also should define properties (getters *and* setters):

payload_nbytes: number of bytes used by payload

frame_nbytes: total number of bytes for header + payload

get_time, set_time, and a corresponding time property:
time at start of payload

Parameters

words : tuple or list of int, or None

header words (generally, 32 bit unsigned int). If `None`, set to a list of zeros for later initialisation. If given as a tuple, the header is immutable.

verify : bool

Whether to do basic verification of integrity. For the base class, checks that the number of words is consistent with the struct size.

Attributes Summary

`mutable`	Whether the header can be modified.
`nbytes`	Size of the header in bytes.

Methods Summary

`copy`(**kwargs)	Create a mutable and independent copy of the header.
`fromfile`(fh, *args, **kwargs)	Read VLBI Header from file.
`fromkeys`(*args, **kwargs)	Initialise a header from parsed values.
`fromvalues`(*args, **kwargs)	Initialise a header from parsed values.
`keys`()	
`tofile`(fh)	Write VLBI frame header to filehandle.
`update`(**kwargs)	Update the header by setting keywords or properties.
`verify`()	Verify that the length of the words is consistent.

Attributes Documentation

mutable
Whether the header can be modified.

nbytes
Size of the header in bytes.

Methods Documentation

copy(***kwargs*)
Create a mutable and independent copy of the header.

Keyword arguments can be passed on as needed by possible subclasses.

classmethod fromfile(*fh*, **args*, ***kwargs*)
Read VLBI Header from file.

Arguments are the same as for class initialisation. The header constructed will be immutable.

classmethod fromkeys(**args*, ***kwargs*)
Initialise a header from parsed values.

Like fromvalues, but without any interpretation of keywords.

Raises
KeyError : if not all keys required are present in `kwargs`

classmethod fromvalues(**args*, ***kwargs*)
Initialise a header from parsed values.

Here, the parsed values must be given as keyword arguments, i.e., for any `header = cls(<words>)`, `cls.fromvalues(**header) == header`.

However, unlike for the `fromkeys` class method, data can also be set using arguments named after header methods, such as `time`.

Parameters
***args**

Possible arguments required to initialize an empty header.

****kwargs**

Values used to initialize header keys or methods.

keys()

tofile(*fh*)
Write VLBI frame header to filehandle.

update(***kwargs*)
Update the header by setting keywords or properties.

Here, any keywords matching header keys are applied first, and any remaining ones are used to set header properties, in the order set by the class (in `_properties`).

Parameters
verify : bool, optional

If `True` (default), verify integrity after updating.

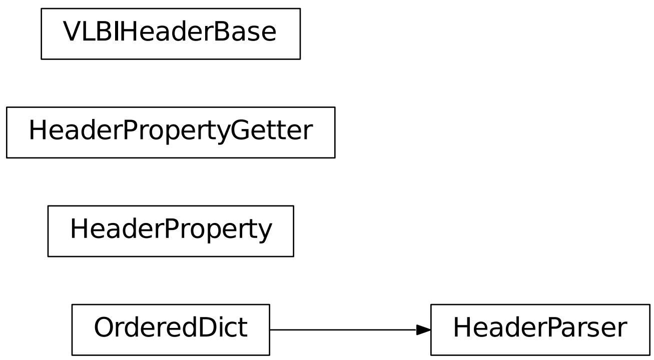
****kwargs**

Arguments used to set keywords and properties.

verify()
Verify that the length of the words is consistent.

Subclasses should override this to do more thorough checks.

Class Inheritance Diagram



11.1.3 baseband.vlbi_base.payload Module

Base definitions for VLBI payloads, used for VDIF and Mark 5B.

Defines a payload class VLBIPayloadBase that can be used to hold the words corresponding to a frame payload, providing access to the values encoded in it as a numpy array.

Classes

`VLBIPayloadBase`(words[, sample_shape, bps, . . .])	Container for decoding and encoding VLBI payloads.

VLBIPayloadBase

class `baseband.vlbi_base.payload.`**`VLBIPayloadBase`**(*words*, *sample_shape=()*, *bps=2*, *complex_data=False*)

Bases: `object`

Container for decoding and encoding VLBI payloads.

Any subclass should define dictionaries `_decoders` and `_encoders`, which hold functions that decode/encode the payload words to/from ndarray. These dictionaries are assumed to be indexed by `bps`.

Parameters

words : `ndarray`

Array containg LSB unsigned words (with the right size) that encode the payload.

sample_shape : tuple

Shape of the samples (e.g., (nchan,)). Default: ().

bps : int

Bits per elementary sample, i.e., per channel and per real or imaginary component. Default: 2.

complex_data : bool

Whether the data are complex. Default: `False`.

Attributes Summary

`data`	Full decoded payload.
`dtype`	Numeric type of the decoded data array.
`nbytes`	Size of the payload in bytes.
`ndim`	Number of dimensions of the decoded data array.
`shape`	Shape of the decoded data array.
`size`	Total number of component samples in the decoded data array.

Methods Summary

`fromdata`(data[, header, bps])	Encode data as a payload.
`fromfile`(fh, *args, **kwargs)	Read payload from filehandle and decode it into data.
`tofile`(fh)	Write payload to filehandle.

Attributes Documentation

`data`
Full decoded payload.

`dtype`
Numeric type of the decoded data array.

`nbytes`
Size of the payload in bytes.

`ndim`
Number of dimensions of the decoded data array.

`shape`
Shape of the decoded data array.

`size`
Total number of component samples in the decoded data array.

Methods Documentation

`classmethod fromdata`(*data*, *header=None*, *bps=2*)
Encode data as a payload.

Parameters

data : `ndarray`

Data to be encoded. The last dimension is taken as the number of channels.

header : header instance, optional

If given, used to infer the bps.

bps : int, optional

Bits per elementary sample, i.e., per channel and per real or imaginary component, used if header is not given. Default: 2.

classmethod fromfile(*fh*, **args*, ***kwargs*)

Read payload from filehandle and decode it into data.

Parameters

fh : filehandle

From which data is read.

payload_nbytes : int

Number of bytes to read (default: as given in `cls._nbytes`).

Any other (keyword) arguments are passed on to the class initialiser.

tofile(*fh*)

Write payload to filehandle.

Class Inheritance Diagram

11.1.4 baseband.vlbi_base.frame Module

Base definitions for VLBI frames, used for VDIF and Mark 5B.

Defines a frame class VLBIFrameBase that can be used to hold a header and a payload, providing access to the values encoded in both.

Classes

`VLBIFrameBase`(header, payload[, valid, verify])	Representation of a VLBI data frame, consisting of a header and payload.

VLBIFrameBase

class baseband.vlbi_base.frame.**VLBIFrameBase**(*header*, *payload*, *valid=True*, *verify=True*)

Bases: `object`

Representation of a VLBI data frame, consisting of a header and payload.

Parameters

header : `baseband.vlbi_base.header.VLBIHeaderBase`

Wrapper around the encoded header words, providing access to the header information.

payload : VLBIPayloadBase

Wrapper around the payload, provding mechanisms to decode it.

valid : bool

Whether the data are valid. Default: True.

verify : bool

Whether to do basic verification of integrity. Default: True.

Notes

The Frame can also be instantiated using class methods:

fromfile : read header and payload from a filehandle

fromdata : encode data as payload

Of course, one can also do the opposite:

tofile : method to write header and payload to filehandle

data : property that yields full decoded payload

One can decode part of the payload by indexing or slicing the frame. If the frame does not contain valid data, all values returned are set to `self.fill_value`.

A number of properties are defined: `shape` and `dtype` are the shape and type of the data array, and `nbytes` the frame size in bytes. Furthermore, the frame acts as a dictionary, with keys those of the header. Any attribute that is not defined on the frame itself, such as `.time` will be looked up on the header as well.

Attributes Summary

data	Full decoded frame.
dtype	Numeric type of the frame data.
fill_value	Value to replace invalid data in the frame.
nbytes	Size of the encoded frame in bytes.
ndim	Number of dimensions of the frame data.
sample_shape	Shape of a sample in the frame (nchan,).
shape	Shape of the frame data.
size	Total number of component samples in the frame data.
valid	Whether frame contains valid data.

Methods Summary

fromdata(data, header, *args, **kwargs)	Construct frame from data and header.
fromfile(fh, *args, **kwargs)	Read a frame from a filehandle.
keys()	
tofile(fh)	Write encoded frame to filehandle.
verify()	Simple verification.

Attributes Documentation

data
Full decoded frame.

dtype
Numeric type of the frame data.

fill_value
Value to replace invalid data in the frame.

nbytes
Size of the encoded frame in bytes.

ndim
Number of dimensions of the frame data.

sample_shape
Shape of a sample in the frame (nchan,).

shape
Shape of the frame data.

size
Total number of component samples in the frame data.

valid
Whether frame contains valid data.

Methods Documentation

classmethod fromdata(*data*, *header*, **args*, ***kwargs*)
Construct frame from data and header.

Parameters

data : `ndarray`

Array holding data to be encoded.

header : `VLBIHeaderBase`

Header for the frame.

***args, **kwargs :**

Any arguments beyond the filehandle are used to help initialize the payload, except for `valid` and `verify`, which are passed on to the header and class initializers.

classmethod fromfile(*fh*, **args*, ***kwargs*)
Read a frame from a filehandle.

Any arguments beyond the filehandle are used to help initialize the payload, except for `valid` and `verify`, which are passed on to the header and class initializers.

keys()

tofile(*fh*)
Write encoded frame to filehandle.

verify()
Simple verification. To be added to by subclasses.

Class Inheritance Diagram

11.1.5 baseband.vlbi_base.base Module

Functions

make_opener(fmt, classes[, doc, append_doc])	Create a baseband file opener.

make_opener

`baseband.vlbi_base.base.make_opener`(*fmt*, *classes*, *doc=u"*, *append_doc=True*)

Create a baseband file opener.

Parameters

fmt : str

Name of the baseband format.

classes : dict

With the file/stream reader/writer classes keyed by names equal to 'FileReader', 'FileWriter', 'StreamReader', 'StreamWriter' prefixed by `fmt`. Typically, one will pass in `classes=globals()`.

doc : str, optional

If given, used to define the docstring of the opener.

append_doc : bool, optional

If `True` (default), append doc to the default docstring rather than override it.

Classes

VLBIFileBase(fh_raw)	VLBI file wrapper, used to add frame methods to a binary data file.
VLBIFileReaderBase(fh_raw)	VLBI wrapped file reader base class.
VLBIStreamBase(fh_raw, header0, sample_rate, ...)	VLBI file wrapper, allowing access as a stream of data.
VLBIStreamReaderBase(fh_raw, header0, ...)	
VLBIStreamWriterBase(fh_raw, header0, ...)	

VLBIFileBase

class baseband.vlbi_base.base.**VLBIFileBase**(*fh_raw*)

Bases: object

VLBI file wrapper, used to add frame methods to a binary data file.

The underlying file is stored in fh_raw and all attributes that do not exist on the class itself are looked up on it.

Parameters

fh_raw : filehandle

Filehandle of the raw binary data file.

Methods Summary

close()	

Methods Documentation

close()

VLBIFileReaderBase

class baseband.vlbi_base.base.**VLBIFileReaderBase**(*fh_raw*)

Bases: baseband.vlbi_base.base.VLBIFileBase

VLBI wrapped file reader base class.

Typically, a subclass will define read_header, read_frame, and find_header methods. This baseclass includes a get_frame_rate method which determines the frame rate by scanning the file for headers, looking for the maximum frame number that occurs before the jump down for the next second. This method requires the subclass to define a read_header method and assumes headers have a 'frame_nr' item, and define a payload_nbytes property (as do all standard VLBI formats).

Parameters

fh_raw : filehandle

Filehandle of the raw binary data file.

Attributes Summary

info	Standardized information on file readers.

Methods Summary

close()	
get_frame_rate()	Determine the number of frames per second.

Attributes Documentation

info

Standardized information on file readers.

The `info` descriptor has a number of standard attributes, which are determined from arguments passed in opening the file, from the first header (`info.header0`) and from possibly scanning the file to determine the duration of frames.

Examples

The most common use is simply to print information:

```
>>> from baseband.data import SAMPLE_MARK5B
>>> from baseband import mark5b
>>> fh = mark5b.open(SAMPLE_MARK5B, 'rb')
>>> fh.info
File information:
format = mark5b
frame_rate = 6400.0 Hz
bps = 2
complex_data = False

missing:  nchan: needed to determine sample shape and rate.
          kday, ref_time: needed to infer full times.

>>> fh.close()

>>> fh = mark5b.open(SAMPLE_MARK5B, 'rb', kday=56000, nchan=8)
>>> fh.info
File information:
format = mark5b
frame_rate = 6400.0 Hz
sample_rate = 32.0 MHz
samples_per_frame = 5000
sample_shape = (8,)
bps = 2
complex_data = False
start_time = 2014-06-13T05:30:01.000000000
>>> fh.close()
```

Attributes

format	(str or None) File format, or None if the underlying file cannot be parsed.
frame_rate	(Quantity) Number of data frames per unit of time.
sample_rate	(Quantity) Complete samples per unit of time.
samples_per_frame	(int) Number of complete samples in each frame.
sample_shape	(tuple) Dimensions of each complete sample (e.g., `(nchan,)`).
bps	(int) Number of bits used to encode each elementary sample.
complex_data	(bool) Whether the data are complex.
start_time	(Time) Time of the first complete sample.
missing	(dict) Entries are keyed by names of arguments that should be passed to the file reader to obtain full information. The associated entries explain why these arguments are needed.

Methods Documentation

close()

get_frame_rate()

Determine the number of frames per second.

The method cycles through headers, starting from the start of the file, finding the largest frame number before it jumps back to 0 for a new second.

Returns

frame_rate : `Quantity`

Frames per second.

Raises

'EOFError'

If the file contains less than one second of data.

VLBIStreamBase

class baseband.vlbi_base.base.**VLBIStreamBase**(*fh_raw*, *header0*, *sample_rate*, *samples_per_frame*, *unsliced_shape*, *bps*, *complex_data*, *squeeze*, *subset=()*, *fill_value=0.0*, *verify=True*)

Bases: `object`

VLBI file wrapper, allowing access as a stream of data.

Attributes Summary

`bps`	Bits per elementary sample.
`complex_data`	Whether the data are complex.

Continued on next page

Table 18 – continued from previous page

header0	First header of the file.
sample_rate	Number of complete samples per second.
sample_shape	Shape of a complete sample (possibly subset or squeezed).
samples_per_frame	Number of complete samples per frame.
squeeze	Whether data arrays have dimensions with length unity removed.
start_time	Start time of the file.
subset	Specific components of the complete sample to decode.
time	Time of the sample pointer's current offset in file.
verify	Whether to do consistency checks on frames being read.

Methods Summary

close()	
tell([unit])	Current offset in the file.

Attributes Documentation

bps
Bits per elementary sample.

complex_data
Whether the data are complex.

header0
First header of the file.

sample_rate
Number of complete samples per second.

sample_shape
Shape of a complete sample (possibly subset or squeezed).

samples_per_frame
Number of complete samples per frame.

squeeze
Whether data arrays have dimensions with length unity removed.

If `True`, data read out has such dimensions removed, and data passed in for writing has them inserted.

start_time
Start time of the file.

See also `time` for the time of the sample pointer's current offset, and (if available) `stop_time` for the time at the end of the file.

subset
Specific components of the complete sample to decode.

The order of dimensions is the same as for `sample_shape`. Set by the class initializer.

time
Time of the sample pointer's current offset in file.

See also start_time for the start time, and (if available) stop_time for the end time, of the file.

verify
Whether to do consistency checks on frames being read.

Methods Documentation

close()

tell(*unit=None*)
Current offset in the file.

Parameters
unit : Unit or str, optional

Time unit the offset should be returned in. By default, no unit is used, i.e., an integer enumerating samples is returned. For the special string 'time', the absolute time is calculated.

Returns
offset : int, Quantity, or Time

Offset in current file (or time at current position).

VLBIStreamReaderBase

class baseband.vlbi_base.base.**VLBIStreamReaderBase**(*fh_raw, header0, sample_rate, samples_per_frame, unsliced_shape, bps, complex_data, squeeze, subset, fill_value, verify*)

Bases: baseband.vlbi_base.base.VLBIStreamBase

Attributes Summary

bps	Bits per elementary sample.
complex_data	Whether the data are complex.
dtype	
fill_value	Value to use for invalid or missing data.
header0	First header of the file.
info	Standardized information on stream readers.
ndim	Number of dimensions of the (squeezed/subset) stream data.
sample_rate	Number of complete samples per second.
sample_shape	Shape of a complete sample (possibly subset or squeezed).
samples_per_frame	Number of complete samples per frame.
shape	Shape of the (squeezed/subset) stream data.
size	Total number of component samples in the (squeezed/subset) stream data.
squeeze	Whether data arrays have dimensions with length unity removed.
start_time	Start time of the file.

Continued on next page

Table 20 – continued from previous page

stop_time	Time at the end of the file, just after the last sample.
subset	Specific components of the complete sample to decode.
time	Time of the sample pointer's current offset in file.
verify	Whether to do consistency checks on frames being read.

Methods Summary

close()	
read([count, out])	Read a number of complete (or subset) samples.
seek(offset[, whence])	Change the stream position.
tell([unit])	Current offset in the file.

Attributes Documentation

bps
Bits per elementary sample.

complex_data
Whether the data are complex.

dtype

fill_value
Value to use for invalid or missing data. Default: 0.

header0
First header of the file.

info
Standardized information on stream readers.

The `info` descriptor provides a few standard attributes, all of which can also be accessed directly on the stream filehandle. More detailed information on the underlying file is stored in its info, accessible via `info.file_info`.

Attributes

start_time	(Time) Time of the first complete sample.
stop_time	(Time) Time of the complete sample just beyond the end of the file.
sample_rate	(Quantity) Complete samples per unit of time.
shape	(tuple) Equivalent shape of the whole file, i.e., combining the number of complete samples and the shape of those samples.
bps	(int) Number of bits used to encode each elementary sample.
complex_data	(bool) Whether the data are complex.

ndim
Number of dimensions of the (squeezed/subset) stream data.

sample_rate
Number of complete samples per second.

sample_shape
Shape of a complete sample (possibly subset or squeezed).

samples_per_frame
Number of complete samples per frame.

shape
Shape of the (squeezed/subset) stream data.

size
Total number of component samples in the (squeezed/subset) stream data.

squeeze
Whether data arrays have dimensions with length unity removed.

If `True`, data read out has such dimensions removed, and data passed in for writing has them inserted.

start_time
Start time of the file.

See also `time` for the time of the sample pointer's current offset, and (if available) `stop_time` for the time at the end of the file.

stop_time
Time at the end of the file, just after the last sample.

See also `start_time` for the start time of the file, and `time` for the time of the sample pointer's current offset.

subset
Specific components of the complete sample to decode.

The order of dimensions is the same as for `sample_shape`. Set by the class initializer.

time
Time of the sample pointer's current offset in file.

See also `start_time` for the start time, and (if available) `stop_time` for the end time, of the file.

verify
Whether to do consistency checks on frames being read.

Methods Documentation

close()

read(*count=None*, *out=None*)
Read a number of complete (or subset) samples.

The range retrieved can span multiple frames.

Parameters
count : int or None, optional

Number of complete/subset samples to read. If `None` (default) or negative, the whole file is read. Ignored if `out` is given.

out : None or array, optional

Array to store the data in. If given, `count` will be inferred from the first dimension; the other dimension should equal `sample_shape`.

Returns

out : `ndarray` of float or complex

The first dimension is sample-time, and the remainder given by `sample_shape`.

seek(*offset*, *whence=0*)

Change the stream position.

This works like a normal filehandle seek, but the offset is in samples (or a relative or absolute time).

Parameters

offset : int, `Quantity`, or `Time`

Offset to move to. Can be an (integer) number of samples, an offset in time units, or an absolute time.

whence : {0, 1, 2, 'start', 'current', or 'end'}, optional

Like regular seek, the offset is taken to be from the start if `whence=0` (default), from the current position if 1, and from the end if 2. One can alternativey use 'start', 'current', or 'end' for 0, 1, or 2, respectively. Ignored if `offset` is a time.

tell(*unit=None*)

Current offset in the file.

Parameters

unit : `Unit` or str, optional

Time unit the offset should be returned in. By default, no unit is used, i.e., an integer enumerating samples is returned. For the special string 'time', the absolute time is calculated.

Returns

offset : int, `Quantity`, or `Time`

Offset in current file (or time at current position).

VLBIStreamWriterBase

class baseband.vlbi_base.base.**VLBIStreamWriterBase**(*fh_raw*, *header0*, *sample_rate*, *samples_per_frame*, *unsliced_shape*, *bps*, *complex_data*, *squeeze*, *subset*, *fill_value*, *verify*)

Bases: `baseband.vlbi_base.base.VLBIStreamBase`

Attributes Summary

`bps`	Bits per elementary sample.
`complex_data`	Whether the data are complex.
`header0`	First header of the file.
`sample_rate`	Number of complete samples per second.
`sample_shape`	Shape of a complete sample (possibly subset or squeezed).
`samples_per_frame`	Number of complete samples per frame.

Continued on next page

Table 22 – continued from previous page

squeeze	Whether data arrays have dimensions with length unity removed.
start_time	Start time of the file.
subset	Specific components of the complete sample to decode.
time	Time of the sample pointer's current offset in file.
verify	Whether to do consistency checks on frames being read.

Methods Summary

close()	
tell([unit])	Current offset in the file.
write(data[, valid])	Write data, buffering by frames as needed.

Attributes Documentation

bps
Bits per elementary sample.

complex_data
Whether the data are complex.

header0
First header of the file.

sample_rate
Number of complete samples per second.

sample_shape
Shape of a complete sample (possibly subset or squeezed).

samples_per_frame
Number of complete samples per frame.

squeeze
Whether data arrays have dimensions with length unity removed.

If `True`, data read out has such dimensions removed, and data passed in for writing has them inserted.

start_time
Start time of the file.

See also `time` for the time of the sample pointer's current offset, and (if available) `stop_time` for the time at the end of the file.

subset
Specific components of the complete sample to decode.

The order of dimensions is the same as for `sample_shape`. Set by the class initializer.

time
Time of the sample pointer's current offset in file.

See also `start_time` for the start time, and (if available) `stop_time` for the end time, of the file.

verify
Whether to do consistency checks on frames being read.

Methods Documentation

close()

tell(*unit=None*)

Current offset in the file.

Parameters

unit : `Unit` or str, optional

Time unit the offset should be returned in. By default, no unit is used, i.e., an integer enumerating samples is returned. For the special string 'time', the absolute time is calculated.

Returns

offset : int, `Quantity`, or `Time`

Offset in current file (or time at current position).

write(*data*, *valid=True*)

Write data, buffering by frames as needed.

Parameters

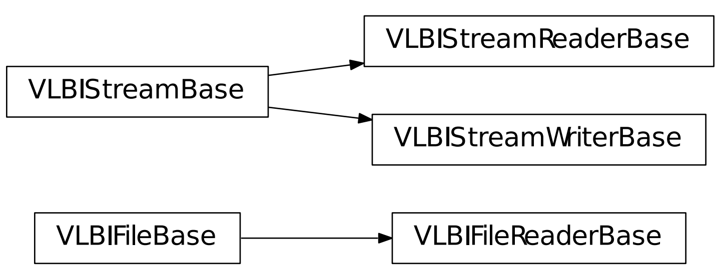
data : `ndarray`

Piece of data to be written, with sample dimensions as given by `sample_shape`. This should be properly scaled to make best use of the dynamic range delivered by the encoding.

valid : bool, optional

Whether the current data are valid. Default: `True`.

Class Inheritance Diagram



11.1.6 baseband.vlbi_base.encoding Module

Encoders and decoders for generic VLBI data formats.

Functions

encode_2bit_base(values)	Generic encoder for data stored using two bits.
encode_4bit_base(values)	Generic encoder for data stored using four bits.
decode_8bit(words)	Generic decoder for data stored using 8 bits.
encode_8bit(values)	Encode 8 bit VDIF data.

encode_2bit_base

baseband.vlbi_base.encoding.**encode_2bit_base**(*values*)

Generic encoder for data stored using two bits.

This returns an unsigned integer array containing encoded sample values that range from 0 to 3. The conversion from floating point sample value to unsigned int is given below, with `lv = TWO_BIT_1_SIGMA = 2.1745`:

Input range	Output
value < -lv	0
-lv < value < 0.	2
0. < value < lv	1
lv < value	3

This does not pack the samples into bytes.

encode_4bit_base

baseband.vlbi_base.encoding.**encode_4bit_base**(*values*)

Generic encoder for data stored using four bits.

This returns an unsigned integer array containing encoded sample values that range from 0 to 15. Floating point sample values are converted to unsigned int by first scaling them by `FOUR_BIT_1_SIGMA = 2.95`, then adding 8.5 (the 0.5 to ensure proper rounding when typecasting to uint8). Some sample output levels are:

Input range	Output
value*scale < -7.5	0
-7.5 < value*scale < -6.5	1
-0.5 < value*scale < +0.5	8
6.5 < value*scale	15

This does not pack the samples into bytes.

decode_8bit

baseband.vlbi_base.encoding.**decode_8bit**(*words*)

Generic decoder for data stored using 8 bits.

We follow mark5access, which assumes the values 0 to 255 encode -127.5 to 127.5, scaled down to match 2 bit data by a factor of 35.5 (`EIGHT_BIT_1_SIGMA`)

For comparison, GMRT phased data treats the 8-bit data values simply as signed integers.

encode_8bit

`baseband.vlbi_base.encoding.encode_8bit(values)`

Encode 8 bit VDIF data.

We follow mark5access, which assumes the values 0 to 255 encode -127.5 to 127.5, scaled down to match 2 bit data by a factor of 35.5 (EIGHT_BIT_1_SIGMA)

For comparison, GMRT phased data treats the 8-bit data values simply as signed integers.

Variables

OPTIMAL_2BIT_HIGH	Optimal high value for a 2-bit digitizer for which the low value is 1.
TWO_BIT_1_SIGMA	Optimal level between low and high for the above OPTIMAL_2BIT_HIGH.
FOUR_BIT_1_SIGMA	Scaling for four-bit encoding that makes it look like 2 bit.
EIGHT_BIT_1_SIGMA	Scaling for eight-bit encoding that makes it look like 2 bit.
decoder_levels	Levels for data encoded with different numbers of bits..

OPTIMAL_2BIT_HIGH

`baseband.vlbi_base.encoding.OPTIMAL_2BIT_HIGH = 3.316505`

Optimal high value for a 2-bit digitizer for which the low value is 1.

It is chosen such that for a normal distribution in which 68.269% of all values are at the low level, this is the mean of the others, i.e.,

$$l = \frac{\int_\sigma^\infty x \exp(-\frac{x^2}{2\sigma^2}) dx}{\int_\sigma^\infty \exp(-\frac{x^2}{2\sigma^2}) dx},$$

where the standard deviation is determined from:

$$1 = \frac{\int_0^\sigma x \exp(-\frac{x^2}{2\sigma^2}) dx}{\int_0^\sigma \exp(-\frac{x^2}{2\sigma^2}) dx}.$$

These give:

$$\sigma = \frac{\sqrt{\frac{\pi}{2}} \mathrm{erf}(\sqrt{1/2})}{1 - \sqrt{1/e}} = 2.174564,$$

and

$$l = \frac{1}{(\sqrt{e} - 1)(1/\mathrm{erf}(\sqrt{1/2}) - 1)} = 3.316505$$

TWO_BIT_1_SIGMA

`baseband.vlbi_base.encoding.TWO_BIT_1_SIGMA = 2.174564`

Optimal level between low and high for the above OPTIMAL_2BIT_HIGH.

FOUR_BIT_1_SIGMA

`baseband.vlbi_base.encoding.FOUR_BIT_1_SIGMA = 2.95`
Scaling for four-bit encoding that makes it look like 2 bit.

EIGHT_BIT_1_SIGMA

`baseband.vlbi_base.encoding.EIGHT_BIT_1_SIGMA = 35.5`
Scaling for eight-bit encoding that makes it look like 2 bit.

decoder_levels

`baseband.vlbi_base.encoding.decoder_levels = {1: array([-1., 1.], dtype=float32), 2: array([-3.316505, -1. ,`
Levels for data encoded with different numbers of bits..

11.1.7 baseband.vlbi_base.utils Module

Functions

`bcd_decode`(value)	
`bcd_encode`(value)	

bcd_decode

`baseband.vlbi_base.utils.bcd_decode`(*value*)

bcd_encode

`baseband.vlbi_base.utils.bcd_encode`(*value*)

Classes

`CRC`(polynomial)	Cyclic Redundancy Check for a bitstream.

CRC

class `baseband.vlbi_base.utils.CRC`(*polynomial*)
Bases: `object`

Cyclic Redundancy Check for a bitstream.

See https://en.wikipedia.org/wiki/Cyclic_redundancy_check

Once initialised, the instance can be used as a function that calculates the CRC, or one can use the `check` method to check that the CRC at the end of a stream is correct.

Parameters

polynomial : int

Binary encoded CRC divisor. For instance, that used by Mark 4 headers is 0x180f, or x^12 + x^11 + x^3 + x^2 + x + 1.

Methods Summary

`__call__`(stream)	Calculate CRC for the given stream.
`check`(stream)	Check that the CRC at the end of the stream is correct.

Methods Documentation

`__call__`(*stream*)

Calculate CRC for the given stream.

Parameters

stream : array of bool or unsigned int

The dimension is treated as the index into the bits. For a single stream, the array should thus be of type `bool`. Integers represent multiple streams. E.g., for a 64-track Mark 4 header, the stream would be an array of `np.uint64` words.

Returns

crc : array

The crc will have the same dtype as the input stream.

`check`(*stream*)

Check that the CRC at the end of the stream is correct.

Parameters

stream : array of bool or unsigned int

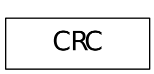
The dimension is treated as the index into the bits. For a single stream, the array should thus be of type `bool`. Integers represent multiple streams. E.g., for a 64-track Mark 4 header, the stream would be an array of `np.uint64` words.

Returns

ok : bool

`True` if the calculated CRC is all zero (which should be the case if the CRC at the end of the stream is correct).

Class Inheritance Diagram



CHAPTER 12

Sample Data Files

12.1 baseband.data Package

Sample files with baseband data recorded in different formats.

12.1.1 Variables

SAMPLE_AROCHIME_VDIF	VDIF sample from ARO, written by CHIME backend.
SAMPLE_DADA	DADA sample from Effelsberg, with header adapted to shortened size.
SAMPLE_DRAO_CORRUPT	Corrupted VDIF sample.
SAMPLE_GSB_PHASED	GSB phased sample.
SAMPLE_GSB_PHASED_HEADER	GSB phased header sample.
SAMPLE_GSB_RAWDUMP	GSB rawdump sample.
SAMPLE_GSB_RAWDUMP_HEADER	GSB rawdump header sample.
SAMPLE_MARK4	Mark 4 sample.
SAMPLE_MARK4_16TRACK	Mark 4 sample.
SAMPLE_MARK4_32TRACK	Mark 4 sample.
SAMPLE_MARK4_32TRACK_FANOUT2	Mark 4 sample.
SAMPLE_MARK5B	Mark 5B sample.
SAMPLE_MWA_VDIF	VDIF sample from MWA.
SAMPLE_PUPPI	GUPPI/PUPPI sample, npol=2, nchan=4.
SAMPLE_VDIF	VDIF sample.
SAMPLE_VLBI_VDIF	VDIF sample.

SAMPLE_AROCHIME_VDIF

`baseband.data.SAMPLE_AROCHIME_VDIF = '/home/docs/checkouts/readthedocs.org/user_builds/baseband/conda/v1.1.1/`

VDIF sample from ARO, written by CHIME backend. EDV=1, nchan=1024, bps=4.

SAMPLE_DADA

baseband.data.**SAMPLE_DADA = '/home/docs/checkouts/readthedocs.org/user_builds/baseband/conda/v1.1.1/lib/pytho**

DADA sample from Effelsberg, with header adapted to shortened size.

SAMPLE_DRAO_CORRUPT

baseband.data.**SAMPLE_DRAO_CORRUPT = '/home/docs/checkouts/readthedocs.org/user_builds/baseband/conda/v1.1.1/l**

Corrupted VDIF sample. bps=4.

First ten frames extracted from b0329 DRAO corrupted raw data file 0059000.dat.

SAMPLE_GSB_PHASED

baseband.data.**SAMPLE_GSB_PHASED = (('/home/docs/checkouts/readthedocs.org/user_builds/baseband/conda/v1.1.1/l**

GSB phased sample. samples_per_frame=8

80 complete samples, starting from seq_nr=9994, from 2013-07-27 GMRT observations of PSR J1810+1744, rewritten so each frame has 8 complete samples.

SAMPLE_GSB_PHASED_HEADER

baseband.data.**SAMPLE_GSB_PHASED_HEADER = '/home/docs/checkouts/readthedocs.org/user_builds/baseband/conda/v1.**

GSB phased header sample.

10 header entries, starting from seq_nr=9994, from 2013-07-27 GMRT observations of PSR J1810+1744.

SAMPLE_GSB_RAWDUMP

baseband.data.**SAMPLE_GSB_RAWDUMP = '/home/docs/checkouts/readthedocs.org/user_builds/baseband/conda/v1.1.1/li**

GSB rawdump sample. samples_per_frame=8192

First 81920 samples of node 5 rawdump data from 2015-04-27 GMRT observations of the Crab pulsar.

SAMPLE_GSB_RAWDUMP_HEADER

baseband.data.**SAMPLE_GSB_RAWDUMP_HEADER = '/home/docs/checkouts/readthedocs.org/user_builds/baseband/conda/v1**

GSB rawdump header sample.

First 10 header entries of node 5 rawdump data from 2015-04-27 GMRT observations of the Crab pulsar.

SAMPLE_MARK4

baseband.data.**SAMPLE_MARK4 = '/home/docs/checkouts/readthedocs.org/user_builds/baseband/conda/v1.1.1/lib/pyth**

Mark 4 sample. ntrack=64, fanout=4, bps=2.

Created from a European VLBI Network/Arecibo PSR B1957+20 observation using dd if=gp052d_ar_no0021 of=sample.m4 bs=128000 count=3

SAMPLE_MARK4_16TRACK

baseband.data.**SAMPLE_MARK4_16TRACK = '/home/docs/checkouts/readthedocs.org/user_builds/baseband/conda/v1.1.1/**

Mark 4 sample. ntrack=16, fanout=4, bps=2.

Created from the first two frames an Arecibo observation of the Crab Pulsar on 2013/11/03. (2013_306_raks02ae/ar/gs033a_ar_no0055.m5a)

SAMPLE_MARK4_32TRACK

baseband.data.**SAMPLE_MARK4_32TRACK = '/home/docs/checkouts/readthedocs.org/user_builds/baseband/conda/v1.1.1/**

Mark 4 sample. ntrack=32, fanout=4, bps=2.

Created from a Arecibo observation simultaneous with RadioAstron using dd if=rg10a_ar_no0014 of=sample_32track.m4 bs=10000 count=17

SAMPLE_MARK4_32TRACK_FANOUT2

baseband.data.**SAMPLE_MARK4_32TRACK_FANOUT2 = '/home/docs/checkouts/readthedocs.org/user_builds/baseband/conda**

Mark 4 sample. ntrack=32, fanout=2, bps=2.

Created from an Arecibo observation of PSR B1133+16 using dd if=gk049c_ar_no0011.m5a of=sample_32track_fanout2.m4 bs=10000 count=18

SAMPLE_MARK5B

baseband.data.**SAMPLE_MARK5B = '/home/docs/checkouts/readthedocs.org/user_builds/baseband/conda/v1.1.1/lib/pyt**

Mark 5B sample. nchan=8, bps=2.

Created from a EVN/WSRT PSR B1957+20 observation.

SAMPLE_MWA_VDIF

baseband.data.**SAMPLE_MWA_VDIF = '/home/docs/checkouts/readthedocs.org/user_builds/baseband/conda/v1.1.1/lib/p**

VDIF sample from MWA. EDV=0, two threads, bps=8

SAMPLE_PUPPI

baseband.data.**SAMPLE_PUPPI = '/home/docs/checkouts/readthedocs.org/user_builds/baseband/conda/v1.1.1/lib/pyth**

GUPPI/PUPPI sample, npol=2, nchan=4.

Created from the first four frames of a 2018-01-14 Arecibo observation of J1810+1744, with payload shortened to 8192 complete samples (with 512 overlap).

SAMPLE_VDIF

baseband.data.**SAMPLE_VDIF = '/home/docs/checkouts/readthedocs.org/user_builds/baseband/conda/v1.1.1/lib/pytho**

VDIF sample. 8 threads, bps=2.

Created from a EVN/VLBA PSR B1957+20 observation. Timestamps of frames with even thread IDs have been corrected to be consistent with odd-ID frames.

SAMPLE_VLBI_VDIF

baseband.data.**SAMPLE_VLBI_VDIF = '/home/docs/checkouts/readthedocs.org/user_builds/baseband/conda/v1.1.1/lib/**

VDIF sample. 8 threads, bps=2.

Created from a EVN/VLBA PSR B1957+20 observation. Uncorrected version of SAMPLE_VDIF.

Part IV

Developer documentation

The developer documentation feature tutorials for supporting new formats or format extensions such as VDIF EDV.

CHAPTER 13

Supporting a New VDIF EDV

Users may encounter VDIF files with unusual headers not currently supported by Baseband. These may either have novel EDV, or they may purport to be a supported EDV but not conform to its formal specification. To handle such situations, Baseband supports implementation of new EDVs and overriding of existing EDVs without the need to modify Baseband's source code.

The tutorials below assumes the following modules have been imported:

```
>>> import numpy as np
>>> import astropy.units as u
>>> from baseband import vdif, vlbi_base as vlbi
```

13.1 VDIF Headers

Each VDIF frame begins with a 32-byte, or eight 32-bit **word**, header that is structured as follows:

where the abbreviated labels are

- I_1 - invalid data
- L_1 - if 1, header is VDIF legacy
- V_3 - VDIF version number
- $\log_2(\#\mathrm{chns})_5$ - $\log_2$ of the number of sub-bands in the frame
- C_1 - if 1, complex data
- EDV_8 - "extended data version" number; see below

Detailed definitions of terms are found on pages 5 to 7 of the VDIF specification document.

Words 4 - 7 hold optional extended user data, using a layout specified by the EDV, in word 4 of the header. EDV formats can be registered on the VDIF website; Baseband aims to support all registered formats (but does not currently support EDV = 4).

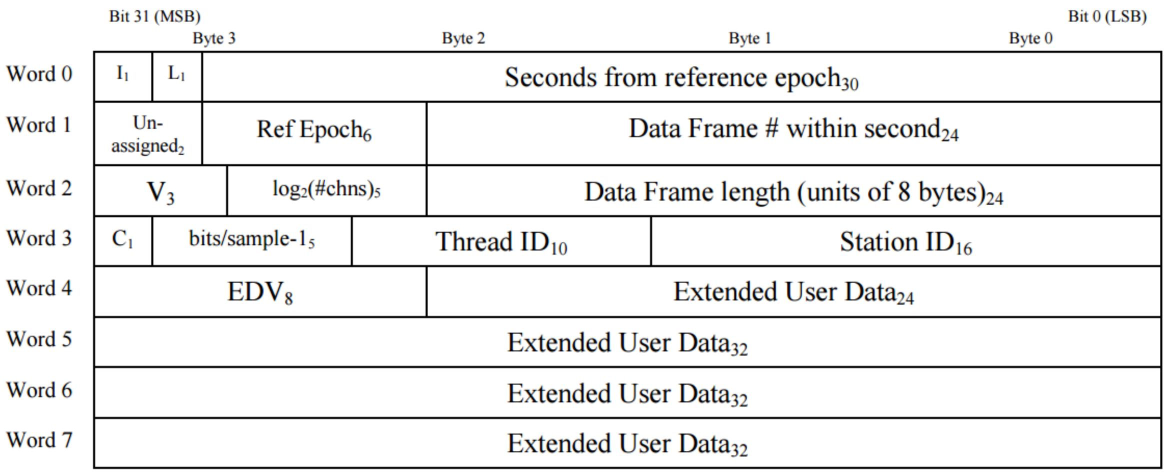


Fig. 1: Schematic of the standard 32-bit VDIF header, from VDIF specification release 1.1.1 document, Fig. 3. 32-bit words are labelled on the left, while byte and bit numbers above indicate relative addresses within each word. Subscripts indicate field length in bits.

13.2 Implementing a New EDV

In this tutorial, we follow the implementation of an EDV=4 header. This would be a first and required step to support that format, but does not suffice, as it also needs a new frame class that allows the purpose of the EDV class, which is to independently store the validity of sub-band channels within a single data frame, rather than using the single invalid-data bit. From the EDV=4 specification, we see that we need to add the following to the standard VDIF header:

- Validity header mask (word 4, bits 16 - 24): integer value between 1 and 64 inclusive indicating the number of validity bits. (This is different than $\log_2(\#\mathrm{chns})_5$, since some channels can be unused.)
- Synchronization pattern (word 5): constant byte sequence `0xACABFEED`, for finding the locations of headers in a data stream.
- Validity mask (words 6 - 7): 64-bit binary mask indicating the validity of sub-bands. Any fraction of 64 sub-bands can be stored in this format, with any unused bands labelled as invalid (`0`) in the mask. If the number of bands exceeds 64, each bit indicates the validity of a group of sub-bands; see specification for details.

See Sec. 3.1 of the specification for best practices on using the invalid data bit I_1 in word 0.

In Baseband, a header is parsed using `VDIFHeader`, which returns a header instance that is a subclass of `VDIFHeader` corresponding to the header EDV. This can be seen in the `header` module class inheritance diagram. To support a new EDV, we create a new subclass to `VDIFHeader`:

```
>>> class VDIFHeader4(vdif.header.VDIFHeader):
...     _edv = 4
...
...     _header_parser = vlbi.header.HeaderParser(
...         (('invalid_data', (0, 31, 1, False)),
...          ('legacy_mode', (0, 30, 1, False)),
...          ('seconds', (0, 0, 30)),
...          ('_1_30_2', (1, 30, 2, 0x0)),
...          ('ref_epoch', (1, 24, 6)),
...          ('frame_nr', (1, 0, 24, 0x0)),
...          ('vdif_version', (2, 29, 3, 0x1)),
```

(continues on next page)

(continued from previous page)

```
...         ('lg2_nchan', (2, 24, 5)),
...         ('frame_length', (2, 0, 24)),
...         ('complex_data', (3, 31, 1)),
...         ('bits_per_sample', (3, 26, 5)),
...         ('thread_id', (3, 16, 10, 0x0)),
...         ('station_id', (3, 0, 16)),
...         ('edv', (4, 24, 8)),
...         ('validity_mask_length', (4, 16, 8, 0)),
...         ('sync_pattern', (5, 0, 32, 0xACABFEED)),
...         ('validity_mask', (6, 0, 64, 0))))
```

VDIFHeader has a metaclass that ensures that whenever it is subclassed, the subclass definition is inserted into the VDIF_HEADER_CLASSES dictionary using its EDV value as the dictionary key. Methods in VDIFHeader use this dictionary to determine the type of object to return for a particular EDV. How all this works is further discussed in the documentation of the VDIF header module.

The class must have a private _edv attribute for it to properly be registered in VDIF_HEADER_CLASSES. It must also feature a _header_parser that reads these words to return header properties. For this, we utilize vlbi_base.header.HeaderParser, available in baseband.vlbi_base.header. To initialize a header parser, we pass it a tuple of header properties, where each entry follows the syntax:

```
('property_name', (word_index, bit_index, bit_length, default))
```

where

- property_name: name of the header property; this will be the key;
- word_index: index into the header words for this key;
- bit_index: index to the starting bit of the part used;
- bit_length: number of bits used, normally between 1 and 32, but can be 64 for adding two words together; and
- default: (optional) default value to use in initialization.

For further details, see the documentation of HeaderParser.

Once defined, we can use our new header like any other:

```
>>> myheader = vdif.header.VDIFHeader.fromvalues(
...     edv=4, seconds=14363767, nchan=1,
...     station=65532, bps=2, complex_data=False,
...     thread_id=3, validity_mask_length=60,
...     validity_mask=(1 << 59) + 1)
>>> myheader
<VDIFHeader4 invalid_data: False,
             legacy_mode: False,
             seconds: 14363767,
             _1_30_2: 0,
             ref_epoch: 0,
             frame_nr: 0,
             vdif_version: 1,
             lg2_nchan: 0,
             frame_length: 0,
             complex_data: False,
             bits_per_sample: 1,
             thread_id: 3,
             station_id: 65532,
             edv: 4,
```

(continues on next page)

(continued from previous page)

```
            validity_mask_length: 60,
            sync_pattern: 0xacabfeed,
            validity_mask: 576460752303423489>
>>> myheader['validity_mask'] == 2**59 + 1
True
```

There is an easier means of instantiating the header parser. As can be seen in the class inheritance diagram for the header module, many VDIF headers are subclassed from other VDIFHeader subclasses, namely VDIFBaseHeader and VDIFSampleRateHeader. This is because many EDV specifications share common header values, and so their functions and derived properties should be shared as well. Moreover, header parsers can be appended to one another, which saves repetitious coding because the first four words of any VDIF header are the same. Indeed, we can create the same header as above by subclassing VDIFBaseHeader:

```
>>> class VDIFHeader4Enhanced(vdif.header.VDIFBaseHeader):
...     _edv = 42
...
...     _header_parser = vdif.header.VDIFBaseHeader._header_parser +\
...                      vlbi.header.HeaderParser((
...                             ('validity_mask_length', (4, 16, 8, 0)),
...                             ('sync_pattern', (5, 0, 32, 0xACABFEED)),
...                             ('validity_mask', (6, 0, 64, 0))))
...
...     _properties = vdif.header.VDIFBaseHeader._properties + ('validity',)
...
...     def verify(self):
...         """Basic checks of header integrity."""
...         super(VDIFHeader4Enhanced, self).verify()
...         assert 1 <= self['validity_mask_length'] <= 64
...
...     @property
...     def validity(self):
...         """Validity mask array with proper length.
...
...         If set, writes both ``validity_mask`` and ``validity_mask_length``.
...         """
...         bitmask = np.unpackbits(self['validity_mask'].astype('>u8')
...                                 .view('u1'))[::-1].astype(bool)
...         return bitmask[:self['validity_mask_length']]
...
...     @validity.setter
...     def validity(self, validity):
...         bitmask = np.zeros(64, dtype=bool)
...         bitmask[:len(validity)] = validity
...         self['validity_mask_length'] = len(validity)
...         self['validity_mask'] = np.packbits(bitmask[::-1]).view('>u8')
```

Here, we set edv = 42 because VDIFHeader's metaclass is designed to prevent accidental overwriting of existing entries in VDIF_HEADER_CLASSES. If we had used _edv = 4, we would have gotten an exception:

```
ValueError: EDV 4 already registered in VDIF_HEADER_CLASSES
```

We shall see how to override header classes in the next section. Except for the EDV, VDIFHeader4Enhanced's header structure is identical to VDIFHeader4. It also contains a few extra functions to enhance the header's usability.

The verify function is an optional function that runs upon header initialization to check its veracity. Ours simply checks that the validity mask length is in the allowed range, but we also call the same function in the superclass (VDIFBaseHeader), which checks that the header is not in 4-word "legacy mode", that the header's EDV matches that read from the words, that there are eight words, and that the sync pattern matches 0xACABFEED.

The validity_mask is a bit mask, which is not necessarily the easiest to use directly. Hence, implement a derived validity property that generates a boolean mask of the right length (note that this is not right for cases whether the number of channels in the header exceeds 64). We also define a corresponding setter, and add this to the private _properties attribute, so that we can use validity as a keyword in fromvalues:

```
>>> myenhancedheader = vdif.header.VDIFHeader.fromvalues(
...     edv=42, seconds=14363767, nchan=1,
...     station=65532, bps=2, complex_data=False,
...     thread_id=3, validity=[True]+[False]*58+[True])
>>> myenhancedheader
<VDIFHeader4Enhanced invalid_data: False,
                     legacy_mode: False,
                     seconds: 14363767,
                     _1_30_2: 0,
                     ref_epoch: 0,
                     frame_nr: 0,
                     vdif_version: 1,
                     lg2_nchan: 0,
                     frame_length: 0,
                     complex_data: False,
                     bits_per_sample: 1,
                     thread_id: 3,
                     station_id: 65532,
                     edv: 42,
                     validity_mask_length: 60,
                     sync_pattern: 0xacabfeed,
                     validity_mask: [576460752303423489]>
>>> assert myenhancedheader['validity_mask'] == 2**59 + 1
>>> assert (myenhancedheader.validity == [True]+[False]*58+[True]).all()
>>> myenhancedheader.validity = [True]*8
>>> myenhancedheader['validity_mask']
array([255], dtype=uint64)
```

Note: If you have implemented support for a new EDV that is widely used, we encourage you to make a pull request to Baseband's GitHub repository, as well as to register it (if it is not already registered) with the VDIF consortium!

13.3 Replacing an Existing EDV

Above, we mentioned that VDIFHeader's metaclass is designed to prevent accidental overwriting of existing entries in VDIF_HEADER_CLASSES, so attempting to assign two header classes to the same EDV results in an exception. There are situations such the one above, however, where we'd like to replace one header with another.

To get VDIFHeader to use VDIFHeader4Enhanced when edv=4, we can manually insert it in the dictionary:

```
>>> vdif.header.VDIF_HEADER_CLASSES[4] = VDIFHeader4Enhanced
```

Of course, we should then be sure that its _edv attribute is correct:

```
>>> VDIFHeader4Enhanced._edv = 4
```

VDIFHeader will now return instances of VDIFHeader4Enhanced when reading headers with edv = 4:

```
>>> myheader = vdif.header.VDIFHeader.fromvalues(
...     edv=4, seconds=14363767, nchan=1,
...     station=65532, bps=2, complex_data=False,
...     thread_id=3, validity=[True]*60)
>>> assert isinstance(myheader, VDIFHeader4Enhanced)
```

Note: Failing to modify `_edv` in the class definition will lead to an EDV mismatch when `verify` is called during header initialization.

This can also be used to override `VDIFHeader`'s behavior *even for EDVs that are supported by Baseband*, which may prove useful when reading data with corrupted or mislabelled headers. To illustrate this, we attempt to read in a corrupted VDIF file originally from the Dominion Radio Astrophysical Observatory. This file can be imported from the baseband data directory:

```
>>> from baseband.data import SAMPLE_DRAO_CORRUPT
```

Naively opening the file with

```
>>> fh = vdif.open(SAMPLE_DRAO_CORRUPT, 'rs')
```

will lead to an AssertionError. This is because while the headers of the file use EDV=0, it deviates from that EDV standard by storing additional information an: an "eud2" parameter in word 5, which is related to the sample time. Furthermore, the `bits_per_sample` setting is incorrect (it should be 3 rather than 4 – the number is defined such that a one-bit sample has a `bits_per_sample` code of 0). Finally, though not an error, the `thread_id` in word 3 defines two parts, `link` and `slot`, which reflect the data acquisition computer node that wrote the data to disk.

To accommodate these changes, we design an alternate header. We first pop the EDV = 0 entry from `VDIF_HEADER_CLASSES`:

```
>>> vdif.header.VDIF_HEADER_CLASSES.pop(0)
<class 'baseband.vdif.header.VDIFHeader0'>
```

We then define a replacement class:

```
>>> class DRAOVDIFHeader(vdif.header.VDIFHeader0):
...     """DRAO VDIF Header
...
...     An extension of EDV=0 which uses the thread_id to store link
...     and slot numbers, and adds a user keyword (illegal in EDV0,
...     but whatever) that identifies data taken at the same time.
...
...     The header also corrects 'bits_per_sample' to be properly bps-1.
...     """
...
...     _header_parser = vdif.header.VDIFHeader0._header_parser + \
...         vlbi.header.HeaderParser((('link', (3, 16, 4)),
...                                   ('slot', (3, 20, 6)),
...                                   ('eud2', (5, 0, 32))))
...
...     def verify(self):
...         pass  # this is a hack, don't bother with verification...
...
...     @classmethod
...     def fromfile(cls, fh, edv=0, verify=False):
...         self = super(DRAOVDIFHeader, cls).fromfile(fh, edv=0,
```

(continues on next page)

(continued from previous page)

```
...                                                             verify=False)
...         # Correct wrong bps
...         self.mutable = True
...         self['bits_per_sample'] = 3
...         return self
```

We override `verify` because `VDIFHeader0`'s `verify` function checks that word 5 contains no data. We also override the `fromfile` class method such that the `bits_per_sample` property is reset to its proper value whenever a header is read from file.

We can now read in the corrupt file by manually reading in the header, then the payload, of each frame:

```
>>> fh = vdif.open(SAMPLE_DRAO_CORRUPT, 'rb')
>>> header0 = DRAOVDIFHeader.fromfile(fh)
>>> header0['eud2'] == 667235140
True
>>> header0['link'] == 2
True
>>> payload0 = vdif.payload.VDIFPayload.fromfile(fh, header0)
>>> payload0.shape == (header0.samples_per_frame, header0.nchan)
True
>>> fh.close()
```

Reading a frame using `VDIFFrame` will still fail, since its `_header_class` is `VDIFHeader`, and so `VDIFHeader.fromfile`, rather than the function we defined, is used to read in headers. If we wanted to use `VDIFFrame`, we would need to set

```
VDIFFrame._header_class = DRAOVDIFHeader
```

before using `open()`, so that header files are read using `DRAOVDIFHeader.fromfile`.

A more elegant solution that is compatible with `VDIFStreamReader` without hacking `VDIFFrame` involves modifying the bits-per-sample code within `__init__()`. Let's remove our previous custom class, and define a replacement:

```
>>> vdif.header.VDIF_HEADER_CLASSES.pop(0)
<class '__main__.DRAOVDIFHeader'>
>>> class DRAOVDIFHeaderEnhanced(vdif.header.VDIFHeader0):
...     """DRAO VDIF Header
...
...     An extension of EDV=0 which uses the thread_id to store link and slot
...     numbers, and adds a user keyword (illegal in EDV0, but whatever) that
...     identifies data taken at the same time.
...
...     The header also corrects 'bits_per_sample' to be properly bps-1.
...     """
...     _header_parser = vdif.header.VDIFHeader0._header_parser + \
...         vlbi.header.HeaderParser((('link', (3, 16, 4)),
...                                   ('slot', (3, 20, 6)),
...                                   ('eud2', (5, 0, 32))))
...
...     def __init__(self, words, edv=None, verify=True, **kwargs):
...         super(DRAOVDIFHeaderEnhanced, self).__init__(
...                 words, verify=False, **kwargs)
...         self.mutable = True
...         self['bits_per_sample'] = 3
...
...     def verify(self):
...         pass
```

We can then use the stream reader without further modification:

```
>>> fh2 = vdif.open(SAMPLE_DRAO_CORRUPT, 'rs', sample_rate=5**12*u.Hz)
>>> fh2.header0['eud2'] == header0['eud2']
True
>>> np.all(fh2.read(1) == payload0[0])
True
>>> fh2.close()
```

Reading frames using `VDIFFileReader.read_frame` will now work as well, but reading frame sets using `VDIFFileReader.read_frameset` will still fail. This is because the frame and thread numbers that function relies on are meaningless for these headers, and grouping threads together using the `link`, `slot` and `eud2` values should be manually performed by the user.

Part V

Project details

CHAPTER 14

Contributors

CHAPTER 15

Authors and Credits

15.1 Baseband Project Contributors

15.1.1 Authors

- Marten van Kerkwijk (@mhvk)
- Chenchong Charles Zhu (@cczhu)

15.1.2 Alphabetical list of contributors

- Rebecca Lin (@00rebe)
- Nikhil Mahajan (@theXYZT)
- Robert Main (@ramain)
- Dana Simard (@danasimard)
- George Stein (@georgestein)

If you have contributed to Baseband but are not listed above, please send one of the authors an e-mail, or open a pull request for this page.

CHAPTER 16

Full Changelog

16.1 1.1.1 (2018-07-24)

16.1.1 Bug Fixes

- Ensure `gsb` times can be decoded with astropy-dev (which is to become astropy 3.1). [#249]
- Fixed rounding error when encoding 4-bit data using `baseband.vlbi_base.encoding.encode_4bit_base`. [#250]
- Added GUPPI/PUPPI to the list of file formats used by `baseband.open` and `baseband.file_info`. [#251]

16.2 1.1 (2018-06-06)

16.2.1 New Features

- Added a new `baseband.file_info` function, which can be used to inspect data files. [#200]
- Added a general file opener, `baseband.open` which for a set of formats will check whether the file is of that format, and then load it using the corresponding module. [#198]
- Allow users to pass a `verify` keyword to file openers reading streams. [#233]
- Added support for the GUPPI format. [#212]
- Enabled `baseband.dada.open` to read streams where the last frame has an incomplete payload. [#228]

16.2.2 API Changes

- In analogy with Mark 5B, VDIF header time getting and setting now requires a frame rate rather than a sample rate. [#217, #218]

- DADA and GUPPI now support passing either a `start_time` or `offset` (in addition to `time`) to set the start time in the header. [#240]

16.2.3 Bug Fixes

16.2.4 Other Changes and Additions

- The `baseband.data` module with sample data files now has an explicit entry in the documentation. [#198]
- Increased speed of VLBI stream reading by changing the way header sync patterns are stored, and removing redundant verification steps. VDIF sequential decode is now 5 - 10% faster (depending on the number of threads). [#241]

16.3 1.0.1 (2018-06-04)

16.3.1 Bug Fixes

- Fixed a bug in `baseband.dada.open` where passing a `squeeze` setting is ignored when also passing header keywords in 'ws' mode. [#211]
- Raise an exception rather than return incorrect times for Mark 5B files in which the fractional seconds are not set. [#216]

16.3.2 Other Changes and Additions

- Fixed broken links and typos in the documentation. [#211]

16.4 1.0.0 (2018-04-09)

- Initial release.

CHAPTER 17

Licenses

17.1 Baseband License

Baseband is licensed under the GNU General Public License v3.0. The full text of the license can be found in LICENSE under Baseband's root directory.

Part VI

Reference/API

CHAPTER 18

baseband Package

Radio baseband I/O.

18.1 Functions

`file_info`(name[, format])	Get format and other information from a baseband file.
`open`(name[, mode, format])	Open a baseband file for reading or writing.
`test`([package, test_path, args, plugins, . . .])	Run the tests using py.test.

18.1.1 file_info

`baseband.`**`file_info`**(*name*, *format=('dada', 'mark4', 'mark5b', 'vdif', 'guppi', 'gsb')*, ***kwargs*)

Get format and other information from a baseband file.

The keyword arguments will only be used if needed, so if one is unsure what format a file is, but knows it was taken recently and has 8 channels, one would put in `ref_time=Time('2015-01-01'), nchan=8`. Alternatively, and perhaps easier, one can first call the function without extra arguments in which case the result will describe what is missing.

Parameters

name : str or filehandle

Raw file for which to obtain information.

format : str, tuple of str, optional

Formats to try. If not given, try all standard formats.

****kwargs**

Any arguments that might help to get information. For instance, Mark 4 and Mark 5B do not have complete timestamps, which can be addressed by passing in `ref_time`. Furthermore, for Mark 5B, it is needed to pass in `nchan`. Arguments are checked for consistency with the file even if not used (see notes below).

Returns

info : `VLBIFileReaderInfo` or `VLBIStreamReaderInfo`

The information on the file. Can be turned info a `dict` by calling it (i.e., `info()`).

Notes

All keyword arguments passed in are classified, ending up in one of the following (mostly useful if the file could be opened as a stream):

- `used_kwargs`: arguments that were needed to open the file.
- `consistent_kwargs`: not needed to open the file, but consistent.
- `inconsistent_kwargs`: not needed to open the file, and inconsistent.
- `irrelevant_kwargs`: provide information irrelevant for opening.

18.1.2 open

`baseband.open`(*name*, *mode='rs'*, *format=('dada', 'mark4', 'mark5b', 'vdif', 'guppi', 'gsb')*, ***kwargs*)

Open a baseband file for reading or writing.

Opened as a binary file, one gets a wrapped filehandle that adds methods to read/write a frame. Opened as a stream, the handle is wrapped further, and reading and writing to the file is done as if the file were a stream of samples.

Parameters

name : str or filehandle

File name or handle.

mode : {'rb', 'wb', 'rs', or 'ws'}, optional

Whether to open for reading or writing, and as a regular binary file or as a stream. Default: 'rs', for reading a stream.

format : str or tuple of str

The format the file is in. For reading, this can be a tuple of possible formats, all of which will be tried in turn. By default, all supported formats are tried.

****kwargs**

Additional arguments needed for opening the file as a stream. For most formats, trying without these will raise an exception that tells which arguments are needed. Opening will not succeed if any arguments are passed in that are inconsistent with the file, or are irrelevant for opening the file.

18.1.3 test

`baseband.test`(*package=None*, *test_path=None*, *args=None*, *plugins=None*, *verbose=False*, *pastebin=None*, *remote_data=False*, *pep8=False*, *pdb=False*, *coverage=False*, *open_files=False*, ***kwargs*)

Run the tests using py.test. A proper set of arguments is constructed and passed to pytest.main.

Parameters

package : str, optional

The name of a specific package to test, e.g. 'io.fits' or 'utils'. If nothing is specified all default tests are run.

test_path : str, optional

Specify location to test by path. May be a single file or directory. Must be specified absolutely or relative to the calling directory.

args : str, optional

Additional arguments to be passed to pytest.main in the `args` keyword argument.

plugins : list, optional

Plugins to be passed to pytest.main in the `plugins` keyword argument.

verbose : bool, optional

Convenience option to turn on verbose output from py.test. Passing True is the same as specifying `'-v'` in `args`.

pastebin : {'failed','all',None}, optional

Convenience option for turning on py.test pastebin output. Set to `'failed'` to upload info for failed tests, or `'all'` to upload info for all tests.

remote_data : bool, optional

Controls whether to run tests marked with @remote_data. These tests use online data and are not run by default. Set to True to run these tests.

pep8 : bool, optional

Turn on PEP8 checking via the pytest-pep8 plugin and disable normal tests. Same as specifying `'--pep8 -k pep8'` in `args`.

pdb : bool, optional

Turn on PDB post-mortem analysis for failing tests. Same as specifying `'--pdb'` in `args`.

coverage : bool, optional

Generate a test coverage report. The result will be placed in the directory htmlcov.

open_files : bool, optional

Fail when any tests leave files open. Off by default, because this adds extra run time to the test suite. Requires the psutil package.

parallel : int, optional

When provided, run the tests in parallel on the specified number of CPUs. If parallel is negative, it will use the all the cores on the machine. Requires the pytest-xdist plugin installed. Only available when using Astropy 0.3 or later.

kwargs

Any additional keywords passed into this function will be passed on to the astropy test runner. This allows use of test-related functionality implemented in later versions of astropy without explicitly updating the package template.

Python Module Index

b

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W