waveline

Release 0.2.0

Lukas Berbuer (Vallen Systeme GmbH)

Dec 23, 2020

LIBRARY DOCUMENTATION

Library to easily interface with Vallen Systeme WaveLine™ devices using the public APIs.

waveline.conditionwave	Module for conditionWave device.
waveline.spotwave	Module for spotWave device.

CHAPTER

ONE

WAVELINE.CONDITIONWAVE

Module for conditionWave device.

All device-related functions are exposed by the *ConditionWave* class.

Classes

ChannelSettings(range_volts, ...)	Channel settings.
ConditionWave(address)	Interface for conditionWave device.
FilterSettings(highpass, lowpass[, order])	Filter settings.

1.1 ChannelSettings

class waveline.conditionwave.**ChannelSettings**(*range_volts*, *decimation_factor*, *filter_settings*)

Channel settings.

__init__(*range_volts*, *decimation_factor*, *filter_settings*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(range_volts, decimation_factor, ...)	Initialize self.

range_volts: **float**
Input range in volts

decimation_factor: **int**
Decimation factor

filter_settings: ***waveline.conditionwave.FilterSettings***
Filter settings

1.2 ConditionWave

class waveline.conditionwave.**ConditionWave**(*address*)

Interface for conditionWave device.

The device is controlled via TCP/IP:

- Control port: 5432
- Streaming ports: 5433 for channel 1 and 5434 for channel 2

The interface is asynchronous and using `asyncio` for TCP/IP communication. This is especially beneficial for this kind of streaming applications, where most of the time the app is waiting for more data packets (read more). Please refer to the examples for implementation details.

__init__(*address*)

Initialize device.

Parameters **address** (`str`) – IP address of device. Use the method *discover* to get IP addresses of available conditionWave devices.

Returns Instance of *ConditionWave*

Example

There are two ways constructing and using the *ConditionWave* class:

1. Without context manager, manually calling the *connect* and *close* method:

```
>>> async def main():
>>>     cw = waveline.ConditionWave("192.168.0.100")
>>>     await cw.connect()
>>>     print(await cw.get_info())
>>>     ...
>>>     await cw.close()
>>> asyncio.run(main())
```

2. Using the async context manager:

```
>>> async def main():
>>>     async with waveline.ConditionWave("192.168.0.100") as cw:
>>>         print(await cw.get_info())
>>>         ...
>>> asyncio.run(main())
```

Methods

__init__(address)	Initialize device.
close()	Close connection.
connect()	Connect to device.
discover([timeout])	Discover conditionWave devices in network.
get_buffersize()	Get buffer size in bytes (only during acquisition).
get_info()	Get device information.
get_temperature()	Get current device temperature in °C (only during acquisition).

continues on next page

Table 3 – continued from previous page

set_decimation(factor)	Set decimation factor.
set_filter([highpass, lowpass, order])	Apply IIR filter settings.
set_range(range_volts)	Set input range.
start_acquisition()	Start data acquisition.
stop_acquisition()	Stop data acquisition.
stream(channel, blocksize)	Async generator to stream channel data.

Attributes

CHANNELS	Valid channels
DEFAULT_SETTINGS	Default settings
MAX_SAMPLERATE	Maximum sampling rate in Hz
PORT	Control port number
RANGES	Mapping of range in volts and range index
connected	Check if connected to device.
decimation	Decimation factor.
filter_settings	Filter settings.
input_range	Input range in volts.

CHANNELS = (1, 2)
Valid channels

MAX_SAMPLERATE = 10000000
Maximum sampling rate in Hz

RANGES = {0.05: 0, 5.0: 1}
Mapping of range in volts and range index

PORT = 5432
Control port number

DEFAULT_SETTINGS = ChannelSettings(range_volts=0.05, decimation_factor=1, filter_setti
Default settings

classmethod discover(*timeout=0.5*)
Discover conditionWave devices in network.

Parameters **timeout** (float) – Timeout in seconds

Return type List[str]

Returns List of IP adresses

property connected
Check if connected to device.

Return type bool

property input_range
Input range in volts.

Return type float

property decimation
Decimation factor.

Return type int

property filter_settings
Filter settings.

Return type *FilterSettings*

async connect()
Connect to device.

async close()
Close connection.

async get_info()
Get device information.

Return type str

async set_range(*range_volts*)
Set input range.

Parameters **range_volts** (float) – Input range in volts (0.05, 5)

async set_decimation(*factor*)
Set decimation factor.

Parameters **factor** (int) – Decimation factor [1, 500]

async set_filter(*highpass=None*, *lowpass=None*, *order=8*)
Apply IIR filter settings.

Default is bypass.

Parameters

- **highpass** (Optional[float]) – Highpass frequency in Hz
- **lowpass** (Optional[float]) – Lowpass frequency in Hz
- **order** (int) – IIR filter order

async start_acquisition()
Start data acquisition.

stream(*channel*, *blocksize*)
Async generator to stream channel data.

Parameters

- **channel** (int) – Channel number [0, 1]
- **blocksize** (int) – Number of samples per block

Yields Tuple of datetime and numpy array (in volts)

Example

```
>>> async with waveline.ConditionWave("192.168.0.100") as cw:
>>>     # apply settings
>>>     await cw.set_range(0.05)
>>>     await cw.set_filter(100e3, 500e3, 8)
>>>     # start daq and streaming
>>>     await cw.start_acquisition()
>>>     async for timestamp, block in cw.stream(channel=1, blocksize=65536):
>>>         # do something with the data
>>>         ...
```

async stop_acquisition()
Stop data acquisition.

get_temperature()
Get current device temperature in °C (only during acquisition).

Return type Optional[int]

get_buffersize()
Get buffer size in bytes (only during acquisition).

Return type int

1.3 FilterSettings

class waveline.conditionwave.**FilterSettings**(*highpass*, *lowpass*, *order=8*)
Filter settings.

__init__(*highpass*, *lowpass*, *order=8*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(highpass, lowpass[, order])	Initialize self.

Attributes

order	Filter order

highpass: Optional[float]
Highpass frequency in Hz

lowpass: Optional[float]
Lowpass frequency in Hz

order: int = 8
Filter order

CHAPTER

TWO

WAVELINE.SPOTWAVE

Module for spotWave device.

All device-related functions are exposed by the *SpotWave* class.

Classes

AERecord(type_, time, amplitude, rise_time, ...)	AE data record, either status or hit data.
Setup(acq_enabled, cont_enabled, ...)	Setup.
SpotWave(port)	Interface for spotWave devices.
Status(device_id, firmware_version, ...)	Status information.
TRRecord(trai, time, samples, data)	Transient data record.

2.1 AERecord

class waveline.spotwave.**AERecord**(*type_*, *time*, *amplitude*, *rise_time*, *duration*, *counts*, *energy*, *trai*, *flags*)

AE data record, either status or hit data.

Todo:

- Documentation or data type with available hit flags

__init__(*type_*, *time*, *amplitude*, *rise_time*, *duration*, *counts*, *energy*, *trai*, *flags*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(type_, time, amplitude, rise_time, ...)	Initialize self.

type_: **str**

Record type (hit or status data)

time: **float**

Time in seconds

amplitude: **float**

Peak amplitude in volts

rise_time: *float*
Rise time in seconds

duration: *float*
Duration in seconds

counts: *int*
Number of positive threshold crossings

energy: *float*
Energy (EN 1330-9) in eu (1e-14 V^2s)

trai: *int*
Transient recorder index (key between *AERecord* and *TRRecord*)

flags: *int*
Hit flags

2.2 Setup

class waveline.spotwave.**Setup**(*acq_enabled*, *cont_enabled*, *log_enabled*, *adc_to_volts*, *threshold_volts*, *ddt_seconds*, *status_interval_seconds*, *filter_highpass_hz*, *filter_lowpass_hz*, *filter_order*, *tr_enabled*, *tr_decimation*, *tr_pretrigger_samples*, *tr_postduration_samples*, *cct_seconds*)

Setup.

__init__(*acq_enabled*, *cont_enabled*, *log_enabled*, *adc_to_volts*, *threshold_volts*, *ddt_seconds*, *status_interval_seconds*, *filter_highpass_hz*, *filter_lowpass_hz*, *filter_order*, *tr_enabled*, *tr_decimation*, *tr_pretrigger_samples*, *tr_postduration_samples*, *cct_seconds*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(acq_enabled, cont_enabled, ...)	Initialize self.

acq_enabled: *bool*
Flag if acquisition is enabled

cont_enabled: *bool*
Flag if continuous mode is enabled

log_enabled: *bool*
Flag if logging mode is enabled

adc_to_volts: *float*
Conversion factor from ADC values to volts

threshold_volts: *float*
Threshold for hit-based acquisition in volts

ddt_seconds: *float*
Duration discrimination time (DDT) in seconds

status_interval_seconds: *float*
Status interval in seconds

filter_highpass_hz: *float*
Highpass frequency in Hz

filter_lowpass_hz: *float*
Lowpass frequency in Hz

filter_order: *int*
Filter order

tr_enabled: *bool*
Flag in transient data recording is enabled

tr_decimation: *int*
Decimation factor for transient data

tr_pretrigger_samples: *int*
Pre-trigger samples for transient data

tr_postduration_samples: *int*
Post-duration samples for transient data

cct_seconds: *float*
Coupling check transmitter (CCT) / pulser interval in seconds

2.3 SpotWave

class waveline.spotwave.**SpotWave**(*port*)
Interface for spotWave devices.

The spotWave device is connected via USB and exposes a virtual serial port for communication.

__init__(*port*)
Initialize device.

Parameters **port** (Union[str, Serial]) – Either the serial port id (e.g. "COM6") or a serial.Serial port instance. Use the method *discover* to get a list of ports with connected spotWave devices.

Returns Instance of *SpotWave*

Example

There are two ways constructing and using the *ConditionWave* class:

1. Without context manager and manually calling the *close* method afterwards:

```
>>> sw = waveline.SpotWave("COM6")
>>> print(sw.get_setup())
>>> ...
>>> sw.close()
```

2. Using the context manager:

```
>>> with waveline.SpotWave("COM6") as sw:
>>>     print(sw.get_setup())
>>>     ...
```

Methods

__init__(port)	Initialize device.
clear_buffer()	Clear input and output buffer.
close()	Close serial connection to the device.
connect()	Open serial connection to the device.
discover()	Discover connected spotWave devices.
get_ae_data()	Get AE data records.
get_data(samples)	Read snapshot of transient data with maximum sampling rate (2 MHz).
get_setup()	Get setup.
get_status()	Get status.
get_tr_data()	Get transient data records.
set_cct(interval_seconds[, sync])	Set coupling check ransmitter (CCT) / pulser interval.
set_continuous_mode(enabled)	Enable/disable continuous mode.
set_datetime([timestamp])	Set current date and time.
set_ddt(microseconds)	Set duration discrimination time (DDT).
set_filter(highpass, lowpass[, order])	Set IIR filter frequencies and order.
set_status_interval(milliseconds)	Set status interval.
set_threshold(microvolts)	Set threshold for hit-based acquisition.
set_tr_decimation(factor)	Set decimation factor of transient data.
set_tr_enabled(enabled)	Enable/disable recording of transient data.
set_tr_postduration(samples)	Set post-duration samples for transient data.
set_tr_pretrigger(samples)	Set pre-trigger samples for transient data.
start_acquisition()	Start acquisition.
stop_acquisition()	Stop acquisition.
stream()	High-level method to continuously acquire data.

Attributes

CLOCK	Internal clock in Hz
PRODUCT_ID	USB product id of SpotWave device
VENDOR_ID	USB vendor id of Vallen Systeme GmbH
connected	Check if the connection to the device is open.

VENDOR_ID = 8849
USB vendor id of Vallen Systeme GmbH

PRODUCT_ID = 272
USB product id of SpotWave device

CLOCK = 2000000
Internal clock in Hz

connect()
Open serial connection to the device.

The *connect* method is automatically called in the constructor. You only need to call the method to reopen the connection after calling *close*.

close()
Close serial connection to the device.

property connected
Check if the connection to the device is open.

Return type bool

classmethod discover()
Discover connected spotWave devices.

Return type List[str]

Returns List of port names

clear_buffer()
Clear input and output buffer.

get_setup()
Get setup.

Return type *Setup*

Returns Dataclass with setup information

get_status()
Get status.

Return type *Status*

Returns Dataclass with status information

set_continuous_mode(*enabled*)
Enable/disable continuous mode.

Threshold will be ignored. The length of the records is determined by *ddt* with *set_ddt*.

Note: The parameters for continuous mode with transient recording enabled (*set_tr_enabled*) have to be chosen with care - mainly the decimation factor (*set_tr_decimation*) and *ddt* (*set_ddt*). The internal buffer of the device can store up to ~200.000 samples.

If the buffer is full, data records are lost. Small latencies in data polling can cause overflows and therefore data loss. One record should not exceed half the buffer size (~100.000 samples). 25% of the buffer size (~50.000 samples) is a good starting point. The number of samples in a record is determined by *ddt* and the decimation factor d: $n = ddt_{\mu s} \cdot f_s/d = ddt_{\mu s} \cdot 2/d \implies ddt_{\mu s} \approx 50.000 \cdot d/2$

On the other hand, if the number of samples is small, more hits are generated and the CPU load increases.

Parameters **enabled** (bool) – Set to *True* to enable continuous mode

set_ddt(*microseconds*)
Set duration discrimination time (DDT).

Parameters **microseconds** (int) – DDT in µs

set_status_interval(*milliseconds*)
Set status interval.

Parameters **milliseconds** (int) – Status interval in ms

set_tr_enabled(*enabled*)
Enable/disable recording of transient data.

Parameters **enabled** (bool) – Set to *True* to enable transient data

set_tr_decimation(*factor*)
Set decimation factor of transient data.

The sampling rate of transient data will be 2 MHz / *factor*.

Parameters **factor** (int) – Decimation factor

set_tr_pretrigger(*samples*)
Set pre-trigger samples for transient data.

Parameters **samples** (int) – Pre-trigger samples

set_tr_postduration(*samples*)
Set post-duration samples for transient data.

Parameters **samples** (int) – Post-duration samples

set_cct(*interval_seconds*, *sync=False*)
Set coupling check ransmitter (CCT) / pulser interval.

The pulser amplitude is 3.3 V.

Parameters

- **interval_seconds** (int) – Pulser interval in seconds
- **sync** (bool) – Synchronize the pulser with the first sample of the *get_data* command

set_filter(*highpass*, *lowpass*, *order=4*)
Set IIR filter frequencies and order.

Parameters

- **highpass** (float) – Highpass frequency in kHz
- **lowpass** (float) – Lowpass frequency in kHz
- **order** (int) – Filter order

set_datetime(*timestamp=None*)
Set current date and time.

Parameters **timestamp** (Optional[datetime]) – datetime.datetime object, current time if *None*

set_threshold(*microvolts*)
Set threshold for hit-based acquisition.

Parameters **microvolts** (float) – Threshold in µV

start_acquisition()
Start acquisition.

stop_acquisition()
Stop acquisition.

get_ae_data()
Get AE data records.

Todo:

- Implement parsing of record start marker

Yields AE data records (either status or hit data)

Return type Iterator[*AERecord*]

get_tr_data()
Get transient data records.

Yields Transient data records

Return type Iterator[*TRRecord*]

stream()
High-level method to continuously acquire data.

Yields AE and TR data records

Example

```
>>> with waveline.SpotWave("COM6") as sw:
>>>     # apply settings
>>>     sw.set_ddt(400)
>>>     for record in sw.stream():
>>>         # do something with the data depending on the type
>>>         if isinstance(record, waveline.spotwave.AERecord):
>>>             ...
>>>         if isinstance(record, waveline.spotwave.TRRecord):
>>>             ...
```

Return type Iterator[Union[*AERecord*, *TRRecord*]]

get_data(*samples*)
Read snapshot of transient data with maximum sampling rate (2 MHz).

Parameters **samples** (int) – Number of samples to read

Return type ndarray

Returns Array with amplitudes in volts

2.4 Status

class waveline.spotwave.**Status**(*device_id*, *firmware_version*, *temperature*, *data_size*, *datetime*)
Status information.

__init__(*device_id*, *firmware_version*, *temperature*, *data_size*, *datetime*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(device_id, firmware_version, ...)	Initialize self.

device_id: **str**
Unique device id

firmware_version: **str**
Firmware version <major>.<minor> as hex codes

temperature: *int*
Device temperature in °C

data_size: *int*
Bytes in buffer

datetime: *datetime.datetime*
Device datetime

2.5 TRRecord

class waveline.spotwave.**TRRecord**(*trai*, *time*, *samples*, *data*)
Transient data record.

__init__(*trai*, *time*, *samples*, *data*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(trai, time, samples, data)	Initialize self.

trai: *int*
Transient recorder index (key between *AERecord* and *TRRecord*)

time: *float*
Time in seconds

samples: *int*
Number of samples

data: *numpy.ndarray*
Array of transient data in volts

CHAPTER

THREE

CHANGELOG

3.1 0.2.0

2020-12-18

Initial public release

CHAPTER

FOUR

TODOS

Todo:

- Documentation or data type with available hit flags

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pywaveline/envs/0.2.0/lib/python3.7/site-packages/waveline/spotwave.py:docstring of waveline.spotwave.AERecord, line 3.)

Todo:

- Implement parsing of record start marker

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pywaveline/envs/0.2.0/lib/python3.7/site-packages/waveline/spotwave.py:docstring of waveline.spotwave.SpotWave.get_ae_data, line 3.)

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

W

INDEX

Symbols

A

C

D

E

F

G

H

I

L

M

O

P

R

S

T

V

W