pyjls

Release 0.9.4

Jetperch LLC

May 10, 2024

CONTENTS

CHAPTER

ONE

JLS

Welcome to the Joulescope® File Format project. The goal of this project is to provide performant data storage for huge, simultaneous, one-dimensional signals. This repository contains:

- The JLS file format specification
- The implementation in C
- Language bindings for Python

1.1 Features

- Cross-platform
  - Microsoft Windows x64
  - Apple macOS x64
  - Apple macOS ARM
  - Linux x64
  - Linux aarch64 (ARM 64-bit). Supports Raspberry Pi 4.
- Support for multiple, simultaneous data sources
- Support for multiple, simultaneous signal waveforms
- Fixed sample rate signals (FSR)
  - Handles missing samples gracefully (interpolate)
  - Multiple data types including:
    - Floating point: f32, f64
    - Unsigned integers: u1, u4, u8, u16, u24, u32, u64
    - Signed integers: i4, i8, i16, i24, i32, i64
    - Fixed-point, signed integers (same bit sizes as signed integers)
    - Boolean (digital) 1-bit signals = u1
- Variable sample rate (VSR) signals
- Fast read performance
  - Signal Summaries
    - "Zoomed out" view with mean, min, max, standard deviation

        * Provides fast waveform load without any additional processing steps
    - Automatic load by summary level
    - Fast seek, next, previous access
- Sample ID to Wall-clock time (UTC) for FSR signals
- Annotations
    - Global VSR annotations
    - Signal annotations, timestamped to sample_id for FSR and UTC time for VSR
    - Support for text, marker, and user-defined (text, binary, JSON)
- User data
    - Arbitrary data included in the same file
    - Support for text, binary, and JSON
- Reliability
    - Integrated integrity checks using CRC32C
    - File data still accessible in the case of improper program termination
    - Uncorrupted data is still accessible in presence of file corruption
    - Write once, except for indices and the doubly-linked list pointers
- Compression options
    - lossless
    - lossy
    - lossy with downsampling below threshold
    - Support level 0 DATA not written (only INDEX & SUMMARY)

Items marked with are under development and coming soon. Items marked with are planned for future release.

As of June 2023, the JLS v2 file structure is well-defined and stable. However, the compression storage formats are not yet defined and corrupted file recovery is not yet implemented.

1.2 Why JLS?

The world is already full of file formats, and we would rather not create another one. However, we could not identify a solution that met these requirements. HDF5 meets the large storage requirements, but not the reliability and rapid load requirements. The Saleae binary export file format v2 is also not suitable since it buffers stores single, contiguous blocks. Sigrok v2 is similar. The Sigrok v3 format (under development as of June 2023) is better in that it stores sequences of "packets" containing data blocks, but it still will does not allow for fast seek or summaries.

Timeseries databases, such as InfluxDB, are powerful tools. However, they are not well-designed for fast sample-rate data.

Media containers are another option, especially the ISO base media file format used by MPEG4 and many others:

- ISO/IEC 14496-14:2020 Specification
- Overview

However, the standard does not include the ability to store the signal summaries and our specific signal types. While we could add these features, these formats are already complicated, greatly reducing the advantage of repurposing them.

1.3 Why JLS v2?

This file format is based upon JLS v1 designed for pyjoulescope and used by the Joulescope test instrument. We leveraged the lessons learned from v1 to make v2 better, faster, and more extensible.

The JLS v1 format has been great for the Joulescope ecosystem and has accomplished the objective of long data captures (days) with fast sampling rates (MHz). However, it now has a long list of issues including:

- Inflexible storage format (always current, voltage, power, current range, GPI0, GPI1).
- Unable to store from multiple sources.
- Unable to store other sources and signals.
- No annotation support: 41, 93.
- Inflexible user data support.
- Inconsistent performance across sampling rates, zoom levels, and file sizes: 48, 103.
- Unable to correlate sample times with UTC: 55.

The JLS v2 file format addressed all of these issues, dramatically improved performance, and added new capabilities, such as signal compression.

1.4 How?

At its lowest layer, JLS is an enhanced tag-length-value (TLV) format. TLV files form the foundation of many reliable image and video formats, including MPEG4 and PNG. The enhanced header contains additional fields to speed navigation and improve reliability. The JLS file format calls each TLV a **chunk**. The enhanced tag-length component the **chunk header** or simply **header**. The file also contains a **file header**, not to be confused with the **chunk header**. A **chunk** may have zero payload length, in which case the next header follows immediately. Otherwise, a **chunk** consists of a **header** followed by a **payload**.

The JLS file format defines **sources** that produce data. The file allows the application to clearly define and label the source. Each source can have any number of associated signals.

Signals are 1-D sequences of values over time consisting of a single, fixed data type. Each signal can have multiple **tracks** that contain data associated with that signal. The JLS file supports two signal types: fixed sample rate (FSR) and variable sample rate (VSR). FSR signals store their sample data in the FSR track using FSR_DATA and FSR_SUMMARY. FSR time is denoted by samples using timestamp. FSR signals also support:

- Sample time to UTC time mapping using the UTC track.
- Annotations with the ANNOTATION track.

VSR signals store their sample data in the VSR track. VSR signals specify time in UTC (wall-clock time). VSR signals also support annotations with the ANNOTATION track. The JLS file format supports VSR signals that only use the ANNOTATION track and not the VSR track. Such signals are commonly used to store UART text data where each line contains a UTC timestamp.

Signals support DATA chunks and SUMMARY chunks. The DATA chunks store the actual sample data. The SUMMARY chunks store the reduced statistics, where each statistic entry represents multiple samples. FSR tracks store the mean, min, max, and standard deviation. Although standard deviation requires the writer to compute the square root, standard deviation keeps the same units and bit depth requirements as the other fields. Variance requires twice the bit size for integer types since it is squared.

Before each SUMMARY chunk, the JLS file will contain the INDEX chunk which contains the starting time and offset for each chunk that contributed to the summary. This SUMMARY chunk enables fast O(log n) navigation of the file. For FSR tracks, the starting time is calculated rather than stored for each entry.

The JLS file format design supports SUMMARY of SUMMARY. It supports the DATA and up to 15 layers of SUMMARIES. timestamp is given as a 64-bit integer, which allows each summary to include only 20 samples and still support the full 64-bit integer timestamp space. In practice, the first level summary increases a single value to 4 values, so summary steps are usually 50 or more.

Many applications, including the Joulescope UI, prioritize read performance, especially visualizing the waveform quickly following open, over write performance. Waiting to scan through a 1 TB file is not a valid option. The reader opens the file and scans for sources and signals. The application can then quickly load the highest summary of summaries for every signal of interest. The application can very quickly display this data, and then start to retrieve more detailed information as requested.

1.5 Example file structure

```
sof
header
USER_DATA(0, NULL)    // Required, point to first real user_data chunk
SOURCE_DEF(0)         // Required, internal, reserved for global annotations
SIGNAL_DEF(0, 0.VSR)  // Required, internal, reserved for global annotations
TRACK_DEF(0.VSR)
TRACK_HEAD(0.VSR)
TRACK_DEF(0.ANNO)
TRACK_HEAD(0.ANNO)
SOURCE_DEF(1)         // input device 1
SIGNAL_DEF(1, 1, FSR) // our signal, like "current" or "voltage"
TRACK_DEF(1.FSR)
TRACK_HEAD(1.FSR)
TRACK_DEF(1.ANNO)
TRACK_HEAD(1.ANNO)
TRACK_DEF(1.UTC)
TRACK_HEAD(1.UTC)
USER_DATA           // just because
TRACK_DATA(1.FSR)
TRACK_DATA(1.FSR)
TRACK_DATA(1.FSR)
TRACK_DATA(1.FSR)
TRACK_INDEX(1.FSR, lvl=0)
TRACK_SUMMARY(1.FSR, lvl=1)
TRACK_DATA(1.FSR)
TRACK_DATA(1.FSR)
TRACK_DATA(1.FSR)
TRACK_DATA(1.FSR)
TRACK_INDEX(1.FSR, lvl=0)
TRACK_SUMMARY(1.FSR, lvl=1)
TRACK_DATA(1.FSR)
TRACK_DATA(1.FSR)
TRACK_DATA(1.FSR)
TRACK_DATA(1.FSR)
TRACK_INDEX(1.FSR, lvl=0)
TRACK_SUMMARY(1.FSR, lvl=1)
TRACK_INDEX(1.FSR, lvl=1)
TRACK_SUMMARY(1.FSR, lvl=2)
USER_DATA           // just because
```

(continues on next page)

(continued from previous page)

```
END
eof
```

Note that TRACK_HEAD(1.FSR) points to the first TRACK_INDEX(1.FSR, lvl=0) and TRACK_INDEX(1.FSR, lvl=1). Each TRACK_DATA(1.FSR) is in a doubly-linked list with its next and previous neighbors. Each TRACK_INDEX(1.FSR, lvl=0) is likewise in a separate doubly-linked list, and the payload of each TRACK_INDEX points to the summarized TRACK_DATA instances. TRACK_INDEX(1.FSR, lvl=1) points to each TRACK_INDEX(1.FSR, lvl=0) instance. As more data is added, the TRACK_INDEX(1.FSR, lvl=1) will also get added to the INDEX chunks at the same level.

1.6 Resources

- source code
- documentation
- pypi
- Joulescope (Joulescope web store)
- forum

1.7 References

- JLS v1: lower-layer, upper-layer.
- Sigrok/v3, which shares many of the same motivations.
- Tag-length-value: Wikipedia.
- Doubly linked list: Wikipedia.

1.8 License

CHAPTER

TWO

C API

2.1 Error Codes

group **jls_ec**

Standardize error code definitions.

See errc

Defines

JLS_ERROR_CODES(X)

The list of error codes for use by X macros.

See also:

https://en.wikipedia.org/wiki/X_Macro

See also:

http://www.drdobbs.com/cpp/the-x-macro/228700289

JLS_ERROR_ENUM(NAME, TEXT)

The macro used to define the error code enum.

JLS_SUCCESS

A shorter, less confusing alias for success.

Enums

enum **jls_error_code_e**

The list of error codes.

Values:

enumerator **JLS_ERROR_CODE_COUNT**

Functions

const char *`jls_error_code_name`(int ec)

Convert an error code into its short name.

Parameters
`ec` – **[in]** The error code (jls_error_code_e).

Returns
The short string name for the error code.

const char *`jls_error_code_description`(int ec)

Convert an error code into its description.

Parameters
`ec` – **[in]** The error code (jls_error_code_e).

Returns
The user-meaningful description of the error.

2.2 JLS Format

group `jls_format`

JLS file format.

Defines

`JLS_FORMAT_VERSION_MAJOR`

`JLS_FORMAT_VERSION_MINOR`

`JLS_FORMAT_VERSION_PATCH`

`JLS_FORMAT_VERSION_U32`

`JLS_HEADER_IDENTIFICATION`

The file identification bytes at the start of the file.

These bytes are not arbitrary. We carefully selected them to provide:

- Identification: Help the application determine that this file is in the correct format with minimal uncertainty.
- Correct endianness: Little endian has won, so this entire format is stored in little endian format.
- Proper binary processing: The different line ending combinations ensure that the reader is not "fixing" the line endings, since this is a binary file format.
- Display: Include "substitute" and "file separator" so that text printers to not show the rest of the file.

The following table describes maps each byte to its purpose:

Value (hex)	Purpose
6A 6C 73 66 6d 74	ASCII “jlsfmt”, when viewed in text editor.
0D 0A	DOS line ending to ensure binary correctness.
20	ASCII space.
0A	UNIX line ending to ensure binary correctness.
20	ASCII space.
1A	Substitute character (stops listing under Windows).
20 20	ASCII spaces.
B2	Ensure that system supports 8-bit data
1C	File separator.

JLS_SOURCE_COUNT

The maximum allowed number of sources.

JLS_SIGNAL_COUNT

The maximum allowed number of signals.

JLS_SUMMARY_LEVEL_COUNT

The number of summary levels.

JLS_TRACK_TAG_FLAG

JLS_TRACK_TAG_PACK(track_type, track_chunk)

Pack the chunk tag.

Parameters

- **track_type** – The jls_track_type_e
- **track_chunk** – The jls_track_chunk_e

Returns

The tag value.

JLS_TRACK_TAG_PACKER(track_type, track_chunk)

JLS_DATATYPE_BASETYPE_INT

JLS_DATATYPE_BASETYPE_UNSIGNED

JLS_DATATYPE_BASETYPE_UINT

JLS_DATATYPE_BASETYPE_FLOAT

JLS_DATATYPE_DEF(basetype, size, q)

Construct a JLS datatype.

Parameters

- **basetype** – The datatype base type, one of [INT, UINT, FLOAT, BOOL]

- **size** – The size in bits. Only the following options are supported:
  - INT: 4, 8, 16, 24, 32, 64
  - UINT: 1, 4, 8, 16, 24, 32, 64
  - FLOAT: 32, 64
  - BOOL = UINT 1
- **q** – The signed fixed-point location, only valid for INT and UINT. Set to 0 for normal, whole numbers. Set to 0 for FLOAT and BOOL. The fixed point is between bits q and q-1. The value is scaled by 2 ** -q.

JLS_DATATYPE_I4

JLS_DATATYPE_I8

JLS_DATATYPE_I16

JLS_DATATYPE_I24

JLS_DATATYPE_I32

JLS_DATATYPE_I64

JLS_DATATYPE_U1

JLS_DATATYPE_U4

JLS_DATATYPE_U8

JLS_DATATYPE_U16

JLS_DATATYPE_U24

JLS_DATATYPE_U32

JLS_DATATYPE_U64

JLS_DATATYPE_BOOL

JLS_DATATYPE_F32

JLS_DATATYPE_F64

Enums

enum **jls_signal_type_e**

The signal type definition.

Values:

enumerator **JLS_SIGNAL_TYPE_FSR**

Fixed sampling rate.

enumerator **JLS_SIGNAL_TYPE_VSR**

Variable sampling rate.

enum **jls_track_type_e**

The available track types that store data over time.

Values:

enumerator **JLS_TRACK_TYPE_FSR**

Block tracks contain fixed sample-rate (FSR) data.

The JLS_TAG_SIGNAL_DEF defines the sampling rate.

enumerator **JLS_TRACK_TYPE_VSR**

Fixed-type, variable-sample-rate (VSR) time series data.

Each data entry consists of time in UTC and the data.

enumerator **JLS_TRACK_TYPE_ANNOTATION**

Annotations contain infrequent, variable-typed data.

Each annotation data entry consists of time and associated annotation data. For FSR, time must be samples_id. For VSR, time must be UTC.

See also:

jls_annotation_type_e for the annotation types.

enumerator **JLS_TRACK_TYPE_UTC**

The UTC track associates sample_id with UTC.

Each utc data entry consists of sample_id, UTC timestamp pairs. This track is only used for FSR signals.

enumerator **JLS_TRACK_TYPE_COUNT**

The total number of track types.

enum **jls_storage_type_e**

The data storage type.

The data storage type applies to directly user-accessible data including annotations and user_data.

Values:

enumerator **JLS_STORAGE_TYPE_INVALID**

Invalid (unknown) storage type.

enumerator **JLS_STORAGE_TYPE_BINARY**

Raw binary data.

enumerator **JLS_STORAGE_TYPE_STRING**

Null-terminated C-style string with UTF-8 encoding.

enumerator **JLS_STORAGE_TYPE_JSON**

JSON serialized data structure with NULL terminator and UTF-8 encoding.

enum **jls_annotation_type_e**

The available annotation types.

Values:

enumerator **JLS_ANNOTATION_TYPE_USER**

Arbitrary user data.

Application-dependent data with no standardized form or purpose.

enumerator **JLS_ANNOTATION_TYPE_TEXT**

UTF-8 formatted text.

Viewers should display this text at the appropriate location. The jls_storage_type_e must be STRING.

enumerator **JLS_ANNOTATION_TYPE_VERTICAL_MARKER**

A vertical marker at a given time.

Marker names can be arbitrary, but the convention is:
- Number strings, like "1", represent a single marker.
- Alpha + number string, like "A1" and "A2", represent a marker pair (dual markers). The jls_storage_type_e must be STRING.

enumerator **JLS_ANNOTATION_TYPE_HORIZONTAL_MARKER**

A horizontal marker at a given y-axis value.

Marker names can be arbitrary, but the convention is:
- Number strings, like "1", represent a single marker.
- Alpha + number string, like "A1" and "A2", represent a marker pair (dual markers). The jls_storage_type_e must be STRING.

enum **jls_track_chunk_e**

The chunks used to store the track information.

Values:

enumerator **JLS_TRACK_CHUNK_DEF**

Track definition chunk.

This chunk contains a zero-length (empty) payload.

enumerator **JLS_TRACK_CHUNK_HEAD**

Track offsets for the first chunk at each level.

This chunk provides fast seek access to the first chunk at each summary level for this track. The payload is *jls_track_head_s* which contains the offset to the first INDEX chunk.

See also:

jls_track_head_s for all track types.

enumerator **JLS_TRACK_CHUNK_DATA**

The data chunk.

The payload varies by track type and data format. All DATA payloads start with *jls_payload_header_s*.

See also:

jls_fsr_f32_data_s for FSR float32.

See also:

jls_annotation_s for ANNOTATION.

See also:

jls_utc_data_s for UTC.

enumerator **JLS_TRACK_CHUNK_INDEX**

Provides the timestamp and offset for each contributing data chunk.

This chunk MUST be immediately follow by a SUMMARY chunk. All INDEX payloads start with *jls_payload_header_s*.

See also:

jls_fsr_index_s for FSR track types.

See also:

jls_index_s for all other track types.

enumerator **JLS_TRACK_CHUNK_SUMMARY**

The summary chunk.

The payload format is defined by the track type. All CHUNK payloads start with *jls_payload_header_s*.

See also:

jls_fsr_f32_summary_s for FSR float32.

See also:

jls_annotation_summary_s for ANNOTATION.

See also:

jls_utc_summary_s for UTC.

enum **jls_tag_e**

The tag definitions.

Values:

enumerator **JLS_TAG_INVALID**

enumerator **JLS_TAG_SOURCE_DEF**

enumerator **JLS_TAG_SIGNAL_DEF**

enumerator **JLS_TAG_TRACK_FSR_DEF**

enumerator **JLS_TAG_TRACK_FSR_HEAD**

enumerator **JLS_TAG_TRACK_FSR_DATA**

enumerator **JLS_TAG_TRACK_FSR_INDEX**

enumerator **JLS_TAG_TRACK_FSR_SUMMARY**

enumerator **JLS_TAG_TRACK_VSR_DEF**

enumerator **JLS_TAG_TRACK_VSR_HEAD**

enumerator **JLS_TAG_TRACK_VSR_DATA**

enumerator **JLS_TAG_TRACK_VSR_INDEX**

enumerator **JLS_TAG_TRACK_VSR_SUMMARY**

enumerator **JLS_TAG_TRACK_ANNOTATION_DEF**

enumerator **JLS_TAG_TRACK_ANNOTATION_HEAD**

enumerator **JLS_TAG_TRACK_ANNOTATION_DATA**

enumerator **JLS_TAG_TRACK_ANNOTATION_INDEX**

enumerator **JLS_TAG_TRACK_ANNOTATION_SUMMARY**

enumerator **JLS_TAG_TRACK_UTC_DEF**

enumerator **JLS_TAG_TRACK_UTC_HEAD**

enumerator **JLS_TAG_TRACK_UTC_DATA**

enumerator **JLS_TAG_TRACK_UTC_INDEX**

enumerator **JLS_TAG_TRACK_UTC_SUMMARY**

enumerator **JLS_TAG_USER_DATA**

enumerator **JLS_TAG_END**

enum **jls_summary_fsr_e**

The summary storage order for each entry.

Values:

enumerator **JLS_SUMMARY_FSR_MEAN**

enumerator **JLS_SUMMARY_FSR_STD**

enumerator **JLS_SUMMARY_FSR_MIN**

enumerator **JLS_SUMMARY_FSR_MAX**

enumerator **JLS_SUMMARY_FSR_COUNT**

Functions

static inline uint8_t **jls_datatype_parse_basetype**(uint32_t dt)

static inline uint8_t **jls_datatype_parse_size**(uint32_t dt)

static inline uint8_t **jls_datatype_parse_q**(uint32_t dt)

struct **jls_source_def_s**

#include <format.h> The source definition.

Public Members

uint16_t **source_id**

The source identifier.

0 reserved for global annotations, must be unique per instance

const char ***name**

The source name string.

const char ***vendor**

The vendor name string.

const char ***model**

The model string.

const char ***version**

The version string.

const char ***serial_number**

The serial number string.

struct **jls_signal_def_s**

#include <format.h> The signal definition.

Public Members

uint16_t **signal_id**

The signal identifier.

0 to JLS_SIGNAL_COUNT - 1, must be unique per instance

uint16_t **source_id**

The source identifier, must match a source_def.

uint8_t **signal_type**

The jls_signal_type_e signal type.

uint32_t **data_type**

The JLS_DATATYPE_* data type for this signal.

uint32_t **sample_rate**

TThe sample rate per second (Hz). 0 for VSR.

uint32_t **samples_per_data**

The number of samples per data chunk. (write suggestion)

uint32_t **sample_decimate_factor**

The number of samples per summary level 1 entry.

uint32_t **entries_per_summary**

The number of entries per summary chunk. (write suggestion)

uint32_t **summary_decimate_factor**

The number of summaries per summary, level >= 2.

uint32_t **annotation_decimate_factor**

The annotation decimate factor for summaries.

uint32_t **utc_decimate_factor**

The UTC decimate factor for summaries.

int64_t **sample_id_offset**

The sample id offset for the first sample. (FSR only)

const char ***name**

The signal name.

const char ***units**

The units string, normally as SI with no scale prefix.

struct **jls_track_head_s**

#include <format.h> The track head payload for JLS_TRACK_CHUNK_HEAD.

union **jls_version_u**

#include <format.h> Union structure for parsing 32-bit versions.

Public Members

uint32_t **u32**

uint16_t **patch**

uint8_t **minor**

uint8_t **major**

struct *jls_version_u*::[anonymous] **s**

struct **jls_file_header_s**

#include <format.h> The JLS file header structure.

Public Members

uint8_t **identification**[16]

The semi-unique file identification header.

See also:

JLS_HEADER_IDENTIFICATION

uint64_t **length**

The file length in bytes.

This is the last field updated on file close. If the file was not closed gracefully, the value will be 0.

union *jls_version_u* **version**

The JLS file format version number.

See also:

JLS_FORMAT_VERSION_U32

uint32_t **crc32**

The CRC32 from the start of the file through version.

struct **jls_chunk_header_s**

#include <format.h> The JLS chunk header structure.

Every chunk starts with this header. This header identifies the chunk's payload, and enables the chunk to participate in one doubly-linked list.

If the length is zero, then the next chuck header immediately follows this chunk header. If the length is not zero, then the chunk consists of:

- A chunk header
- payload of length bytes
- Zero padding of 0-7 bytes, so that the entire chunk will end on a multiple of 8 bytes. This field ends on: 8 * k - 4
- crc32 over the payload.

Public Members

uint64_t **item_next**

The next item.

The chunk header enables each chunk to participate in a single, doubly-linked list. This field indicates the location for the next item in the list. The value is relative to start of the file. 0 indicates end of list.

This field allows simple, linear traversal of data, but the next chunk is not known when the chunk is first created. Therefore, this field requires that chunk headers are updated when the software writes the next item to the file.

uint64_t **item_prev**

The previous item.

The chunk header enables each chunk to participate in a single, doubly-linked list. This field indicates the location for the previous item in the list. The value is relative to start of the file. 0 indicates start of list.

uint8_t **tag**

The tag.

The jls_tag_e value that identifies the contents of this chunk.

uint8_t **rsv0_u8**

Reserved for future use. (compression?)

uint16_t **chunk_meta**

The metadata associated with this chunk.

Each tag is free to define the purpose of this field.

However, all data tags use this definition:

- chunk_meta[7:0] is the signal/time series identifier from 0 to 255.
- chunk_meta[11:8] is reserved.
- chunk_meta[15:12] contains the depth for this chunk from 0 to 15.
  - 0 = block (sample level)
  - 1 = First-level summary of block samples
  - 2 = Second-level summary of first-level summaries.

User-data reserves chunk_meta[15:12] to store the storage_type and another internal indications. chunk_meta[11:0] may be assigned by the specific application.

uint32_t **payload_length**

The length of the payload in bytes.

Can be 0.

In addition to defining the payload size, this value is also used for forward chunk traversal.

uint32_t **payload_prev_length**

The length of the previous payload in bytes.

Used for reverse chunk traversal.

uint32_t **crc32**

The CRC32 over the header, excluding this field.

struct **jls_payload_header_s**

#include <format.h> The payload header for DATA, INDEX, and SUMMARY chunks.

Public Members

int64_t **timestamp**

The sample_id for the first entry.

uint32_t **entry_count**

The total number of entries.

uint16_t **entry_size_bits**

The size of each entry, in bits.

uint16_t **rsv16**

Reserved.

struct **jls_fsr_data_s**

#include <format.h> The FSR data chunk format.

Public Members

struct *jls_payload_header_s* **header**

The payload header.

float **data**[]

The summary data.

Although data's type is float, the actual data type depends upon the signal definition.

struct **jls_fsr_index_s**

#include <format.h> The payload for JLS_TAG_TRACK_FSR_INDEX chunks.

Since FSR has a fixed sample rate, the header contains enough information to fully identify the timestamp for each offset. Therefore, no additional time information is required per entry.

See also:

jls_index_s for all other INDEX chunk types.

Public Members

uint64_t **offsets**[]

The chunk file offsets, spaced by fixed time intervals.

struct **jls_fsr_f32_summary_s**

#include <format.h> The FSR summary chunk format with f32 values.

This summary format is used by all types except u32, u64, i32, i64, f64.

Public Members

struct *jls_payload_header_s* **header**

The payload.

float **data**[][*JLS_SUMMARY_FSR_COUNT*]

The summary data, each entry is 4 x f32: mean, std, min, max.

struct **jls_fsr_f64_summary_s**

#include <format.h> The FSR summary chunk format with f64 values.

This summary format is used by u64, i64, f64.

Public Members

struct *jls_payload_header_s* **header**

The payload.

double **data**[][*JLS_SUMMARY_FSR_COUNT*]

The summary data, each entry is 4 x f64: mean, std, min, max.

struct **jls_index_entry_s**

#include <format.h> The entry format for JLS_TRACK_CHUNK_INDEX payloads.

See also:

jls_index_s

Public Members

int64_t **timestamp**

The timestamp for this entry. sample_id for FSR, UTC for VSR.

uint64_t **offset**

The chunk file offset.

struct **jls_index_s**

#include <format.h> The payload for JLS_TRACK_CHUNK_INDEX chunks.

The INDEX payload maps timestamps to offsets to allow fast seek. However, the JLS_TAG_TRACK_FSR_INDEX uses the *jls_fsr_index_s* since the timestamp provides unnecessary, duplicative information for FSR tracks.

See also:

jls_fsr_index_s for JLS_TAG_TRACK_FSR_INDEX.

struct **jls_annotation_s**

#include <format.h> Hold a single annotation record.

This structure is used by both the API and JLS_TAG_TRACK_ANNOTATION_DATA.

Public Members

int64_t **timestamp**

The timestamp for this annotation. sample_id for FSR, UTC for VSR.

uint64_t **rsv64_1**

Reserved, write to 0.

uint8_t **annotation_type**

The jls_annotation_type_e.

uint8_t **storage_type**

The jls_storage_type_e.

uint8_t **group_id**

The optional group identifier. If unused, write to 0.

uint8_t **rsv8_1**

Reserved, write to 0.

float **y**

The y-axis value or NAN to automatically position.

uint32_t **data_size**

The size of data in bytes.

uint8_t **data**[]

The annotation data.

struct **jls_annotation_summary_entry_s**

#include <format.h> The entry format for JLS_TAG_TRACK_ANNOTATION_SUMMARY.

See also:

jls_annotation_summary_s

Public Members

int64_t **timestamp**

The timestamp (duplicates INDEX).

uint8_t **annotation_type**

The *jls_annotation_s.annotation_type*.

uint8_t **group_id**

The *jls_annotation_s.group_id*.

uint8_t **rsv8_1**

Reserved, write to 0.

uint8_t **rsv8_2**

Reserved, write to 0.

float **y**

The *jls_annotation_s.y*.

struct **jls_annotation_summary_s**

#include <format.h> The payload format for JLS_TAG_TRACK_ANNOTATION_SUMMARY chunks.

struct **jls_utc_data_s**

#include <format.h> The entry format for JLS_TAG_TRACK_UTC_DATA.

This same format is reused for summary entries. UTC.DATA only exists to provide recovery in the event that the file is not properly closed.

See also:

jls_utc_summary_s

Public Members

int64_t **timestamp**

The timestamp in UTC.

struct **jls_utc_summary_entry_s**

#include <format.h> The entry format for JLS_TAG_TRACK_UTC_SUMMARY.

This same format is reused for summary entries. UTC.DATA only exists to provide recovery in the event that the file is not properly closed.

See also:

jls_utc_summary_s

Public Members

int64_t **sample_id**

The timestamp in sample ids (duplicates INDEX).

int64_t **timestamp**

The timestamp in UTC.

struct **jls_utc_summary_s**

#include <format.h> The payload format for JLS_TAG_TRACK_UTC_SUMMARY chunks.

2.3 JLS Reader

group **jls_reader**

JLS reader.

Typedefs

typedef int32_t (***jls_rd_annotation_cbk_fn**)(void *user_data, const struct *jls_annotation_s* *annotation)

The function called for each annotation.

See also:

jls_rd_annotations

Param user_data
The arbitrary user data.

Param annotation
The annotation which only remains valid for the duration of the call.

Return
0 to continue iteration or any other value to stop.

typedef int32_t (***jls_rd_user_data_cbk_fn**)(void *user_data, uint16_t chunk_meta, enum *jls_storage_type_e* storage_type, uint8_t *data, uint32_t data_size)

The function called for each user data entry.

See also:

jls_rd_user_data

Param user_data
The arbitrary user data.

Param chunk_meta
The chunk meta value.

Param storage_type
The data storage type.

Param data
The data.

Param data_size
The size of data in bytes.

Return
0 to continue iteration or any other value to stop.

typedef int32_t (*`jls_rd_utc_cbk_fn`)(void *user_data, const struct *jls_utc_summary_entry_s* *utc, uint32_t size)

The function called for each UTC entry.

See also:

jls_rd_annotations

Param user_data
The arbitrary user data.

Param utc
The array of utc entries, which are sample_id / timestamp pairs.

Param size
The number of utc entries for this callback.

Return
0 to continue iteration or any other value to stop.

Functions

int32_t `jls_rd_open`(struct jls_rd_s **instance, const char *path)

Open a JLS file to read contents.

Call *jls_rd_close()* when done.

Parameters

- `instance` – **[out]** The new JLS read instance.
- `path` – The JLS file path.

Returns
0 or error code.

void `jls_rd_close`(struct jls_rd_s *self)

Close a JLS file opened with *jls_rd_open()*.

Parameters
`self` – The JLS read instance.

int32_t **jls_rd_sources**(struct jls_rd_s *self, struct *jls_source_def_s* **sources, uint16_t *count)

Get the array of sources in the file.

Parameters

- **self** – The reader instance.
- **sources** – **[out]** The array of sources.
- **count** – **[out]** The number of items in sources.

Returns
0 or error code.

int32_t **jls_rd_signals**(struct jls_rd_s *self, struct *jls_signal_def_s* **signals, uint16_t *count)

Get the array of signals in the file.

Parameters

- **self** – The reader instance.
- **signals** – **[out]** The array of signals.
- **count** – **[out]** The number of items in signals.

Returns
0 or error code.

int32_t **jls_rd_signal**(struct jls_rd_s *self, uint16_t signal_id, struct *jls_signal_def_s* *signal)

Get the signal by signal_id.

Parameters

- **self** – The reader instance.
- **signal_id** – The signal id to get.
- **signal** – **[out]** The signal definition.

Returns
0 or error code.

int32_t **jls_rd_fsr_length**(struct jls_rd_s *self, uint16_t signal_id, int64_t *samples)

Get the number of samples in an FSR signal.

Parameters

- **self** – The reader instance.
- **signal_id** – The signal id.
- **samples** – **[out]** The number of samples in the signal.

Returns
0 or error code.

int32_t **jls_rd_fsr**(struct jls_rd_s *self, uint16_t signal_id, int64_t start_sample_id, void *data, int64_t data_length)

Read fixed sample rate (FSR) data.

Parameters

- **self** – The reader instance.
- **signal_id** – The signal id. The first recorded sample is always 0. To get the first recorded sample_id, see *jls_signal_def_s.sample_id_offset*.

- **start_sample_id** – The starting sample id to read.
- **data** – **[out]** The samples read.
- **data_length** – The number of samples to read. data is at least this many samples. For data types less than 8 bits long, you need to provide an extra byte to allow for data shifting. Therefore data is at least 1 + (data_length * entry_size_bits) / 8 bytes.

Returns
0 or error code

int32_t **jls_rd_fsr_f32**(struct jls_rd_s *self, uint16_t signal_id, int64_t start_sample_id, float *data, int64_t data_length)

Read fixed sample rate (FSR) float32 data.

Parameters

- **self** – The reader instance.
- **signal_id** – The signal id. The first recorded sample is always 0. To get the first recorded sample_id, see *jls_signal_def_s.sample_id_offset*.
- **start_sample_id** – The starting sample id to read.
- **data** – **[out]** The samples read.
- **data_length** – The number of samples to read. data is also at least this many entries (4 * data_length bytes).

Returns
0 or error code

int32_t **jls_rd_fsr_statistics**(struct jls_rd_s *self, uint16_t signal_id, int64_t start_sample_id, int64_t increment, double *data, int64_t data_length)

Read the statistics data for a fixed sampling rate signal.

For data_length 1, the statistics are sample-accurate. For larger data_lengths, the external boundaries for start and end are computed exactly. The internal boundaries are approximated, perfect for waveform display, but perhaps not suitable for other use cases. If you need sample accurate statistics over multiple increments, call this function repeatedly with data_length 1.

Parameters

- **self** – The reader instance.
- **signal_id** – The FSR signal.
- **start_sample_id** – The starting sample id to read. The first recorded sample is always 0. To get the first recorded sample_id, see *jls_signal_def_s.sample_id_offset*.
- **increment** – The number of samples that form a single output summary.
- **data** – **[out]** The statistics information, in the shape of data[data_length][JLS_SUMMARY_FSR_COUNT]. The elements are mean, standard_deviation, min, max. Use JLS_SUMMARY_FSR_MEAN, JLS_SUMMARY_FSR_STD, JLS_SUMMARY_FSR_MIN, and JLS_SUMMARY_FSR_MAX to index the values.
- **data_length** – The number of statistics points to populate. data is at least JLS_SUMMARY_FSR_COUNT * data_length elements, each of float64 type (8 bytes). This argument allows efficient computation over many consecutive windows, as is common for displaying waveforms.

Returns
0 or error code.

int32_t **jls_rd_annotations**(struct jls_rd_s *self, uint16_t signal_id, int64_t timestamp, *jls_rd_annotation_cbk_fn* cbk_fn, void *cbk_user_data)

Iterate over the annotations for a signal.

Parameters

- **self** – The reader instance.
- **signal_id** – The signal id.
- **timestamp** – The starting timestamp. Skip all prior annotations.
- **cbk_fn** – The callback function that *jls_rd_annotations()* will call once for each matching annotation. Return 0 to continue to the next annotation or a non-zero value to stop iteration.
- **cbk_user_data** – The arbitrary data provided to cbk_fn.

Returns
0 or error code.

int32_t **jls_rd_user_data**(struct jls_rd_s *self, *jls_rd_user_data_cbk_fn* cbk_fn, void *cbk_user_data)

Iterate over user data entries.

Parameters

- **self** – The reader instance.
- **cbk_fn** – The callback function that *jls_rd_user_data()* will call once for each user data entry. Return 0 to continue to the next user data entry or a non-zero value to stop iteration.
- **cbk_user_data** – The arbitrary data provided to cbk_fn.

Returns
0 or error code.

int32_t **jls_rd_utc**(struct jls_rd_s *self, uint16_t signal_id, int64_t sample_id, *jls_rd_utc_cbk_fn* cbk_fn, void *cbk_user_data)

Iterate over the UTC timestamps for a FSR signal.

Parameters

- **self** – The reader instance.
- **signal_id** – The signal id.
- **sample_id** – The starting sample_id. Skip all prior sample ids.
- **cbk_fn** – The callback function that *jls_rd_utc()* will call once for UTC entries. Return 0 to continue to the next UTC entries or a non-zero value to stop iteration.
- **cbk_user_data** – The arbitrary data provided to cbk_fn.

Returns
0 or error code.

int32_t **jls_rd_sample_id_to_timestamp**(struct jls_rd_s *self, uint16_t signal_id, int64_t sample_id, int64_t *timestamp)

Convert from sample_id to timestamp for FSR signals.

Parameters

- **self** – The reader instance.

- **signal_id** – The signal id.
- **sample_id** – The sample id to convert.
- **timestamp** – **[out]** The JLS timestamp for the sample_id.

Returns
0 or error code.

int32_t **jls_rd_timestamp_to_sample_id**(struct jls_rd_s *self, uint16_t signal_id, int64_t timestamp, int64_t *sample_id)

Convert from timestamp to sample_id for FSR signals.

Parameters

- **self** – The reader instance.
- **signal_id** – The signal id.
- **timestamp** – The JLS timestamp to convert.
- **sample_id** – **[out]** The sample_id for timestamp.

Returns
0 or error code.

2.4 JLS Threaded Writer

group **jls_threaded_writer**

JLS threaded writer.

This module wraps writer. For normal operation while recording signals, prefer this threaded writer over writer.

Enums

enum **jls_twr_flag_e**

The threaded writer flags.

Values:

enumerator **JLS_TWR_FLAG_DROP_ON_OVERFLOW**

Drop on overflow when set, block otherwise.

Functions

int32_t **jls_twr_open**(struct jls_twr_s **instance, const char *path)

Open a JLS file for writing.

Call *jls_twr_close()* when done.

Parameters

- **instance** – **[out]** The JLS writer instance.

- **path** – The JLS file path.

Returns
0 or error code.

int32_t **jls_twr_close**(struct jls_twr_s *self)

Close a JLS file.

Parameters
self – The JLS writer instance from *jls_twr_open()*.

Returns
0 or error code.

uint32_t **jls_twr_flags_get**(struct jls_twr_s *self)

Parameters

- **Get** – threaded writer flags.
- **self** – The JLS writer instance from *jls_twr_open()*.
- **flags** – The jls_twr_flag_e bits.

Returns
0 or error code.

int32_t **jls_twr_flags_set**(struct jls_twr_s *self, uint32_t flags)

Parameters

- **Set** – threaded writer flags.
- **self** – The JLS writer instance from *jls_twr_open()*.
- **flags** – The jls_twr_flag_e bits.

Returns
0 or error code.

int32_t **jls_twr_flush**(struct jls_twr_s *self)

Flush a JLS file to disk.

Parameters
self – The JLS writer instance from *jls_twr_open()*.

Returns
0 or error code.

int32_t **jls_twr_source_def**(struct jls_twr_s *self, const struct *jls_source_def_s* *source)

Define a new source.

This JLS file format supports multiple sources, which are usually different instruments. Each source can provide multiple signals.

Parameters

- **self** – The JLS writer instance.
- **source** – The source definition.

Returns
0 or error code.

int32_t **jls_twr_signal_def**(struct jls_twr_s *self, const struct *jls_signal_def_s* *signal)

Define a new signal.

Parameters

- **self** – The JLS writer instance.
- **signal** – The signal definition.

Returns
0 or error code.

int32_t **jls_twr_user_data**(struct jls_twr_s *self, uint16_t chunk_meta, enum *jls_storage_type_e* storage_type, const uint8_t *data, uint32_t data_size)

Add arbitrary user data.

Parameters

- **self** – The writer instance.
- **chunk_meta** – The arbitrary data. Bits 15:12 are reserved, but bits 11:0 may be assigned by the application.
- **storage_type** – The storage type for data.
- **data** – The user data to store.
- **data_size** – The size of data for JLS_STORAGE_TYPE_BINARY. Ignored for all other storage types.

Returns
0 or error code.

int32_t **jls_twr_fsr**(struct jls_twr_s *self, uint16_t signal_id, int64_t sample_id, const void *data, uint32_t data_length)

Write fixed-rate sample data to a signal.

Parameters

- **self** – The JLS writer instance.
- **signal_id** – The signal id.
- **sample_id** – The sample id for data[0].
- **data** – The sample data array. Data must be packed with no spacing between samples. u1 stores 8 samples per byte, and u4 stores 2 samples per byte.
- **data_length** – The length of data in samples.

Returns
0 or error code

int32_t **jls_twr_fsr_f32**(struct jls_twr_s *self, uint16_t signal_id, int64_t sample_id, const float *data, uint32_t data_length)

Write sample data to a float32 FSR signal.

Parameters

- **self** – The JLS writer instance.
- **signal_id** – The signal id.
- **sample_id** – The sample id for data[0].
- **data** – The sample data array.

- **data_length** – The length of data in floats (bytes / 4).

Returns
0 or error code

int32_t **jls_twr_fsr_omit_data**(struct jls_twr_s *self, uint16_t signal_id, uint32_t enable)

Omit level 0 data chunks from the signal's stream.

Enable is delayed by one sample block to ensure any pending data is writen. Disable takes effect immediately.

On read, the level 0 data is reconstructed using the summaries.

As of Sep 2023, this setting is ignored for u1, u4, u8, i4, and i8 FSR data types. These data types are omitted whenever the payload contains a constant data value regardless of this setting.

Parameters

- **self** – The JLS writer instance
- **signal_id** – The signal id.
- **enable** – 0 to disable (default), 1 to enable.

Returns
0 or error code.

int32_t **jls_twr_annotation**(struct jls_twr_s *self, uint16_t signal_id, int64_t timestamp, float y, enum *jls_annotation_type_e* annotation_type, uint8_t group_id, enum *jls_storage_type_e* storage_type, const uint8_t *data, uint32_t data_size)

Add an annotation to a signal.

Parameters

- **self** – The writer instance.
- **signal_id** – The signal id.
- **timestamp** – The x-axis timestamp in sample_id for FSR and UTC for VSR.
- **y** – The y-axis value or NAN to automatically position.
- **annotation_type** – The annotation type.
- **group_id** – The optional group identifier. If unused, set to 0.
- **storage_type** – The storage type.
- **data** – The data for the annotation.
- **data_size** – The length of data for JLS_STORAGE_TYPE_BINARY storage_type. Set to 0 for all other storage types.

Returns
0 or error code.

int32_t **jls_twr_utc**(struct jls_twr_s *self, uint16_t signal_id, int64_t sample_id, int64_t utc)

Add a mapping from sample_id to UTC timestamp for an FSR signal.

Parameters

- **self** – The writer instance.
- **signal_id** – The signal id.

- **sample_id** – The sample_id for FSR.
- **utc** – The UTC timestamp.

Returns
0 or error code.

2.5 JLS Time

group **jls_time**

JLS time representation.

The C standard library includes time.h which is very inconvenient for any precision application. This module defines a much simpler 64-bit fixed point integer for representing time. The value is 34Q30 with the upper 34 bits to represent whole seconds and the lower 30 bits to represent fractional seconds. A value of 2**30 (1 << 30) represents 1 second. This representation gives a resolution of 2 ** -30 (approximately 1 nanosecond) and a range of +/- 2 ** 33 (approximately 272 years). The value is signed to allow for simple arithmetic on the time either as a fixed value or as deltas.

Certain elements may elect to use floating point time given in seconds. The macros *JLS_TIME_TO_F64()* and JLS_F64_TO_TIME() facilitate converting between the domains. Note that double precision floating point is not able to maintain the same resolution over the time range as the 64-bit representation. *JLS_TIME_TO_F32()* and JLS_F32_TO_TIME() allow conversion to single precision floating point which has significantly reduce resolution compared to the 34Q30 value.

Defines

JLS_TIME_Q

The number of fractional bits in the 64-bit time representation.

JLS_TIME_MAX

The maximum (positive) time representation.

JLS_TIME_MIN

The minimum (negative) time representation.

JLS_TIME_EPOCH_UNIX_OFFSET_SECONDS

The offset from the standard UNIX (POSIX) epoch.

This offset allows translation between jls time and the standard UNIX (POSIX) epoch of Jan 1, 1970.

The value was computed using python3:

```
import dateutil.parser
dateutil.parser.parse('2018-01-01T00:00:00Z').timestamp()
```

JLS chooses a different epoch to advance “zero” by 48 years!

JLS_TIME_SECOND

The fixed-point representation for 1 second.

JLS_FRACT_MASK

The mask for the fractional bits.

JLS_TIME_MILLISECOND

The approximate fixed-point representation for 1 millisecond.

JLS_TIME_MICROSECOND

The approximate fixed-point representation for 1 microsecond.

CAUTION: this value is 0.024% accurate (240 ppm)

JLS_TIME_NANOSECOND

The approximate fixed-point representation for 1 nanosecond.

WARNING: this value is only 6.7% accurate!

JLS_TIME_MINUTE

The fixed-point representation for 1 minute.

JLS_TIME_HOUR

The fixed-point representation for 1 hour.

JLS_TIME_DAY

The fixed-point representation for 1 day.

JLS_TIME_WEEK

The fixed-point representation for 1 week.

JLS_TIME_MONTH

The average fixed-point representation for 1 month (365 day year).

JLS_TIME_YEAR

The approximate fixed-point representation for 1 year (365 days).

JLS_TIME_TO_F64(x)

Convert the 64-bit fixed point time to a double.

Parameters

- **x** – The 64-bit signed fixed point time.

Returns

The time as a double p. Note that IEEE 747 doubles only have 52 bits of precision, so the result will be truncated for very small deltas.

JLS_TIME_TO_F32(x)

Convert the 64-bit fixed point time to single precision float.

Parameters

- **x** – The 64-bit signed fixed point time.

Returns
The time as a float p in seconds. Note that IEEE 747 singles only have 23 bits of precision, so the result will likely be truncated.

`JLS_TIME_TO_SECONDS(x)`

Convert to 32-bit unsigned seconds.

Parameters

- **x** – The 64-bit signed fixed point time.

Returns
The 64-bit unsigned time in seconds, rounded to nearest.

`JLS_TIME_TO_MILLISECONDS(x)`

Convert to milliseconds.

Parameters

- **x** – The 64-bit signed fixed point time.

Returns
The 64-bit signed time in milliseconds, rounded to nearest.

`JLS_TIME_TO_MICROSECONDS(x)`

Convert to microseconds.

Parameters

- **x** – The 64-bit signed fixed point time.

Returns
The 64-bit signed time in microseconds, rounded to nearest.

`JLS_TIME_TO_NANOSECONDS(x)`

Convert to nanoseconds.

Parameters

- **x** – The 64-bit signed fixed point time.

Returns
The 64-bit signed time in nanoseconds, rounded to nearest.

`JLS_SECONDS_TO_TIME(x)`

Convert to 64-bit signed fixed point time.

Parameters

- **x** – he 32-bit unsigned time in seconds.

Returns
The 64-bit signed fixed point time.

`JLS_MILLISECONDS_TO_TIME(x)`

Convert to 64-bit signed fixed point time.

Parameters

- **x** – The 32-bit unsigned time in milliseconds.

Returns
The 64-bit signed fixed point time.

JLS_MICROSECONDS_TO_TIME(x)

Convert to 64-bit signed fixed point time.

Parameters

- **x** – The 32-bit unsigned time in microseconds.

Returns

The 64-bit signed fixed point time.

JLS_NANOSECONDS_TO_TIME(x)

Convert to 64-bit signed fixed point time.

Parameters

- **x** – The 32-bit unsigned time in microseconds.

Returns

The 64-bit signed fixed point time.

Functions

static inline int64_t JLS_F64_TO_TIME(double x)

Convert the double precision time to 64-bit fixed point time.

Parameters

x – The double-precision floating point time in seconds.

Returns

The time as a 34Q30.

static inline int64_t JLS_F32_TO_TIME(float x)

Convert the single precision float time to 64-bit fixed point time.

Parameters

x – The single-precision floating point time in seconds.

Returns

The time as a 34Q30.

static inline int64_t JLS_TIME_TO_COUNTER(int64_t x, uint64_t z)

Convert to counter ticks, rounded to nearest.

Parameters

- **x** – The 64-bit signed fixed point time.
- **z** – The counter frequency in Hz.

Returns

The 64-bit time in counter ticks.

static inline int64_t JLS_TIME_TO_COUNTER_RZERO(int64_t x, uint64_t z)

Convert to counter ticks, rounded towards zero.

Parameters

- **x** – The 64-bit signed fixed point time.
- **z** – The counter frequency in Hz.

Returns

The 64-bit time in counter ticks.

static inline int64_t **JLS_TIME_TO_COUNTER_RINF**(int64_t x, uint64_t z)

Convert to counter ticks, rounded towards infinity.

Parameters

- **x** – The 64-bit signed fixed point time.
- **z** – The counter frequency in Hz.

Returns
The 64-bit time in counter ticks.

static inline int64_t **JLS_COUNTER_TO_TIME**(uint64_t x, uint64_t z)

Convert a counter to 64-bit signed fixed point time.

Parameters

- **x** – The counter value in ticks.
- **z** – The counter frequency in Hz.

Returns
The 64-bit signed fixed point time.

static inline int64_t **JLS_TIME_ABS**(int64_t t)

Compute the absolute value of a time.

Parameters
t – The time.

Returns
The absolute value of t.

int64_t **jls_now**(void)

Get the UTC time as a 34Q30 fixed point number.

At power-on, the time will start from 0 unless the system has a real-time clock. When the current time first synchronizes to an external host, it may have a large skip.

Be sure to verify your time for each platform using python:

```
python
import datetime
import dateutil.parser
epoch = dateutil.parser.parse('2018-01-01T00:00:00Z').timestamp()
datetime.datetime.fromtimestamp((my_time >> 30) + epoch)
```

Returns
The current time. This value is not guaranteed to be monotonic. The device may synchronize to external clocks which can cause discontinuous jumps, both backwards and forwards.

struct *jls_time_counter_s* **jls_time_counter**(void)

Get the monotonic platform counter.

The platform implementation may select an appropriate source and frequency. The JLS library assumes a nominal frequency of at least 1000 Hz, but we frequencies in the 1 MHz to 100 MHz range to enable profiling. The frequency should not exceed 10 GHz to prevent rollover.

The counter must be monotonic. If the underlying hardware is less than the full 64 bits, then the platform must unwrap and extend the hardware value to 64-bit.

The JLS authors recommend this counter starts at 0 when the system powers up.

Returns
The monotonic platform counter.

static inline int64_t **jls_time_rel**(void)

Get the monotonic platform time as a 34Q30 fixed point number.

Returns
The monotonic platform time based upon the *jls_time_counter()*. The platform time has no guaranteed relationship with UTC or wall-clock calendar time. This time has both offset and scale errors relative to UTC.

struct **jls_time_counter_s**

#include <time.h> The platform counter structure.

Public Members

uint64_t **value**

The counter value.

uint64_t **frequency**

The approximate counter frequency.

CHAPTER

THREE

PYTHON API

Python binding for the native JLS implementation.

class pyjls.binding.**AnnotationType**

The annotation type enumeration.

class pyjls.binding.**DataType**

The signal data type enumeration.

unsigned integers: U1, U4, U8, U16, U32, U64 signed integer: I4, I8, I16, I32, I64 floating point: F32, F64

class pyjls.binding.**Reader**

Open a JLS v2 file for reading.

Parameters
path – The path to the JLS file.

annotations(*signal_id*, *timestamp*, *cbk_fn*)

Read annotations from a signal.

Parameters

- **signal_id** – The signal id.
- **timestamp** – The starting timestamp. FSR uses sample_id. VSR uses utc.
- **cbk** – The function(timestamp, y, annotation_type, group_id, data) to call for each annotation. Return True to stop iteration over the annotations or False to continue iterating.

close()

Close the JLS file and free all resources.

fsr(*signal_id*, *start_sample_id*, *length*)

Read the FSR data.

Parameters

- **signal_id** – The signal id.
- **start_sample_id** – The starting sample id to read.
- **length** – The number of samples to read.

Returns

The data, which varies depending upon the FSR data type.

u1 and u4 data will be packed in little endian order.

For u1, unpack with:
np.unpackbits(y, bitorder='little')[:len(x)]

For u4, unpack with
d = np.empty(len(y) * 2, dtype=np.uint8) d[0::2] = np.bitwise_and(y, 0x0f) d[1::2] = np.bitwise_and(np.right_shift(y, 4), 0x0f)

`fsr_statistics`(*signal_id*, *start_sample_id*, *increment*, *length*)

Read FSR statistics (mean, stdev, min, max).

Parameters

- **signal_id** – The signal id for a fixed sampling rate (FSR) signal.
- **start_sample_id** – The starting sample id to read. The sample_id of the first recorded sample in a signal is 0.
- **increment** – The number of samples represented per return value.
- **length** – The number of return values to generate.

Returns

The 2-D array[summary][stat] of np.float32. * Each summary entry represents the statistics computed

approximately over increment samples starting from start_sample_id + <index> * increment. It has length given by the length argument.

- stat is length 4 with columns defined by SummaryFSR which are mean (average), standard deviation, minimum, and maximum.

For length 1, the return statistics are sample-accurate. For larger lengths, the external boundaries for index 0 (first) and index -1 (last) are computed exactly. The internal boundaries are approximated, perfect for waveform display, but perhaps not suitable for other use cases. If you need sample accurate statistics over multiple increments, call this function repeatedly with length 1.

`sample_id_to_timestamp`(*signal_id*, *sample_id*)

Convert sample_id to UTC timestamp for FSR signals.

Parameters

- **signal_id** – The signal id.
- **sample_id** – The sample_id to convert.

Returns
The JLS timestamp corresponding to sample_id.

Raises
RuntimeError – on error.

`signal_lookup`(*spec*) → *SignalDef*

Look up a signal.

Parameters
spec – The signal id or name.

Returns
The signal definition:

Raises
ValueError – If not found.

signals

Return the dict mapping signal_id to SignalDef.

sources

Return the dict mapping source_id to SourceDef.

timestamp_to_sample_id(*signal_id*, *utc_timestamp*)

Convert UTC timestamp to sample_id for FSR signals.

Parameters

- **signal_id** – The signal id.
- **utc_timestamp** – The UTC timestamp to convert.

Returns

The sample_id corresponding to utc_timestamp.

Raises

RuntimeError – on error.

user_data(*cbk_fn*)

Get the user data.

Parameters

cbk_fn – The callable(chunk_meta_u16, data) called for each user_data entry. Return True to stop iterating over subsequent user data entries or False to continue iterating.

utc(*signal_id*, *sample_id*, *cbk_fn*)

Read the sample_id / utc pairs from a FSR signal.

Parameters

- **signal_id** – The signal id.
- **sample_id** – The starting sample_id.
- **cbk** – The function(entries) to call for each annotation. Entries is an Nx2 numpy array of [sample_id, utc_timestamp]. Return True to stop iteration over the annotations or False to continue iterating.

class pyjls.binding.**SignalDef**(*signal_id: int*, *source_id: int*, *signal_type: int = 0*, *data_type: int = 0*, *sample_rate: int = 0*, *samples_per_data: int = 0*, *sample_decimate_factor: int = 0*, *entries_per_summary: int = 0*, *summary_decimate_factor: int = 0*, *annotation_decimate_factor: int = 0*, *utc_decimate_factor: int = 0*, *sample_id_offset: int = 0*, *name: str = None*, *units: str = None*, *length: int = 0*)

Define a signal.

Variables

- **signal_id** – The source identifier. 0 reserved for global annotations, must be unique per instance.
- **source_id** – The source identifier. Must match a SourceDef entry.
- **signal_type** – The pyjls.SignalType for this signal.
- **data_type** – The pyjls.DataType for this signal.
- **sample_rate** – The sample rate per second (Hz). 0 for VSR.
- **samples_per_data** – The number of samples per data chunk. (write suggestion)

- **sample_decimate_factor** – The number of samples per summary level 1 entry.
- **entries_per_summary** – The number of entries per summary chunk. (write suggestion)
- **summary_decimate_factor** – The number of summaries per summary, level >= 2.
- **annotation_decimate_factor** – The annotation decimate factor for summaries.
- **utc_decimate_factor** – The UTC decimate factor for summaries.
- **sample_id_offset** – The sample id offset for the first sample. (FSR only)
- **name** – The signal name string.
- **units** – The signal units string.
- **length** – The length in samples.

class pyjls.binding.**SignalType**

The signal type enumeration.

class pyjls.binding.**SourceDef**(*source_id: int*, *name: str = None*, *vendor: str = None*, *model: str = None*, *version: str = None*, *serial_number: str = None*)

Define a source.

Variables

- **source_id** – The source identifier.
- **name** – The source name string.
- **vendor** – The vendor string.
- **model** – The model string.
- **version** – The version string.
- **serial_number** – The serial number string.

class pyjls.binding.**SummaryFSR**

The FSR column enumeration.

class pyjls.binding.**Writer**

Create a new JLS writer.

Parameters

path – The output JLS file path.

annotation(*signal_id*, *timestamp*, *y*, *annotation_type*, *group_id*, *data*)

Add an annotation to a signal.

Parameters

- **signal_id** – The signal id.
- **timestamp** – The x-axis timestamp in sample_id for FSR and UTC for VSR.
- **y** – The y-axis value or NAN to automatically position.
- **annotation_type** – The annotation type.
- **data** – The data for the annotation.

Raise

On error.

close()

Close the JLS file and release all resources.

flush()

Flush any pending data to the JLS file.

fsr(*signal_id*, *sample_id*, *data*)

Add FSR data to a signal.

Parameters

- **signal_id** – The signal id for the data.
- **sample_id** – The sample id for the first sample in data.
- **data** – The data to add.

Raise
On error.

fsr_f32(*signal_id*, *sample_id*, *data*)

Add 32-bit floating-point FSR data to a signal.

Parameters

- **signal_id** – The signal id for the data.
- **sample_id** – The sample id for the first sample in data.
- **data** – The 32-bit floating point data to add.

signal_def(*signal_id*, *source_id*, *signal_type=None*, *data_type=None*, *sample_rate=None*, *samples_per_data=None*, *sample_decimate_factor=None*, *entries_per_summary=None*, *summary_decimate_factor=None*, *annotation_decimate_factor=None*, *utc_decimate_factor=None*, *name=None*, *units=None*)

Define a signal.

signal_def_from_struct(*s:* SignalDef)

Define a signal.

source_def(*source_id*, *name=None*, *vendor=None*, *model=None*, *version=None*, *serial_number=None*)

Define a source.

source_def_from_struct(*s:* SourceDef)

Define a source.

user_data(*chunk_meta*, *data*)

Add user data to the file.

Parameters

- **chunk_meta** – The arbitrary u16 metadata value.
- **data** – The bytes to store.

Raise
On error.

utc(*signal_id*, *sample_id*, *utc_i64*)

Add a mapping from sample_id to UTC timestamp for an FSR signal. :param signal_id: The signal id. :param sample_id: The sample_id for FSR. :param utc: The UTC timestamp. :raise: On error.

pyjls.binding.**copy**(*src*, *dst*, *msg_fn=None*, *progress_fn=None*)

Copy a JLS file.

Parameters

- **src** – The source path.
- **dst** – The destination path.
- **msg_fn** – The optional function to call with status messages.
- **progress_fn** – The optional function to call with progress from 0.0 to 1.0.

CHAPTER

FOUR

CHANGELOG

This file contains the list of changes made to the JLS project.

4.1 0.9.4

2024 May 10

- Fixed unicode filename support on Windows #12

4.2 0.9.3

2024 Mar 7

- Added “export” python subcommand.

4.3 0.9.2

2024 Feb 14

- Fixed index error in file repair operation.
- Fixed FSR summary reconstruction on truncation #10

4.4 0.9.1

2023 Nov 13

- Bumped rev to fix publish to pypi.

4.5 0.9.0

2023 Nov 13

- Added jls_copy with "jls copy" command and pyjls "copy" entry point.
- Publish release assets including "jls.exe".

4.6 0.8.2

2023 Oct 25

- Added file truncation repair for never closed files.
- Added support for python 3.12.

4.7 0.8.1

2023 Sep 18

- Improved python binding Reader.fsr_statistics documentation #9.
- Fixed occasional missing last byte in jls_rd_fsr for 1-bit signals.

4.8 0.8.0

2023 Sep 16

- Added file truncation repair.
- Added jls_raw_truncate and jls_raw_backend.
- Refactored reader and writer into new core to enable repairer.
- Added jls_twr_flags_get/set.
- Added JLS_TWR_FLAG_DROP_ON_OVERFLOW which drops samples on overflow.
- Added jls_wr_fsr_omit_data and jls_twr_fsr_omit_data.
- Automatically omit constant 1, 4, & 8 bit entry FSR data chunks.
- Modified default signal_def settings for improved performance.
- Added jls_dt_str for pretty-printing the data type.
- Added u1 and u4 FSR data type support to performance tool.
- Fixed jls_rd_fsr out of bounds memory access.
- Added read_fuzzer tool to jls example.

4.9 0.7.3

2023 Jul 24

- Added “noexcept” to python callbacks. Cython 3.0 deprecates implicit noexcept.

4.10 0.7.2

2023 Jun 14

- Fixed build for Raspberry Pi.
- Added JLS_OPTIMIZE_CRC cmake option.

4.11 0.7.1

2023 Jun 7

- Added Reader.signal_lookup.
- Improved documentation.
- Added Read The Docs integration.
- Improved python “export” subcommand to specify “–signal” by name.
- Added python subcommand “plot”.
- Improved build process, migrated to GitHub Actions.
- Bumped minimum python version from 3.8 to 3.9.

4.12 0.7.0

2023 May 31

- Fixed incorrect write timestamp stride in FSR index/summary entries. Any recording over 5.77 hours was incorrect.
- Improved threaded writer.
  - Removed jls_wr_flush during close due to UI performance problems.
  - Release the GIL on some python Writer operations.
  - Reduced buffer size from 100,000,000 B to 64 MB.
  - Save string null termination byte for annotations and user_data.
  - Increased thread priority on Windows.
  - Do not quit until all messages are processed.
- Added jls executable to examples.
- Improved reader logging and error handling.

4.13 0.6.3

2023 May 16

- Added support for building a shared library. Initialize build subdir with "cmake -DBUILD_SHARED_LIBS=ON .."

4.14 0.6.2

2023 Apr 28

- Improved UTC read processing.
- Added "–utc" option to info entry point.
- Fixed incorrect NaNs in summary on write.
- Added "export" pyjls entry point.

4.15 0.6.1

2023 Apr 27

- Added FSR support for missing and duplicate data.
- Added FSR support for unaligned u1 and u4 data.
- Improved log messages.

4.16 0.6.0

2023 Apr 26

- Fixed JLS to handle non-zero sample_id for first FSR data sample.
- Added pyjls.Reader.timestamp_to_sample_id and sample_id_to_timestamp.

4.17 0.5.3

2023 Apr 19

- Fixed build warnings for fn() declarations.

4.18 0.5.2

2023 Mar 30

- Reduced default log level to WARNING.

4.19 0.5.1

2023 Mar 16

- Added zero length check to jls_wr_fsr_data.

4.20 0.5.0

2023 Mar 9

- Added support for data_type strings (not just enum integers).
- Modified python reader bindings to release GIL.
- Migrated to time64 representation for all API calls. Use utc_to_jls and jls_to_utc to convert as needed to/from python timestamps.
- Added data_type_as_enum and data_type_as_str conversion functions.
- Improved exception messages.

4.21 0.4.3

2022 Nov 30

- Changed windows dependency from deprecated pypiwin32 to pywin32.
- Bumped numpy dependencies.
- Added build system requirements for pip.

4.22 0.4.2

2022 Mar 17

- Fixed pyjls build for Raspberry Pi OS 64-bit.

4.23 0.4.1

2022 Mar 7

- Fixed build for Linux and macOS.

4.24 0.4.0

2022 Mar 5

- Added support for additional data types (was only f32):
  - Signed Integer: i4, i8, i16, i24, i32, i64
  - Unsigned Integer: u1, u4, u8, u16, u24, u32, u64
  - Float: f64
  - Fixed point signed & unsigned integers.
- Renamed jls_rd_fsr_f32_statistics to jls_rd_fsr_statistics, which now always returns statistics as double (f64).
- Improved documentation.

4.25 0.3.4

2021 Oct 28

- Added all version fields to pyjls module.
- Fixed writer not correctly serializing null and empty strings #6.

4.26 0.3.3

2021 Jul 7

- Added reader sample_id bounds checks to FSR functions.
- Cached jls_rd_fsr_length results.

4.27 0.3.2

2021 Jul 7

- Fixed incorrect statistics computation when using summaries.
- Added example/jls_read.c.
- Connected example/generate.py arguments to work correctly.
- Fixed documentation for jls_rd_fsr_f32_statistics().
- Added GitHub action.

4.28 0.3.1

2021 Apr 13

- Fixed y annotation argument order.
- Added horizontal marker annotation.
- Fixed Python API to automatically convert to/from UTC python timestamps.
- Fixed reader seek when contains multiple annotations at same timestamp.

4.29 0.3.0

2021 Apr 8

Yanked this release, use 0.3.1

- NOT BACKWARDS COMPATIBLE with 0.2.6 and earlier.
- Modified annotation to contain optional y-axis position. API change.
- Improved file format consistency and improved format.h.
- Added UTC track writer and reader for FSR signals.
- Added INDEX and SUMMARY writer for ANNOTATION and UTC tracks.
- Added reader seek to timestamp for ANNOTATION and UTC using INDEX & SUMMARY.

4.30 0.2.6

2021 Mar 18

- Fixed uninitialized variables in POSIX backend (thanks Valgrind!).
- Fixed memory leaks (thanks Valgrind!).

4.31 0.2.5

2021 Mar 18

- Added 32-bit ARMv7 support (Raspberry Pi OS).
- Added 64-bit ARM support for Apple silicon (M1).

4.32 0.2.4

2021 Mar 16

- Added support for aarch64 (Raspberry Pi 4). Untested on mac M1.
- Fixed POSIX time.

4.33 0.2.3

2021 Mar 10

- Fixed sdist to include native files for pyjls cython build.
- Added CREDITS.html.
- Included credits, license & readme with pyjls source distribution.

4.34 0.2.2

2021 Mar 9

- Fixed build using Clang and AppleClang.

4.35 0.2.1

2021 Mar 8

- Fixed timeout on close when bursting data.
- Added annotate_stress example.
- Added flush.

4.36 0.2.0

2021 Mar 8

- Added group_id parameter to annotation.
- Fixed example/generate.py length argument.
- Updated numpy dependency to 1.20.
- Fixed pyjls console_script.
- Changed logging interface and connected native to python logging.

4.37 0.1.0

2021 Mar 1

- Initial public release.

4.38 0.0.1

2021 Feb 5

- Initial documentation.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

M

P

R

S

T

U

W