# trsfile Documentation

***Release 0.2.0***

**Kevin Valk**

**Jan 23, 2019**

# Contents

Riscure Inspector uses the `.trs` file format to save and read traces from disk. To better assist reading and writing trace set files from third parties, Riscure published this Python library.

CHAPTER 1

# Quick start

This library supports reading and writing of `.trs` files, but it does not (*yet*) support modifying existing `.trs` files. Both the `TrcFile` and the `Trace` class emulate all the functionality of a `list`, so slice to your heart's content!

## 1.1 Installation

This library is available on PyPi for Python 3 and up. Just add `trsfile` to your `requirements.txt` or install it via the command line:

```
pip install trsfile
```

## 1.2 Reading `.trs` files

```
import trsfile

with trsfile.open('trace-set.trs', 'r') as traces:
    # Show all headers
    for header, value in trs_file.get_headers().items():
        print(header, '=', value)
    print()

    # Iterate over the first 25 traces
    for i, trace in enumerate(trs_file[0:25]):
        print('Trace {0:d} contains {1:d} samples'.format(i, len(trace)))
        print('  - minimum value in trace: {0:f}'.format(min(trace)))
        print('  - maximum value in trace: {0:f}'.format(max(trace)))
```

## 1.3 Creating `.trs` files

```
import random, os, trsfile
from trsfile import trs_open, Trace, SampleCoding, TracePadding, Header

with trs_open(
        'trace-set.trs',                # File name of the trace set
        'w',                            # Mode: r, w, x, a (default to x)
        # Zero or more options can be passed (supported options depend on the storage␣
↪engine)
        engine = 'TrsEngine',           # Optional: how the trace set is stored␣
↪(defaults to TrsEngine)
        headers = {                     # Optional: headers (see Header class)
            Header.LABEL_X: 'Testing X',
            Header.LABEL_Y: 'Testing Y',
            Header.DESCRIPTION: 'Testing trace creation',
        },
        padding_mode = TracePadding.AUTO,# Optional: padding mode (defaults to␣
↪TracePadding.AUTO)
        live_update = True              # Optional: updates the TRS file for live␣
↪preview (small performance hit)
                                        #   0 (False): Disabled (default)
                                        #   1 (True) : TRS file updated after every␣
↪trace
                                        #   N        : TRS file is updated after N␣
↪traces
    ) as trs_file:
    # Extend the trace file with 100 traces with each 1000 samples
    trs_file.extend([
        Trace(
            SampleCoding.FLOAT,
            [random.uniform(-255, 255) for _ in range(0, 1000)],
            data = os.urandom(16)
        )
        for _ in range(0, 100)]
    )

    # Replace 5 traces (the slice [0:10:2]) with random length traces.
    # Because we are creating using the TracePadding.PAD mode, all traces
    # will be clipped or padded on the first trace length
    trs_file[0:10:2] = [
        Trace(
            SampleCoding.FLOAT,
            [random.uniform(0, 255) for _ in range(0, random.randrange(1000))],
            data = os.urandom(16),
            title = 'Clipped trace'
        )
        for _ in range(0, 5)
    ]

    # Adding one Trace
    trs_file.append(
        Trace(
            SampleCoding.FLOAT,
            [random.uniform(-255, 255) for _ in range(0, 1000)],
            data = os.urandom(16)
        )
```

(continues on next page)

(continued from previous page)

```
    )

    # We cannot delete traces with the TrsEngine, other engines do support this␣
↪feature
    #del trs_file[40:50]

    # We can only change headers with a value that has the same length as the␣
↪previous value
    # with the TrsEngine, other engines can support dynamically adding, deleting or␣
↪changing
    # headers.
    #trs_file.update_header(Header.LABEL_X, 'Time')
    #trs_file.update_header(Header.LABEL_Y, 'Voltage')
    #trs_file.update_header(Header.DESCRIPTION, 'Traces created for some purpose!')

    print('Total length of new trace set: {0:d}'.format(len(trs_file)))
```

## 1.4 Converting `TraceSet` from one type to another

```
import random, os, trsfile

with \
    trsfile.open(
        'trace-set',                    # Previously create trace set
        'r',                            # Read only mode
        engine='FileEngine'             # Using the FileEngine
    ) as traces, \
    trsfile.open(                       # Note: TrsEngine is the default
        'trace-set.trs',                # Name of the new trace set
        'w',                            # Write mode
        headers=traces.get_headers()    # Copy the headers
    ) as new_traces:
    new_traces.extend(traces)           # Extend the new trace set with the
                                        # traces from the old trace set
```

CHAPTER 2

# Documentation

The full documentation is available in the `docs` folder with a readable version on Read the Docs.

CHAPTER 3

# Testing

The library supports Python `unittest` module and the tests can be executed with the following command:

```
python -m unittest
```

CHAPTER 4

# License

BSD 3-Clause Clear License

CHAPTER 5

# Architecture overview

This diagram gives a quick overview on a conceptual level how different concepts are related.

CHAPTER 6

# The API documentation

If you are looking for information on a specific function, class, or method, this part of the documentation is for you.

## 6.1 The API documentation

This part of the documentation covers all the interfaces of trsfile.

### 6.1.1 Overview

This section gives an overview of the main classes and their descriptions.

`trsfile.`**`open`**(*path*, *mode='r'*, ***options*)

Reads, modifies or creates a `TraceSet` with a specific storage engine (defaults to `TrsEngine`).

**Parameters**

- **`path`** (*`str`*) – path to the file or directory
- **`mode`** (*`str`*) – mode how to open the file or directory (same as the default Python *`open`*)
- **`options`** (*`dict(str, any)`*) – zero or more options that are passed down to the `TraceSet` and the storage engine. Available options can be found in the different storage engines. The storage engine can be selected with `engine = 'TrsEngine'` (default value).

**Returns** instance of a new or initialized `TraceSet`

**Return type** *TraceSet*

`trsfile.`**`trs_open`**(*path*, *mode='r'*, ***options*)

Reads, modifies or creates a `TraceSet` with a specific storage engine (defaults to `TrsEngine`).

**Parameters**

- **`path`** (*`str`*) – path to the file or directory

- **mode** (*str*) – mode how to open the file or directory (same as the default Python *open*)
- **options** (*dict(str, any)*) – zero or more options that are passed down to the TraceSet and the storage engine. Available options can be found in the different storage engines. The storage engine can be selected with `engine = 'TrsEngine'` (default value).

**Returns** instance of a new or initialized TraceSet

**Return type** *TraceSet*

**class** trsfile.trace.**Trace**(*sample_coding*, *samples*, *data=b''*, *title='trace'*, *headers={}*)

The *Trace* class behaves like a list object were each item in the list is a sample of the trace.

When a *Trace* is initialized the samples are (optionally) converted to a numpy.array depending on the current type of the samples and the provided sample_coding.

**class** trsfile.trace_set.**TraceSet**(*path*, *mode='r'*, ***options*)

The *TraceSet* class behaves like a list object were each item in the list is a Trace.

Storing the *TraceSet* requires knowledge on the format which is resolved through the usage of storage engines (Engine).

**class** trsfile.engine.trs.**TrsEngine**(*path*, *mode='x'*, ***options*)

This engine supports .trs files from Riscure as specified in the "Trace set coding" document in Inspector.

This engine supports the following options:

| Option | Description |
|---|---|
| headers | Dictionary containing zero or more headers, see *trsfile.common.Header* |
| live_update | Performs live update of the TRS file every N traces. True for updating after every trace and False for never. |
| padding_mode | Padding mode to use. The supported values are: *trsfile.common.TracePadding.NONE* and *trsfile.common.TracePadding.AUTO* (default) |

**class** trsfile.engine.file.**FileEngine**(*path*, *mode='x'*, ***options*)

This engine tries to save traces to disk in the most versatile and simple manner available. No known tools support this file format and serve only as an intermediate step to later convert it to a supported format.

This is can be useful when the trace length (number of samples) varies as this is often not supported in trace files.

After acquisition, the file can be converted to the proper format with the correct padding mode.

This engine supports the following options:

| Option | Description |
|---|---|
| headers | Dictionary containing zero or more headers, see *trsfile.common.Header* |

**class** trsfile.common.**Header**

All headers that are currently supported in the .trs file format as defined in the inspector manual (2018). The storage engine shall try to always store all headers regardless if they are used or not. However, some file formats will have no way of storing arbitrary headers. As such, optional headers can be dropped.

Some headers can be used by *trsfile.trace_set.TraceSet* or *trsfile.trace.Trace* to augment their functionality. An example of this is the *trsfile.trace.Trace.get_key()* method.

**class** trsfile.common.**SampleCoding**

Defines the encoding of all the samples in the trace. Bit 4 specifies if it is a float (1) or an integer (0), bits 0 to 3 specifies the length of the value. Finally, bits 5-7 are currently reserved and set to 000.

This class is just a simple lookup table.

**class** trsfile.common.**TracePadding**

Defines the padding mode of the samples in each trace. This can be helpful when not all traces will be the same length. This can be set in *trsfile.open()*, *trsfile.trs_open()*

| Mode | Description |
|---|---|
| NONE | No padding will be used and an exception will be thrown when traces are not of the same length. |
| PAD | All traces will be padded with zeroes to the maximum trace length. |
| TRUNCATE | All traces will be truncated to the minimum trace length. |
| AUTO | Traces will be clipped or padded in the best possible way the storage engine supports. This could mean data is lost which because retroactive padding is not supported. |

### 6.1.2 Common

**class** trsfile.common.**Header**

Bases: enum.Enum

All headers that are currently supported in the .trs file format as defined in the inspector manual (2018). The storage engine shall try to always store all headers regardless if they are used or not. However, some file formats will have no way of storing arbitrary headers. As such, optional headers can be dropped.

Some headers can be used by *trsfile.trace_set.TraceSet* or *trsfile.trace.Trace* to augment their functionality. An example of this is the *trsfile.trace.Trace.get_key()* method.

**ACQUISITION_COUPLING_OF_SCOPE = 86**

**ACQUISITION_DEVICE_ID = 89**

**ACQUISITION_FREQUENCY_FILTER = 91**

**ACQUISITION_INPUT_IMPEDANCE = 88**

**ACQUISITION_OFFSET_OF_SCOPE = 87**

**ACQUISITION_RANGE_FILTER = 92**

**ACQUISITION_RANGE_OF_SCOPE = 85**

**ACQUISITION_TYPE_FILTER = 90**

**DESCRIPTION = 71**

**EXTERNAL_CLOCK_BASE = 103**

**EXTERNAL_CLOCK_FREQUENCY = 102**

**EXTERNAL_CLOCK_MULTIPLIER = 98**

**EXTERNAL_CLOCK_PHASE_SHIFT = 99**

**EXTERNAL_CLOCK_RESAMPLER_ENABLED = 101**

**EXTERNAL_CLOCK_RESAMPLER_MASK = 100**

**EXTERNAL_CLOCK_THRESHOLD = 97**

**EXTERNAL_CLOCK_USED = 96**

**GO_LAST_TRACE = 106**

**INPUT_LENGTH = 110**

**INPUT_OFFSET = 107**

**KEY_LENGTH = 112**

**KEY_OFFSET = 109**

**LABEL_X = 73**

**LABEL_Y = 74**

**LENGTH_DATA = 68**

**LOGARITHMIC_SCALE = 78**

**NUMBER_SAMPLES = 66**

**NUMBER_TRACES = 65**

**NUMBER_VIEW = 104**

**OFFSET_X = 72**

**OUTPUT_LENGTH = 111**

**OUTPUT_OFFSET = 108**

**SAMPLE_CODING = 67**

**SCALE_X = 75**

**SCALE_Y = 76**

**TITLE_SPACE = 69**

**TRACE_BLOCK = 95**

**TRACE_OFFSET = 77**

**TRACE_OVERLAP = 105**

**TRACE_TITLE = 70**

**class** trsfile.common.**SampleCoding**
Bases: enum.Enum

Defines the encoding of all the samples in the trace. Bit 4 specifies if it is a float (1) or an integer (0), bits 0 to 3 specifies the length of the value. Finally, bits 5-7 are currently reserved and set to 000.

This class is just a simple lookup table.

**BYTE = 1**

**FLOAT = 20**

**INT = 4**

**SHORT = 2**

**class** trsfile.common.**TracePadding**
Bases: enum.Enum

Defines the padding mode of the samples in each trace. This can be helpful when not all traces will be the same length. This can be set in *trsfile.open()*, *trsfile.trs_open()*

| Mode | Description |
|---|---|
| NONE | No padding will be used and an exception will be thrown when traces are not of the same length. |
| PAD | All traces will be padded with zeroes to the maximum trace length. |
| TRUNCATE | All traces will be truncated to the minimum trace length. |
| AUTO | Traces will be clipped or padded in the best possible way the storage engine supports. This could mean data is lost which because retroactive padding is not supported. |

**AUTO = 3**

**NONE = 0**

**PAD = 1**

**TRUNCATE = 2**

### 6.1.3 Trace

**class** trsfile.trace.**Trace**(*sample_coding*, *samples*, *data=b”*, *title='trace'*, *headers={}*)

Bases: object

The *Trace* class behaves like a list object were each item in the list is a sample of the trace.

When a *Trace* is initialized the samples are (optionally) converted to a numpy.array depending on the current type of the samples and the provided sample_coding.

**get_input**()

**get_key**()

**get_output**()

### 6.1.4 TraceSet

**class** trsfile.trace_set.**TraceSet**(*path*, *mode='r'*, ***options*)

Bases: object

The *TraceSet* class behaves like a list object were each item in the list is a Trace.

Storing the *TraceSet* requires knowledge on the format which is resolved through the usage of storage engines (Engine).

**append**(*trace*)

**close**()

**extend**(*traces*)

**get_header**(*header*)

**get_headers**()

**insert**(*index*, *trace*)

**is_closed**()

**reverse**()

**update_header**(*header*, *value*)

**update_headers**(*headers*)

### 6.1.5 Storage Engines

The TraceSet behaves like a list (it is a list of Traces). Each Trace also behaves like a list (it is a list of samples). This is all on a conceptual level and the storage engine specifies how this conceptual model is translated to a specific file format. This behavior also makes it easy to convert from any (supported) file format to another one.



#### TrsEngine

**class** trsfile.engine.trs.**TrsEngine**(*path*, *mode='x'*, ***options*)

Bases: *trsfile.engine.engine.Engine*

This engine supports .trs files from Riscure as specified in the "Trace set coding" document in Inspector.

This engine supports the following options:

| Option | Description |
|---|---|
| headers | Dictionary containing zero or more headers, see *trsfile.common.Header* |
| live_update | Performs live update of the TRS file every N traces. True for updating after every trace and False for never. |
| padding_mode | Padding mode to use. The supported values are: *trsfile.common.TracePadding.NONE* and *trsfile.common.TracePadding.AUTO* (default) |

**close**()

Closes the open file handle if it is opened

**get_traces**(*index*)

Retrieves zero or more traces from the trace set

**Parameters** **index** (*slice, int*) – the slice or index that specifies which traces to get

**Returns** a list of zero or more traces from the trace set

**Return type** *Trace*, list[*Trace*]

**is_closed**()

Returns if the file backing the trace set is closed

**Returns** True if the file is closed, otherwise False

**Return type** boolean

**length**()

Returns the total number of traces

**Returns** total number of traces

**Return type** int

**set_traces**(*index*, *traces*)

Inserts zero or more traces into the trace set

**Parameters**

- **index** (*slice, int*) – the slice or index that specifies were to insert traces
- **traces** (*Trace, list[Trace]*) – zero or more traces to insert into the trace set

**Returns** None

**update_headers**(*headers*)
Updates zero or more headers

**Parameters** **headers** (*dict(Header, any)*) – dictionary of header, value pairs to update

**Returns** a list of the headers that changed

**Return type** list[*Header*]

### FileEngine

**class** trsfile.engine.file.**FileEngine**(*path*, *mode='x'*, ***options*)
Bases: *trsfile.engine.engine.Engine*

This engine tries to save traces to disk in the most versatile and simple manner available. No known tools support this file format and serve only as an intermediate step to later convert it to a supported format.

This is can be useful when the trace length (number of samples) varies as this is often not supported in trace files.

After acquisition, the file can be converted to the proper format with the correct padding mode.

This engine supports the following options:

| Option | Description |
|---|---|
| headers | Dictionary containing zero or more headers, see *trsfile.common.Header* |

**INFO_FILE = 'traceset.pickle'**

**close**()
Closes the file backing the trace set. It also could perform the final writes to synchronize the file with the trace set.

**Returns** None

**del_traces**(*index*)
Deletes zero or more traces from the trace set

**Parameters** **index** (*slice, int*) – the slice or index that specifies which traces to delete

**Returns** None

**get_traces**(*index*)
Retrieves zero or more traces from the trace set

**Parameters** **index** (*slice, int*) – the slice or index that specifies which traces to get

**Returns** a list of zero or more traces from the trace set

**Return type** *Trace*, list[*Trace*]

**is_closed**()
Returns if the file backing the trace set is closed

**Returns** True if the file is closed, otherwise False

**Return type** boolean

**length**()
Returns the total number of traces

**Returns** total number of traces

**Return type** int

**set_traces**(*index*, *traces*)
Inserts zero or more traces into the trace set

**Parameters**

- **index** (*slice, int*) – the slice or index that specifies were to insert traces
- **traces** (*Trace, list[Trace]*) – zero or more traces to insert into the trace set

**Returns** None

**update_headers**(*headers*)
Updates zero or more headers

**Parameters** **headers** (*dict(Header, any)*) – dictionary of header, value pairs to update

**Returns** a list of the headers that changed

**Return type** list[*Header*]

### Engine

**class** trsfile.engine.engine.**Engine**(*path*, *mode='x'*, ***options*)
Bases: object

**close**()
Closes the file backing the trace set. It also could perform the final writes to synchronize the file with the trace set.

**Returns** None

**del_traces**(*index*)
Deletes zero or more traces from the trace set

**Parameters** **index** (*slice, int*) – the slice or index that specifies which traces to delete

**Returns** None

**get_traces**(*index*)
Retrieves zero or more traces from the trace set

**Parameters** **index** (*slice, int*) – the slice or index that specifies which traces to get

**Returns** a list of zero or more traces from the trace set

**Return type** *Trace*, list[*Trace*]

**is_closed**()
Returns if the file backing the trace set is closed

**Returns** True if the file is closed, otherwise False

**Return type** boolean

**is_read_only**()
Returns if the trace set is read-only

**Returns** True if the file is read-only, otherwise False

**Return type** boolean

**length**()
Returns the total number of traces

**Returns** total number of traces

**Return type** int

**read_only = False**

**set_traces**(*index*, *traces*)
Inserts zero or more traces into the trace set

**Parameters**

- **index** (*slice, int*) – the slice or index that specifies were to insert traces
- **traces** (Trace, *list[Trace]*) – zero or more traces to insert into the trace set

**Returns** None

**update_header**(*header*, *value*)
Updates one specific header

**Parameters**

- **header** (Header) – header to update
- **value** (*any*) – value of the header to update

**Returns** a list of the headers that changed

**Return type** list[*Header*]

**update_headers**(*headers*)
Updates zero or more headers

**Parameters** **headers** (*dict(Header, any)*) – dictionary of header, value pairs to update

**Returns** a list of the headers that changed

**Return type** list[*Header*]

# Python Module Index

## t

# Index