npTDMS Documentation

Release 1.3.0

Adam Reeve

May 09, 2021

Contents

npTDMS is a cross-platform Python package for reading and writing TDMS files as produced by LabVIEW, and is built on top of the numpy package. Data is read from TDMS files as numpy arrays, and npTDMS also allows writing numpy arrays to TDMS files.

CHAPTER 1

Contents

1.1 Installation and Quick Start

npTDMS is available from the Python Package Index, so the easiest way to install it is by running:

```
pip install npTDMS
```

There are optional features available that require additional dependencies. These are `hdf` for hdf export, `pandas` for pandas DataFrame export, and `thermocouple_scaling` for reading files that use thermocouple scalings. You can specify these extra features when installing npTDMS to also install the dependencies they require:

```
pip install npTDMS[hdf,pandas,thermocouple_scaling]
```

Alternatively, after downloading the source code you can extract it and change into the new directory, then run:

```
python setup.py install
```

Typical usage when reading a TDMS file might look like:

```
from nptdms import TdmsFile

tdms_file = TdmsFile.read("path_to_file.tdms")
for group in tdms_file.groups():
    group_name = group.name
    for channel in group.channels():
        channel_name = channel.name
        # Access dictionary of properties:
        properties = channel.properties
        # Access numpy array of data for channel:
        data = channel[:]
        # Access a subset of data
        data_subset = channel[100:200]
```

Or to access a channel by group name and channel name directly:

```
group = tdms_file[group_name]
channel = group[channel_name]
```

The `TdmsFile.read` method reads all data into memory immediately. When you are working with large TDMS files or don't need to read all channel data, you can instead use `TdmsFile.open`. This is more memory efficient but accessing data can be slower:

```
with TdmsFile.open("path_to_file.tdms"):
    channel = tdms_file[group_name][channel_name]
    channel_data = channel[:]
```

npTDMS also has rudimentary support for writing TDMS files. Using npTDMS to write a TDMS file looks like:

```
from nptdms import TdmsWriter, ChannelObject
import numpy

with TdmsWriter("path_to_file.tdms") as tdms_writer:
    data_array = numpy.linspace(0, 1, 10)
    channel = ChannelObject('Group', 'Channel1', data_array)
    tdms_writer.write_segment([channel])
```

1.2 Reading TDMS files

To read a TDMS file, create an instance of the *TdmsFile* class using one of the static *nptdms.TdmsFile.read()* or *nptdms.TdmsFile.open()* methods, passing the path to the file, or an already opened file. The *read()* method will read all channel data immediately:

```
tdms_file = TdmsFile.read("my_file.tdms")
```

If using the *open()* method, only the file metadata will be read initially, and the returned *TdmsFile* object should be used as a context manager to keep the file open and allow channel data to be read on demand:

```
with TdmsFile.open("my_file.tdms") as tdms_file:
    # Use tdms_file
    ...
```

Using an instance of *TdmsFile*, groups within the file can be accessed by indexing into the file with a group name, or all groups can be retrieved as a list with the *groups()* method:

```
group = tdms_file["group name"]
all_groups = tdms_file.groups()
```

A group is an instance of the *TdmsGroup* class, and can contain multiple channels of data. You can access channels in a group by indexing into the group with a channel name or retrieve all channels as a list with the *channels()* method:

```
channel = group["channel name"]
all_group_channels = group.channels()
```

Channels are instances of the *TdmsChannel* class and act like arrays. They can be indexed with an integer index to retrieve a single value or with a slice to retrieve all data or a subset of data as a numpy array:

```
all_channel_data = channel[:]
data_subset = channel[100:200]
first_channel_value = channel[0]
```

If the channel contains waveform data and has the `wf_start_offset` and `wf_increment` properties, you can get an array of relative time values for the data using the *time_track()* method:

```
time = channel.time_track()
```

In addition, if the `wf_start_time` property is set, you can pass `absolute_time=True` to get an array of absolute times in UTC.

A TDMS file, group and channel can all have properties associated with them, so each of the *TdmsFile*, *TdmsGroup* and *TdmsChannel* classes provide access to these properties as a dictionary using their `properties` attribute:

```
# Iterate over all items in the file properties and print them
for name, value in tdms_file.properties.items():
    print("{0}: {1}".format(name, value))

# Get a single property value from the file
property_value = tdms_file.properties["my_property_name"]

# Get a group property
property_value = tdms_file["group name"].properties["group_property_name"]

# Get a channel property
property_value = tdms_file["group name"]["channel name"].properties["channel_property_
↪name"]
```

In addition to the properties dictionary, all groups and channels have `name` and `path` attributes. The name is the human readable name of the group or channel, and the path is the full path to the TDMS object, which includes the group name for channels:

```
group = tdms_file["group name"]
channel = group["channel name"]
print(group.name)     # Prints "group name"
print(group.path)     # Prints "/'group name'"
print(channel.name)   # Prints "channel name"
print(channel.path)   # Prints "/'group name'/'channel name'"
```

1.2.1 Reading large files

TDMS files are often too large to easily fit in memory so npTDMS offers a few ways to deal with this. A TDMS file can be opened for reading without reading all the data immediately using the static *open()* method, then channel data is read as required:

```
with TdmsFile.open(tdms_file_path) as tdms_file:
    channel = tdms_file[group_name][channel_name]
    all_channel_data = channel[:]
    data_subset = channel[100:200]
```

TDMS files are written in multiple segments, where each segment can in turn have multiple chunks of data. When accessing a value or a slice of data in a channel, npTDMS will read whole chunks at a time. npTDMS also allows streaming data from a file chunk by chunk using *nptdms.TdmsFile.data_chunks()*. This is a generator that produces instances of *DataChunk*. For example, to compute the mean of a channel:

```
channel_sum = 0.0
channel_length = 0
with TdmsFile.open(tdms_file_path) as tdms_file:
    for chunk in tdms_file.data_chunks():
        channel_chunk = chunk[group_name][channel_name]
        channel_length += len(channel_chunk)
        channel_sum += channel_chunk[:].sum()
channel_mean = channel_sum / channel_length
```

This approach can be useful to stream TDMS data to another format on disk or into a data store. It's also possible to stream data chunks for a single channel using *nptdms.TdmsChannel.data_chunks()*:

```
with TdmsFile.open(tdms_file_path) as tdms_file:
    channel = tdms_file[group_name][channel_name]
    for chunk in channel.data_chunks():
        channel_chunk_data = chunk[:]
```

If you don't need to read the channel data at all and only need to read metadata, you can also use the static *read_metadata()* method:

```
tdms_file = TdmsFile.read_metadata(tdms_file_path)
```

In cases where you need to work with large arrays of channel data as if all data was in memory, you can also pass the `memmap_dir` argument when reading a file. This will read data into memory mapped numpy arrays on disk, and your operating system will then page data in and out of memory as required:

```
with tempfile.TemporaryDirectory() as temp_memmap_dir:
    tdms_file = TdmsFile.read(tdms_file_path, memmap_dir=temp_memmap_dir)
```

1.2.2 Timestamps

By default, timestamps are read as numpy datetime64 objects with microsecond precision. However, TDMS files are capable of storing times with a precision of 2^{-64} seconds. If you need access to this higher precision timestamp data, all methods for constructing a *TdmsFile* accept a `raw_timestamps` parameter. When this is true, any timestamp properties will be returned as a *TdmsTimestamp* object. This has `seconds` and `second_fractions` attributes which are the number of seconds since the epoch 1904-01-01 00:00:00 UTC, and a positive number of 2^{-64} fractions of a second. This class has methods for converting to a numpy datetime64 object or datetime.datetime. For example:

```
>>> timestamp = channel.properties['wf_start_time']
>>> timestamp
TdmsTimestamp(3670436596, 11242258187010646344)
>>> timestamp.seconds
3670436596
>>> timestamp.second_fractions
11242258187010646344
>>> print(timestamp)
2020-04-22T21:43:16.609444
>>> timestamp.as_datetime64('ns')
numpy.datetime64('2020-04-22T21:43:16.609444037')
>>> timestamp.as_datetime()
datetime.datetime(2020, 4, 22, 21, 43, 16, 609444)
```

When setting `raw_timestamps` to true, channels with timestamp data will return data as a *TimestampArray* rather than as a `datetime64` array. This is a subclass of `numpy.ndarray` with additional properties and

an *as_datetime64()* method for converting to a datetime64 array, and elements in the array are returned as *TdmsTimestamp* instances:

```
>>> timestamp_data = channel[:]
>>> timestamp_data
TimestampArray([(8942011409353408512, 3670436596), (9643130391967563776, 3670436596),
                (9661619779500244992, 3670436596), ..., (1366710545511612416,␣
↪3670502040),
                (1476995959824056320, 3670502040), (1587685994415521792, 3670502040)],
               dtype=[('second_fractions', '<u8'), ('seconds', '<i8')])
>> timestamp_data[0]
TdmsTimestamp(3670436596, 8942011409353408512)
>>> timestamp_data.seconds
array([3670436596, 3670436596, 3670436596, ..., 3670502040, 3670502040, 3670502040],␣
↪dtype=int64)
>>> timestamp_data.second_fractions
array([8942011409353408512, 9643130391967563776, 9661619779500244992, ...,␣
↪1366710545511612416,
       1476995959824056320, 1587685994415521792], dtype=uint64)
>>> timestamp_data.as_datetime64('us')
array(['2020-04-22T21:43:16.484747', '2020-04-22T21:43:16.522755', '2020-04-
↪22T21:43:16.523757', ...,
       '2020-04-23T15:54:00.074089', '2020-04-23T15:54:00.080068', '2020-04-
↪23T15:54:00.086068'],
      dtype='datetime64[us]')
```

Timestamps in TDMS files are stored in UTC time and npTDMS does not do any timezone conversions. If timestamps need to be converted to the local timezone, the arrow package is recommended. For example:

```
import datetime
import arrow

timestamp = channel.properties['wf_start_time']
local_time = arrow.get(timestamp.astype(datetime.datetime)).to('local')
print(local_time.format())
```

Here we first convert the numpy datetime64 object to Python's built in datetime type before converting it to an arrow time, then convert it from UTC to the local timezone.

1.2.3 Scaled data

The TDMS format supports different ways of scaling data, and DAQmx raw data in particular is usually scaled. The data retrieved from a *TdmsChannel* has scaling applied. If you have opened a TDMS file with *read()*, you can access the raw unscaled data with the *raw_data* property of a channel. Note that DAQmx channels may have multiple raw scalers rather than a single raw data channel, in which case you need to use the *raw_scaler_data* property to access the raw data as a dictionary of scaler id to raw data array.

When you've opened a TDMS file with *open()*, you instead need to use *read_data*, passing `scaled=False`:

```
with TdmsFile.open(tdms_file_path) as tdms_file:
    channel = tdms_file[group_name][channel_name]
    unscaled_data = channel.read_data(scaled=False)
```

This will return an array of raw data, or a dictionary of scaler id to raw scaler data for DAQmx data.

1.2.4 Conversion to other formats

npTDMS has convenience methods to convert data to Pandas DataFrames or HDF5 files. The *TdmsFile* class has *as_dataframe()* and *as_hdf()* methods to convert a whole file to a DataFrame or HDF5 file. In addition there is an *as_dataframe()* method on *TdmsGroup* and an *as_dataframe()* method on *TdmsChannel* for converting a single group or channel to a Pandas DataFrame.

1.2.5 Thread safety

When a TDMS file is opened with *open()*, the returned *TdmsFile* object is not thread-safe and reading from it concurrently will result in undefined behaviour. If you need to read from the same file concurrently you should open a new *TdmsFile* per thread.

When a TDMS file is read with *read()*, the returned *TdmsFile* is safe to read from concurrently as all data has been read from the file upfront.

1.3 Writing TDMS files

npTDMS has rudimentary support for writing TDMS files. The full set of optimisations supported by the TDMS file format for speeding up the writing of files and minimising file size are not implemented by npTDMS, but the basic functionality required to write TDMS files is available.

To write a TDMS file, the *TdmsWriter* class is used, which should be used as a context manager. The *__init__()* method accepts the path to the file to create, or a file that has already been opened in binary write mode:

```
with TdmsWriter("my_file.tdms") as tdms_writer:
    # write data
```

The *write_segment()* method is used to write a segment of data to the TDMS file. Because the TDMS file format is designed for streaming data applications, it supports writing data one segment at a time as data becomes available. If you don't require this functionality you can simple call `write_segment` once with all of your data.

The *write_segment()* method takes a list of objects, each of which must be an instance of one of:

- *nptdms.RootObject*. This is the TDMS root object, and there may only be one root object in a segment.
- *nptdms.GroupObject*. This is used to group the channel objects.
- *nptdms.ChannelObject*. An object that contains data.
- *nptdms.TdmsGroup* or *nptdms.TdmsChannel*. A TDMS object that was read from a TDMS file using *nptdms.TdmsFile*.

Each of *RootObject*, *GroupObject* and *ChannelObject* may optionally have properties associated with them, which are passed into the `__init__` method as a dictionary. The data types supported as property values are:

- Integers
- Floating point values
- Strings
- datetime or numpy datetime64 objects
- Boolean values

For more control over the data type used to represent a property value, for example to use an unsigned integer type, you can pass an instance of one of the data types from the *nptdms.types* module.

A complete example of writing a TDMS file with various object types and properties is given below:

```
from nptdms import TdmsWriter, RootObject, GroupObject, ChannelObject

root_object = RootObject(properties={
    "prop1": "foo",
    "prop2": 3,
})
group_object = GroupObject("group_1", properties={
    "prop1": 1.2345,
    "prop2": False,
})
data = np.array([1.0, 2.0, 3.0, 4.0, 5.0])
channel_object = ChannelObject("group_1", "channel_1", data, properties={})

with TdmsWriter("my_file.tdms") as tdms_writer:
    # Write first segment
    tdms_writer.write_segment([
        root_object,
        group_object,
        channel_object])
    # Write another segment with more data for the same channel
    more_data = np.array([6.0, 7.0, 8.0, 9.0, 10.0])
    channel_object = ChannelObject("group_1", "channel_1", more_data, properties={})
    tdms_writer.write_segment([channel_object])
```

You could also read a TDMS file and then re-write it by passing *TdmsGroup* and *TdmsChannel* instances to the `write_segment` method. If you want to only copy certain channels for example, you could do something like:

```
from nptdms import TdmsFile, TdmsWriter, RootObject

original_file = TdmsFile("original_file.tdms")
original_groups = original_file.groups()
original_channels = [chan for group in original_groups for chan in group.channels()]

with TdmsWriter("copied_file.tdms") as copied_file:
    root_object = RootObject(original_file.properties)
    channels_to_copy = [chan for chan in original_channels if include_channel(chan)]
    copied_file.write_segment([root_object] + original_groups + channels_to_copy)
```

Note that this isn't suitable for copying channels with scaled data, as the channel data will already have scaling applied.

1.4 npTDMS API Reference

1.4.1 Reading TDMS Files

class nptdms.**TdmsFile**

Reads and stores data from a TDMS file.

There are two main ways to create a new TdmsFile object. TdmsFile.read will read all data into memory:

```
tdms_file = TdmsFile.read(tdms_file_path)
```

or you can use TdmsFile.open to read file metadata but not immediately read all data, for cases where a file is too large to easily fit in memory or you don't need to read data for all channels:

```
with TdmsFile.open(tdms_file_path) as tdms_file:
    # Use tdms_file
    ...
```

This class acts like a dictionary, where the keys are names of groups in the TDMS files and the values are TdmsGroup objects. A TdmsFile can be indexed by group name to access a group within the TDMS file, for example:

```
tdms_file = TdmsFile.read(tdms_file_path)
group = tdms_file[group_name]
```

Iterating over a TdmsFile produces the names of groups in this file, or you can use the groups method to directly access all groups:

```
for group in tdms_file.groups():
    # Use group
    ...
```

static read(*file*, *raw_timestamps=False*, *memmap_dir=None*)

Creates a new TdmsFile object and reads all data in the file

Parameters

- **file** – Either the path to the tdms file to read as a string or pathlib.Path, or an already opened file.
- **raw_timestamps** – By default TDMS timestamps are read as numpy datetime64 but this loses some precision. Setting this to true will read timestamps as a custom TdmsTimestamp type.
- **memmap_dir** – The directory to store memory mapped data files in, or None to read data into memory. The data files are created as temporary files and are deleted when the channel data is no longer used. tempfile.gettempdir() can be used to get the default temporary file directory.

static open(*file*, *raw_timestamps=False*, *memmap_dir=None*)

Creates a new TdmsFile object and reads metadata, leaving the file open to allow reading channel data

Parameters

- **file** – Either the path to the tdms file to read as a string or pathlib.Path, or an already opened file.
- **raw_timestamps** – By default TDMS timestamps are read as numpy datetime64 but this loses some precision. Setting this to true will read timestamps as a custom TdmsTimestamp type.
- **memmap_dir** – The directory to store memory mapped data files in, or None to read data into memory. The data files are created as temporary files and are deleted when the channel data is no longer used. tempfile.gettempdir() can be used to get the default temporary file directory.

static read_metadata(*file*, *raw_timestamps=False*)

Creates a new TdmsFile object and only reads the metadata

Parameters

- **file** – Either the path to the tdms file to read as a string or pathlib.Path, or an already opened file.
- **raw_timestamps** – By default TDMS timestamps are read as numpy datetime64 but this loses some precision. Setting this to true will read timestamps as a custom TdmsTimestamp type.

groups()
Returns a list of the groups in this file

Return type List of TdmsGroup.

properties
Return the properties of this file as a dictionary

These are the properties associated with the root TDMS object.

as_dataframe(*time_index=False*, *absolute_time=False*, *scaled_data=True*)
Converts the TDMS file to a DataFrame. DataFrame columns are named using the TDMS object paths.

Parameters

- **time_index** – Whether to include a time index for the dataframe.
- **absolute_time** – If time_index is true, whether the time index values are absolute times or relative to the start time.
- **scaled_data** – By default the scaled data will be used. Set to False to use raw unscaled data. For DAQmx data, there will be one column per DAQmx raw scaler and column names will include the scale id.

Returns The full TDMS file data.

Return type pandas.DataFrame

as_hdf(*filepath*, *mode='w'*, *group='/'*)
Converts the TDMS file into an HDF5 file

Parameters

- **filepath** – The path of the HDF5 file you want to write to.
- **mode** – The write mode of the HDF5 file. This can be 'w' or 'a'
- **group** – A group in the HDF5 file that will contain the TDMS data.

data_chunks()
A generator that streams chunks of data from disk. This method may only be used when the TDMS file was opened without reading all data immediately.

Return type Generator that yields *DataChunk* objects

close()
Close the underlying file if it was opened by this TdmsFile

If this TdmsFile was initialised with an already open file then the reference to it is released but the file is not closed.

class nptdms.**TdmsGroup**
Represents a group of channels in a TDMS file.

This class acts like a dictionary, where the keys are names of channels in the group and the values are TdmsChannel objects. A TdmsGroup can be indexed by channel name to access a channel in this group, for example:

```
channel = group[channel_name]
```

Iterating over a TdmsGroup produces the names of channels in this group, or you can use the channels method to directly access all channels:

```
for channel in group.channels():
    # Use channel
    ...
```

Variables **properties** – Dictionary of TDMS properties defined for this group.

path
Path to the TDMS object for this group

name
The name of this group

channels()
The list of channels in this group

Return type A list of TdmsChannel

as_dataframe(*time_index=False*, *absolute_time=False*, *scaled_data=True*)
Converts the TDMS group to a DataFrame. DataFrame columns are named using the channel names.

Parameters

- **time_index** – Whether to include a time index for the dataframe.
- **absolute_time** – If time_index is true, whether the time index values are absolute times or relative to the start time.
- **scaled_data** – By default the scaled data will be used. Set to False to use raw unscaled data. For DAQmx data, there will be one column per DAQmx raw scaler and column names will include the scale id.

Returns The TDMS object data.

Return type pandas.DataFrame

class nptdms.**TdmsChannel**
Represents a data channel in a TDMS file.

This class acts like an array, you can get the length of a channel using `len(channel)`, and can iterate over values in the channel using a for loop, or index into a channel using an integer index to get a single value:

```
for value in channel:
    # Use value
    ...
first_value = channel[0]
```

Or you can index using a slice to retrieve a range of data as a numpy array. To get all data in this channel as a numpy array:

```
all_data = channel[:]
```

Or to retrieve a subset of data:

```
data_subset = channel[start:stop]
```

Variables **properties** – Dictionary of TDMS properties defined for this channel, for example the start time and time increment for waveforms.

path
Path to the TDMS object for this channel

name
The name of this channel

group_name
The name of the group that contains this channel

dtype
NumPy data type of the channel data

For data with a scaling this is the data type of the scaled data

Return type numpy.dtype

data
If the TdmsFile was created by reading all data, this property provides direct access to the numpy array containing the data for this channel.

Indexing into the channel with a slice should be preferred to using this property, for example:

```
channel_data = channel[:]
```

raw_data
If the TdmsFile was created by reading all data, this property provides direct access to the numpy array of raw, unscaled data. For unscaled objects this is the same as the data property.

raw_scaler_data
If the TdmsFile was created by reading all data, this property provides direct access to the numpy array of raw DAQmx scaler data as a dictionary mapping from scale id to raw data arrays.

data_chunks()
A generator that streams chunks data for this channel from disk. This method may only be used when the TDMS file was opened without reading all data immediately.

Return type Generator that yields *ChannelDataChunk* objects

read_data(*offset=0*, *length=None*, *scaled=True*)
Reads data for this channel from the TDMS file and returns it as a numpy array

Indexing into the channel with a slice should be preferred over using this method, but this method is needed if you want to read raw, unscaled data.

Parameters

- **offset** – Initial position to read data from.
- **length** – Number of values to attempt to read. Fewer values will be returned if attempting to read beyond the end of the available data.
- **scaled** – By default scaling will be applied to the returned data. Set this parameter to False to return raw unscaled data. For DAQmx data a dictionary of scaler id to raw scaler data will be returned.

time_track(*absolute_time=False*, *accuracy='ns'*)
Return an array of time or the independent variable for this channel

This depends on the object having the wf_increment and wf_start_offset properties defined. Note that wf_start_offset is usually zero for time-series data. If you have time-series data channels with different start times, you should use the absolute time or calculate the time offsets using the wf_start_time property.

For larger timespans, the accuracy setting should be set lower. The default setting is 'ns', which has a timespan of [1678 AD, 2262 AD]. For the exact ranges, refer to http://docs.scipy.org/doc/numpy/reference/arrays.datetime.html section "Datetime Units".

Parameters

- **absolute_time** – Whether the returned time values are absolute times rather than relative to the start time. If true, the wf_start_time property must be set.
- **accuracy** – The accuracy of the returned datetime64 array.

Return type NumPy array.

Raises KeyError if required properties aren't found

as_dataframe(*time_index=False*, *absolute_time=False*, *scaled_data=True*)

Converts the TDMS channel to a DataFrame. The DataFrame column is named using the channel path.

Parameters

- **time_index** – Whether to include a time index for the dataframe.
- **absolute_time** – If time_index is true, whether the time index values are absolute times or relative to the start time.
- **scaled_data** – By default the scaled data will be used. Set to False to use raw unscaled data. For DAQmx data, there will be one column per DAQmx raw scaler and column names will include the scale id.

Returns The TDMS object data.

Return type pandas.DataFrame

class nptdms.**DataChunk**

A chunk of data in a TDMS file

Can be indexed by group name to get the data for a group in this channel, which can then be indexed by channel name to get the data for a channel in this chunk. For example:

```
group_chunk = data_chunk[group_name]
channel_chunk = group_chunk[channel_name]
```

groups()

Returns chunks of data for all groups

Return type List of *GroupDataChunk*

class nptdms.**GroupDataChunk**

A chunk of data for a group in a TDMS file

Can be indexed by channel name to get the data for a channel in this chunk. For example:

```
channel_chunk = group_chunk[channel_name]
```

Variables **name** – Name of the group

channels()

Returns chunks of channel data for all channels in this group

Return type List of *ChannelDataChunk*

class nptdms.**ChannelDataChunk**
A chunk of data for a channel in a TDMS file

Is an array-like object that supports indexing to access data, for example:

```
chunk_length = len(channel_data_chunk)
chunk_data = channel_data_chunk[:]
```

Variables

- **name** – Name of the channel
- **offset** – Starting index of this chunk of data in the entire channel

1.4.2 Writing TDMS Files

class nptdms.**TdmsWriter**(*file*, *mode='w'*)
Writes to a TDMS file.

A TdmsWriter should be used as a context manager, for example:

```
with TdmsWriter(path) as tdms_writer:
    tdms_writer.write_segment(segment_data)
```

__init__(*file*, *mode='w'*)
Initialise a new TDMS writer

Parameters

- **file** – Either the path to the tdms file to open or an already opened file.
- **mode** – Either 'w' to open a new file or 'a' to append to an existing TDMS file.

write_segment(*objects*)
Write a segment of data to a TDMS file

Parameters **objects** – A list of TdmsObject instances to write

class nptdms.**RootObject**(*properties=None*)
The root TDMS object containing properties for the TDMS file

__init__(*properties=None*)
Initialise a new GroupObject

Parameters **properties** – A dictionary mapping property names to their value.

path
The string representation of the root path

class nptdms.**GroupObject**(*group*, *properties=None*)
A TDMS object for a group

__init__(*group*, *properties=None*)
Initialise a new GroupObject

Parameters

- **group** – The name of this group.
- **properties** – A dictionary mapping property names to their value.

path
The string representation of this group's path

class nptdms.**ChannelObject**(*group*, *channel*, *data*, *properties=None*)
A TDMS object for a channel with data

__init__(*group*, *channel*, *data*, *properties=None*)
Initialise a new ChannelObject

Parameters

- **group** – The name of the group this channel is in.
- **channel** – The name of this channel.
- **data** – 1-D Numpy array of data to be written.
- **properties** – A dictionary mapping property names to their value.

path
The string representation of this channel's path

1.4.3 Data Types for Property Values

class nptdms.types.**Int8**(*value*)

class nptdms.types.**Int16**(*value*)

class nptdms.types.**Int32**(*value*)

class nptdms.types.**Int64**(*value*)

class nptdms.types.**Uint8**(*value*)

class nptdms.types.**Uint16**(*value*)

class nptdms.types.**Uint32**(*value*)

class nptdms.types.**Uint64**(*value*)

class nptdms.types.**SingleFloat**(*value*)

class nptdms.types.**DoubleFloat**(*value*)

class nptdms.types.**String**(*value*)

class nptdms.types.**Boolean**(*value*)

class nptdms.types.**TimeStamp**(*value*)

1.4.4 Timestamps

class nptdms.timestamp.**TdmsTimestamp**(*seconds*, *second_fractions*)
A Timestamp from a TDMS file

The TDMS format stores timestamps as a signed number of seconds since the epoch 1904-01-01 00:00:00 UTC and number of positive fractions (2^-64) of a second.

Variables

- **seconds** – Seconds since the epoch as a signed integer
- **second_fractions** – A positive number of 2^-64 fractions of a second

as_datetime64(*resolution='us'*)
Convert this timestamp to a numpy datetime64 object

Parameters **resolution** – The resolution of the datetime64 object to create as a numpy unit code. Must be one of 's', 'ms', 'us', 'ns' or 'ps'

as_datetime()
Convert this timestamp to a Python datetime.datetime object

class nptdms.timestamp.**TimestampArray**
A numpy array of TDMS timestamps

Indexing into a TimestampArray returns TdmsTimestamp objects.

seconds
The number of seconds since the TDMS epoch (1904-01-01 00:00:00 UTC) as a numpy array

second_fractions
The number of 2**-64 fractions of a second as a numpy array

as_datetime64(*resolution='us'*)
Convert to an array of numpy datetime64 objects

Parameters **resolution** – The resolution of the datetime64 objects to create as a numpy unit code. Must be one of 's', 'ms', 'us', 'ns' or 'ps'

1.4.5 Indices and Tables

- genindex
- modindex
- search

1.5 The tdmsinfo Command

npTDMS comes with a command line program, `tdmsinfo`, which lists the contents of a TDMS file. Usage looks like:

```
tdmsinfo [--properties] tdms_file
```

Passing the `--properties` or `-p` argument will include TDMS object properties in the printed information as well as the data type and length of channels.

The output of tdmsinfo including properties will look something like:

```
/
  properties:
    name: test_file
  /'group_1'
    properties:
      group_property: property value
    /'group_1'/'channel_1'
      data type: Uint32
      length: 2000
      properties:
        wf_start_time: 2016-12-3014:56:00+00:00
        wf_increment: 0.0005
        wf_samples: 200
```

There is also a `--debug` or `-d` argument that will output debug information to stderr, which can be useful when debugging a problem with a TDMS file.

1.6 Limitations

npTDMS currently doesn't support reading TDMS files with XML headers (TDM files), or files with extended precision floating point data.

Python Module Index

n

Index

S

T

U

W