dlisio Documentation

Release 0.3.6

Equinor

Jun 03, 2022

CONTENTS

Welcome to dlisio. dlisio is a python package for reading Digital Log Interchange Standard (DLIS) v1. Version 2 exists, and has been around for quite a while, but it is our understanding that most dlis files out there are still version 1. Hence dlisio's focus is put on version 1[1], for now.

As of version 0.3.0, dlisio is extended to also read Log Information Standard 79 (LIS79)[2]. An extended version of the LIS79 standard called LIS84/Enhanced LIS exists, but this version is currently not supported by dlisio.

Before you get started we recommended that you familiarize yourself with some basic concepts of the DLIS- and LIS file formats. These are non-trivial formats and some knowledge about them is required for effective work. A good place to start is the user guides: *DLIS User Guide* and *LIS User Guide*.

Warning: DLIS and LIS files are often "wrapped" in "container"-formats. Essentially this is just extra information needed for the file to be correctly read by a tape-reader and does not add anything to the well-logs themselves. Maybe the most common one is the TapeImageFormat (TIF). TIF is automatically detected by dlisio and both DLIS and LIS files wrapped in TIF can be read without any special care from the user. There are other container formats or TIF-modifications with unknown origins that dlisio does not support. **DLIS and LIS files using these formats or modifications will fail to read.**

dlisio can be installed with pip

```
$ python3 -m pip install dlisio
```

Alternatively, you can grab the latest source code from GitHub.

[1] API RP66 v1, http://w3.energistics.org/RP66/V1/Toc/main.html
[2] LIS79, http://w3.energistics.org/LIS/lis-79.pdf

CHAPTER

ONE

ABOUT THE PROJECT

dlisio attempts to abstract away a lot of the pain of LIS and DLIS and give access to the data in a simple and easy-to-use manner. It gives the user the ability to work with these files without having to know *all* the details of the standard itself. Its main focus is making the data accessible while putting little assumptions on how the data is to be used.

dlisio is written and maintained by Equinor ASA as a free, simple, easy-to-use library to read well logs that can be tailored to our needs, and as a contribution to the open-source community.

dlisio is divided into 3 subpackages:

- `dlisio.dlis`, *Digital Log Interchange Standard*
- `dlisio.lis` , *Log Interchange Standard*
- `dlisio.common`, *Common API Reference*

CHAPTER TWO

TABLE OF CONTENTS

2.1 Changelog

All notable changes to this project will be documented in this file, for a complete overview of changes, please refer to the git log.

The format is based on Keep a Changelog, but most notably, without sectioning changes into type-of-change.

2.1.1 0.3.6 - 2022.03.18

- Prebuilt wheels for python 3.10
- Prebuilt wheels for linux aarch64

2.1.2 0.3.5 - 2021.05.31

- Added support for UTF-8 filepaths on Windows
- dlisio has been given its own logger. `#df40d7e`
- Fixes a bug that caused dlisio to fail when opening files >2GB on Windows
- DLIS metadata is now cached by default `#810d077`
- DLIS attribute units are better documented
- Removed deprecated `dlis.LogicalFile.__getitem__`
- Removed deprecated `dlis.LogicalFile.match`
- Added `curves_metadata` to the lis interface
- Added `index_mnem` to `DataFormatSpec`
- Added `index_units` to `DataFormatSpec`
- Added `sample_rates` to `DataFormatSpec`
- LIS Component Blocks are validated when parsed
- Added a class-like interface for LIS Spec Block 1 Process Indicators

2.1.3 0.3.4 - 2021.05.07

- Full support for LIS Spec Block sub-type 0 and 1.
- dlisio now uses the python type 'bytes' to represent the LIS79 defined datatype 'mask'.
- The content of DFSRs is checked more thoroughly before attempting to read the curves.
- Integral parts of LIS Entry Block are sanity checked upon parsing.
- Fixes a bug that resulted in the curve parsing routines allocating to little memory for the numpy array. That ultimately resulted in a segfault. This bug affected both DLIS and LIS, but only occurs for files that are written in a specific way.
- Fixes a bug in the DLIS curve parsing that caused dlisio to incorrectly restore state after attempting to read a broken frame from the file, resulting in the following frames being incorrectly written to the outputted numpy array.

2.1.4 0.3.3 - 2021.04.30

- `lis.load` now raises by default when a file cannot be successfully indexed. In previous versions only a warning was issued and a partially indexed file was returned.
- `lis.load` now accepts a `dlisio.ErrorHandler`-instance similar to `dlis.load`. This allows the caller to decide how load behaves when a file cannot be properly indexed.
- dlisio has learned to parse the following LIS records: Operator Command Inputs, Operator Response Inputs, System Outputs to Operator and FLIC Comment.
- LIS Data Format Specification Records have been given a more user-friendly interface through `lis.DataFormatSpec`.
- dlisio has learned to read LIS curves where the index is recorded in depth recording mode 1.
- dlisio has learned to read LIS curves that are sampled at a higher rate than the recorded index (Fast Channels)
- dlisio has learned to read LIS curves with non-scalar sample-values.
- dlisio has learned to read LIS curves containing strings.
- Added safe-guards against ZeroDevisionError's in lis.curves
- The predecessor and successor bits in the LIS Physical Records Headers are checked for consistency during the indexing routines.
- Error messages emitted from the indexing routines at `lis.load` are now properly communicated to the caller.

2.1.5 0.3.2 - 2021.03.19

- Added support for reading the LIS79 record types: Job Identification, Wellsite Data and Tool String Info.
- Fixes a bug that led dlisio to attempt to read curves with multiple entries in each sample, even though such support was never implemented. Now a NotImplementedError is raised.

2.1.6 0.3.1 - 2021.03.10

- Solves an issue in the Windows deploy pipeline on Appveyor that resulted in the pipeline failing to upload the python wheels to PyPi.

2.1.7 0.3.0 - 2021.03.09

- Added an initial pass at a Log Information Standard 79 (LIS79) reader. Like the DLIS reader, the new LIS reader is mainly implemented in C/C++ with python bindings on top. The reader is not feature complete at this point. But it can read most curves, with a few exceptions (see the docs of dlisio.lis.curves). Basic metadata such as LIS Header/trailer Records (RHLR, RTLR, THLR, TTLR, FHLR, FTLR) can also be read. Support for more complex LIS Records such Information Records is not yet added. The LIS reader resides in the python submodule `dlisio.lis`.
- The python module is restructured to accommodate the new LIS reader. Most notably, all DLIS related functionality is moved to the submodule `dlisio.dlis`. I.e. this release breaks the main entry point of dlisio `dlisio.load` which from this release and onwards is moved to `dlisio.dlis.load`. For a full overview of the restructuring see commit `#736d545`.
- The documentation on readthedocs has been given an overhaul to fit the new module structure and LIS documentation is added.
- Added support for DLIS NOFORM objects.
- Better debug information for broken DLIS files.
- Better error message when passing a directory as path to `dlisio.dlis.load`.
- Nicer error message when failing to construct datetime objects due to invalid dates in the file.
- Added support for python 3.9
- Dropped support for python 3.5
- Restructuring the C/C++ core of dlisio, please refer to the git log for a full overview of the restructuring.
- The C and C++ targets are merged into one target `dlisio`, and `dlisio-extension` ceased to exist.

2.1.8 0.2.6 - 2020.12.16

- Fixes a bug that caused `dlisio.load` to fail on files >2GB on Windows.
- dlisio can now read data from truncated files, this feature is opt-in.
- dlisio can now read data from files that are padded at the end, this feature is opt-in.
- How dlisio handles spec-violations in files is now customisable.
- `describe()` includes attributes units
- `Batch` has been renamed to `physicalfile`.
- `dlis` has been renamed to `logicalfile`
- `dlis.match` (now `logicalfile.match`) is deprecated in favor of `logicalfile.find`

2.1.9 0.2.5 - 2020.10.20

- Fixed a bug where dlisio silently misinterpreted vax-floats.
- More robust handling of encoded strings.
- Internal restructuring. Metadata handling is partially moved to C++.

2.1.10 0.2.4 - 2020.07.27

- fixes a bug in `dl::findoffsets` that caused an infinite loop for certain broken files.

2.1.11 0.2.3 - 2020.06.19

- Fixes a bug in `dlisio.load()` that caused it to leak open file handles when load failed.
- Added official support and distributed wheels for python 3.8.
- Better error message is reported when attempting to load files which do not exist.
- dlisio can now read files which contain empty logical records.
- The cli tool describe.cpp is removed as it has not been maintained and used.

2.1.12 0.2.2 - 2020.06.15

- Fixes a bug in `dlisio.load()` that caused it to leak an open file handle.

2.1.13 0.2.1 - 2020.06.05

- Fixes a bug in the build script that creates the macos wheels. The lfp library was not properly included, resulting in an import error when importing dlisio.

2.1.14 0.2.0 - 2020.06.04

- dlisio can now read files wrapped in Tape Image Format (tif).
- dlisio can now read files that do not contain a Storage Unit Label.
- The numpy array returned by `frame.curves()` can now be indexed with fingerprints in addition to the normal mnemonic indexing. Fingerprints are a more reliable indexing method as these are required to be unique by the standard, unlike mnemonics. This should mainly be of interest to automation pipelines where reliable indexing is key.
- dlisio can now read frames with duplicated channels. This behavior is explicitly forbidden by the spec. However, it is frequently violated. By default, `frame.curves()` still fails, but this can now be bypassed with `strict=False`.
- dlisio no longer accepts files where the last Visible Record is truncated, but the last Logical Record is intact. Support for such truncated files was never intended in the first place, but happened to work.
- `Channel.curves()` fails more gracefully when there is no recorded curve data.
- The documentation has been revamped and new sections focusing on understanding the content and structure of dlis-files are added.

- Fixes a bug that caused `channel.curves()` to use too much memory.
- Fixes a bug that causes `dlisio.load()` to fail if the file contained encrypted fdata record(s).
- Fixes a bug that caused `dlisio.load()` to fail if the obname of a fdata record spanned multiple Visible Records.
- Fixes a bug that re-read unknown objects from disk even if they were cached from previous reads.

2.1.15 0.1.16 - 2020.01.16

- Fixes a bug were `dlisio.load()` did not properly close the memory mapping to the file when loading failed.
- Fixes a bug where `dlis.match()` and `dlis.object()` returned the same object multiple times under certain circumstances.
- `dlis.describe()` again includes the object-count of each object-type.
- `dlisio.load()` now warns if a file contains `Update`-objects. The current lack of support for such objects means that dlisio may wrongfully present data in files with `Update`-objects.
- There is now a list of organization codes on readthedocs
- Fixes a bug in the Process-docs

2.1.16 0.1.15 - 2019.12.18

- Metadata objects are now parsed and loaded when needed, rather than all at once in `dlisio.load()`. This is not directly observable for the user, other than it improves performance for `dlisio.load()`. For files with a lot of metadata, the performance gain is huge.
- dlisio can now read even more curve-data. Specifically, where multiple FDATA (rows) are stored in the same IFLR.
- The array from `Frame.curves()` now includes FRAMENO as the first column. FRAMENO are the row numbers as represented in the file. It might happen that there are missing rows or that they are out-of-order in the file, that is now observable by inspecting FRAMENO.
- Better support for non-ascii strings. It is now possible to tell dlisio which string encodings to try if decoding with 'utf-8' fails. Supply a list of encodings o `set_encodings()` and dlisio will try them in order.
- `Frame.index` now returns the Channel mnemonic, not the `Channel`-object.
- `Channel.index` is removed.
- Validated types are now represented as tuples, not lists.
- Fixes a bug were microseconds in datetime objects were interpreted as milliseconds.
- Better error message when incomplete Channels objects cause parsing of curves to fail as a result.

2.1.17 0.1.14 - 2019.10.14

- dlisio has learned to read curves with variable length data types. Thus, every data-type that the standard allows for curves is now supported by dlisio.
- `Frame`- and `Channel`-objects now have an index-property. `index` returns the `Channel`-object that serves as the index-channel for the given Frame/Channel.

2.1.18 0.1.13 - 2019.10.3

- The sphinx documentation on readthedocs has a few new sections: About the project, an introduction to some dlis-concepts and a quick guide to help new users to get started with dlisio.
- API documentation has seen some improvements as well. The `dlis`-class documentation is revamped to better help users to work with logical files and accessing objects. `Frame` and `Channel` are more thoroughly documented, and more examples on how to work with curve data are provided.
- Direct access to specific objects has been made more convenient with `dlis.object()`.
- `dlis.match()` is no longer case sensitive.
- `dlis.fileheader` now returns the `Fileheader`-object directly, not wrapped as dict_values.
- `dlis.objects` has been removed
- CircleCI is added to the ci-pipeline for building and testing on linux
- Python test suite has seen some refactoring
- It is now possible to build the python module with `setup.py`, provided the core library is already installed on the system.

2.1.19 0.1.12 - 2019.08.15

- Output a readable summary of any metadata-object, logical file or batch-object with `.describe()`.
- Access to curves directly through `Frame`- and `Channel`-objects.
- dlisio has learned to read the following metadata-objects: `Process`, `Path`, `Splice`, `Well reference point`, `Group`, `Message`, `Comment`.
- `dlis.match()` lets you search for objects with a regular expression.
- dlisio now reads even more files. Restrictions such as number-of-objects in an object_set and missing representation codes in templates have been lifted.
- The parsing routine has seen some improvements. This includes giving the user more freedom to customize object-parsing.
- Multidimensional metadata attributes are handled correctly.
- `BasicObject.update_stash` has been removed.
- `dlis.getobjects()` has been removed.
- `dlis.object_set` has been renamed to `dlis.indexedobjects`.
- `Computation.source` is now a scalar, not vector.
- `BasicObject`'s `type` and `attic` is now attributes, not properties.
- Objects are allowed to have empty ids (name/mnemonic).

- The API documentation has seen some minor updates.
- dlisio uses endianness.h rather than its own implementation.
- Some of the binary test files have been simplified.
- core functionality such as `findfdata`, `findsul`, `findvrl`, `findoffsets` and `stream.at` are more thoroughly tested.
- Parts of the Python test suite have been refactored.
- Fixed a bug were long obnames were allocated insufficient memory.
- Fixed a bug were multi-dimensional fdata were interpreted incorrectly.
- Fixed a bug that caused incorrectly partitioning from physical- to logical file(s).
- Fixed a bug that caused parsing of a encrypted logical record to fail.

2.1.20 0.1.11 - 2019.06.04

- Support for logical files - dlisio now partitions the loaded physical file into logical files. This has resulted in a behavioral change were `dlisio.load()` now returns a tuple-like object of n-logical files.

2.2 DLIS Specification

dlis is a binary file format for well logs, developed by Schlumberger in the late 80's and published by the American Petroleum Institute (API) in 1991. In 1998 the stewardship was handed off to Petrotechnical Open Software Corporation (POSC), now known as energistics[1]

When developing dlis standard, the main emphasis was put on easy writing. The files are structured in such a way that data can be written directly while acquiring the logs. This is very handy for the producers of the files, and equally tedious for the consumer that wants to read them later on. Additionally it is a very tolerant standard. It specifies general data-structures within the file, such as channels and frames, but allows the producers to modify these to fit their needs. It also allows for completely new structures, not defined by the standard itself, such as vendor-specific object-types. This all sums up to a fairly complex standard with a lot of quirks and weirdness to it. It is safe to say that dlis a particularly difficult format to work with.

2.3 DLIS User Guide

This is a quick guide to get you started with dlisio. Note that all classes and functions are more thoroughly documented under *DLIS API Reference*. Please refer there for more information about them.

The same documentation is also available directly in your favorite python interpreter and in the unix console, just type `help(function)` or `pydoc function`, respectively. In the interpreter, help can be used directly on class instances. E.g: `help(frame)` or `help(frame.curves)`

[1] POSC, https://www.energistics.org/

2.3.1 Opening files

Load all the *LogicalFile*:

```
>>> from dlisio import dlis
>>> with dlis.load(filename) as files:
...     for f in files:
...         pass
```

The returned `files` is an instance of *PhysicalFile* that can be iterated over and operations can be applied to each logical file.

If you only want to work with one logical file at the time, *load()* supports automatic unpacking of logical files. The following syntax unpacks the first logical file into `f` and stores the rest (0-n) logical files into `*tail`.

```
>>> with dlis.load(filename) as (f, *tail):
...     pass
```

Or, if the number of logical files is known:

```
>>> with dlis.load(filename) as (f1, f2, f3):
...     pass
```

When a file is loaded, you can output some basic information about the physical file:

```
>>> with dlis.load(filename) as files:
...     files.describe()
-------------
Physical File
-------------
Number of Logical Files : 3

Description : LogicalFile(DDBC1)
Frames      : 0
Channels    : 0

Description : LogicalFile(DDBC2)
Frames      : 2
Channels    : 22

Description : LogicalFile(DDBC3)
Frames      : 2
Channels    : 160
```

Or about a logical file:

```
>>> with dlis.load(filename) as (f, *tail):
...     f.describe()
------------
Logical File
------------
Description  : LogicalFile(MSCT_200LTP)
Frames       : 2
Channels     : 104

```

(continues on next page)

(continued from previous page)

```
Known objects
--
FILE-HEADER             : 1
ORIGIN                  : 3
CALIBRATION-COEFFICIENT : 8
CHANNEL                 : 104
FRAME                   : 2

Unknown objects
--
440-CHANNEL             : 93
440-OP-CORE_TABLES      : 17
440-OP-CHANNEL          : 101
```

2.3.2 Accessing objects

Think of *Logical Files* as pools of objects with different types. All objects of a type can be reached by name, e.g. channels or coefficients:

```
>>> for ch in f.channels:
...     pass
```

See *Logical Files* for a full list of all object types.

`LogicalFile.object()` lets you access a specific object:

```
>>> obj = f.object('CHANNEL', 'TDEP')
```

Objects can also be searched for with *`LogicalFile.find()`*:

```
>>> objs = f.find('CHANNEL', 'T.*')
```

Inspect an object with the `.describe()`-method:

```
>>> obj.describe()
-----
Frame
-----
name   : 800T
origin : 2
copy   : 0

Channel indexing
--
Indexed by       : TIME
Interval         : [33354518, 35194520]
Direction        : INCREASING
Constant spacing : 800
Index channel    : Channel(TIME)

Channels
--
```

(continues on next page)

(continued from previous page)

```
TIME TDEP ETIM LMVL UMVL CFLA OCD  RCMD RCPP CMRT
RCNU DCFL DFS  DZER RHMD HMRT RHV  RLSW MNU  S1CY
S2CY RSCU RSTS UCFL CARC CMDV CMPP CNU  HMDV HV
LSWI SCUR SSTA RCMP RHPP RRPP CMPR HPPR RPPV SMSC
CMCU HMCU CMLP
```

2.3.3 Frames and Channels

See *DLIS Curves* for information about the relationship between Channels and Frames. Have a look at *DLIS Channel* and *DLIS Frame*, they contain some useful metadata in addition to the curve-values!

Channels belonging to a Frame can be accessed directly through `Frame.channels`:

```
>>> frame.channels[0]
Channel(TDEP)
```

Likewise, the parent-frame of a Channel can be accessed through the channel:

```
>>> ch.frame
Frame(800T)
```

The actual curve data of a Channel is accessed by `Channel.curves()`, which returns a structured numpy array that support common slicing operations:

```
>>> curve = ch.curves()
>>> curve[0:5]
array([852606., 852606., 852606., 852606., 852606.], dtype=float32)
```

Note that its almost always considerably faster to read curves-data with `Frame.curves()`. Please refer to `Channel.curves()` for further elaboration on why this is.

Access all curves in a frame with `Frame.curves()`. The returned structured numpy array can be indexed by Channel mnemonics and/or sliced by samples:

```
>>> curves = fr.curves()
>>> curves[[fr.index, 'TENS_SL']][0:5]
array([(16677259., 2233.), (16678259., 2237.), (16679259., 2211.),
       (16680259., 2193.), (16681259., 2213.)])
```

Note that double brackets are needed in order to access muliple channels at once.

As long as the frame only contains channels with scalar samples, it can be trivially converted to a pandas DataFrame:

```
>>> import pandas as pd
>>> curves = pd.DataFrame(frame.curves())
```

For more examples of how to work with the curve-data, please refer to `Frame.curves()` and `Channel.curves()`

2.4 DLIS Metadata

Together with the actual logs, there is often an abundance of metadata related to the acquisition of the logs. In a DLIS-file metadata is structured into different `dict`-like objects that describe certain pieces of information. RP66v1 defines over 20 object-types. In practice, only a handful see widespread use. Please refer to the *DLIS API Reference* to get a full[1] overview of the different object-types. Here are some examples of the frequently used ones:

`Origin`: Contains general information about the file, and the circumstances in which it was produced.

`Channel`: Description of a specific curve in the logical file.

`Frame`: A grouping of channels that all share the same index, typically a logpass. The actual curve-data are accessed through Frames.

`Tool`: Describes a physical tool that was used for the acquisition of the logs.

`Parameter`: Contains some parameter value and a description of it.

2.4.1 Identifying specific objects

Common for *all* objects are the four fields: *`type`*, *`name`* (mnemonic), *`origin`* and *`copynumber`*. Together, these form a unique identifier for the object. RP66v1 states that no two objects from the *same logical file* can have the same value for all four fields.

origin: The origin field states which origin the object is a part of. Its interger value implicitly refers to the *`Origin`*-object that has the same value in its origin field.

copynumber: The copynumber is used to distinguish two objects that otherwise have an identical signature. E.g. if there are two *`Channel`* objects with the same name/mnemonic and both belong to the same origin.

To access a specific object use *`LogicalFile.object()`*. Or search for objects matching a regular expression with *`LogicalFile.find()`*

```
>>> channel = f.object('CHANNEL', 'GR', origin=1, copynumber=0)
>>> channel.long_name
'Gamma Ray'

>>> f.find('CHANNEL', '.*GR.*')
[Channel(GR), Channel(RGR)]
```

Note: Note that *`LogicalFile.object()`* allows you to ommit the origin and/or copynumber, but will raise if it's unable to uniquely identify the object. The documentation for *`LogicalFile.object()`* and *`LogicalFile.find()`* offers more examples.

[1] There is currently a handful of object-types (mostly from Chaper 7: semantics: dictionaries of RP66v1) that dlisio does not implement full support for. I.e. they are not translated into their own python class, but rather use the more generic and rough interface of *`BasicObject`*. These rarely see the light of day, but if present they can be accessed through *`LogicalFile.unknowns`*.

2.4.2 Relationship between metadata objects

A key feature of RP66v1 is that objects refer to each other. Object-to-object referencing is used to establish a relationship between two objects. This relationship can serve as an implicit context to the object. Many objects rely on this context and make little sense without it.

Object-references are conveyed through the object's attributes. A concrete example is *Tool.channels*, which references all the channels that are produced directly by that tool. dlisio automatically resolves object references to make it easy to work with programmatically:

```
>>> tool = f.object('TOOL', 'USIT')
>>> tool.channels
[TDEP, BI, CBL, CBLF, CBSL, ..., CMCG, WF1, WF1N, WF2, WF2N]

>>> channel = tool.channels[1]
>>> channel.long_name
'Bond Index'
```

2.4.3 Units

Alongside attribute values, a DLIS-file also contains a corresponding **units** field. This is true for all object attributes, even when units make little sense, such as for *Channel.long_name*.

Currently there is no *nice* interface for accessing units with dlisio. They are, however, reachable through some of the more basic interfaces. Sensible units are also printed in *BasicObject.describe()*.

Each metadata object has in itself an attribute attic, *BasicObject.attic*. The attic is a complete dict-like representation of the object, but without the syntactic sugar provided by the specialized class of the object.

At the moment attribute units are only accessible through the attic: `obj.attic['ATTRIBUTE'].units`.

Every attribute has a *RP66V1 name* section in its docs that corresponds to the `ATTRIBUTE` value, which must be used to retrieve the raw attribute.

For example, given the object `frame` and docs for *Frame.spacing*:

```
>>> frame.spacing
... 800
>>> frame.attic['SPACING'].units
... '0.5 ms'
```

2.4.4 Multiple origins

RP66v1 allows for a single logical file to have multiple *Origin*-objects. Origin describes the source of the data, such as which field and well it's from. It's therefore theoretically possible that the data contained in a logical file stem from different sources when there are more than one *Origin*-objects. Such files obviously need special care. More precisely, the origin field of each object that is accessed needs to be examined in order to determine which origin it belongs to.

The majority of files do *not* contain multiple origins, which makes it safe to ignore the origin field. However, it is considered a good practice to check the origin count when opening a new file, e.g. by issuing a warning if there are more than one:

```
import logging
from dlisio import dlis
```

(continues on next page)

(continued from previous page)

```
with dlis.load(path) as (f, *tail):
   if len(f.origins) > 1: logging.warning('File contains multiple origins')
```

2.4.5 Vendor-specific metadata

In addition to the many object-structures defined by RP66v1 itself, vendors are free to specify their own metadata objects. However, with often cryptic naming and minimal explanation, such objects can be challenging to decipher without any external explanation of the intent of these objects.

The vendor-specific objects are reachable through `f.unknowns`:

```
with dlis.load(path) as (f, *tail):
   f.unknowns
```

2.5 DLIS Curves

The primary data of a dlis-file are curves, also referred to as channels. Typically a curve is the recorded measurements taken along the borehole, indexed against an axis, like depth or time. But a curve can also be a computation of some sort, e.g. a calibrated version of a measured curve.

In dlisio, curves are accessed through *`Frame`*- or *`Channel`*-objects, by calling *`Frame.curves()`* or *`Channel.curves()`*. The primary data type for curves is structured numpy.ndarray. This enables quick and easy mathematical operations on the data you care about.

2.5.1 Frame-objects

Curves are organized in frames. Conceptually, a frame can be seen as a table of data where each column is a curve, like in the time-indexed frame below. Frames almost always have an index channel that provides the position in e.g. depth or time at which the rest of the values in the row were measured. Each frame usually corresponds to a log run, but otherwise frames impose little structure except grouping channels that have a common index. There is no restrictions on the number of channels per frame, or the number of frames in a file.

Frames are described by *`Frame`*-objects. These contain a list of *`Channel`*-objects, which describe the individual curves in more detail. Frame objects also contain additional information about the index channel, such as its min/max values, spacing, direction and type-of-index. See *`Frame`* for a full list of its attributes.

Table 1: A TIME indexed frame

FRAMENO	TIME	TDEP	ETIM	LMVL	UMVL	CFLA	OCD
1	16677259	852606.0	0.	585	635	18	6789.05
2	16677659	852606.0	0.4	585	635	18	6789.05
3	16678059	852606.0	0.8	585	635	0	6789.05
4	16678459	852606.0	1.2	585	635	0	6789.05
5	16678859	852606.0	1.6	585	635	0	6789.05

Note: FRAMENO: The first column of every Frame is always FRAMENO, which is the row number. Generally FRAMENO is not that interesting, but it can aid in debugging strange-looking curve-data. For example if you are suspecting that some of the data is missing (or even is out-of-order).

2.5.2 Channel-objects

Curve-metadata is recorded in `Channel`-objects. Each `Channel`-object describes a single curve, e.g. TDEP, GR, VDL, WF1, etc. The metadata includes a description, the dimension of each sample and their units, which `Frame` the curve belongs to and the source of the curve. The source is the entity that produced the curve. For measured curves, that is typically a `Tool`. For computed curves, the source might be a `Computation`. Additionally, `Channel` contains a list of property indicators. These indicate the general intrinsic properties of the Channel. Some examples are MEASURED, COMPUTED, DEPTH-MATCHED and AVERAGED. Appendix C of RP66v1 defines the full list of property indicators with descriptions.

2.5.3 N-dimensional curve samples

By far the most common case is that each channel has scalar samples i.e. a single measured numerical value per row, however RP66v1 allows each sample to be a multi- dimensional array. For example, this is common for ultrasonic logs, where some channels contain a one-dimensional array of values per row, representing measurements made at different azimuthal angles. For other channels each sample may even be a 2- or 3-dimensional array.

RP66v1 imposes no limit on the dimensionality that a channel-sample can have, nor does dlisio.

Note: Thanks to numpy's structured numpy.ndarray, dlisio is able to support n-dimensional curves of any shape and form. However, other popular formats such as Pandas DataFrame or CSV are only able to handle tabular data, i.e. frames where all channels have scalar sample values. Hence, conversion to these formats may not always be possible.

2.5.4 Channels with no data

It is not uncommon for files to have `Channel`-objects describing curves that are not present in the file. I.e. there is metadata describing a curve, but the curve itself is missing.

This typically happens because metadata like `Tool`-objects and all the curves it can produce is recorded prior to the acquisition. However when acquisition starts only a subset of the channels is actually recorded, but the metadata for the unused channels are never removed.

Note: Calls to `Channel.curves()` and `Channel.frame` returns `None` when there is no recorded curve data.

2.6 DLIS API Reference

2.6.1 Load a DLIS-file

`dlisio.dlis.`**`load`**(*path*, *error_handler=None*)

Loads a file and returns one filehandle pr logical file.

Load does more than just opening the file. A DLIS file has no random access in itself, so load scans the entire file and creates its own index to emulate random access.

DLIS-files are segmented into Logical Files, see *`dlisio.dlis.PhysicalFile`* and *`dlisio.dlis.LogicalFile`*. The partitioning into Logical Files also happens at load.

Parameters

- **path** (*str_like*) –
- **error_handler** (`dlisio.common.ErrorHandler`, *optional*) – Error handling rules. Default rules will apply if none supplied. Handler will be added to all the logical files, so users may modify the behavior at any time.

Returns **dlis**

Return type *dlisio.dlis.PhysicalFile*

Notes

It's not uncommon that DLIS files are stored with different file extensions than *.DLIS*. For example *.TIF* [1]. Load does not care about file extension at all. As long as the content adheres to the Digital Log Interchange Standard, load will read it as such.

[1] TIF in this case refers to TapeImageFormat, and is not to be confused with the image format TIFF.

Examples

Load is designed to work with python's `with`-statement:

```
>>> from dlisio import dlis
>>> with dlis.load(filename) as files:
...     for f in files:
...         pass
```

Automatically unpack the first logical file and store the remaining logical files in tail

```
>>> with dlis.load(filename) as (f, *tail):
...     pass
```

Note that the parentheses are needed when unpacking directly in the with- statement. The asterisk allows an arbitrary number of extra logical files to be stored in tail. Use len(tail) to check how many extra logical files there are.

2.6.2 Physical File

class dlisio.dlis.**PhysicalFile**

A Physical File

A physical DLIS file, i.e. a regular file on disk, is segmented into multiple Logical Files (LF). Think of a DLIS file as a folder-structure. The DLIS-file itself is the folder, and the Logical Files are the actual files:

```
your_file.dlis
|
|-> Logical File 0
|-> Logical File 1
|-> Logical File 2
...
|-> Logical File n
```

Each LF is a logical grouping of log data and metadata related to the acquisition of those logs. The LF's are independent of each other and can be thought of as separate files.

This class is essentially a tuple of all the Logical Files in a regular file, and is what's being returned by `dlisio.dlis.load()`. The LFs can be unpacked following standard Python tuple unpacking rules.

Notes

The DLIS-specification, rp66v1, opens up for a Logical File to span multiple physical files. dlisio does not currently have cross-file support. We have yet to see any files in the wild using this feature. Note that dlisio will still be able to open such files, but any internal cross-referencing will be lost.

See also:

`dlisio.dlis.load` Open and Index a DLIS-file

`dlisio.dlis.LogicalFile` Interface for working with a single Logical File

close()

Close the file handles of all Logical Files

describe(*width=80*, *indent=''*)

Describe

Returns a human-readable and printable summary of the current Physical File.

2.6.3 Logical Files

class dlisio.dlis.**LogicalFile**

Logical file (LF)

This class supplies the main interface for working with a single Logical File. A Logical File contains log data and metadata related to the acquisition of the logs. The metadata is stored as 'objects' - in lack of a better word. There are different object-types for different types of data. The logs can be acquired through Frame- and Channel-objects (`dlisio.dlis.Frame` and `dlisio.dlis.Channel`). There is also an abundance of object-types for storing other metadata: Tool, Parameter, Measurement and Calibration to name a few.

`object()` and `find()` let you access specific objects or search for objects matching a regular expression. Unknown objects such as vendor-specific objects are accessible through the 'unknown'-attribute.

Uniquely identifying an object within a logical file can be a bit cumbersome, and confusing. It requires no less than 4 attributes! Namely, the object's type, name, origin and copynumber. The standard allows for two objects of the same type to have the same name, as long as either their origin or copynumber differ. E.g. there may exist two channels with the same name/mnemonic.

Metadata caching

All metadata (e.g. Frame, Channel, Parameter) are cached by default. This avoids re-creation of the same objects.

Note that changes to settings such as *`dlisio.common.set_encodings()`* or `LogicalFile.error_handler` *after* loading the file will not propagate to already-cached objects. It's therefore advisable that such settings are set before loading the file. Alternatively, you can manually toggle on and off caching to clear it with *`cache_metadata()`*.

`types = {}`

Dispatch-table for turning `dlisio.core.basic_object` into type-specific python objects like Channel and Frame. This is mainly intended for internal use so the typical user can safely ignore this attribute.

Object-types not present in the table are considered as unknowns. They can still be reached through *`object()`* and *`find()`* but lack the syntactic sugar added by the type-specific classes.

It is possible to monkey-patch the dispatch-table with your own custom classes. However this is considered to be a fairly advanced usage and it's then the users responsibility of ensuring correctness for the custom class.

Type dict

`close``()`

Close the file handle

It is not necessary to call this method if you're using the *with* statement, which will close the file for you.

`cache_metadata``(`*cache*`)`

Toggle caching of metadata objects

By default, the metadata objects are cached to avoid unnecessary re-creation of objects. Like all caching, this trades CPU time for memory usage.

Parameters **`cache`** (*`bool`*) – Toggle caching on/off

`property fileheader`

Return the Fileheader

Returns **fileheader**

Return type *Fileheader*

`axes = None`

`calibrations = None`

`channels = None`

`coefficients = None`

`comments = None`

`computations = None`

`equipments = None`

`frames = None`

`groups = None`

`longnames = None`

`measurements = None`

`messages = None`

`origins = None`

`parameters = None`

`paths = None`

`processes = None`

`splices = None`

`tools = None`

`wellrefs = None`

`zones = None`

`noformats = None`

`property unknowns`

Return all objects that are unknown to dlisio.

Unknown objects are object-types that dlisio does not know about. By default, any metadata object not defined by rp66v1 [1]. The are all parsed as *`dlisio.dlis.Unknown`*, that implements a dict interface.

[1] http://w3.energistics.org/rp66/v1/Toc/main.html

Notes

Adding a custom python class for an object-type to dlis.types will in-effect remove all objects of that type from unknowns.

Returns **objects** – A defaultdict index by object-type

Return type defaultdict(dict)

`find`(*objecttype*, *objectname=None*, *matcher=None*)

Find objects in the logical file

Find searches and returns all objects matching objecttype, and if specified, objectname. By default find uses python's re module with case-insensitivity when searching for objects matching objecttype and objectname. See examples for how to use.

Parameters

- **objectype** (*str*) – type of objects to be looked for
- **objectname** (*str, optional*) – name / mnemonic of objects to be looked for
- **matcher** (*Any matcher derived from dlisio.core.matcher, optional*) – matcher object to be used when comparing objecttype, objectname to file content. Default is `dlisio.dlis.regex`.

Returns **objects** – A list of all objects in the logicalfile that matches objecttype (and objectname)

Return type list

See also:

`dlisio.dlis.utils.exact_matcher` str comparison w/ str.__eq__

`dlisio.dlis.utils.regex_matcher` str comparison w/ python's re module

Examples

Query for all Channels where the name / mnemonic contains 'GR':

```
>>> f.find('CHANNEL', '.*GR.*')
[Channel(GR), Channel(GR1)]
```

Query for all Channel-like objects. I.e. both regular Channels and vendor-specific ones:

```
>>> f.find('.*CHANNEL')
[Channel(TDEP), Channel(GR), Channel(GR1), 440channel(GR2)]
```

The two queries above can be combined:

```
>>> f.find('.*CHANNEL', '.*GR.*')
[Channel(GR), Channel(GR1), 440channel(GR2)]
```

Omitting the objectname yields *all* objects matching objecttype:

```
>>> f.find('FRAME')
[Frame(60B), Frame(20B), Frame(10B)]
```

If your query does not include a regular expression and you care about performance, tell find to use the default exact matcher. For large files there is a significant difference between comparing strings with self.exact and compiling- and comparing regex expressions with self.regex.

```
>>> f.find('FRAME', matcher=dlisio.dlis.exact)
[Frame(60B), Frame(20B), Frame(10B)]
```

`object`(*type*, *name*, *origin=None*, *copynr=None*)

Direct access to a single object. dlis-objects are uniquely identifiable by the combination of type, name, origin and copynumber of the object. However, in most cases type and name are sufficient to identify a specific object. If origin and/or copynr is omitted in the function call, and there are multiple objects matching the type and name, a ValueError is raised.

Parameters

- **`type`** (*`str`*) – type as specified in RP66
- **`name`** (*`str`*) – name
- **`origin`** (*`number, optional`*) – number specifying which origin object the current object belongs to
- **`copynr`** (*`number, optional`*) – number specifying the copynumber of current object

Returns **obj**

Return type searched object

Examples

```
>>> ch = f.object("CHANNEL", "MKAP", 2, 0)
>>> print(ch.name)
MKAP
```

describe(*width=80*, *indent=''*)

Printable summary of the logical file

Parameters

- **width** (*int*) – maximum width of each line.
- **indent** (*str*) – string that will be prepended to each line.

Returns **summary** – A printable summary of the logical file

Return type dlisio.dlis.utils.Summary

load()

Force load all objects - mainly indended for debugging

storage_label()

Return the storage label of the physical file

Notes

This method is mainly intended for internal use.

2.6.4 DLIS Frame

class dlisio.dlis.**Frame**(*BasicObject*)

Frame

A Frame is a logical gathering of multiple Channels (curves), typically Channels from the same run. A Frame containing three Channels would look something like this:

```
 TDEP     AIBK    TENS_SL
-------  -------  -------
|          |           |
 |          |         |
  |         |        |
   |       |        |
    |       |       |
     |     |         |
```

Usually, the first channel in the channel list is considered to be the index-channel of the Frame. See:attr:*index_type* for more information about the index channels.

All Channels belonging to a Frame are directly accessible through `channels`. A full table of all the curve-data can be accessed with `curves`.

description

Textual description of the Frame.

RP66V1 name: *DESCRIPTION*

Type str

`channels`

Channels in the frame

RP66V1 name: *CHANNELS*

Type list(*Channel*)

`index_type`

The measurement of the index, e.g. borehole-depth. If **not** None, the first channel is considered to be an index channel. If index_type is None, then the Frame has no index channel and is implicitly indexed by samplenumber i.e. 0, 1, . . . , n.

RP66V1 name: *INDEX-TYPE*

Type str

`direction`

Direction of the index (Increasing or decreasing)

RP66V1 name: *DIRECTION*

Type str

`spacing`

Constant spacing in the index

RP66V1 name: *SPACING*

`index_min`

Minimum value of the index

RP66V1 name: *INDEX-MIN*

`index_max`

Maximum value of the index

RP66V1 name: *INDEX-MAX*

`encrypted`

If the frame was encrypted

RP66V1 name: *ENCRYPTED*

Type bool

See also:

`BasicObject` The basic object that Frame is derived from

`Channel` Channel objects

Notes

The Frame object reflects the logical record type FRAME, defined in rp66. FRAME objects are listed in Appendix A.2 - Logical Record Types, and described in detail in Chapter 5.7.1 - Static and Frame Data, FRAME objects.

`curves`(*strict=True*)

All curves belonging to this frame

Get all the curves in this frame as a structured numpy array. The frame includes the frame number (FRAMENO), to detect errors such as missing entries and out-of-order frames.

Parameters `strict` (*boolean, optional*) – By default (strict=True) curves() raises a ValueError if there are multiple channel with the same values for both name, origin and copynumber. This would be a clear violation of the dlis-spec. Setting strict=False lifts this restriction and dlisio will append numerical values (i.e. 0, 1, 2 ..) to the labels used for column-names in the returned array.

Returns **curves** – curves with dtype = self.dtype

Return type np.ndarray

Raises `ValueError` – If there multiple channels with identical name, origin, copynumber in Frame.channels. This can be suppressed by passing strict=False

See also:

`Channel.curves` Access the curve-data directly through the Channel objects

`Frame.dtype` dtype of the array

Examples

The returned array supports both horizontal- and vertical slicing. Do a vertical slice by specifying a single Channel

```
>>> curves = frame.curves()
>>> curves['CHANN1']
array([16677259., 852606., 16678259., 852606.])
```

Access a subset of Channels, note the double-bracket syntax

```
>>> curves[['CHANN2', 'CHANN3']]
array([
    (16677259., 852606.),
    (16678259., 852606.),
    (16679259., 852606.),
    (16680259., 852606.)
])
```

Do a horizontal slice of all Channels, i.e. read a subset of samples from all channels

```
>>> curves[0:2]
array([
    (16677259., 852606., 2233., 852606.),
    (16678259., 852606., 2237., 852606.)])
```

Horizontal and vertical slicing can be combined

```
>>> curves['CHANN2'][0]
16677259.0
```

And here the subscription order is irrelevant

```
>>> curves[0]['CHANN2']
16677259.0
```

Some curves, like image curves, have multi-dimensional samples. Accessing a single sample from a 2-dimensional curve

```
>>> curves = frame.curves()
>>> sample = curves['MULTI_D_CHANNEL'][0]
>>> sample
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11,  12]])
```

This sample is a 2-dimensional array of size 4x3. We can continue to slice this sample. Note that now the subscription order here **does** matter. Here we read the two last rows

```
>>> sample[-2:]
array([[  7,  8,  9],
       [ 10, 11, 12]])
```

Lets read every second column

```
>>> sample[:,::2]
array([[ 1, 3],
       [ 4, 6],
       [ 7, 9],
       [10, 12]])
```

Note the syntax. Within the brackets, everything before the ‘,’ is row- operations and everything after are column-operations. Read it as: keep all rows (:) and then from first to last column, keep every 2nd column (::2). The comma syntax for indexing different axes extends to array’s with higher orders as well.

Combine the two to read a specific element

```
>>> sample[0,0]
1
```

If you prefer to work with pandas over numpy, the conversion is trivial

```
>>> import pandas as pd
>>> curves = pd.DataFrame(frame.curves())
```

Note that pandas (and CSV) *only* supports scalar sample values. I.e. frames containing one or more channels that have none-scalar sample values cannot be converted to pandas.DataFrame or CSV directly.

If the Frame contains an index Channel, use that as index in the DataFrame

```
>>> curves = frame.curves()
>>> pdcurves = pd.DataFrame(curves, index=curves[frame.index])
>>> pdcurves.index.name = frame.index
```

By default the returned np.ndarray have column-names that reflects the mnemonics of each Channel

```
>>> frame.channels
[Channel(TDEP), Channel(TIME), Channel(GR)]
```

```
>>> curves = frame.curves()
>>> curves.dtype.names
('FRAMENO', 'TDEP', 'TIME', 'GR')
```

Sometimes a Frame can contain multiple Channels with the same name/mnemonic, in that case the labels are augmented with the origin and copynumber:

```
>>> frame.channels
[Channel(TDEP), Channel(TDEP), Channel(GR)]
```

```
>>> curves = frame.curves()
>>> curves.dtype.names
('FRAMENO', 'TDEP.0.0', 'TDEP.0.1', 'GR')
```

This augmentation is customizable by changing `dtype_fmt`. See *dtype*. Some frames have multiple instances of the same channel or mulitple channels where name, origin and copynumber are identical. This is a clear violation of the dlis spec and *curves()* will raise an ValueError by default.

```
>>> frame.channels
[Channel(TDEP), Channel(TDEP), Channel(GR)]
```

```
>>> Frame.curves()
ValueError: field 'TDEP.0.0' occurs more than once
```

However, *curves()* offers an escape-hatch to get the underlying data. By passing strict=False dlisio appends numerical values to the identical channels

```
>>> curves = frame.curves(strict=False)
>>> curves.dtype.names
('FRAMENO', 'TDEP.0.0(0)', 'TDEP.0.0(1)', 'GR')
```

dtype(*strict=True*)

data-type of each frame, i.e. the sum of channel.dtype of each channel in self.channels. The first column is always FRAMENO.

If all curve mnemonics are unique, then dtype.names == [‘FRAMENO’] + [ch.name for ch in self.channels]. If there are more than one channel with the same name for this frame, all duplicated mnemonics are enriched with origin and copynumber.

Consider a frame with the channels mnemonics [(‘TIME’, 0, 0), (‘TDEP’, 0, 0), (‘TIME, 1, 0)]: dtype.names for this frame would be (‘FRAMENO’, ‘TIME.0.0’, ‘TDEP’, ‘TIME.1.0’).

Duplicated mnemonics are formatted by the dtype_fmt attribute. To use a custom format for a specific frame instance, set dtype_fmt for the Frame object. If you want to have some other formatting for *all* dtypes, set the dtype_format class attribute. It has to be a 3-element format-string taking a string and two ints. Custom formatting is particularly useful for peculiar files where the full stop (.) appears in the mnemonic itself, and a consistent way of parsing origin and copynumber are needed.

In addition to the customizable dtype.names, ch.fingerprint is always used as field title, which serves as an alias for the name.

Returns **dtype**

Return type np.dtype

Examples

A frame with two TIME channels:

```
>>> dtype = frame.dtype()
>>> dtype.names
dtype([('FRAMENO', '<i4'),
       (('T.CHANNEL-I.TIME-O.0-C.0','TIME.0.0'), '<f4'),
       (('T.CHANNEL-I.TDEP-O.0-C.0','TDEP'), '<i2'),
       (('T.CHANNEL-I.TIME-O.1-C.0','TIME.1.0'), '<i2')])
```

Override instance-specific mnemonic formatting

```
>>> frame.dtype().names
(FRAMENO', 'TIME.0.0', 'TDEP', 'TIME.1.0')
>>> frame.dtype_fmt = '{:s}-{:d}-{:d}'
>>> frame.dtype()
(FRAMENO','TIME-0-0', 'TDEP','TIME-1-0')
```

property index

Mnemonic of the channel all channels in this Frame are indexed against, if any. See *Frame.index_type* for definition of existing index channel.

Returns **mnemonic**

Return type str

2.6.5 DLIS Channel

class dlisio.dlis.**Channel**(*BasicObject*)

A channel is a sequence of measured or computed samples that are indexed against some physical quantity e.g. depth or time. The standard supports multi-dimensional samples. Each sample can be a scalar or a n-dimensional array. In addition to giving access to the actual curve-data, the Channel object contains metadata about the curve.

All Channels are a part of one, and only one, Frame. The parent Frame can be reached directly through *frame*.

Refer to the *curves()* to see some examples on how to access the curve-data.

long_name

Descriptive name of the channel.

RP66V1 name: *LONG-NAME*

Type str or *Longname*

reprc

Representation code

RP66V1 name: *REPRESENTATION-CODE*

Type int

`units`

Physical units of each element in the channel's sample arrays

RP66V1 name: *UNITS*

Type str

`properties`

Property indicators that summarizes the characteristics of the channel and the processing that have produced it.

RP66V1 name: *PROPERTIES*

Type list(str)

`dimension`

Dimensions of the samples

RP66V1 name: *DIMENSION*

Type list(int)

`axis`

Coordinate axes of the samples

RP66V1 name: *AXIS*

Type list(*Axis*)

`element_limit`

The maximum size of the sample dimensions

RP66V1 name: *ELEMENT-LIMIT*

Type list(int)

`source`

The source of the channel. Returns the source object, if any

RP66V1 name: *SOURCE*

`frame`

Frame to which channel belongs to

Type *Frame*

See also:

`BasicObject` The basic object that Channel is derived from

Notes

The Channel object reflects the logical record type CHANNEL, defined in rp66. CHANNEL records are listed in Appendix A.2 - Logical Record Types and described in detail in Chapter 5.5.1 - Static and Frame Data, CHANNEL objects.

`curves()`

Returns a numpy ndarray with the curves-values.

Returns **curves**

Return type np.ndarray or None

See also:

Frame.curves

Notes

This method should only be used if there is only *one* channel of interest in a particular frame.

Due to the memory-layout of dlis-files, reading a single channel from disk and reading the entire frame is almost equally fast. That means reading channels from the same frame one-by-one with this method is _way_ slower than reading the entire frame with `Frame.curves()` and then indexing on the channels-of-interest.

Examples

Read the full curve

```
>>> curve = channel.curves()
>>> curve
array([1.1, 2.2, 3.3, 4.4])
```

The returned array supports common slicing operations

```
>>> curve[::2]
array([1.1, 3.3])
```

Read the full curve from a multidimensional channel

```
>>> curve = multichannel.curves()
>>> curve
array([[[  1,  2,  3],
        [  4,  5,  6]],
       [[  7,  8,  9],
        [ 10, 11,  12]]])
```

This curve has two samples, that both are of size 2x3. From the 1st sample, read the element located in the 2nd row, 3rd column

```
>>> curve[0][1][2]
6
```

property dtype

data-type of each sample in the channel's sample array. The dtype-label is *channel.name.id*.

Returns **dtype**

Return type np.dtype

2.6.6 Other Metadata

Basic Object

`class dlisio.dlis.BasicObject`

A Basic object that all other object-types derive from

BasicObject is mainly an implementation detail. Its the least common denominator of all object-types.

When working with dlisio you need not care too much about the BasicObject. However, keep in mind that all other object-types inherit BasicObject's attributes and methods and they can be called directly on the objects. Hence it might be a good idea to somewhat familiarize yourself with its features.

When reading the documentation keep in mind that the attributes defined on the BasicObject are **not** documented again on the derived object.

An object is uniquely identifiable within a logical file by the combination of its type, name, origin and copynumber. I.e. no two objects can have the same type, name, origin AND copynumber. This means that the standard allows for e.g. two Channels to have the same name/mnemonic, which can easily become a bit confusing.

`type`

Type of the object, e.g. rp66 object-types as CHANNEL or FRAME

Type str

`name`

Mnemonic / name

Type str

`origin`

Defines which origin the object belongs to

Type int

`copynumber`

There may exist several copies of the same object, the copynumber is used to distinguish them

Type int

`attic`

Attic refers the underlying basic_object, which is a dict-representation of the data on disk. The attic can be None if this particular instance was not loaded from disk.

Type dict_like

`attributes = {}`

Parsing guide for native dlis objects. The typical user can safely ignore this attribute. It is mainly intended for internal use and advanced customization of dlisio's parsing routine.

In short, this dictionary tells dlisio parse each attribute for a given object. By default it implements rules defined by the dlis-spec. However it can easily be customized if you'd like dlisio to parse your file in a different way.

Examples

Lets take a look at Channel.attributes, it looks like this:

```
attributes = {
    'LONG-NAME'          : scalar,
    'REPRESENTATION-CODE': scalar,
    'UNITS'              : scalar,
    'PROPERTIES'         : vector,
    'DIMENSION'          : reverse,
    'AXIS'               : reverse,
    'ELEMENT-LIMIT'      : reverse,
    'SOURCE'             : scalar,
}
```

The keys are from the dlis-spec, i.e. this is how the attributes of a CHANNEL-object are named in the file. The values tells dlisio how to interpret these keys. In the dlis-spec it's defined that 'UNITS' only contains a single value. This is communicated to dlisio with the 'scalar'-keyword. I.e. Channels __getitem__ will return a single value:

```
>>> channel['UNITS']
'm/s'
```

The __getitem__ is not intended for direct use. However it is called internally from all properties of the Channel-object. I.e. you observe the same here:

```
>>> channel.units
'm/s'
```

The following example might be a bit absurd, but keep in mind that this approach can be applied to _any_ attribute of _any_ object-type, even Unknown object-types.

But now let's say that you are in possession of a file that you know is structured differently from what the standard specifies. It contains some weird Channel's where there are multiple units per Channel. Simply update the attribute-dict before loading the file and dlisio will parse 'UNITS' as a list:

```
>>> from dlisio.dlis.utils import vector
>>> Channel.attributes['UNITS'] = vector
>>> with dlisio.dlis.load('file.dlis') as (f, *_):
...     ch = f.object('CHANNEL', 'TDEP')
...     ch.units
['m/s', 'rad/s']
```

The same can be achieved for a _specific_ object. Forcing a copy of the attribute-dict for a given object before altering it makes sure your changes only apply _that_ object

```
>>> ch = f.object('CHANNEL', 'TDEP')
>>> ch.attributes = dict(ch.attributes)
>>> Channel.attributes['UNITS'] = vector
>>> ch.units
['m/s', 'rad/s']
```

References

[1] http://w3.energistics.org/RP66/V1/Toc/main.html

Type dict

linkage = {}

Defines which attributes contain references to other objects. The typical user can safely ignore this attribute. It is mainly intended for internal use and advanced customization of dlisio's parsing routine.

Object-to-object references often contains implicit information. E.g. Frame.channels implicitly reference Channel object, so the type 'CHANNEL' is not specified in the file. Hence dlisio needs to be told about this.

Like for attributes, this behavior is customizable.

Examples

Change how dlisio parses Channel.source:

```
>>> from dlisio.dlis.utils import obname
>>> Channel.linkage['SOURCE'] = obname('PARAMETER')
>>> with dlisio.dlis.load('file.dlis') as (f, *_):
...     ch = f.object('channel', 'TDEP')
...     ch.source
Parameter('2000T')
```

The same can be achieved for a _specific_ object. Forcing a copy of the linkage-dict makes sure your changes only apply to that specific object:

```
>>> ch = f.object('channel', 'TDEP')
>>> ch.linkage = dict(ch.linkage)
>>> ch.linkage['SOURCE'] = dlisio.plumbing.parse.obname('PARAMETER')
>>> ch.source
Parameter('2000T')
```

Type dict

property fingerprint

Object fingerprint

Return the fingerprint, a unique identifier, for this object. This is basically an objref type from the RP66 standard, but with a pythonic flavour, and suitable for keys in dicts.

Returns fingerprint

Return type str

property stash

Attributes unknown to dlisio

It is not uncommon for objects to have 'extra' attributes that are not defined by the standard. Because such attributes are unknown to dlisio, they cannot be reach through normal attributes.

Returns stash – all attributes not defined in *attributes*

Return type dict

describe(*width=80*, *indent=''*, *exclude='er'*)

Printable summary of the object

Parameters

- **width** (*int*) – the maximum width of each line.
- **indent** (*str*) – string that will be prepended to each line.
- **exclude** (*str*) – exclude certain parts of the object in the summary.

Returns **summary** – A printable summary of the object

Return type utils.Summary

Notes

The exclude parameter gives the option to omit parts of the summary. The table below states the different modes available.

option	Description
'h'	header
'a'	known attributes
's'	attributes from stash
'u'	units
'i'	attributes that violates the standard [1]
'e'	attributes with empty values (default) [2]

[1] Only applicable to attributes that should be interpreted in a specific way, such as Parameter.values. If not applicable, it is ignored.

[2] Do not print attributes that have no value.

Axis

class dlisio.dlis.**Axis**(*BasicObject*)

Axis

The Axis object describes the coordinate axis of an array, e.g. the sample array of channels. One axis object describes only one coordinate axis. I.e a three dimensional array is described by three Axis objects.

axis_id

Axis identifier

RP66V1 name: *AXIS-ID*

Type str

coordinates

Explicit coordinate value along the axis

RP66V1 name: *COORDINATES*

Type list

`spacing`

Constant, signed spacing along the axis between successive coordinates

RP66V1 name: *SPACING*

See also:

`BasicObject` The basic object that Axis is derived from

Notes

The Axis object reflects the logical record type AXIS, defined in rp66. AXIS objects are listed in Appendix A.2 - Logical Record Types, and described in detail in Chapter 5.3.1 - Static and Frame Data, Axis Objects.

Calibration

`class dlisio.dlis.Calibration(`*BasicObject*`)`

Calibration objects are a collection of measurements and coefficients that defines the calibration process of channel objects.

`method`

Computational method used to calibrate the channels

RP66V1 name: *METHOD*

Type str

`calibrated`

Calibrated channels

RP66V1 name: *CALIBRATED-CHANNELS*

Type list(*Channel*)

`uncalibrated`

Uncalibrated channels. I.e. the channels as they where before calibration

RP66V1 name: *UNCALIBRATED-CHANNELS*

Type list(*Channel*)

`coefficients`

Coefficients

RP66V1 name: *COEFFICIENTS*

Type list(*Coefficient*)

`measurements`

Measurements

RP66V1 name: *MEASUREMENTS*

Type list(*Measurement*)

`parameters`

Parameters containing numerical and textual information assosiated with the calibration process.

RP66V1 name: *PARAMETERS*

Type list(*Parameter*)

See also:

BasicObject The basic object that Calibration is derived from

Notes

The Calibration reflects the logical record type CALIBRATION, defined in rp66. CALIBRATION records are listen in Appendix A.2 - Logical Record Types and described detail in Chapter 5.8.7.3 - Static and Frame Data, CALIBRATION objects.

Coefficent

class dlisio.dlis.**Coefficient**(*BasicObject*)

Records of measurements, references, and tolerances used in the calibration of channels.

label

Identify the coefficient-role in the calibration process

RP66V1 name: *LABEL*

Type str

coefficients

Coefficients corresponding to the label

RP66V1 name: *COEFFICIENTS*

Type list

references

Nominal values for each coefficient

RP66V1 name: *REFERENCES*

Type list

plus_tolerance

Maximum value that a sample can exceed the reference and still be "within tolerance"

RP66V1 name: *PLUS-TOLERANCES*

Type list

minus_tolerance

Maximum value that a sample can fall below the reference and still be "within tolerance"

RP66V1 name: *MINUS-TOLERANCES*

Type list

See also:

BasicObject The basic object that Coefficient is derived from

Notes

The Coefficient object reflects the logical record type CALIBRATION-COEFFICIENT, defined in rp66. CALIBRATION-COEFFICIENT records are listed in Appendix A.2 - Logical Record Types and described in detail in Chapter 5.8.7.2 - Static and Frame Data, CALIBRATION-COEFFICIENT objects.

Comment

class dlisio.dlis.**Comment**(*BasicObject*)

Comment objects contains arbitrary messages that may be interesting for the consumer e.g. a drilling report.

text

Textual comments

RP66V1 name: *TEXT*

Type list(str)

See also:

BasicObject The basic object that Comment is derived from

Notes

The Comment object reflects the logical record type COMMENT, described in rp66. COMMENT objects are defined in Appendix A.2 - Logical Record Types, described in detail in Chapter 6.1.2 - Transient Data, Comment objects.

Computation

class dlisio.dlis.**Computation**(*BasicObject*)

Results of computations that are more appropriately expressed as static information rather than as channels.

The computation value(s) may be scalars or an array. In the later case, the structure of the array is defined in the dimension attribute. The zones attribute specifies which zones the computations is defined. If there are no zones the computation is defined everywhere.

The axis attribute, if present, defines axis labels for multidimensional value(s).

long_name

Descriptive name of the computation

RP66V1 name: *LONG-NAME*

Type str or *Longname*

properties

Property indicators that summarizes the characteristics of the computation and the processing that has occurred to produce it

RP66V1 name: *PROPERTIES*

Type list(str)

`dimension`

Array structure of a single value

RP66V1 name: *DIMENSION*

Type list(int)

`axis`

Coordinate axes of the values

RP66V1 name: *AXIS*

Type list(*Axis*)

`zones`

Mutually disjoint zones over which the value of the current computation is constant

RP66V1 name: *ZONES*

Type list(*Zone*)

`source`

The immediate source of the Computation

RP66V1 name: *SOURCE*

See also:

`BasicObject` The basic object that Computation is derived from

Notes

The Computation object reflects the logical record type COMPUTATION, defined in rp66. COMPUTATION objects are listed in Appendix A.2 - Logical Record Types, and described in detail in Chapter 5.8.6 - Static and Frame Data, COMPUTATION objects.

`property values`

Computation values

Computation value(s) may be scalar or array's. The size/dimensionallity of each value is defined in the dimensions attribute.

Each value may or may not be zoned, i.e. it is only defined in a certain zone. If this is the case the first zone, computation.zones[0], will correspond to the first value, computation.values[0] and so on. If there is no zones, there should only be one value, which is said to be unzoned, i.e. it is defined everywhere.

RP66V1 name: *VALUES*

Raises `ValueError` – Unable to structure the values based on the information available.

Returns **values**

Return type structured np.ndarray

Notes

If dlisio is unable to structure the values due to insufficient or contradictory information in the object, an ValueError is raised. The raw array can still be accessed through attic, but note that in this case, the semantic meaning of the array is undefined.

Examples

First value:

```
>>> computation.values[0]
[10, 20, 30]
```

Zone (if any) where that parameter value is valid:

```
>>> computation.zones[0]
Zone('ZONE-A')
```

Equipment

`class dlisio.dlis.`**`Equipment`**`(`*BasicObject*`)`

Equipment objects contains information about individual pieces of surface and downhole equipment used in the acquistion of the data. Typically, tools (specified by the Tool object) is a composition of equipment.

`trademark_name`

The producer's name for the equipment

RP66V1 name: *TRADEMARK-NAME*

Type str

`status`

Operational status

RP66V1 name: *STATUS*

Type bool

`generic_type`

Generic type

RP66V1 name: *TYPE*

Type str

`serial_number`

Serial number

RP66V1 name: *SERIAL-NUMBER*

Type str

`location`

General location of equipment during acqusition

RP66V1 name: *LOCATION*

Type str

`height`
Heigth

RP66V1 name: *HEIGHT*

`length`
Length

RP66V1 name: *LENGTH*

`diameter_min`
Minimum diameter

RP66V1 name: *MINIMUM-DIAMETER*

`diameter_max`
Maximum diameter

RP66V1 name: *MAXIMUM-DIAMETER*

`volume`
Volume

RP66V1 name: *VOLUME*

`weight`
Weight

RP66V1 name: *WEIGHT*

`hole_size`
Hole size

RP66V1 name: *HOLE-SIZE*

`pressure`
Pressure

RP66V1 name: *PRESSURE*

`temperature`
Temperature

RP66V1 name: *TEMPERATURE*

`vertical_depth`
Vertical depth

RP66V1 name: *VERTICAL-DEPTH*

`radial_drift`
Radial drift

RP66V1 name: *RADIAL-DRIFT*

`angular_drift`
Angular drift

RP66V1 name: *ANGULAR-DRIFT*

See also:

`BasicObject` The basic object that Equipment is derived from

Notes

The Equipment object reflects the logical record type EQUIPMENT, defined in rp66. EQUIPMENT records are listed in Appendix A.2 - Logical Record Types and described in detail in Chapter 5.8.3 - Static and Frame Data, EQUIPMENT objects.

Fileheader

`class dlisio.dlis.`**`Fileheader`**`(`*BasicObject*`)`

The Fileheader is an identifier for the Logical File. Below follows a description of the relationship between a DLIS-file, Logical File, File Set, and Storeage Set:

DLIS-file - single dlis-file may or may not consists of multiple logical files:

```
 ---------------------------------------
| Logical File 1 | ... | Logical File n |
 ---------------------------------------
```

Logical File - Each Logical File has exactly one Fileheader, but can have mutiple origins:

```
 ---------------------------------------------
| Fileheader | Origin | Frame | Channel | ... |
 ---------------------------------------------
```

File Set - A File set consists of multiple Logical Files which may spand across multiple DLIS-files. Logical Files are grouped into File Sets by producer defined criterias:

```
 ---------------------------------------
| Logical File 1 | ... | Logical File n |
 ---------------------------------------
```

Storage Set - A Storage Set consist of multiple DLIS-files. A Storage Set may or may not coincide with a File Set:

```
 ---------------------------------
| DLIS-file 1 | ... | DLIS-File n |
 ---------------------------------
```

`sequencenr`

Sequential position of the logical file in a storage set

RP66V1 name: *SEQUENCE-NUMBER*

Type str

`id`

Descriptive identification of the logical file

RP66V1 name: *ID*

Type str

See also:

`BasicObject` The basic object that Fileheader is derived from

Notes

The Fileheader object reflects the logical record type FILE-HEADER, defined in rp66. FILE-HEADER records are listed in Appendix A.2 - Logical Record Types and described in Chapter 5.1 - Static and Frame Data, File Header Logical Record (FHLR).

Group

`class dlisio.dlis.Group(`*BasicObject*`)`

Group Objects indicate logical groupings of other Objects. A Group is defined by the producer by any associations deemed useful.

`description`

RP66V1 name: *DESCRIPTION*

Type str

`object_type`

Specifies the type of object that is referenced in the object list attribute.

RP66V1 name: *OBJECT-TYPE*

Type str

`object_list`

References to arbitrary objects.

RP66V1 name: *OBJECT-LIST*

Type list

`group_list`

Reference to other Group objects

RP66V1 name: *GROUP-LIST*

Type list(*Group*)

Notes

The Group object reflects the logical record type Group, defined in rp66. Group records are listed in Appendix A.2 - Logical Record Types and described in detail in Chapter 5.8.8 - Static and Frame Data, Group objects.

Longname

`class dlisio.dlis.Longname(`*BasicObject*`)`

Structured names of other objects.

`modifier`

General modifier

RP66V1 name: *GENERAL-MODIFIER*

Type list(str)

`quantity`
Something that is measureable E.g. the diameter of a pipe

RP66V1 name: *QUANTITY*

Type str

`quantity_mod`
Specialization of a quantity

RP66V1 name: *QUANTITY-MODIFIER*

Type list(str)

`altered_form`
Altered form of the quantity. E.g. standard deviation is an altered form of a temperature quantity.

RP66V1 name: *ALTERED-FORM*

Type str

`entity`
The entity of which the quantity is measured. E.g. entity = borehole, quantity = diameter

RP66V1 name: *ENTITY*

Type str

`entity_mod`
Specialization of an entity

RP66V1 name: *ENTITY-MODIFIER*

Type list(str)

`entity_nr`
Distinguishes multiple instances of the same entity

RP66V1 name: *ENTITY-NUMBER*

Type str

`entity_part`
Part of an entity

RP66V1 name: *ENTITY-PART*

Type str

`entity_part_nr`
Distinguishes multiple instances of the same entity part

RP66V1 name: *ENTITY-PART-NUMBER*

Type str

`generic_source`
The source of the information

RP66V1 name: *GENERIC-SOURCE*

Type str

`source_part`

A specific part of the source information. E.g. "transmitter"

RP66V1 name: *SOURCE-PART*

Type list(str)

`source_part_nr`

Distinguishes multiple instances of the same source part

RP66V1 name: *SOURCE-PART-NUMBER*

Type list(str)

`conditions`

Conditions applicable at the time the information was acquired or generated

RP66V1 name: *CONDITIONS*

Type list(str)

`standard_symbol`

Industry-standardized symbolic name by which the information is known. The possible values are specified by POSC

RP66V1 name: *STANDARD-SYMBOL*

Type str

`private_symbol`

Association between the recorded information and corresponding records or objects of the Producer's internal or corporate database

RP66V1 name: *PRIVATE-SYMBOL*

Type str

See also:

`BasicObject` The basic object that Longname is derived from

Notes

The Longname object reflects the logical record type LONG-NAME, defined in rp66. LONG-NAME objects are listed in Appendix A.2 - Logical Record Types, and described in detail in Chapter 5.4.1 - Static and Frame Data, Long-Name Objects.

Measurement

`class dlisio.dlis.Measurement(`*BasicObject*`)`

Records of measurements, references, and tolerances used to compute calibration coefficients.

`phase`

In what phase of the overall job sequence the measurement as acquired

RP66V1 name: *PHASE*

Type str

`source`

Source the measurement

RP66V1 name: *MEASUREMENT-SOURCE*

`mtype`

Type of measurement

RP66V1 name: *TYPE*

Type str

`dimension`

Structure of the sample array

RP66V1 name: *DIMENSION*

Type list(int)

`axis`

Coordinate axis of the sample array

RP66V1 name: *AXIS*

Type list(*Axis*)

`samplecount`

Number of samples used to compute the max/std_deviation

RP66V1 name: *SAMPLE-COUNT*

Type int

`begin_time`

Time of the sample acquisition. Either an absolute time represented by datetime or elapsed time from file-creation (see `Origin.creation_time`).

RP66V1 name: *BEGIN-TIME*

`duration`

Time duration of the sample acquisition

RP66V1 name: *DURATION*

`standard`

Measurable quantity of the calibration standard used to produce the sample

RP66V1 name: *STANDARD*

See also:

`BasicObject` The basic object that Measurement derived from

Notes

The Measurement object reflects the logical record type CALIBRATION-MEASUREMENT, defined in rp66. CHANNEL records are listed in Appendix A.2 - Logical Record Types and described in detail in Chapter 5.8.7.1 - Static and Frame Data, CALIBRATION-MEASUREMENT objects.

property max_deviation

Maximum deviation

Only applicable when the sample attribute contains mean values. In that case, this is maximum deviation from the mean of any value used to compute the mean.

Each sample may be a scalar of ndarray, but should have the same structure as the samples in the sample attribute.

RP66V1 name: *MAXIMUM-DEVIATION*

property minus_tolerance

The maximum value that any sample (in samples) can fall below the reference and still be 'within tolerance'. Should be all non-negative numbers. If this attribute is empty, the minus tolerance is implicity infinite.

Each sample may be a scalar of ndarray, but should have the same structure as the samples in the sample attribute.

RP66V1 name: *MINUS-TOLERANCE*

property plus_tolerance

The maximum value that any sample (in samples) can exceed the reference and still be 'within tolerance'. Should be all non-negative numbers. If this attribute is empty, the plus tolerance is implicity infinite.

Each sample may be a scalar of ndarray, but should have the same structure as the samples in the sample attribute.

RP66V1 name: *PLUS-TOLERANCE*

property reference

The nominal value of each sample in the samples attribute

Each sample may be a scalar of ndarray, but should have the same structure as the samples in the sample attribute.

RP66V1 name: *REFERENCE*

property samples

Measurment samples

The type of measurment is described by the type attribute. Each sample may be either a scalar or ndarray

RP66V1 name: *MEASUREMENT*

property std_deviation

Standard deviation

Only applicable when the sample attribute contains mean values. In that case, this is the standard deviation of the samples used to compute the mean.

Each sample may be a scalar of ndarray, but should have the same structure as the samples in the sample attribute.

RP66V1 name: *STANDARD-DEVIATION*

Message

class `dlisio.dlis.Message`(*BasicObject*)

Textual messages tied to other data by means of time and/or position of tool zero point when the message was recorded

`message_type`

source and purpose of the message.

RP66V1 name: *TYPE*

Type str

`time`

time the message was issued. Either an absolute time represented by datetime or elapsed time from file-creation (see `Origin.creation_time`).

RP66V1 name: *TIME*

`borehole_drift`

borehole drift of the tool zero point when message was issued.

RP66V1 name: *BOREHOLE-DRIFT*

`vertical_depth`

vertical depth of the tool zero point when message was issued.

RP66V1 name: *VERTICAL-DEPTH*

`radial_drift`

radial drift of the tool zero point when message was issued.

RP66V1 name: *RADIAL-DRIFT*

`angular_drift`

angular drift of the tool zero point when message was issued.

RP66V1 name: *ANGULAR-DRIFT*

`text`

message(s).

RP66V1 name: *TEXT*

Type list(str)

See also:

`BasicObject` The basic object that Message is derived from

Notes

The Message object reflects the logical record type MESSAGE, described in rp66. MESSAGE objects are defined in Appendix A.2 - Logical Record Types, described in detail in Chapter 6.1.1 - Transient Data, message objects.

No Format

class dlisio.dlis.**Noformat**(*BasicObject*)

Noformat objects contain description of unformatted data present in files.

consumer_name

Client-provided name for the data, for example an external file specification

RP66V1 name: *CONSUMER-NAME*

Type str

description

Textual description

RP66V1 name: *DESCRIPTION*

Type str

See also:

BasicObject The basic object that Nofromat is derived from

Notes

The Noformat object reflects the logical record type NO-FORMAT, described in rp66. NO-FORMAT objects are defined in Appendix A.2 - Logical Record Types, described in detail in Chapter 5.10.1 Static and Frame Data, No-Format Objects.

data()

Raw data

A raw byte buffer of all the records corresponding to the noform-object. Content of this data is not defined by rp66v1. However, the name and description of the no-format object might give an indication about its content.

Returns **data** – Raw bytes

Return type bytes

Examples

Look at the description of the no-format object to see if it contains information about nature of the data

```
>>> print(noformat.describe())
---------
No-format
---------
name   : 2
origin : 0
copy   : 0
---
Consumer name : FILE
Description   : /home/files.txt
```

Save the data

```
>>> out = open('noformat.txt', 'wb')
>>> out.write(noformat.data())
>>> out.close()
```

It is not uncommon that the no-format object itself is not enough to understand its content. Some vendors store additional information in other custom object types. In that case some manual digging into the file might reveal the information necessary to correctly understand the no-format content:

```
>>> images = f.find('.*IMAGE.*', matcher=dlisio.dlis.utils.regex_matcher())
>>> for image in images:
>>>     print(image.describe())
---------
my_image
---------
name   : MYIMAGE
origin : 0
copy   : 0
---
Unknown attributes
--
NAME          : MYIMAGE.PNG
NOFORMAT-NAME : dlisio.core.obname(id='MYIMAGE', origin=0, copynum=0)
```

Check for no-format records:

```
>>> for noformat in f.noformats:
>>>     print(noformat.describe())
---------
No-format
---------
name   : NOFORMAT-MYIMAGE
origin : 0
copy   : 0
---
Consumer name : MYIMAGE
Description   : My very important image
```

Store the result:

```
>>> noformat = f.object("NO-FORMAT", "NOFORMAT-MYIMAGE")
>>> out = open('MYIMAGE.PNG', 'wb')
>>> out.write(noformat.data())
>>> out.close()
```

Origin

class dlisio.dlis.**Origin**(*BasicObject*)

Describes the creation of the logical file.

Origin objects is an unique indentifier for a Logical File and it describes the circumstances under which the file was created. The Origin object also spesify the Logical File's relation to a DLIS-file and to which Logical Set it belongs.

A logical file may have several Origin objects, whereas the first Origin object is the Defining object. No two logical files should have identical Defining Origins.

file_id

An exact copy of Fileheader.id

RP66V1 name: *FILE-ID*

Type str

file_set_name

The name of the File Set that the Logical File is a part of

RP66V1 name: *FILE-SET-NAME*

Type str

file_set_nr

The number of the File Set that the Logical File is a part of

RP66V1 name: *FILE-SET-NUMBER*

Type int

file_nr

The file number of the Logical File within a File Set

RP66V1 name: *FILE-NUMBER*

Type int

file_type

A producer spesified File-Type that signifies the content of the DLIS-file

RP66V1 name: *FILE-TYPE*

Type str

product

Name of the software product that produced the DLIS-file

RP66V1 name: *PRODUCT*

Type str

version

The version of the software product that created the DLIS-file

RP66V1 name: *VERSION*

Type str

`programs`

Other programs and services that was a part of the software that created the DLIS-file

RP66V1 name: *PROGRAMS*

Type list(str)

`creation_time`

Date and time at which the DLIS-File was created

RP66V1 name: *CREATION-TIME*

Type datetime

`order_nr`

An unique accounting number assosiated with the creation of the DLIS-File

RP66V1 name: *ORDER-NUMBER*

Type str

`descent_nr`

The meaning of this number must be obtained directly from the producer

RP66V1 name: *DESCENT-NUMBER*

`run_nr`

The meaning of this number must be obtained directly from the company

RP66V1 name: *RUN-NUMBER*

`well_id`

Id of the well at which the measurements where acquired

RP66V1 name: *WELL-ID*

`well_name`

Name of the well at which the measurements where acquired

RP66V1 name: *WELL-NAME*

Type str

`field_name`

The field to which the well belongs

RP66V1 name: *FIELD-NAME*

Type str

`producer_code`

The producer's identifying code

RP66V1 name: *PRODUCER-CODE*

Type int

`producer_name`

The producer's name

RP66V1 name: *PRODUCER-NAME*

Type str

`company`

The name of the client company which the log was produced for

RP66V1 name: *COMPANY*

Type str

`namespace_name`

(DLIS internal) A producer-defined namespace for which the object names for this origin are defined under

RP66V1 name: *NAME-SPACE-NAME*

Type str

`namespace_version`

(DLIS internal) The version of the namespace.

RP66V1 name: *NAME-SPACE-VERSION*

Type int

See also:

`BasicObject` The basic object that Origin is derived from

`Fileheader` Fileheader

Notes

The Origin object reflects the logical record type ORIGIN, defined in rp66. ORIGIN records are listed in Appendix A.2 - Logical Record Types and described in detail in Chapter 5.1 - Static and Frame Data, Origin objects.

Parameter

`class dlisio.dlis.Parameter(`*BasicObject*`)`

Parameter

A parameter object describes a parameter used in the acquisition and processing of data. The parameter value(s) may be scalars or an array. In the later case, the structure of the array is defined in the dimension attribute. The zones attribute specifies which zones the parameter is defined. If there are no zones the parameter is defined everywhere.

The axis attribute, if present, defines axis labels for multidimensional value(s).

`long_name`

Descriptive name of the channel.

RP66V1 name: *LONG-NAME*

Type *Longname*

`dimension`

Dimensions of the parameter values

RP66V1 name: *DIMENSION*

Type list(int)

`axis`

Coordinate axes of the parameter values

RP66V1 name: *AXIS*

Type list(*Axis*)

`zones`

Mutually disjoint intervals where the parameter values is constant

RP66V1 name: *ZONES*

Type list(*Zone*)

See also:

`BasicObject` The basic object that Parameter is derived from

Notes

The Parameter object reflects the logical record type PARAMETER, described in rp66. PARAMETER objects are defined in Appendix A.2 - Logical Record Types, described in detail in Chapter 5.8.2 - Static and Frame Data, PARAMETER objects.

`property values`

Parameter values

Parameter value(s) may be scalar or array's. The size/dimensionallity of each value is defined in the dimensions attribute.

Each value may or may not be zoned, i.e. it is only defined in a certain zone. If this is the case the first zone, parameter.zones[0], will correspond to the first value, parameter.values[0] and so on. If there is no zones, there should only be one value, which is said to be unzoned, i.e. it is defined everywere.

RP66V1 name: *VALUES*

Raises `ValueError` – Unable to structure the values based on the information available.

Returns **values**

Return type structured np.ndarray

Notes

If dlisio is unable to structure the values due to insufficient or contradictory information in the object, an ValueError is raised. The raw array can still be accessed through attic, but note that in this case, the semantic meaning of the array is undefined.

Examples

First value:

```
>>> parameter.values[0]
[10, 20, 30]
```

Zone (if any) where that parameter value is valid:

```
>>> parameter.zones[0]
Zone('ZONE-A')
```

Path

`class dlisio.dlis.Path(`*BasicObject*`)`

Path Objects defines Channels in the Data Frames of a given Frame Type and are combined to define part or all of a Data Path, and variation in the alignment.

`frame_type`

The frame in which the channel's of the current path are recorded.

RP66V1 name: *FRAME-TYPE*

Type *Frame*

`well_reference_point`

Well Reference Point

RP66V1 name: *WELL-REFERENCE-POINT*

Type *Wellref*

`value`

Value Channel for the current Path.

RP66V1 name: *VALUE*

Type list(*Channel*)

`borehole_depth`

Specifies the constant Borehole Depth coordinate for the current Path, It may or may not be described by a channel object.

RP66V1 name: *BOREHOLE-DEPTH*

`vertical_depth`

Specifies the constant Vertical Depth coordinate for the current Path, It may or may not be described by a channel object.

RP66V1 name: *VERTICAL-DEPTH*

`radial_drift`

Specifies the constant Radial Drift coordinate for the current Path, It may or may not be described by a channel object.

RP66V1 name: *RADIAL-DRIFT*

angular_drift

Specifies the constant Angular Drift coordinate for the current Path, It may or may not be described by a channel object.

RP66V1 name: *ANGULAR-DRIFT*

time

Specifies the constant Time coordinate for the current Path, It may or may not be described by a channel object. Either an absolute time represented by datetime or elapsed time from file-creation (see `Origin.creation_time`).

RP66V1 name: *TIME*

depth_offset

Specifies the Depth Offset, which indicates how much the *value* is "off depth".

RP66V1 name: *DEPTH-OFFSET*

measure_point_offset

Specifies a Measure Point Offset, which indicates a fixed distance along Borehole Depth from the Value Channel's Measure Point to a Data Reference Point.

RP66V1 name: *MEASURE-POINT-OFFSET*

tool_zero_offset

Distance of the Data Reference Point for the current Path above the tool string's Tool Zero Point.

RP66V1 name: *TOOL-ZERO-OFFSET*

See also:

`BasicObject` The basic object that Parameter is derived from

Notes

The Path object reflects the logical record type PATH, defined in rp66. PATH records are listed in Appendix A.2 - Logical Record Types and described in detail in Chapter 5.7.2 - Static and Frame Data, PATH objects.

Process

class dlisio.dlis.**Process**(*BasicObject*)

Process objects describes a specific process or computation applied to input objects to get output objects.

description

RP66V1 name: *DESCRIPTION*

Type str

trademark_name

Trademark name refers to the process and its products.

RP66V1 name: *TRADEMARK-NAME*

Type str

`version`

Software version.

RP66V1 name: *VERSION*

Type str

`properties`

Properties that applies to the output of the process, as a result of the process.

RP66V1 name: *PROPERTIES*

Type list(str)

`status`

Indicated the status of the process. It's typically updated to indicate when the process is completed or aborted.

RP66V1 name: *STATUS*

Type str

`input_channels`

Channels that are used directly by this Process.

RP66V1 name: *INPUT-CHANNELS*

Type list(*Channel*)

`output_channels`

Channels that are produced directly by this Process.

RP66V1 name: *OUTPUT-CHANNELS*

Type list(*Channel*)

`input_computations`

Computations that are used directly by this Process.

RP66V1 name: *INPUT-COMPUTATIONS*

Type list(*Computation*)

`output_computations`

Computations that are produced directly by this Process.

RP66V1 name: *OUTPUT-COMPUTATIONS*

Type list(*Computation*)

`parameters`

Parameters that are used by the Process or that directly affect the operation of the Process.

RP66V1 name: *PARAMETERS*

Type list(*Parameter*)

`comments`

Comments contains information specific to the particular execution of the process.

RP66V1 name: *COMMENTS*

Type list(str)

See also:

`BasicObject` The basic object that Parameter is derived from

Notes

The Process object reflects the logical record type Process, defined in rp66. PROCESS records are listed in Appendix A.2 - Logical Record Types and described in detail in Chapter 5.8.5 - Static and Frame Data, Process objects.

Splice

class dlisio.dlis.**Splice**(*BasicObject*)

Splice describes the process of concatinating multiple channels into one. The concatination is defined by the zone objects, where the first zone object corresponds to the first input channel and so on. The zones must be mutually disjoint but the ordering is arbitrary:

```
Input Ch 1    input Ch 2   -> Output
------------------------------------------------
    |            |             |
    |           |              |         Zone1
    |          |               |
------------------------------------------------
    |         |               None
------------------------------------------------
    |           |               |
    |            |               |       Zone2
    |             |               |
------------------------------------------------
```

output_channel

Concatenation of all input channels

RP66V1 name: *OUTPUT-CHANNELS*

Type *Channel*

input_channels

Channels that where used to create the output channel

RP66V1 name: *INPUT-CHANNELS*

Type list(*Channel*)

zones

Zones of each input channel that is used in the concatination process

RP66V1 name: *ZONES*

Type list(*Zone*)

See also:

`BasicObject` The basic object that Splice is derived from

Notes

The Splice object reflects the logical record type SPLICE, defined in rp66. SPLICE records are listed in Appendix A.2 - Logical Record Types and described in detail in Chapter 5.8.9 - Static and Frame Data, SPLICE objects.

Tool

`class dlisio.dlis.Tool(BasicObject)`

A tool is an ensembly of equiptment that as a whole provide measurements or services. The list of equiptment that makes up the tool can be found in *tool.parts*. Tools objects also keep a list of all channels that where produced by the tool in *tool.channels*.

`description`

Textual description of the tool

RP66V1 name: *DESCRIPTION*

Type str

`trademark_name`

The producer's name for the tool

RP66V1 name: *TRADEMARK-NAME*

Type str

`generic_name`

The name generally used by the industry to describe such a tool

RP66V1 name: *GENERIC-NAME*

Type str

`status`

If the tool is enabled to provide information to the acquisition system

RP66V1 name: *STATUS*

Type bool

`parts`

Equipments that makes up the tool

RP66V1 name: *PARTS*

Type list(*Equipment*)

`channels`

Channels that are produced by this tool

RP66V1 name: *CHANNELS*

Type list(*Channel*)

`parameters`

Parameters that directly affect or reflect the operation of the tool

RP66V1 name: *PARAMETERS*

Type list(*Parameter*)

See also:

`BasicObject` The basic object that Tool is derived from

Notes

The tool object reflects the logical record type TOOL defined in rp66. TOOL objects are listed in Appendix A.2 - Logical Record Types, described in detail in Chapter 5.8.4 - Static and Frame Data, TOOL objects.

Wellref

class dlisio.dlis.**Wellref**(*BasicObject*)

Well reference defines origin of well with coordinates.

permanent_datum

Level from where vertical distance is measured

RP66V1 name: *PERMANENT-DATUM*

Type str

vertical_zero

Vertical zero is an entity that corresponds to zero depth.

RP66V1 name: *VERTICAL-ZERO*

Type str

permanent_datum_elevation

Permanent datum, structure or entity from which the vertical distance can be measured.

RP66V1 name: *PERMANENT-DATUM-ELEVATION*

above_permanent_datum

Distance of permanent Datum above mean sea level. Negative values indicates that the Permanent datum is below mean sea level

RP66V1 name: *ABOVE-PERMANENT-DATUM*

magnetic_declination

Angle between the line of direction to geographic north and the line of direction to magnetic north. This defines angle with vertex at well reference point.

RP66V1 name: *MAGNETIC-DECLINATION*

coordinate

Independent spatial coordinates. Typically, latitude, longitude and elevation

RP66V1 names: *COORDINATE-1-NAME*, *COORDINATE-1-VALUE*, *COORDINATE-2-NAME*, *COORDINATE-2-VALUE*, *COORDINATE-3-NAME*, *COORDINATE-3-VALUE*

Type dict

See also:

`BasicObject` The basic object that Wellref is derived from

Notes

The Well Reference object reflects the well reference point of a well, defined in rp66. Well reference records are listed in Appendix A.2 - Logical Record Types are described in detail in Chapter 5.2.2 - Static and Frame Data, Well reference objects.

Zone

`class dlisio.dlis.Zone(BasicObject)`

A Zone objects specifies a single interval in depth or time. Other objects use zones to define spesific regions in wells or time intervals where the object data is valid.

`description`

Description of the zone

RP66V1 name: *DESCRIPTION*

Type str

`domain`

Type of interval, e.g. borhole-depth, time or vertical-depth

RP66V1 name: *DOMAIN*

Type str

`maximum`

Latest time or deepest point, not inclusive

RP66V1 name: *MAXIMUM*

`minimum`

Earliest time or shallowest point, inclusive

RP66V1 name: *MINIMUM*

See also:

`BasicObject` The basic object that Zone is derived from

Notes

The Zone object reflects the logical record type ZONE, defined in rp66. ZONE objects are listed in Appendix A.2 - Logical Record Types, and described in detail in Chapter 5.8.1 - Static and Frame Data, Zone Objects.

Unknown

`class dlisio.dlis.Unknown(BasicObject)`

The unknown object is intended as a fall-back object if the object-type is not recognized by dlisio, e.g. vendor spesific object types

See also:

`BasicObject` The basic object that Unknown is derived from

2.7 Common errors

If you received an error while working with dlis file, you might find explanation for it in this section.

Note: Refer to *dlisio.common.ErrorHandler* for possible remedies.

2.7.1 Errors on load

Although there are still some features of the DLIS spec that dlisio doesn't implement full support for, it should be able to load all dlis files that conform to the spec.

From experience we know that there exists a lot of files out there that don't strictly adhere to the spec, and these might cause load to fail. It's rather common that files are truncated, the file is zeroed out from some point or that there are some extra "trash" bytes somewhere in the file. All of which can throw dlisio parsing routines off. Hence described errors are mostly manifestations of the same set of issues. The exact error text is dependent on where in the file the issue occurs.

Load errors are in general non-recoverable and prevent user from further reading the file. However they can be bypassed by using *dlisio.common.ErrorHandler* which helps to read the file up to the point of failure.

could not find visible record envelope pattern

```
searched 200 bytes, but could not find visible record envelope pattern
`0xFF 0x01`
```

This error might indicate:

- The file is not actually a .dlis file
- The file contains trash at the start or at the end of the file

incorrect format version

```
rp66: Incorrect format version in Visible Record 1234
```

This error might indicate:

- The file is completely zeroed-out from certain point
- The file contains a zeroed-out chunk of data
- The file is missing a chunk of data in the middle
- The file is a non-standard TIF file

too short logical record

```
Too short logical record. Length can't be less than 4, but was 0
```

This error might indicate:

- The file is completely zeroed-out from certain point
- The file contains a zeroed-out chunk of data

unexpected EOF when reading / file truncated

```
rp66: unexpected EOF when reading record - got 1234 bytes, expected
there to be 5678 more
```

```
File truncated ...
```

This error might indicate:

- The file is incomplete, unknown amount of information is missing from the end

logical record segment expects successor

```
End of logical file, but last logical record segment expects successor
```

```
Reached EOF, but last logical record segment expects successor
```

This error might indicate:

- Logical file is incomplete, end of file or new logical file follows.

2.7.2 Parsing errors

Some errors which occur during parsing of objects or curves.

error parsing / unexpected end-of-record

```
error parsing object descriptor
```

```
unexpected end-of-record
```

```
unexpected end-of-record in template
```

These (and similar) errors indicate that the file breaks specification in a way that is too ambiguous. When encountered, dlisio no longer knows how to interpret data that follows, so it stops processing data further.

Some common situations that might lead to this error:

- Incorrect invariant attribute in template
- Invalid LRS padbytes value
- Fileheader set is declared to be named, but is not

field occurs more than once

```
field 'DEPTH.1.0' occurs more than once
```

According to specification, every channel may appear in the frame once and only once. So this error may indicate:

- Channel is repeated for user convenience. It's likely the case if repeated channel can be assumed to be index of the frame, like for example "DEPTH"
- Something is broken in frame-channel relationships and you should be cautious with trusting the data

To bypass this error call:

```
frame.curves(strict=False)
```

fmtstr would read past end

```
corrupted record: fmtstr would read past end
```

There is less data than what the metadata claims there to be. Either the metadata is incorrect or the file is missing data, or both. It's not obvious which one it is. The inconsistency between the metadata and the raw data makes it impossible for dlisio to correctly parse the data. Hence there is no recovery from this.

Possible reasons:

- Invalid LRS padbytes value

day is out of range/month must be

```
day is out of range for month
```

```
month must be in 1..12
```

Standard Python errors on creation of invalid date. Error would mean that date stored in the file is broken, like (1990, 0, 0, 0, 0, 0, 0).

2.7.3 Warnings

unable to decode

```
UnicodeWarning: unable to decode string
```

Data in the file wasn't written in UTF-8 as it should have been. dlisio doesn't know which encoding was used, so it doesn't know how to represent this data correctly. You might experiment with *dlisio.common.set_encodings()*.

Hint: If unsure, try "latin1". It's likely to present you with relevant result.

2.8 Organization codes

Organizations are assigned their own organization codes by energistics. These organization codes typically pop up in the metadata of dlis files, e.g. to identify the producer.

The rp66v1 standard allows vendors to specify their own metadata objects. It also specifies that the type of such vendor-specific objects should always start with the organization code like so:

```
>>> f.unknowns
dict_keys(['440-FILE', '440-OP-TOOL', '440-CHANNEL'])
```

From he naming we can see that these objects are defined by Schlumberger. The semantic meaning of such objects may only be known to the producer.

Name (Code)	Description (Organization Name)
0	Subcommittee On Recommended Format For Digital Well Data, Basic Schema
1	Operator
2	Driller
3	Mud Logger
4	Abyssus Marine Services AS
6	ALT – Advanced Logic Technology (added 2014-09-18)
9	Amerada Hess
10	Analysts, The
12	ArenaPetro
15	Baker Hughes Inteq
20	Baroid
30	Birdwell
40	Reeves (1 Jan 99; formerly BPB)
50	Brett Exploration
58	Canrig (added 2009-09-09)
60	Cardinal
65	Center Line Data
66	Subcommittee On Recommended Format For Digital Well Data, DLIS Schema
70	Century Geophysical
77	CGG Logging, Massey France
80	Charlene Well Surveying
85	China Oilfield Services Limited (COSL) (added 2019-02-11)
90	Compagnie de Services Numerique
95	Comprobe
100	Computer Data Processors
110	Computrex
115	COPGO Wood Group
120	Core Laboratories
125	CRC Wireline, Inc.
126	Crocker Data Processing Pty Ltd
127	Tucker Wireline Services (formerly Davis Great Guns Logging, Wichita, KS)
128	Datalog Technology (added 2009-09-09)
130	Digigraph
137	Tucker Technologies (formerly Digital Logging Inc.), Tulsa, OK.
140	Digitech
145	Deines Perforating

continues on next page

Table 2 – continued from previous page

Name (Code)	Description (Organization Name)
148	Drillog Petro-Dynamics Limited
150	Baker Atlas (formerly Dresser Atlas)
155	Dynamic Technologies (DTCC)
160	Earthworm Drilling
170	Electronic Logging Company
180	Elgen
190	El Toro
200	Empire
205	Encom Technology, Ltd.
206	Ensigh Geophysics, Ltd.
208	Epoch (merged with Canrig, Nov 2008; added 2009-09-09)
210	Frontier
213	GeoEnergy, Inc.
214	Geokinetics Inc.
215	Geolog
216	Geophysical Data Systems (added 2015-01-13)
217	Geoshare
218	GEO·X Systems Ltd.
220	G O International
225	GOWell Petroleum (added 2012-04-02)
230	Gravilog
240	Great Guns Servicing
250	Great Lakes Petroleum Services
260	GTS
268	Guardian Data Seismic Pty. Ltd.
270	Guns
280	Halliburton Logging
283	Harvey Rock Physics (added 2015-01-13)
285	Horizon Production Logging
290	Husky
293	INOVA Geophysical Equipment Limited (updated 2012-04-02)
295	Input/Output, Inc.
297	iO Data AS
300	Jetwell
305	Landmark Graphics
310	Lane Wells
315	Logicom Computer Services (UK) Ltd
320	Magnolia
330	McCullough Tool
332	Mitchell Energy Corporation
333	MST Ltd. (Modern Seismic Technologies) – added 2014-08-15)
335	Paradigm Geophysical (formerly Mincom Pty Ltd)
337	DPTS Limited (formerly MR-DPTS Limited, changed as of 2008-08-12)
338	NRI On-Line Inc
339	Oilware, Inc.
340	Pan Geo Atlas
342	Pathfinder Energy Services
345	Perfco
350	Perfojet Services

continues on next page

Table 2 – continued from previous page

Name (Code)	Description (Organization Name)
360	Perforating Guns of Canada
361	Petcom, Inc.
362	CGG (FKA Petroleum Exploration Computer Consultants, Ltd).
363	Petrologic Limited
364	PetroMar Technologies
366	Phillips Petroleum Company
367	Phoenixdata Services Pty Ltd.
368	Petroleum Geo-Services (PGS)
370	Petroleum Information
380	Petrophysics
390	Pioneer
392	The Practical Well Log Standards Group
395	IHS Energy Log Services (formerly Q. C. Data Collectors)
400	Ram Guns
410	Riley's Datashare
418	RODE
420	Roke
430	Sand Surveys
440	Schlumberger
450	Scientific Software
455	Seismic Instruments, Inc.
460	Seismograph Service
462	SEGDEF
463	SEG Technical Standards High Density Media Format Subcommittee
464	Shell Services Company
465	Stratigraphic Systems, Inc.
466	Spectrum ASA
467	Sperry-Sun Drilling Services
468	SEPTCO
469	Sercel, Inc.
470	Triangle
471	Thrubit Logging(added 2009-09-09)
472	TGS
475	Troika International
480	Welex
490	Well Reconnaissance
495	Wellsite Information Transfer Specification (WITS)
500	Well Surveys
510	Western
520	Westronics
525	Winters Wireline
530	Wireline Electronics
540	Worth Well
560	Z & S Consultants Limited
999	Reserved for local schemas
1000	Energistics (formerly POSC, changed as of 2006-11-06)

2.9 LIS Specification

The Log Information Standard (LIS)[1] is a binary file format for well logs, developed by Schlumberger in 1974. The 1974 version is commonly referred to as LIS79 or the 79 Subset. An extension to the LIS format was introduced by Schlumberger in 1984. This version is commonly referred to as LIS84 or Enhanced LIS. LIS is the predecessor to the *DLIS Specification*.

When developing LIS standard, the main emphasis was put on easy writing. The files are structured in such a way that data can be written directly while acquiring the logs. This is very handy for the producers of the files, and equally tedious for the consumer that wants to read them later on.

LIS was designed to work with the physical medium tape. Physical tapes have a lot of limitations that modern computers do not have, such as a fixed storage size. LIS implements several layers of file segmentation mechanisms to effectively handle this. Some LIS files also embed access mechanics that let tape-readers effectively seek the tape. This is all generally uninteresting information for a modern computer, but as these mechanics are embedded in the files, modern software still has to account for it. This adds additional layers of complexity.

2.10 LIS User Guide

First things first, import the LIS submodule:

```
>>> from dlisio import lis
```

The main entry point for the LIS reader is `dlisio.lis.load()`, which is designed to work with python's `with`-statement:

```
>>> with lis.load('myfile.lis') as files:
...     pass
```

You can also use `load` without the `with`-statement, but then the exercise of closing the filehandle is left to you:

```
>>> files = lis.load('myfile.lis')
>>> # Work with your file, then close when done
>>> files.close()
```

`load` returns an instance of `dlisio.lis.PhysicalFile` - a tuple-like object containing all the Logical Files (LF) in `myfile.lis`.

Note: If you are unfamiliar with Logical Files and the internal structure of a LIS file, please refer to `dlisio.lis.LogicalFile` and `dlisio.lis.PhysicalFile`.

Lets have a closer look at one of the Logical Files returned by load:

```
>>> f, *tail = files
```

The Logical File, `f`, is an instance of `dlisio.lis.LogicalFile` which is the main interface for interacting with Logical Files.

A Logical File contains a File Header Logical Header (FTLR), and optionally a File Trailer Logical Record (FTLR). These contain general information specific to *this* LF such as the file name and date of generation:

[1] LIS79, http://w3.energistics.org/LIS/lis-79.pdf

```
>>> header = f.header()
>>> trailer = f.trailer()
```

While the FHLR and FTLR are specific to one Logical File, the Reel Header/Trailer (RHLR/RTLR) and Tape Header/Trailer (THLR/TTLR) contain general information that applies to the reel and tape, respectively. These records can also be reached directly from the Logical File:

```
>>> header = f.reel.header()
>>> trailer = f.reel.trailer()
```

For a full overview of the content off all these records, please refer to *LIS Logical Records*.

2.10.1 Working with the curves

Getting the overview

Within a LIS Logical File, curves are defined and organized by Data Format Specification Records (DFSR). A DFSR defines a set of channels/curves that are sampled along the same index. There might be multiple Data Format Specification Records in a Logical File, each defining different channels/curves and index.

Note: LIS79 opens for the presence of duplicated DFSR, for redundancy. Currently, dlisio has no support for identifying redundant DFSRs, and `dlisio.lis.curves()` will return empty arrays for these.

The `dlisio.lis.DataFormatSpec` can be accessed directly from the `dlisio.lis.LogicalFile`:

```
>>> formatspecs = f.data_format_specs()
```

A DFSR contains information about the logset in general, such as logging direction and information about the index. In addition it contains a list of Spec Blocks, `dlisio.lis.DataFormatSpec.specs`. Each Spec Block describes one of the channels/curves in the logset. But let's begin by looking at the index:

```
>>> format_spec = formatspecs[0]
>>> format_spec.index_mnem
'DEPT'
>>> format_spec.index_units
'.1IN'
>>> format_spec.spacing
60
>>> format_spec.spacing_units
'.1IN'
>>> format_spec.direction
255
```

This tells us that the current logset is indexed against depth, and that the depth is measured in 1/10 of an inch. There is also a constant spacing of 60 .1IN (or 6 IN) between samples. Note that there is no general requirement that the index is evenly spaced. In that case `spacing` and `spacing_units` might not be recorded. Lastly we see that the measurements are taken going down-hole (refer to `dlisio.lis.DataFormatSpec.direction`).

Now lets look at the individual channels/curves in the logset. We can list the mnemonics and the units of measurement by simply looping the Spec Blocks:

```
>>> for spec in format_spec.specs:
...     print("Index: {} (units={})".format(spec.mnem, spec.units))
"Index: 'DEPT', units: '.1IN')"
"Index: 'TIME', units: 'ms  ')"
"Index: 'CHX ', units: 'M   ')"
"Index: 'RCCL', units: '    ')"
```

Note: Please refer to *dlisio.lis.DataFormatSpec* and *dlisio.core.spec_block_0* / *dlisio.core.spec_block_1* for the complete documentation of DFSR and Spec Blocks, respectively.

Reading the curves

The *dlisio.lis.DataFormatSpec* only contains the metadata about the logset. To read the actual curves, pass the DFSR format_spec to *dlisio.lis.curves()* together with the filehandle f:

```
>>> curves = lis.curves(f, format_spec)
```

This returns a structured numpy.ndarray containing all the curves in the logset. The array can be indexed by the mnemonics from the Spec Blocks.

Fast Channels

There is one caveat to reading the curves from a logset that is necessary to be aware of. LIS supports a concept it refers to as 'Fast Channels'. In short this means that channels within the *same* logset can have different sampling rates.

When a logset contains curves with different sampling rates, these have to be read separately. I.e. multiple calls to the *dlisio.lis.curves()* are necessary in order to read all the curves. The sampling rate of a channel is recorded in its Spec Block:

```
>>> ch = format_spec.specs[-1]
>>> ch.mnem
'RCCL'
>>> ch.samples
6
```

The last channel in this Data Format Spec, RCCL, has a sampling rate of 6, which means it's sampled 6 times as frequent as the index channel. Remember, that the index DEPT was sampled once every 6th inch. That implicitly means that RCCL is sampled once every inch.

You can also get a set of all sample rates in the DataFormatSpec:

```
>>> format_spec.sample_rates()
{1, 6, 30}
```

There are 3 different sampling rates. Only channels with the same sample rate can be read in one go. To read all the curves several calls to *dlisio.lis.curves()* are necessary:

```
>>> data_01 = lis.curves(f, format_spec, sample_rate=1)
>>> data_06 = lis.curves(f, format_spec, sample_rate=6)
>>> data_30 = lis.curves(f, format_spec, sample_rate=30)
```

Note: Please refer to *dlisio.lis.curves()* for a more comprehensive explanation of this topic.

Note: When reading channels with a higher sampling rate, the index is still included in the resulting numpy array, and the missing values are linearly interpolated between the samples in the file. This behaviour is defined by LIS79 itself.

Associate curves and metadata

dlisio.lis.curves() has a sister function *dlisio.lis.curves_metadata()*. This function can be useful if you want to associate the curves and metadata (Spec Blocks):

```
>>> data = lis.curves(f, format_spec, sample_rate=6)
>>> meta = lis.curves_metadata(format_spec, sample_rate=6)
```

When passing the same DFSR and `sample_rate` to *dlisio.lis.curves_metadata()* and *dlisio.lis.curves()* you get a dict with the corresponding Spec Blocks. Like the numpy array returned by `curves()` this dict uses the mnemonics from the Spec Blocks as keys. This makes it easy to get both the curve itself and its metadata:

```
>>> data['RCCL']
array([-0.00488281, -0.00488281, -0.00488281, ...,  0.01904297,
    0.01806641,  0.01806641], dtype=float32)

>>> meta['RCCL']
dlisio.core.spec_block0('mnemonic=RCCL')
```

A complete example

Putting this all together to read *all* the curves- and metadata from the Logical File, `f`, we get:

```
>>> for format_spec in f.data_format_specs():
...     for sample_rate in format_spec.sample_rates():
...         data = lis.curves(f, format_spec, sample_rate)
...         meta = lis.curves_metadata(format_spec, sample_rate)
...         # Do something fun with it
```

2.10.2 Reading parameters and other metadata from Information Records

The records *Job Identification*, *Tool String Info* and *Wellsite Data* may contain useful information such as company- and well-name, or parameters of some kind. These records are structured identically, and in dlisio they share the common interface of *dlisio.lis.InformationRecord*.

Note: The LIS79 specification does not have a precise definition of when to use which record type. Hence it's up to the producers to decide which of the above record types to use.

The records can be accessed through their own methods on *dlisio.lis.LogicalFile*. E.g. all the Wellsite Data Records can be read with *dlisio.lis.LogicalFile.wellsite_data()*:

```
>>> records = f.wellsite_data()
>>> print(records)
[InformationRecord(type=lis_rectype.wellsite_data, ltell=62)]
```

This particular file only contains a single Wellsite Data Record, but it's not uncommon that a file contains multiple Information Records of the same type. Let's have a closer look at its content:

```
>>> inforec = records[0]
>>> inforec.components()
[dlisio.core.component_block(mnem='TYPE', units='    ', component='CONS'),
 dlisio.core.component_block(mnem='MNEM', units='    ', component='WN  '),
 dlisio.core.component_block(mnem='STAT', units='    ', component='ALLO'),
 dlisio.core.component_block(mnem='PUNI', units='    ', component='    '),
 dlisio.core.component_block(mnem='TUNI', units='    ', component='    '),
 dlisio.core.component_block(mnem='VALU', units='    ', component='15/9-F-15 '),
 dlisio.core.component_block(mnem='MNEM', units='    ', component='CN  '),
 dlisio.core.component_block(mnem='STAT', units='    ', component='ALLO'),
 dlisio.core.component_block(mnem='PUNI', units='    ', component='    '),
 dlisio.core.component_block(mnem='TUNI', units='    ', component='    '),
 dlisio.core.component_block(mnem='VALU', units='    ', component='StatoilHydro')]
```

As you can see, Information Records are essentially a list of *dlisio.core.component_block*. Each Component Block contains a specific piece of information, e.g. a parameter.

Tables in Information Records

Some Information Records are intended to be read as tables. I.e. each Component Block is an entry in a table. This can be checked by calling *dlisio.lis.InformationRecord.isstructured()*:

```
>>> inforec.isstructured()
True
```

If so, dlisio can create the table for you, in form of a Numpy Structured Array. The mnemonics from the Component Blocks are used as field names (column names):

```
>>> inforec.table(simple=True)
[('WN  ', 'ALLO', '    ', '    ', '15/9-F-15 '),
 ('CN  ', 'ALLO', '    ', '    ', 'StatoilHydro')]
```

Setting the argument `simple=True` means that only the values from the Component Blocks are used in the table. Setting it to `False`, which is the default behavior, means that the entire Component Blocks are put into the array. This is useful if you e.g. want to preserve other parts of the Component Blocks, such as units.

Note: The data returned from `table()` is the same data as returned from `components()`, it's just formatted differently.

2.11 LIS API Reference

2.11.1 Load a LIS-file

`dlisio.lis.load`(*path*, *error_handler=None*)

Loads and indexes a LIS file

Load does more than just opening the file. A LIS file has no random access in itself, so load scans the entire file and creates its own index to enumlate random access. The file is also segmented into Logical Files, see *`PhysicalFile`* and *`LogicalFile`*.

** Incorrectly written files **

It is not uncommon that LIS files are written incorrectly, meaning they are violating the LIS79 specification. Typically it's only a very small portion of the file that is incorrect, often down to a couple of bytes. Unfortunately, because of how the internal structure of LIS files is defined, the bytes following the incorrect part become unreadable too. In most cases there is little dlisio can do about this as it's unclear what the original intention was.

However, it may be possible to read the file up until the incorrect part occurs. load has an escape hatch for this, which essentially returns everything that it believes to be successfully indexed prior to the point-of-failure. The caveat being that there is no guarantee that the file is interpreted correctly by dlisio at this point. This escape hatch is controlled by the parameter `error_handler`. Please refer to the examples section for more details of it's use.

Notes

It's not uncommon that LIS files are stored with different file extensions than *.LIS*. For example *.LTI* or *.TIF*. Load does not care about file extension at all. As long as the content adheres to the Log Interchange Standard, load will read it as such.

Parameters

- **path** (*str_like*) – path to lis-file
- **error_handler** (`dlisio.common.ErrorHandler`, *optional*) – Defines how load will behave when encountering any errors while indexing the file.

Returns lis

Return type *dlisio.lis.PhysicalFile*

Examples

Opening a file is straightforward. *`dlisio.lis.load()`* is designed to work like python's own `open()`. That is, it can be used both with- and without python's `` ` ``with-statments:

```
>>> from dlisio import lis
>>> with lis.load(filepath) as files:
...     pass
```

If `load` raises a RuntimeError this is likely due to the file being incorrectly written. You can instruct dlisio to return to you what it did manage to index before it failed:

```
>>> from dlisio.common import ErrorHandler, Actions
>>> handler = ErrorHandler(critical=Actions.LOG_ERROR)
>>> with lis.load(filepath, handler) as files:
...     pass
```

What this effectively does is to turn the `RuntimeError` into a `logging.error` and load now returns a partially indexed file. dlisio does not guarantee that what's being returned is correct at this point, and you should verify for yourself that the data it serves you looks sane.

2.11.2 LIS Physical File

class dlisio.lis.**PhysicalFile**

Physical File - A regular file on disk

Think of a LIS file as a directory. The top directory is your regular file on disk. The regular file is divided into sub-structures (or subfolders if you will) called 'Reels'. Each Reel if further divided into 'Tapes', which again contains Logical Files:

```
your_file.lis
|
|-> Reel 0
|    |
|    |-> Tape 0
|    |    |
|    |    |-> Logical File 0
|    |    |-> Logical File 1
|    |
|    |-> Tape 1
|    |    |
|    |    |-> Logical File 2
|
|-> Reel 1
     |
     |-> Tape 0
          |
          |-> Logical File 3
```

Each Logical File can be thought of as an independent regular file, containing log data.

Each Reel and Tape has its own corresponding Header and Trailer. These contain general information about the Reel or Tape.

When reading the LIS file with *dlisio.lis.load()*, dlisio will flatten the above tree structure and simply return a tuple-like object (*PhysicalFile*) of all the Logical Files. The Reel and Tape in which a Logical File belongs to can be queried directly from the Logical File.

Notes

More often than not, the tree structure of a LIS file is trivial, containing just a couple of logical files - all belonging to the same Tape and Reel.

Notes

The LIS79 specification opens for the possibility of a Reel spanning across multiples regular files. dlisio does not currently have cross-file support. Each regular file can still be read as a stand-alone LIS-file.

See also:

`dlisio.lis.load` Open and Index a LIS-file

`dlisio.lis.LogicalFile` A wrapper for a single Logical File

2.11.3 LIS Logical File

class `dlisio.lis.`**`LogicalFile`**

Logical File (LF)

This class is the main interface for working with a single LF. A LF is essentially a series of Logical Records (LR). There are many different LR types, each designed to carry a specific piece of information. For example a Logical File Header Record contains static information about the file, while the Data Format Specification Records contain information about curve-data, and how that should be parsed.

This class provides an interface for easy interaction and extraction of the various Logical Records within the Logical File. It is completely independent of other LF's, and even has its own IO-device. It stores a pre-built index of all LR's for random access when reading from disk.

Notes

No parsed records are cached by this class. Thus it's advisable that the result of each record read is cached locally.

`path`

Path to the file as passed to *`dlisio.lis.load()`*

Type str

`io`

The underlying lis-aware IO-device that acts on the file. The iodevice implements primitive io-operations such as seek, read and tell, but also higher level abstractions such as read_record. For normal workflows it should not be necessary to interact directly with the io-device.

Type dlisio.core.lis_stream

`index`

A dlisio-created index of all Logical Records (LR) in the current Logical File (LF). The index is created at load and gives dlisio random access to the LR's. The index can be iterated and records can be extracted using *`dlisio.lis.LogicalFile.io`*. For normal workflow it should not be necessary to interact directly with the index.

Type dlisio.core.lis_record_index

`reel`

The reel that this Logical File (LF) belongs to.

See `dlisio.lis.PhysicalFile` for more on the relationship between LIS reels and LF's. See `dlisio.core.reel_header` and `dlisio.core.reel_trailer` for more on the Reel Logical Records (RHLR, RTLR).

Type *dlisio.lis.HeaderTrailer*

`tape`

The tape that this Logical File (LF) belongs to.

See `dlisio.lis.PhysicalFile` for more on the relationship between LIS tapes and LF's. See `dlisio.core.tape_header` and `dlisio.core.tape_trailer` for more on the Tape Logical Records (THLR, TTLR).

Type *dlisio.lis.HeaderTrailer*

`close()`

Close the file handle

It is not necessary to call this method if you're using the *with* statement, which will close the file for you.

`data_format_specs()`

Data Format Specification Records (DFSR)

A DFSR contains all relevant information to extract a specific logset - a logset being a set of channels/curves all sampled along a common index.

See also:

`dlisio.lis.curves` Read all the curves from a given DFSR

Returns **records**

Return type list of dlisio.lis.DataFormatSpec

`flic_comment()`

Comment Record

Returns **records**

Return type list of `dlisio.core.text_record`

`header()`

Logical File Header

Reads and parses the Logical File Header from disk - if present.

Returns **header**

Return type *dlisio.core.file_header* or None

`job_identification()`

Job Identification Logical Records

Returns **records**

Return type list of `dlisio.lis.InformationRecord` `operator_command_inputs()`

Operator Command Inputs

Returns **records**

Return type list of *`dlisio.core.text_record`*

`operator_response_inputs()`

Operator Response Inputs

Returns **records**

Return type list of *`dlisio.core.text_record`*

`parse_records(`*rectype*`)`

Parse Explicit records of given type

`system_outputs_to_operator()`

System Outputs to Operator

Returns **records**

Return type list of *`dlisio.core.text_record`*

`tool_string_info()`

Tool String Info Logical Records

Returns **records**

Return type list of *`dlisio.lis.InformationRecord`*

`trailer()`

Logical File Trailer

Reads and parses the Logical File Header from disk - if present.

Returns **trailer**

Return type *dlisio.core.file_trailer* or None

`wellsite_data()`

Wellsite Data Logical Records

Returns **records**

Return type list of *`dlisio.lis.InformationRecord`*

2.11.4 LIS Curves

`dlisio.lis.curves()`

Read curves

Read the curves described by the *Data Format Specification* Record (DFSR). The curves are read into a Numpy Structured Array [1]. The mnemonics - as described by the DFSR - of the channels are used as column names.

Fast Channels

Some log sets contain curves that are sampled at unequal sampling rate. More specifically, curves can be sampled at a greater frequency than the recorded index. These are referred to in the LIS79 specification as Fast Channels. The sample rate is not absolute, but rather a factor relative to the index. E.g. a sampling rate of 1 means that the curve are sampled at the same frequency as the index, while a sample rate of 6 means the channel is sampled at 6 times the frequency of the index.

When the sampling rate of a curve is higher than the index, the index is linearly interpolated to create the missing values samples. This is the LIS79 defined behavior [2]. Note that this is *only* true for the index. Other channels will not be re-sampled.

These LIS79 mechanics of multiple sampling rates within a logset do mean that the result of the entire logset is a sparse array. That is, curves at lower sampling rates will have a lot of undefined samples. Take this example with 2 channels, `CH01` and `CH02`, where `CH01` has a sample rate of 1, and `CH02` has a sample rate of 3. The full log set would then look like this:

```
DEPTH       CH01        CH02
----------------------------
0           -           1
0           -           2
300         500         3
310         -           4
320         -           5
330         510         6
```

The only depth samples that are recorded in the file are `300` and `330`. The rest is interpolated between these depth samples. Notice how the first depth samples are 0. That is because there is no prior recorded depth to interpolate against - and constant spacing is not guaranteed.

Looking at `CH01` in the above logset we see that only a third of the samples contains actual measurements. Remember, no interpolation rules are defined for curves that are not the index. This is an undesirable situation both from dlisio's point-of-view - and from the consumer of the log. dlisio would have to fill all these undefined samples with some value on behalf of the user, as there is no such thing as a sparse numpy array. This will have a negative effect on memory consumption. And, depending on the application of course, the consumer of the log would likely want to filter out the "absent-values" and re-sample the logs anyway. To avoid this, **only equally sampled curves can be read into the same numpy array by** *`dlisio.lis.curves()`*. **This means that multiple calls to this function are required to read all the curves in the logset.** The parameter `sample_rate` is used to define which curves to read. Please refer to the example section for more details.

[1] Structured Arrays, https://numpy.org/doc/stable/user/basics.rec.html

[2] LIS79 ch 3.3.2.2, http://w3.energistics.org/LIS/lis-79.pdf

Parameters

- **f** (`LogicalFile`) – The logcal file that the dfsr belongs to
- **dfsr** (`dlisio.lis.DataFormatSpec`) – Data Format Specification Record
- **sample_rate** (*None*) – Read channels matching a specific sampling rate (relative to the recorded index). If all curves are sampled equally compared to the index, this can be omitted. If not, a value must be given to tell which subset of the curves in the log set should be read.
- **strict** (*boolean, optional*) – By default (strict=True) curves() raises a ValueError if there are multiple channels with the same mnemonic. Setting strict=False lifts this restriction and dlisio will append numerical values (i.e. 0, 1, 2 ..) to the labels used for column-names in the returned array.

Returns **curves** – Numpy structured ndarray with mnemonics as column names

Return type np.ndarray

Raises

- **ValueError** – If the DFSR contains the same mnemonic multiple times. Numpy Structured Array requires all column names to be unique. See parameter *strict* for workaround

- **NotImplementedError** – If the DFSR contains one or more channel where the type of the samples is lis::mask

Examples

The returned array supports both horizontal- and vertical slicing. Slice on a subset of channels:

```
>>> curves = dlisio.lis.curves(f, dfsr)
>>> curves[['CHANN2', 'CHANN3']]
array([
    (16677259., 852606.),
    (16678259., 852606.),
    (16679259., 852606.),
    (16680259., 852606.)
])
```

Or slice a subset of the samples:

```
>>> curves = dlisio.lis.curves(f, dfsr)
>>> curves[0:2]
array([
    (16677259., 852606., 2233., 852606.),
    (16678259., 852606., 2237., 852606.)])
```

When the logset described by the DFSR contains channels with different sampling rates, multiple calls to curves are necessary to read all the curves. E.g. we can read *all* curves that are sampled at the same rate as the index:

```
>>> dlisio.lis.curves(f, dfsr, sample_rate=1)
array([(300, 500),
       (330, 510)],
  dtype=[('DEPT', '<i4'), ('CH01', '<i4')])
```

As we can see the outputted logset contains the index curve (DEPT) and CH01. However, we know from examination of the current dfsr that there is another channel in this logset, CH02, which is sampled at 3 times the rate of the index. This curve (and all other curves which are sampled equally to it) can be read by a separate call to curves:

```
>>> dlisio.lis.curves(f, dfsr, sample_rate=3)
array([(  0, 1),
       (  0, 2),
       (300, 3),
       (310, 4),
       (320, 5),
       (330, 6)],
  dtype=[('DEPT', '<i4'), ('CH02', '<i4')])
```

Note that it's the same index curve as previously, only re-sampled to fit the higher sampling rate of CH02.

dlisio.lis.**curves_metadata**()

Get the metadata corresponding to curves()

This is a sister-function to *dlisio.lis.curves()*, that returns the metadata objects (Spec Blocks) corresponding to the curves returned by *dlisio.lis.curves()*.

The keys of the returned dict exactly match the (column) names in the numpy array returned by *curves.lis.curves*. The values are the corresponding Spec Blocks.

Notes

`curves_metadata()` and `curves()` should be called with the same values for both parameters *sample_rate* and *strict* for the metadata and curves to match.

If dfsr.depth_mode is 1, then there is no Spec Block for the index. In this case None is used as value for the index in the returned dict.

See also:

`DataFormatSpec.index_mnem` The mnemonic of the index

`DataFormatSpec.index_units` The units of the index

Returns metadata

Return type dict

Examples

Lets say that you want to read all curves that are sampled 6x the index, and you don't care if the mnemonics are repeated. And that you are also interested in the metadata of the curves you read:

```
>>> rate = 6
>>> strict = False
>>> curves = lis.curves(f, fs, sample_rate=rate, strict=strict)
>>> metadata = lis.curves_metadata( fs, sample_rate=rate, strict=strict)
```

You now have the curves and their metadata. The metadata lookup mirrors the (column) names in the data. E.g. access data and metadata for the *first* RCCL curve in this log set:

```
>>> data = curves['RCCL(0)']
>>> spec = metadata['RCCL(0)']
```

2.11.5 LIS Logical Records

Note: dlisio's LIS reader is not yet feature complete and the following list does not reflect the full set of Logical Records defined in LIS79.

File Header (FHLR)

class dlisio.core.**file_header**

File Header Logical Record (FHLR)

file_name

File Name - A unique, fixed length, name for a Logical File within a Logical Tape. The file name consists of two parts:

Service name - A 6 character Name of the service or program that created the tape

File number - A 3 character counter that counts the files in a logical tape

The service name and file number are seperated by a dot (".")

Type str

`service_sublvl_name`

Service Sub Level Name - Subdevision of the Service ID that is used to further classify the source of data.

Type str

`version_number`

Version number for the software that wrote the original data

Type str

`date_of_generation`

Date of generation for the software that wrote the original data. Format: Year/Month/Day

Type str

`max_pr_length`

Maximum physical record length

Type str

`file_type`

Indicator for the kind of information in the file. For example: LL for label, LO for log data or CA for Calibration

Type str

`prev_file_name`

Optional previous file number. When used, it has the same format as *`file_name`*.

Type str

File Trailer (FTLR)

`class dlisio.core.file_trailer`

File Trailer Logical Record (FTLR)

The FTLR is an optional last record of a Logical File. It's identical to the *`dlisio.core.file_header`* with exception of the attribute *`dlisio.core.file_header.prev_file_name`*, which in the trailer is replaced with *`next_file_name`*.

`file_name`

`service_sublvl_name`

`version_number`

`date_of_generation`

`max_pr_length`

`file_type`

`next_file_name`

Optional next file number. When used, it has the same format as *`file_name`*.

Type str

Tape Header (THLR)

class dlisio.core.**tape_header**

Tape Header Logical Record (THLR)

service_name

The name of the service or program that created the tape. This name is used as the service name in *dlisio.core.file_header.file_name* (and *dlisio.core.file_trailer.file_name*) for all Logical Files in the tape

Type str

date

Date when the original data was acquired. Format: Year/Month/Day

Type str

origin_of_data

The system that originally acquired or created the data

Type str

name

Tape name - An ID that can aid to identify the Logical Tape, where applicable

Type str

continuation_number

Tape continuation number - sequential ordering of tapes on the same reel

Type str

comment

Any relevant remarks concerning the Logical Tape or the content of it

Type str

prev_tape_name

An ID that can be used to identify the previous Logical Tape, where applicable. Should be blank for the first tape

Type str

Tape Trailer (TTLR)

class dlisio.core.**tape_trailer**

Tape Trailer Logical Record (TTLR)

The TTLR is optional. It's identical to the *dlisio.core.tape_header* with exception of the attribute *dlisio.core.tape_header.prev_tape_name*, which in the trailer is replaced with *next_tape_name*.

service_name

date

origin_of_data

name

continuation_number

comment

next_tape_name

An ID that can be used to identify the next Logical Tape, where applicable. Should be blank for the last tape

Type str

Reel Header (RHLR)

class dlisio.core.**reel_header**

Reel Header Logical Record (RHLR)

service_name

The name of the service or program that created the tape. This name is used as the service name in *dlisio.core.file_header.file_name* (and *dlisio.core.file_trailer.file_name*) for all Logical Files in the tape

Type str

date

Date when the physical reel was created. Format: Year/Month/Day

Type str

origin_of_data

The system that originally acquired or created the data

Type str

name

Reel name - A 8 character name used to identify a specific reel of tape. This name matches the visual identification written on the tape container

Type str

continuation_number

A number sequentially ordering multiple physical reels. The value is in the range 1 to 99

Type str

comment

Any relevant remarks related to the physical reel of tape

prev_reel_name

An ID that can be used to identify the previous Physical Reel, where applicable.

Type str

Reel Trailer (RTLR)

`class dlisio.core.reel_trailer`

Reel Trailer Logical Record (RTLR)

The RTLR is optional. It's identical to the `dlisio.core.reel_header` with exception of the attribute `dlisio.core.reel_header.prev_reel_name`, which in the trailer is replaced with `next_reel_name`.

`service_name`

`date`

`origin_of_data`

`name`

`continuation_number`

`comment`

`next_reel_name`

An ID that can be used to identify the next Physical Reel, where applicable.

Type str

Data Format Specification

`class dlisio.lis.DataFormatSpec`

Data Format Specification Record (DFSR)

This is dlisio's main interface for accessing Data Format Specification Records. A DFSR describes some arbitrary set of channels/curves that are recorded together along some common index.

The DFSR contains two main categories of data: Entry Blocks & Spec Blocks.

Entry Blocks (EB)

EBs are a set of well-defined properties that applies to the DFSR. These are implemented as properties on this class. E.g. `dlisio.lis.DataFormatSpec.depth_units` and `dlisio.lis.DataFormatSpec.direction`.

Spec Blocks (SB)

SB's contain information about a single channel/curve in this DFSR such as mnemonics, units and dimensions of the curves. These can be accessed through `dlisio.lis.DataFormatSpec.specs`.

Notes

For those familiar with DLIS, DFSRs are analogous to DLIS frames `dlisio.dlis.Frame`.

See also:

`dlisio.lis.curves` Read the curves described by a DataFormatSpec

`property absent_value`

Absent Value

A default value that is used in the frame to indicate that the entry has no valid data and should be ignored.

`property depth_mode`

Depth Recording Mode

The mode in which the depth is recorded in the file. This is mainly an implementation detail and the mode used will not affect the end-user of dlisio in any way.

`property depth_reprc`

Datatype of depth/index channel

The datatype that the depth/index channel is recorded as. This is a number corresponding to LIS79-specific types.

`property depth_units`

Units of depth/index channel

This is typically only defined when *`depth_mode`* is 1. When depth mode is 0, the depth channel is defined by the first Spec Block in *`specs`*.

If you don't care about the depth recoding mode and just want the units of the index, use *`index_units`* instead.

`property direction`

Direction of the acquisition

This UP/DOWN flag indicated whether the measurements where taken going upwards or downwards in the well. A value of 1 means UP, 255 means DOWN while 0 means "neither". Any other value is unspecified by LIS79.

`property entries`

Entry Blocks (EB)

Gives access to the underlying Entry Blocks. The average user is advised to interact with the Entry Blocks through the properties of this class. E.g. absent_value, depth_units etc..

`property frame_size`

The size of one frame (row)

This refers to the size (in bytes) of one row of data in the file.

Notes

LIS79 uses custom datatypes which do not exist on modern computers. This means that the frame size (dtype.itemsize) of the numpy arrays produced by dlisio may not always correspond to the frame size on disk.

`property index_mnem`

Mnemonic of the index

Returns **mnemonic**

Return type str

`property index_units`

Units of the index

Returns **units**

Return type str

`property max_frames`

Maximum frames per Logical Record

The maximum frames (rows) recorded per Logical Record. This refers to the fact that the log data is partitioned onto multiple Logical Record, but that is an implementation detail that most users do not need to care about.

`property optical_log_depth_units`

Optical Log Depth Scale Units

This flag specifies the depth units used on the optical log on the original recording. A value of 1 means 'feet', 255 means 'meters' while 0 means 'time'. Any other value is unspecified by LIS79.

`property record_type`

Data Record Type

Indicated the Logical Record type which is used to store the data which is described by this Data Format Specification Record.

Notes

Only Logical Record Type 0 (normal data) was ever defined by LIS79. Hence this attribute has little semantic value.

`property reference_point`

Data Reference Point

LIS79 defined the Data Reference Point as:

> Data Reference Point - There is a point on the tool string called the tool reference point. Its distance from the surface corresponds to measured depth. The data reference point is another point which is fixed relative to the tool string. At any instant during the real-time acquisition of data, the data reference point stands opposite the part of the hole to which the current output corresponds. This value is the distance of the data reference point above the tool reference point. It may be positive or negative. It is useful in determining the significance of data, such as tension or frame duration, which are not actually a function of depth. If absent, the value is undefined on the tape.

`property reference_point_units`

Reference point units

Units of the reference point described by *`dlisio.lis.DataFormatSpec.reference_point`*.

`sample_rates()`

Return all sample rates used by (non-index) Channels in this DFSR

Returns rates

Return type set of ints

`property spacing`

Frame Spacing

Depth difference between consecutive frames

`property spacing_units`

Frame Spacing Units

Units of the frame spacing described by *`dlisio.lis.DataFormatSpec.spacing`*.

property spec_block_subtype

Spec Block Subtype

There are two different subtype of the Spec Blocks (0 or 1). These have slightly different properties.

property spec_block_type

Spec Block Type

Defines which Block type is being used. Default is 0.

property specs

Spec Blocks (SB)

Gives access to the underlying Spec Blocks. A Spec Block is the LIS79 structure that defines channels/curves. Each Spec Block defines one channel and its properties.

LIS79 defines 2 different Spec Block types, namely subtype 0 and 1. These mostly share the same attributes, but there are a couple of attributes that differ between the 2 subtypes. *spec_block_type* defines which of the 2 subtypes is being used in the DFSR.

See also:

dlisio.core.spec_block_0	Speck Block - subtype 0
dlisio.core.spec_block_1	Speck Block - subtype 1

Job Identification

Job Identification Logical Records implement the interface of *dlisio.lis.InformationRecord*

Tool String Info

Tool String Info Logical Records implement the interface of *dlisio.lis.InformationRecord*

Wellsite Data

Wellsite Data Logical Records implement the interface of *dlisio.lis.InformationRecord*

Operator Command Inputs

Operator Command Inputs Records implement the interface of *dlisio.core.text_record*

Operator Response Inputs

Operator Response Inputs Records are used to store input issued to the operator in response to a system request for information. They implement the interface of *dlisio.core.text_record*

System Outputs to Operator

System Outputs to Operator Records are used to store system output messages issued by the operator. They implement the interface of *dlisio.core.text_record*

FLIC Comment

Comment Records implement the interface of *dlisio.core.text_record*

2.11.6 LIS Structures

Other structures defined by LIS79

class dlisio.core.**spec_block_0**

Spec Block - Subtype 0

A Spec Block contains information needed to correctly parse one channel from a frame. It also contains useful information such as the units of the curve measurement.

Note: For those familiar with DLIS, Spec Blocks are analogous to DLIS Channels *dlisio.dlis.Channel*.

mnemonic

Name of the channel

Type str

service_id

The service ID identifies the tool, the tool string used to measure the datum, or the name of the computed product.

Type str

service_order_nr

A unique number which identifies the logging trip to the well-site.

Type str

units

Units of the channel

Type str

file_nb

Indicates the file number at the time the data was first acquired and written (for well-site data acquisitions only). This number, together with service_id and service_order_nr will uniquely identify any data string for the purpose of merging or other processing.

Type int

reserved_size

The number of bytes reserved for this channel in the frame. If size is negative, the output is suppressed. The space is still reserved, and is the absolute value of this entry.

Type int

`samples`

The number of samples recorded per frame. When samples == 1, the channel is sampled at the same interval as the index of the frame.

Type int

`reprc`

The type of the recorded channel data. This number refers to one of the LIS79-defined data types.

Type int

`api_log_type`

This, together with the attributes *`api_curve_type`*, *`api_curve_class`* and *`api_modifier`* form a largely outdated log/curve code system utilizing a 2 digit curve code. Ref API Bulletin D-9 Feb '79.

Type int

`api_curve_type`

See *`api_log_type`*.

Type int

`api_curve_class`

See *`api_log_type`*.

Type int

`api_modifier`

See *`api_log_type`*.

Type int

`process_level`

Process level is a measure of the amount of processing done to obtain the curve. The size of the number increases in proportion to the amount of processing. However, the system has never been objectively defined.

Type int

class `dlisio.core.`**`spec_block_1`**

Spec Block - Subtype 1

Spec Blocks subtype 1 share most of its attributes with *`dlisio.core.spec_block_0`*. However, there are 3 notable differences:

- The `api_*`-attributes are replaced with *`api_codes`*
- Subtype 1 has no `process_level`
- Subtype 1 defines *`process_indicators`*, which do not exist in subtype 0.

`mnemonic`

`service_id`

`service_order_nr`

`units`

`file_nb`

`reserved_size`

samples

reprc

api_codes

API codes form a log/curve system featuring a 3-digit curve code (Ref: API Bulletin D-9 Jul '79). The API codes are represented as a 32 bit integer. The 8-digit number can be masked out to dd/ddd/dd/d. E.g `45310011` should be interpreted as:

```
- Log Type    = 45
- Curve Type  = 310
- Curve Class = 01
- Modifier    = 1
```

Type int

process_indicators

Process Indicators are used to define different processes or corrections that have been performed on the channel. The process indicators are defined as a bit-mask of 40 bits:

Bit nr	Definition
0-1	original logging direction [1]
2	true vertical depth correction
3	data channel not on depth
4	data channel is filtered
5	data channel is calibrated
6	computed (processed thru a function former)
7	derived (computed from more than one tool)
8	tool defined correction nb 2
9	tool defined correction nb 1
10	mudcake correction
11	lithology correction
12	inclinometry correction
13	pressure correction
14	hole size correction
15	temperature correction
22	auxiliary data flag
23	schlumberger proprietary

A value of 1 for a specific bit means that the correction or process is applied. Note that bits not listed in the table are undefined / unassigned by LIS79.

[1] Bits 0 and 1 form a single entry that defines the original logging direction for this channel. A value of '01' (1) indicates down-hole. '10' (2) indicated up-hole, while '00' (0) indicates an ambiguous direction. I.e. stationary. '11' (3) is undefined.

For convenience the bitmask is expanded to an object with all the above definitions as attributes.

Type *dlisio.core.process_indicators*

class dlisio.core.**process_indicators**

original_logging_direction: int

true_vertical_depth_correction: bool

data_channel_not_on_depth: bool

data_channel_is_filtered: bool

data_channel_is_calibrated: bool

computed: bool

derived: bool

tool_defined_correction_nb_2: bool

tool_defined_correction_nb_1: bool

mudcake_correction: bool

lithology_correction: bool

inclinometry_correction: bool

pressure_correction: bool

hole_size_correction: bool

temperature_correction: bool

auxiliary_data_flag: bool

schlumberger_proprietary: bool

class `dlisio.core.`**component_block**

Component Block (CB)

Component Blocks are the basic structure of an Information Record. Each CB contains an individual piece of information.

type_nb

Helps define how a *`dlisio.lis.InformationRecord`* is formatted. E.g. as a series of individual pieces of information or a table of information.

Type int

reprc

The type of *`component`*

Type int

size

The size of *`component`*

Type int

category

Category is undefined by LIS79

Type int

`mnemonic`

The name of the Component Block

Type str

`units`

The units of measurement for *`component`*

Type str

`component`

The actual data of this Component Block

`class dlisio.lis.`**`InformationRecord`**`(`*`attic`*`)`

Information Record

The 3 LIS Logical Record types Job Identification, Wellsite Data and Tool String Info are structured identically. This class implements the interface for them all.

An Information Record can contain:

- Identification of company name, well name and similar
- Parameters used in computation
- Data environments, such as how a curve is presented on a graphical display

The content of a Information Record can be formatted in one of 2 ways:

- As a table (structured)
- As a list of single parameter values (unstructured)

Whether the record is structured as a table or not, each individual entry is represented by a LIS Component Block (CB) *`dlisio.core.component_block`*.

Notes

The LIS79 Specification is ambiguous on which record types that count as *Information Records*. The table in `figure 3.9:  Logical Record Types` includes Encrypted Table Dump and Table dump in the group called Information Records. The rest of the specification refers to Information Records as Job Identification, Wellsite Data and Tool String Info Records. This is how dlisio defines an Information Record too, as the structure of these are identical while Table Dump and Encrypted Table Dump Record share a different record structure.

`components``()`

Component Blocks

Return all the Component Blocks in the record in an unstructured list - Regardless of the intended structure of the record.

Returns Component Blocks

Return type list of *`dlisio.core.component_block`*

`isstructured``()`

Is the record structured as a table

Return True if the content of the record is structured as a table, otherwise returns False. Empty records - that is records with zero Component Blocks are considered unstructured.

Returns isstructured

Return type bool

table(*fill=None*, *simple=False*)

Format the Information Record components as a table

LIS explicitly allows Tables in Information Records to be sparse. That is, missing entries are allowed. If some table entries are not recorded in the file dlisio will fill that table cell with a default value, given by argument `fill`. E.g. if the following table information is recorded in the Information Record:

```
MNEM GCOD GDEC DEST DSCA
------------------------
1    E2E  2         S5
2    BBB  -    PF2
```

Then row 1 of 'DEST' and row 2 of 'DSCA' will be filled by dlisio.

Parameters

- **fill** – A default value to fill into cells with no data
- **simple** (*bool*) – If simple=False, the table will be populated with the `component_block` directly. If simple=True, then the resulting table will be populated with the values from the *dlisio.core.component_block*.

Raises

- **ValueError** – If the content of the record is not structured as a table.
- **ValueError** – If the content of the record is ill-formed.

Returns **table** – The Information Record structured in a Numpy Structured Array

Return type np.ndarray

table_name()

Table name

When the record contains a table, the name of the table is defined by the first Component Block in the record

Raises **ValueError** – If the record contains unstructured data. I.e. the content is not formatted as a table.

Returns A CB containing the name of the table stored in this record

Return type *dlisio.core.component_block*

2.11.7 Utilities

class dlisio.lis.**HeaderTrailer**(*header=None*, *trailer=None*)

Container for Header-Trailer pairs

Both Reels and Tapes have a Header Logical Record (RHLR and THLR) - and optionally a Trailer Logical Record (RTLR / TTLR).

The Trailer Records have an identical structure to their corresponding header, except for the prev_reel_name/prev_tape_name, which in the trailer is named next_reel_name/next_tape_name.

header()

Header Record

Returns the Reel or Tape Header Logical Record (RHLR or THLR), depending on the context in which the current instance lives.

Returns header – Returns None if the Header Record is missing.

Return type *core.reel_header*, *core.tape_header* or None

`trailer()`

Trailer Record

Returns the Reel or Tape Trailer Logical Record (RTLR or TTLR), depending on the context in which the current instance lives.

Returns trailer – Returns None if the Trailer Record is missing.

Return type *core.reel_trailer*, *core.tape_trailer* or None

`class dlisio.core.text_record`

Describes Miscellaneous records which contain one text field.

`message`

Complete content of the record. Text might be fully human-readable, partly human-readable or be just a bytes sequence. To get more control over the returned value refer to *Strings and encodings*.

Type str or bytes

2.12 Common API Reference

2.12.1 Strings and encodings

`dlisio.common.set_encodings(`*encodings*`)`

Set codepages to use for decoding strings

RP66 specifies that all strings should be in ASCII, meaning 7-bit. Strings in ASCII have identical bitwise representation in UTF-8, and python strings are in UTF-8. However, a lot of files contain strings that aren't ASCII, but are encoded in some way - a common is the degree symbol[1]. Plenty of files use other encodings too.

LIS does not explicitly mention that strings should be ASCII, but it also doesn't mention any encodings.

This function sets the code pages that dlisio will try *in order* when decoding the string-types specified by LIS and DLIS. UTF-8 will always be tried first, and is always correct if the file behaves according to spec.

Available encodings can be found in the Python docs[2].

If none of the encodings succeed, all strings will be returned as a bytes object.

Parameters encodings (`list of str`) – Ordered list of encodings to try

Warns UnicodeWarning – When no decode was successful, and a bytes object is returned

Warning: There is no place in the LIS or DLIS spec to put or look for encoding information, decoding is a wild guess. Plenty of strings are valid in multiple encodings, so there's a high chance that decoding with the wrong encoding will give a valid string, but not the one the writer intended.

[1] https://stackoverflow.com/questions/8732025/why-degree-symbol-differs
[2] https://docs.python.org/3/library/codecs.html#standard-encodings

Warning: It is possible to change the encodings at any time. However, only strings created after the change will use the new encoding. Having strings that are out of sync w.r.t encodings might lead to unexpected behaviour. It is recommended that the file is reloaded after changing the encodings to ensure that all strings use the same encoding.

See also:

get_encodings currently set encodings

Notes

Strings are decoded using Python's bytes.decode(errors = 'strict').

References

Examples

Decoding of the same string under different encodings

```
>>> from dlisio import dlis, common
>>> common.set_encodings([])
>>> with dlis.load('file.dlis') as (f, *_):
...     print(getchannel(f).units)
b'custom unit\xb0'
>>> common.set_encodings(['latin1'])
>>> with dlis.load('file.dlis') as (f, *_):
...     print(getchannel(f).units)
'custom unit°'
>>> common.set_encodings(['utf-16'])
>>> with dlis.load('file.dlis') as (f, *_):
...     print(getchannel(f).units)
''
```

dlisio.common.**get_encodings**()

Get codepages to use for decoding strings

Get the currently set codepages used when decoding strings.

Returns **encodings**

Return type list

See also:

set_encodings

2.12.2 Open

`dlisio.common.open(path, offset=0)`

Open a file

Open a low-level file handle. This is not intended for end-users - rather, it's an escape hatch for very broken files that dlisio cannot handle.

Parameters

- **path** (*str_like*) –
- **offset** (*int*) – Physical file offset at which handle must be opened

Returns **stream**

Return type dlisio.core.stream

See also:

`dlisio.dlis.load`, `dlisio.lis.load`

2.12.3 Error handling

`class dlisio.common.ErrorHandler`

Defines rules about error handling

Many .dlis files happen to be not compliant with specification or simply broken. This class gives user some control over handling of such files.

When dlisio encounters a specification violation, it categories the issue based on the severity of the violation. Some issues are easy to ignore while other might force dlisio to give up on its current task. ErrorHandler supplies an interface for changing how dlisio reacts to different violation in the file.

Different categories are *info*, *minor*, *major* and *critical*:

Severity	Description
critical	Any issue that forces dlisio stop its current objective prematurely is categorised as critical. By default a critical error raises a RuntimeError. An example would be file indexing, which happens at load. Suppose the indexing fails midways through the file. There is no way for dlisio to reliably keep indexing the file. However, it is likely that the file is readable up until the point of failure. Changing the behaviour of critical from raising an Exception to logging would in this case mean that a partially indexed file is returned by load.
major	Result of a direct specification violation in the file. dlisio makes an assumption about what broken information [1] should have been and continues parsing the file on this assumption. If no other major or critical issues are reported, it's likely that assumption was correct and that dlisio parsed the file correctly. However, no guarantees can be made. By default a warning is logged. [1] Note that "information" in this case refers to the data in the file that tells dlisio how the file should be parsed, not to the actual parsed data.
minor	Like Major issues, this is also a result of a direct specification violation. dlisio makes similar assumptions to keep parsing the file. Minor issues are generally less severe and, in contrast to major issues, are more likely to be handled correctly. However, still no guarantees can be made about the parsed data. By default an info message is logged.
info	Issue doesn't contradict specification, but situation is peculiar. By default a debug message is logged.

ErrorHandler only applies to issues related to parsing information from the file. These are issues that otherwise would force dlisio to fail, such as direct violations of the RP66v1 specification. It does not apply to inconsistencies and issues in the parsed data. This means that cases where dlisio enforces behaviour of the parsed data, such as object-to-object references, are out of scope for the ErrorHandler.

Please also note that ErrorHandler doesn't redefine issues categories, it only changes default behavior.

`info`
: Action for merely information message

`minor`
: Action for minor specification violation

`major`
: Action for major specification violation

`critical`
: Action for critical specification violation

Warning: Escaping errors is a good solution when user needs to read as much data as possible, for example, to have a general overview over the file. However user must be careful when using this mode during close inspection. If user decides to accept errors, they must be aware that some returned data will be spoiled. Most likely it will be data which is stored in the file near the failure.

Warning: Be careful not to ignore too much information when investigating files. If you want to debug a broken part of the file, you should look at all issues to get a full picture of the situation.

Examples

Define your own rules:

```
>>> from dlisio.common import ErrorHandler, Actions
>>> def myhandler(msg):
...     logging.getLogger('custom').info("error in dlisio")
...     raise RuntimeError("Custom handler: " + msg)
>>> errorhandler = ErrorHandler(
...     info     = Actions.SWALLOW,
...     minor    = Actions.LOG_WARNING,
...     major    = Actions.RAISE,
...     critical = myhandler)
```

Parse a file:

```
>>> from dlisio import dlis
>>> files = dlis.load(path)
RuntimeError: "...."
>>> handler = ErrorHandler(critical=Actions.LOG_ERROR)
>>> files = dlis.load(path, error_handler=handler)
[ERROR] "...."
>>> for f in files:
...  pass
```

class dlisio.common.**Actions**

Actions available for various specification violations

LOG_DEBUG()

logging.debug

LOG_ERROR()

logging.error

LOG_INFO()

logging.info

LOG_WARNING()

logging.warning

RAISE()

raise RuntimeError

SWALLOW()

pass

2.13 Logging

dlisio is using Python's standard library logging module to emit several logs. dlisio only supplies the loggers, while configuring them is left to the user. If not, the logging modules default configuration applies.

For example, next may be done to make all modules to print to stderr info messages, not just warnings and errors as it happens by default:

```
>>> import logging
>>> logger = logging.getLogger('dlisio')
>>> logger.setLevel(logging.DEBUG)
>>> ch = logging.StreamHandler()
>>> ch.setLevel(logging.INFO)
>>> formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
>>> ch.setFormatter(formatter)
>>> logger.addHandler(ch)
```

CHAPTER

THREE

INDICES AND TABLES

- genindex

INDEX