metamoth Documentation

Release 1.2.0

Santiago Martinez Balvanera

Apr 29, 2023

CONTENTS:

Welcome to the documentation for metamoth, a Python package for extracting the metadata from AudioMoth recordings.

CHAPTER

ONE

METAMOTH

Metamoth is a Python package for parsing the metadata of AudioMoth files. Check the full documentation at https://metamoth.readthedocs.io.

1.1 Motivation

AudioMoth devices store valuable information in the audio file header. This includes the device ID, the date and time of recording, gain settings and battery state. The number of fields in the metadata is growing as new features are added to the AudioMoth firmware.

However, the metadata is not designed to be easily parsed in a programmatic way. The data is stored as a string comment making it difficult to retrieve the individual metadata fields. Additionally, the comment format is not well documented and changes between AudioMoth firmware versions.

This package helps by **quickly** parsing the metadata and returning an object containing the metadata.

1.2 Usage

The `metamoth` package provides a single function, `parse_metadata`, which parses the metadata of an AudioMoth file and returns an object containing the metadata.

```
from metamoth import parse_metadata

metadata = parse_metadata('path/to/file')
```

The extracted metadata can be accessed as attributes of the object, as shown

```
duration = metadata.duration_s
path = metadata.path
# etc.
```

1.2.1 Extracted Metadata

The `metadata` variable is an object (of type AMMetadata) containing the metadata of the file.

The extracted metadata contains:

- `path`: the path of the audio file
- `firmware_version`: the firmware version of the AudioMoth that recorded the file. Since the AudioMoth firmware version is not stored in the recording, this is an estimate and may be incorrect.

Media Information

- `duration_s`: the duration of the file in seconds.
- `samplerate_hz`: the sample rate of the file in Hz.
- `channels`: the number of channels in the file.
- `samples`: the number of audio samples in the file.

Data extracted from the AudioMoth comment string:

- `datetime`: the date and time of the file in a datetime object.
- `timezone`: the timezone of the file as a timezone object.
- `audiomoth_id`: the ID of the AudioMoth that recorded the file.
- `battery_state_v`: the battery state of the AudioMoth that recorded the file in Volts.
- `low_battery`: a boolean indicating if the battery state is low.
- `gain`: the gain setting of the AudioMoth that recorded the file.
- `comment`: the full comment string in the WAV header.

The following fields are only available for some AudioMoth firmware versions. Nonetheless, they are always present in the metadata object, but may be `None`.

- `recording_state`: the recording state of the AudioMoth that recorded the file.
- `temperature_c`: the temperature of the AudioMoth in Celsius.
- `amplitude_threshold`: information concerning the wether an amplitude threshold was used and the threshold value.
- `frequency_filter`: information concerning the wether a frequency filter was used and the filter settings.
- `deployment_id`: the deployment ID as set by the user.
- `external_microphone`: a boolean indicating if an external microphone was used.
- `minimum_trigger_duration_s`: the minimum trigger duration in seconds.
- `frequency_trigger`: information concerning the wether a frequency trigger was used and the trigger settings.

The following table shows the fields available for each AudioMoth firmware.

version	recording_state	temperature_c	amplitude_threshold	frequency_filter	deployment_id	external_microphone	minimum_trigger_duration_s	frequency_trigger
1.0	×	×	×	×	×	×	×	×
1.0.1	×	×	×	×	×	×	×	×
1.1.0	×	×	×	×	×	×	×	×
1.2.0	×	×	×	×	×	×	×	×
1.2.1	✓	×	×	×	×	×	×	×
1.2.2	✓	×	×	×	×	×	×	×
1.3.0	✓	×	×	×	×	×	×	×
1.4.0	✓	✓	✓	✓	×	×	×	×
1.4.1	✓	✓	✓	✓	×	×	×	×
1.4.2	✓	✓	✓	✓	×	×	×	×
1.4.3	✓	✓	✓	✓	×	×	×	×
1.4.4	✓	✓	✓	✓	×	×	×	×
1.5.0	✓	✓	✓	✓	✓	✓	×	×
1.6.0	✓	✓	✓	✓	✓	✓	✓	×
1.7.0	✓	✓	✓	✓	✓	✓	✓	×
1.7.1	✓	✓	✓	✓	✓	✓	✓	×
1.8.0	✓	✓	✓	✓	✓	✓	✓	✓
1.8.1	✓	✓	✓	✓	✓	✓	✓	✓

1.3 Supported AudioMoth Firmware Versions

In the table below you can find the supported AudioMoth firmware versions.

Table 1: Supported AudioMoth Firmware Versions

Firmware	Supported
1.0.0	✓
1.0.1	✓
1.1.0	✓
1.2.0	✓
1.2.1	✓
1.2.2	✓
1.3.0	✓
1.4.0	✓
1.4.1	✓
1.4.2	✓
1.4.3	✓
1.4.4	✓
1.5.0	×
1.6.0	✓
1.7.0	×
1.7.1	×
1.8.0	×
1.8.1	×

Support for newer firmware versions is planned, see the CONTRIBUTING section if you want to help!

1.4 Performance

The `metamoth` package is designed to be fast. It extracts all the required information from the first few bytes and avoids loading the audio data. Thus `metamoth` parsing times are not affected by the size of the audio file.

The following table shows the parsing times of `metamoth` compared to exif tool.

File Size (MB)	metamoth (ms)	exiftool (ms)	Speedup
7.3	0.0845	80	~1000x
44	0.0850	91.86	~1000x

1.5 Installation

The `metamoth` package can be installed using `pip`:

```
pip install metamoth
```

Check the installation section of the documentation for more information.

1.6 Documentation

The documentation for the `metamoth` package is available at https://metamoth.readthedocs.io/en/latest/index.html.

CHAPTER

TWO

INSTALLATION

2.1 Stable release

To install metamoth, run this command in your terminal:

```
$ pip install metamoth
```

This is the preferred method to install metamoth, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for metamoth can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/mbsantiago/metamoth
```

Or download the tarball:

```
$ curl -OJL https://github.com/mbsantiago/metamoth/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ pip install .
```

CHAPTER

THREE

USAGE

3.1 Simple example

The `metamoth` package provides a single function, `parse_metadata`, which parses the metadata of an AudioMoth file and returns an object containing the metadata.

```
from metamoth import parse_metadata

metadata = parse_metadata('path/to/file')
```

The extracted metadata can be accessed as attributes of the object, as shown

```
duration = metadata.duration_s
path = metadata.path
# etc.
```

3.2 Metadata

The `metadata` variable is an object (of type *metamoth.metadata.AMMetadata*) containing the metadata of the file.

The extracted metadata contains:

- `path`: the path of the audio file
- `firmware_version`: the firmware version of the AudioMoth that recorded the file. Since the AudioMoth firmware version is not stored in the recording, this is an estimate and may be incorrect.

3.2.1 Media Information

- `duration_s`: the duration of the file in seconds.
- `samplerate_hz`: the sample rate of the file in Hz.
- `channels`: the number of channels in the file.
- `samples`: the number of audio samples in the file.

3.2.2 AudioMoth Information

Data extracted from the AudioMoth comment string:

- `datetime`: the date and time of the file in a datetime object.
- `timezone`: the timezone of the file as a timezone object.
- `audiomoth_id`: the ID of the AudioMoth that recorded the file.
- `battery_state_v`: the battery state of the AudioMoth that recorded the file in Volts.
- `low_battery`: a boolean indicating if the battery state is low.
- `gain`: the gain setting of the AudioMoth that recorded the file.
- `comment`: the full comment string in the WAV header.

The following fields are only available for some AudioMoth firmware versions. Nonetheless, they are always present in the metadata object, but may be `None`.

- `recording_state`: the *Recording State* of the AudioMoth that recorded the file.
- `temperature_c`: the temperature of the AudioMoth in Celsius.
- `amplitude_threshold`: information concerning the whether an *Amplitude Threshold* was used and the threshold value.
- `frequency_filter`: information concerning the whether a *Frequency Filter* was used and the filter settings.
- `deployment_id`: the deployment ID as set by the user.
- `external_microphone`: a boolean indicating if an external microphone was used.
- `minimum_trigger_duration_s`: the minimum trigger duration in seconds.
- `frequency_trigger`: information concerning the whether a *Frequency Trigger* was used and the trigger settings.

3.2.3 Recording State

The recording state is an object (of type *metamoth.enums.RecordingState*) indicating the state of the AudioMoth when the recording was made. It can be one of the following:

- `RECORDING_OKAY`: the AudioMoth was recording normally.
- `FILE_SIZE_LIMITED`: The AudioMoth stopped recording because the file size limit was reached.
- `SUPPLY_VOLTAGE_LOW`: The AudioMoth stopped recording because the battery was low.
- `SWITCH_CHANGED`: The AudioMoth stopped recording because the switch was changed.
- `MICROPHONE_CHANGED`: The AudioMoth stopped recording because the microphone was changed.
- `MAGNETIC_SWITCH`: The AudioMoth stopped recording because the magnetic switch was triggered.
- `SDCARD_WRITE_ERROR`: The AudioMoth stopped recording because of an error writing to the SD card.

3.2.4 Amplitude Threshold

The amplitude threshold is an object (of type *metamoth.metadata.AmplitudeThreshold*) that holds information about the amplitude threshold used to trigger the recording. It has the following attributes:

- `enabled`: a boolean indicating if an amplitude threshold was used.
- `threshold`: the amplitude threshold in dB.

3.2.5 Frequency Filter

The frequency filter is an object (of type *metamoth.metadata.FrequencyFilter*) that holds information about the frequency filters used on the recording. It has the following attributes:

- `type`: the type of the filter as a *metamoth.enums.FilterType* object. It can be one of the following
  - `LOW_PASS`
  - `HIGH_PASS`
  - `NO_FILTER`
  - `BAND_PASS`
- `lower_frequency_hz`: the lower frequency of the filter in Hz. If the filter is a high-pass filter, this will be `None`.
- `higher_frequency_hz`: the upper frequency of the filter in Hz. If the filter is a low-pass filter, this will be `None`.

3.2.6 Frequency Trigger

The frequency trigger is an object (of type *metamoth.metadata.FrequencyTrigger*) that holds information about the frequency trigger used to trigger the recording. It has the following attributes:

- `enabled`: a boolean indicating if a frequency trigger was used.
- `centre_frequency_hz`: the centre frequency of the trigger in Hz.
- `window_length_shift`: the window length shift in samples of the trigger.

3.2.7 Fields available for each AudioMoth firmware

The following table shows the fields available for each AudioMoth firmware.

version	recording_state	temperature_c	amplitude_threshold	frequency_filter	deployment_id	external_microphone	minimum_trigger_duration_s	frequency_trigger
1.0	×	×	×	×	×	×	×	×
1.0.1	×	×	×	×	×	×	×	×
1.1.0	×	×	×	×	×	×	×	×
1.2.0	×	×	×	×	×	×	×	×
1.2.1	✓	×	×	×	×	×	×	×
1.2.2	✓	×	×	×	×	×	×	×
1.3.0	✓	×	×	×	×	×	×	×
1.4.0	✓	✓	✓	✓	×	×	×	×
1.4.1	✓	✓	✓	✓	×	×	×	×
1.4.2	✓	✓	✓	✓	×	×	×	×
1.4.3	✓	✓	✓	✓	×	×	×	×
1.4.4	✓	✓	✓	✓	×	×	×	×
1.5.0	✓	✓	✓	✓	✓	✓	×	×
1.6.0	✓	✓	✓	✓	✓	✓	✓	×
1.7.0	✓	✓	✓	✓	✓	✓	✓	×
1.7.1	✓	✓	✓	✓	✓	✓	✓	×
1.8.0	✓	✓	✓	✓	✓	✓	✓	✓
1.8.1	✓	✓	✓	✓	✓	✓	✓	✓

CHAPTER

FOUR

METAMOTH

4.1 metamoth package

Metamoth: A Python package for parsing AudioMoth metadata.

Metamoth is a Python package for parsing metadata from AudioMoth recordings. It provides a function to parse the metadata from a WAV file and returns an object with the metadata as attributes.

`metamoth.`**`parse_metadata`**(*path: Union[PathLike, str]*) → *AMMetadata*

Parse the metadata from an AudioMoth recording.

4.1.1 Parameters

path : PathLike

4.1.2 Returns

Metadata
Parse metadata from the recording at *path*. The metadata is returned as a `AMMetadata` object.

4.1.3 Submodules

4.1.4 metamoth.artist module

Functions for reading the artist chunk from a WAV file.

`metamoth.artist.`**`get_am_artist`**(*wav: BinaryIO*, *chunk:* Chunk) → Optional[str]

Get the artist string from the WAV file.

Parameters

wav
: [BinaryIO] The WAV file.

chunk
: [Chunk] The RIFF chunk, which is the root chunk.

Returns

str
: The artist string. If the artist is not found, returns None.

4.1.5 metamoth.audio module

Audio file utilities.

`metamoth.audio.`**`is_riff`**(*path: Union[PathLike, str]*) → bool

Return True if path is a RIFF file.

An RIFF file is a IFF file with the RIFF chunk ID. The RIFF chunk ID is the first 4 bytes of the file.

Parameters

path : PathLike

Returns

bool

`metamoth.audio.`**`is_wav`**(*path: Union[PathLike, str]*) → bool

Return True if path is a WAV file.

A WAV file is a RIFF file with the WAVE chunk ID. The WAVE chunk ID is the 8th byte of the file.

Parameters

path : PathLike

Returns

bool

`metamoth.audio.`**`is_wav_filename`**(*filename: Union[PathLike, str]*) → bool

Return True if filename is a WAV file.

4.1.6 metamoth.chunks module

Parse a RIFF file into chunks and subchunks.

This module is based on the RIFF specification: https://www.loc.gov/preservation/digital/formats/fdd/fdd000001.shtml

The RIFF file format is a container format for storing data in tagged chunks. Each chunk consists of a 4-byte chunk ID, a 4-byte little-endian chunk size, and the chunk data. The chunk data is padded with a null byte if the chunk size is odd.

The RIFF file format is used for storing audio and video data. The RIFF file format is also used for storing other types of data, such as text, images, and metadata.

class metamoth.chunks.**Chunk**(*chunk_id: str*, *size: int*, *position: int*, *identifier: ~typing.Optional[str] = None*, *subchunks: ~typing.List[~metamoth.chunks.Chunk] = <factory>*)

Bases: `object`

A chunk of a RIFF file.

Parameters

chunk_id
[str] The chunk ID.

position: int
The position of the chunk in the file.

chunk_size
[int] The chunk size.

subchunks
[List[Chunk]] The subchunks of the chunk.

chunk_id: `str`

identifier: `Optional[str] = None`

position: `int`

size: `int`

subchunks: `List[`*Chunk*`]`

metamoth.chunks.**parse_into_chunks**(*riff: BinaryIO*) → *Chunk*

Return the RIFF file chunk and subchunks.

Parameters

riff
[BinaryIO] Open file object of the RIFF file.

Returns

Chunk

4.1.7 metamoth.comments module

Functions for reading the comment chunk of an AudioMoth WAV file.

metamoth.comments.**get_am_comment**(*wav: BinaryIO*, *chunk:* Chunk) → str

Return the comment from the WAV file.

Parameters

wav
[BinaryIO] Open file object of the WAV file.

chunk
[Chunk] The RIFF chunk info, which is the root chunk. Should include the LIST chunk as a subchunk.

Returns

comment : str

4.1.8 metamoth.config module

Module with AudioMoth configuration classes for different versions.

class metamoth.config.**Config1_0**(*time: int = 0*, *gain: int = 2*, *clock_band: int = 4*, *clock_divider: int = 2*, *acquisition_cycles: int = 2*, *oversample_rate: int = 16*, *sample_rate: int = 48000*, *sleep_duration: int = 0*, *record_duration: int = 60*, *enable_led: bool = True*, *active_start_stop_periods: bool = False*, *start_stop_period: ~typing.List[~metamoth.config.StartStopPeriod] = <factory>*)

Bases: object

AudioMoth configuration for version 1.0.

Also valid for version 1.0.1

acquisition_cycles: int = 2

active_start_stop_periods: bool = False

clock_band: int = 4

clock_divider: int = 2

enable_led: bool = True

gain: int = 2

oversample_rate: int = 16

record_duration: int = 60

sample_rate: int = 48000

sleep_duration: int = 0

start_stop_period: List[StartStopPeriod]

time: int = 0

class metamoth.config.**Config1_1_0**(*time: int = 0*, *gain: int = 2*, *clock_divider: int = 4*, *acquisition_cycles: int = 16*, *oversample_rate: int = 1*, *sample_rate: int = 384000*, *sample_rate_divider: int = 8*, *sleep_duration: int = 0*, *record_duration: int = 60*, *enable_led: bool = True*, *active_start_stop_periods: bool = False*, *start_stop_period: ~typing.List[~metamoth.config.StartStopPeriod] = <factory>*)

Bases: object

AudioMoth configuration for version 1.1.0.

acquisition_cycles: int = 16

active_start_stop_periods: bool = False

clock_divider: int = 4

enable_led: bool = True

gain: int = 2

oversample_rate: int = 1

record_duration: int = 60

sample_rate: int = 384000

sample_rate_divider: int = 8

sleep_duration: int = 0

start_stop_period: List[StartStopPeriod]

time: int = 0

class metamoth.config.**Config1_2_0**(*time: int = 0*, *gain: int = 2*, *clock_divider: int = 4*, *acquisition_cycles: int = 16*, *oversample_rate: int = 1*, *sample_rate: int = 384000*, *sleep_duration: int = 0*, *record_duration: int = 60*, *enable_led: bool = True*, *active_start_stop_periods: bool = False*, *start_stop_period: ~typing.List[~metamoth.config.StartStopPeriod] = <factory>*, *timezone: int = 0*)

Bases: object

AudioMoth configuration for version 1.2.0.

acquisition_cycles: int = 16

active_start_stop_periods: bool = False

clock_divider: int = 4

enable_led: bool = True

gain: int = 2

oversample_rate: int = 1

record_duration: int = 60

sample_rate: int = 384000

sleep_duration: int = 0

start_stop_period: List[StartStopPeriod]

time: int = 0

timezone: int = 0

class metamoth.config.**Config1_2_1**(*time: int = 0*, *gain: int = 2*, *clock_divider: int = 4*, *acquisition_cycles: int = 16*, *oversample_rate: int = 1*, *sample_rate: int = 384000*, *sleep_duration: int = 0*, *record_duration: int = 60*, *enable_led: bool = True*, *active_start_stop_periods: bool = False*, *start_stop_period: ~typing.List[~metamoth.config.StartStopPeriod] = <factory>*, *timezone: int = 0*, *enable_battery_check: bool = False*, *disable_battery_level_display: bool = False*)

Bases: object

AudioMoth configuration for version 1.2.1.

acquisition_cycles: int = 16

active_start_stop_periods: bool = False

clock_divider: int = 4

disable_battery_level_display: bool = False

enable_battery_check: bool = False

enable_led: bool = True

gain: int = 2

oversample_rate: int = 1

record_duration: int = 60

sample_rate: int = 384000

sleep_duration: int = 0

start_stop_period: List[StartStopPeriod]

time: int = 0

timezone: int = 0

class metamoth.config.**Config1_2_2**(*time: int = 0*, *gain: int = 2*, *clock_divider: int = 4*, *acquisition_cycles: int = 16*, *oversample_rate: int = 1*, *sample_rate: int = 384000*, *sample_rate_divider: int = 8*, *sleep_duration: int = 0*, *record_duration: int = 60*, *enable_led: bool = True*, *active_start_stop_periods: bool = False*, *start_stop_period: ~typing.List[~metamoth.config.StartStopPeriod] = <factory>*, *timezone_hours: int = 0*, *enable_battery_check: bool = False*, *disable_battery_level_display: bool = False*, *timezone_minutes: int = 0*)

Bases: object

AudioMoth configuration for version 1.2.2.

Also valid for version 1.3.0.

acquisition_cycles: int = 16

active_start_stop_periods: bool = False

clock_divider: int = 4

disable_battery_level_display: bool = False

enable_battery_check: bool = False

enable_led: bool = True

gain: int = 2

oversample_rate: int = 1

record_duration: int = 60

sample_rate: int = 384000

sample_rate_divider: int = 8

sleep_duration: int = 0

start_stop_period: List[StartStopPeriod]

time: int = 0

timezone_hours: int = 0

timezone_minutes: int = 0

class metamoth.config.**Config1_4_0**(*time: int = 0*, *gain: int = 2*, *clock_divider: int = 4*, *acquisition_cycles: int = 16*, *oversample_rate: int = 1*, *sample_rate: int = 384000*, *sample_rate_divider: int = 8*, *sleep_duration: int = 5*, *record_duration: int = 55*, *enable_led: bool = True*, *active_start_stop_periods: bool = False*, *start_stop_period: ~typing.List[~metamoth.config.StartStopPeriod] = <factory>*, *timezone_hours: int = 0*, *enable_low_voltage_cutoff: bool = False*, *disable_battery_level_display: bool = False*, *timezone_minutes: int = 0*, *disable_sleep_record_cycle: bool = False*, *earliest_recording_time: int = 0*, *latest_recording_time: int = 0*, *lower_filter_freq: int = 0*, *higher_filter_freq: int = 0*, *amplitude_threshold: int = 0*)

Bases: object

AudioMoth configuration for version 1.4.0.

Also valid for version 1.4.1, 1.4.2, 1.4.3, 1.4.4,

acquisition_cycles: int = 16

active_start_stop_periods: bool = False

amplitude_threshold: int = 0

clock_divider: int = 4

disable_battery_level_display: bool = False

disable_sleep_record_cycle: bool = False

earliest_recording_time: int = 0

enable_led: bool = True

enable_low_voltage_cutoff: bool = False

gain: int = 2

higher_filter_freq: int = 0

latest_recording_time: int = 0

lower_filter_freq: int = 0

oversample_rate: int = 1

record_duration: int = 55

sample_rate: int = 384000

sample_rate_divider: int = 8

sleep_duration: int = 5

start_stop_period: List[StartStopPeriod]

time: int = 0

timezone_hours: int = 0

timezone_minutes: int = 0

class metamoth.config.**Config1_5_0**(*time: int = 0*, *gain: int = 2*, *clock_divider: int = 4*, *acquisition_cycles: int = 16*, *oversample_rate: int = 1*, *sample_rate: int = 384000*, *sample_rate_divider: int = 8*, *sleep_duration: int = 5*, *record_duration: int = 55*, *enable_led: bool = True*, *active_start_stop_periods: bool = False*, *start_stop_period: ~typing.List[~metamoth.config.StartStopPeriod] = <factory>*, *timezone_hours: int = 0*, *enable_low_voltage_cutoff: bool = False*, *disable_battery_level_display: bool = False*, *timezone_minutes: int = 0*, *disable_sleep_record_cycle: bool = False*, *earliest_recording_time: int = 0*, *latest_recording_time: int = 0*, *lower_filter_freq: int = 0*, *higher_filter_freq: int = 0*, *amplitude_threshold: int = 0*, *require_acoustic_configuration: bool = False*, *battery_level_display_type: ~metamoth.enums.BatteryLevelDisplayType = BatteryLevelDisplayType.BATTERY_LEVEL*, *minimum_amplitude_threshold_duration: int = 0*)

Bases: object

AudioMoth configuration for version 1.5.0.

acquisition_cycles: int = 16

active_start_stop_periods: bool = False

amplitude_threshold: int = 0

battery_level_display_type: *BatteryLevelDisplayType* = 1

clock_divider: int = 4

disable_battery_level_display: bool = False

disable_sleep_record_cycle: bool = False

earliest_recording_time: int = 0

enable_led: bool = True

enable_low_voltage_cutoff: bool = False

gain: int = 2

higher_filter_freq: int = 0

latest_recording_time: int = 0

lower_filter_freq: int = 0

minimum_amplitude_threshold_duration: int = 0

oversample_rate: int = 1

record_duration: int = 55

require_acoustic_configuration: bool = False

sample_rate: int = 384000

sample_rate_divider: int = 8

sleep_duration: int = 5

start_stop_period: List[StartStopPeriod]

time: int = 0

timezone_hours: int = 0

timezone_minutes: int = 0

class metamoth.config.**Config1_6_0**(*time: int = 0*, *gain: int = 2*, *clock_divider: int = 4*, *acquisition_cycles: int = 16*, *oversample_rate: int = 1*, *sample_rate: int = 384000*, *sample_rate_divider: int = 8*, *sleep_duration: int = 5*, *record_duration: int = 55*, *enable_led: bool = True*, *active_start_stop_periods: bool = True*, *start_stop_period: ~typing.List[~metamoth.config.StartStopPeriod] = <factory>*, *timezone_hours: int = 0*, *enable_low_voltage_cutoff: bool = True*, *disable_battery_level_display: bool = False*, *timezone_minutes: int = 0*, *disable_sleep_record_cycle: bool = False*, *earliest_recording_time: int = 0*, *latest_recording_time: int = 0*, *lower_filter_freq: int = 0*, *higher_filter_freq: int = 0*, *amplitude_threshold: int = 0*, *require_acoustic_configuration: bool = False*, *battery_level_display_type: ~metamoth.enums.BatteryLevelDisplayType = BatteryLevelDisplayType.BATTERY_LEVEL*, *minimum_trigger_duration: int = 0*, *enable_amplitude_threshold_decibel_scale: bool = False*, *amplitude_threshold_decibels: int = 0*, *enable_amplitude_threshold_percentage_scale: bool = False*, *amplitude_threshold_percentage_mantissa: int = 0*, *amplitude_threshold_percentage_exponent: int = 0*, *enable_energy_saver_mode: bool = False*, *disable_48_hz_dc_blocking_filter: bool = False*)

Bases: object

AudioMoth configuration for version 1.6.0.

acquisition_cycles: int = 16

active_start_stop_periods: bool = True

amplitude_threshold: int = 0

amplitude_threshold_decibels: int = 0

amplitude_threshold_percentage_exponent: int = 0

amplitude_threshold_percentage_mantissa: int = 0

battery_level_display_type: *BatteryLevelDisplayType* = 1

clock_divider: int = 4

disable_48_hz_dc_blocking_filter: bool = False

disable_battery_level_display: bool = False

disable_sleep_record_cycle: bool = False

earliest_recording_time: int = 0

enable_amplitude_threshold_decibel_scale: bool = False

enable_amplitude_threshold_percentage_scale: bool = False

enable_energy_saver_mode: bool = False

enable_led: bool = True

enable_low_voltage_cutoff: bool = True

gain: int = 2

higher_filter_freq: int = 0

latest_recording_time: int = 0

lower_filter_freq: int = 0

minimum_trigger_duration: int = 0

oversample_rate: int = 1

record_duration: int = 55

require_acoustic_configuration: bool = False

sample_rate: int = 384000

sample_rate_divider: int = 8

sleep_duration: int = 5

start_stop_period: List[StartStopPeriod]

time: int = 0

timezone_hours: int = 0

timezone_minutes: int = 0

class metamoth.config.**Config1_7_0**(*time: int = 0*, *gain: ~metamoth.enums.GainSetting = GainSetting.AM_GAIN_MEDIUM*, *clock_divider: int = 4*, *acquisition_cycles: int = 16*, *oversample_rate: int = 1*, *sample_rate: int = 384000*, *sample_rate_divider: int = 8*, *sleep_duration: int = 5*, *record_duration: int = 55*, *enable_led: bool = True*, *active_start_stop_periods: bool = True*, *start_stop_period: ~typing.List[~metamoth.config.StartStopPeriod] = <factory>*, *timezone_hours: int = 0*, *enable_low_voltage_cutoff: bool = True*, *disable_battery_level_display: bool = False*, *timezone_minutes: int = 0*, *disable_sleep_record_cycle: bool = False*, *earliest_recording_time: int = 0*, *latest_recording_time: int = 0*, *lower_filter_freq: int = 0*, *higher_filter_freq: int = 0*, *amplitude_threshold: int = 0*, *require_acoustic_configuration: bool = False*, *battery_level_display_type: ~metamoth.enums.BatteryLevelDisplayType = BatteryLevelDisplayType.BATTERY_LEVEL*, *minimum_trigger_duration: int = 0*, *enable_amplitude_threshold_decibel_scale: bool = False*, *amplitude_threshold_decibels: int = 0*, *enable_amplitude_threshold_percentage_scale: bool = False*, *amplitude_threshold_percentage_mantissa: int = 0*, *amplitude_threshold_percentage_exponent: int = 0*, *enable_energy_saver_mode: bool = False*, *disable_48_hz_dc_blocking_filter: bool = False*, *enable_time_settings_from_gps: bool = False*, *enable_magnetic_switch: bool = False*, *enable_low_gain_range: bool = False*)

Bases: object

AudioMoth configuration for version 1.7.0.

Also valid for version 1.7.1.

acquisition_cycles: int = 16

active_start_stop_periods: bool = True

amplitude_threshold: int = 0

amplitude_threshold_decibels: int = 0

amplitude_threshold_percentage_exponent: int = 0

amplitude_threshold_percentage_mantissa: int = 0

battery_level_display_type: *BatteryLevelDisplayType* = 1

clock_divider: int = 4

disable_48_hz_dc_blocking_filter: bool = False

disable_battery_level_display: bool = False

disable_sleep_record_cycle: bool = False

earliest_recording_time: int = 0

enable_amplitude_threshold_decibel_scale: bool = False

enable_amplitude_threshold_percentage_scale: bool = False

enable_energy_saver_mode: bool = False

enable_led: bool = True

enable_low_gain_range: bool = False

enable_low_voltage_cutoff: bool = True

enable_magnetic_switch: bool = False

enable_time_settings_from_gps: bool = False

gain: *GainSetting* = 2

higher_filter_freq: int = 0

latest_recording_time: int = 0

lower_filter_freq: int = 0

minimum_trigger_duration: int = 0

oversample_rate: int = 1

record_duration: int = 55

require_acoustic_configuration: bool = False

sample_rate: int = 384000

sample_rate_divider: int = 8

sleep_duration: int = 5

start_stop_period: List[StartStopPeriod]

time: int = 0

timezone_hours: int = 0

timezone_minutes: int = 0

class metamoth.config.**Config1_8_0**(*time: int = 0*, *gain: ~metamoth.enums.GainSetting = GainSetting.AM_GAIN_MEDIUM*, *clock_divider: int = 4*, *acquisition_cycles: int = 16*, *oversample_rate: int = 1*, *sample_rate: int = 384000*, *sample_rate_divider: int = 8*, *sleep_duration: int = 5*, *record_duration: int = 55*, *enable_led: bool = True*, *active_start_stop_periods: bool = True*, *start_stop_period: ~typing.List[~metamoth.config.StartStopPeriod] = <factory>*, *timezone_hours: int = 0*, *enable_low_voltage_cutoff: bool = True*, *disable_battery_level_display: bool = False*, *timezone_minutes: int = 0*, *disable_sleep_record_cycle: bool = False*, *earliest_recording_time: int = 0*, *latest_recording_time: int = 0*, *lower_filter_freq: int = 0*, *higher_filter_freq: int = 0*, *amplitude_threshold: int = 0*, *frequency_trigger_centre_frequency: int = 0*, *require_acoustic_configuration: bool = False*, *battery_level_display_type: ~metamoth.enums.BatteryLevelDisplayType = BatteryLevelDisplayType.BATTERY_LEVEL*, *minimum_trigger_duration: int = 0*, *frequency_trigger_window_length_shift: int = 0*, *frequency_trigger_threshold_percentage_mantissa: int = 0*, *frequency_trigger_threshold_percentage_exponent: int = 0*, *enable_amplitude_threshold_decibel_scale: bool = False*, *amplitude_threshold_decibels: int = 0*, *enable_amplitude_threshold_percentage_scale: bool = False*, *amplitude_threshold_percentage_mantissa: int = 0*, *amplitude_threshold_percentage_exponent: int = 0*, *enable_energy_saver_mode: bool = False*, *disable_48_hz_dc_blocking_filter: bool = False*, *enable_time_settings_from_gps: bool = False*, *enable_magnetic_switch: bool = False*, *enable_low_gain_range: bool = False*, *enable_frequency_trigger: bool = False*, *enable_daily_folders: bool = False*)

Bases: `object`

AudioMoth configuration for version 1.8.0.

Also valid for version 1.8.1.

acquisition_cycles: int = 16

active_start_stop_periods: bool = True

amplitude_threshold: int = 0

amplitude_threshold_decibels: int = 0

amplitude_threshold_percentage_exponent: int = 0

amplitude_threshold_percentage_mantissa: int = 0

battery_level_display_type: *BatteryLevelDisplayType* = 1

clock_divider: int = 4

disable_48_hz_dc_blocking_filter: bool = False

disable_battery_level_display: bool = False

```
disable_sleep_record_cycle: bool = False
earliest_recording_time: int = 0
enable_amplitude_threshold_decibel_scale: bool = False
enable_amplitude_threshold_percentage_scale: bool = False
enable_daily_folders: bool = False
enable_energy_saver_mode: bool = False
enable_frequency_trigger: bool = False
enable_led: bool = True
enable_low_gain_range: bool = False
enable_low_voltage_cutoff: bool = True
enable_magnetic_switch: bool = False
enable_time_settings_from_gps: bool = False
frequency_trigger_centre_frequency: int = 0
frequency_trigger_threshold_percentage_exponent: int = 0
frequency_trigger_threshold_percentage_mantissa: int = 0
frequency_trigger_window_length_shift: int = 0
gain: GainSetting = 2
higher_filter_freq: int = 0
latest_recording_time: int = 0
lower_filter_freq: int = 0
minimum_trigger_duration: int = 0
oversample_rate: int = 1
record_duration: int = 55
require_acoustic_configuration: bool = False
sample_rate: int = 384000
sample_rate_divider: int = 8
sleep_duration: int = 5
start_stop_period: List[StartStopPeriod]
time: int = 0
timezone_hours: int = 0
timezone_minutes: int = 0
```

4.1.9 metamoth.enums module

Enums for the AudioMoth configurations and state.

class metamoth.enums.**BatteryLevelDisplayType**(*value*)

Bases: Enum

Battery level display type.

BATTERY_LEVEL = 1

NIMH_LIPO_BATTERY_VOLTAGE = 2

class metamoth.enums.**BatteryState**(*value*)

Bases: Enum

Battery state.

AM_BATTERY_3V6 = 1

AM_BATTERY_3V7 = 2

AM_BATTERY_3V8 = 3

AM_BATTERY_3V9 = 4

AM_BATTERY_4V0 = 5

AM_BATTERY_4V1 = 6

AM_BATTERY_4V2 = 7

AM_BATTERY_4V3 = 8

AM_BATTERY_4V4 = 9

AM_BATTERY_4V5 = 10

AM_BATTERY_4V6 = 11

AM_BATTERY_4V7 = 12

AM_BATTERY_4V8 = 13

AM_BATTERY_4V9 = 14

AM_BATTERY_FULL = 15

AM_BATTERY_LOW = 0

property volts: float

Return the battery voltage.

class metamoth.enums.**ExtendedBatteryState**(*value*)

Bases: Enum

Extended battery state.

AM_EXT_BAT_2V5 = 1

AM_EXT_BAT_2V6 = 2

AM_EXT_BAT_2V7 = 3

AM_EXT_BAT_2V8 = 4

AM_EXT_BAT_2V9 = 5

AM_EXT_BAT_3V0 = 6

AM_EXT_BAT_3V1 = 7

AM_EXT_BAT_3V2 = 8

AM_EXT_BAT_3V3 = 9

AM_EXT_BAT_3V4 = 10

AM_EXT_BAT_3V5 = 11

AM_EXT_BAT_3V6 = 12

AM_EXT_BAT_3V7 = 13

AM_EXT_BAT_3V8 = 14

AM_EXT_BAT_3V9 = 15

AM_EXT_BAT_4V0 = 16

AM_EXT_BAT_4V1 = 17

AM_EXT_BAT_4V2 = 18

AM_EXT_BAT_4V3 = 19

AM_EXT_BAT_4V4 = 20

AM_EXT_BAT_4V5 = 21

AM_EXT_BAT_4V6 = 22

AM_EXT_BAT_4V7 = 23

AM_EXT_BAT_4V8 = 24

AM_EXT_BAT_4V9 = 25

AM_EXT_BAT_FULL = 26

AM_EXT_BAT_LOW = 0

property volts: float

Return the battery voltage.

class metamoth.enums.**FilterType**(*value*)

Bases: Enum

Filter type.

BAND_PASS = 3

HIGH_PASS = 4

LOW_PASS = 2

NO_FILTER = 1

class metamoth.enums.**GainSetting**(*value*)

Bases: Enum

Gain setting.

AM_GAIN_HIGH = 4

AM_GAIN_LOW = 0

AM_GAIN_LOW_MEDIUM = 1

AM_GAIN_MEDIUM = 2

AM_GAIN_MEDIUM_HIGH = 3

class metamoth.enums.**RecordingState**(*value*)

Bases: Enum

Recording state.

FILE_SIZE_LIMITED = 2

MAGNETIC_SWITCH = 6

MICROPHONE_CHANGED = 5

RECORDING_OKAY = 1

SDCARD_WRITE_ERROR = 7

SUPPLY_VOLTAGE_LOW = 3

SWITCH_CHANGED = 4

4.1.10 metamoth.mediainfo module

Get media information from a WAV file.

The WAV file must be a PCM WAV file. The WAV file must have a fmt chunk and a data chunk. The WAV file must have a samplerate and channels information in the fmt chunk.

class metamoth.mediainfo.**MediaInfo**(*samplerate_hz: int*, *duration_s: float*, *samples: int*, *channels: int*)

Bases: object

Media information.

channels: int

Number of channels.

duration_s: float

Duration in seconds.

samplerate_hz: int

Sample rate in Hz.

samples: int

Number of samples.

metamoth.mediainfo.**get_media_info**(*wav: BinaryIO*, *chunk:* Chunk) → *MediaInfo*

Return the media information from the WAV file.

Parameters

wav

[BinaryIO] Open file object of the WAV file.

chunk

[Chunk] The RIFF chunk info, which is the root chunk. Should include the fmt and data chunks as sub-chunks.

Returns

MediaInfo

4.1.11 metamoth.metadata module

Module with AM Metadata structure for different firmware versions.

This module contains the AMMetadata class and its subclasses for different firmware versions.

class metamoth.metadata.**AMMetadata**(*path: str*, *firmware_version: str*, *samplerate_hz: int*, *duration_s: float*, *samples: int*, *channels: int*, *audiomoth_id: str*, *datetime: datetime*, *timezone: timezone*, *gain:* GainSetting, *comment: str*, *low_battery: bool*, *battery_state_v: float*, *recording_state: Optional[*RecordingState*] = RecordingState.RECORDING_OKAY*, *temperature_c: Optional[float] = None*, *amplitude_threshold: Optional[*AmplitudeThreshold*] = None*, *frequency_filter: Optional[*FrequencyFilter*] = None*, *deployment_id: Optional[int] = None*, *external_microphone: bool = False*, *minimum_trigger_duration_s: Optional[int] = None*, *frequency_trigger: Optional[*FrequencyTrigger*] = None*)

Bases: *CommentMetadataV6*, *MediaInfo*, *ExtraMetadata*

AudioMoth recording metadata.

class metamoth.metadata.**AmplitudeThreshold**(*enabled: bool*, *threshold: int*)

Bases: object

Amplitude threshold applied to the recording.

enabled: bool

True if the amplitude threshold is enabled.

threshold: int

Amplitude threshold.

class metamoth.metadata.**CommentMetadata**(*audiomoth_id: str*, *datetime: datetime*, *timezone: timezone*, *gain:* GainSetting, *comment: str*, *low_battery: bool*, *battery_state_v: float*)

Bases: object

Base class for AudioMoth metadata stored in comment.

audiomoth_id: str

AudioMoth ID. This is the serial number of the AudioMoth.

battery_state_v: float

Battery state in volts.

comment: str

Full comment string.

datetime: datetime

Datetime of the recording.

gain: *GainSetting*

Gain setting of the AudioMoth.

low_battery: bool

True if the battery is low.

timezone: timezone

Timezone of the recording.

class metamoth.metadata.**CommentMetadataV1**(*audiomoth_id: str*, *datetime: datetime*, *timezone: timezone*, *gain:* GainSetting, *comment: str*, *low_battery: bool*, *battery_state_v: float*)

Bases: *CommentMetadata*

AudioMoth recording metadata in comment string.

Valid for versions 1.0, 1.0.1, 1.1.0, 1.2.0,

Timezone is always UTC for versions 1.0 and 1.0.1. For versions 1.1.0 and 1.2.0, the an UTC hour offset can be set in the AudioMoth settings.

audiomoth_id: str

AudioMoth ID. This is the serial number of the AudioMoth.

battery_state_v: float

Battery state in volts.

comment: str

Full comment string.

datetime: datetime

Datetime of the recording.

gain: *GainSetting*

Gain setting of the AudioMoth.

low_battery: bool

True if the battery is low.

timezone: timezone

Timezone of the recording.

class metamoth.metadata.**CommentMetadataV2**(*audiomoth_id: str*, *datetime: datetime*, *timezone: timezone*, *gain:* GainSetting, *comment: str*, *low_battery: bool*, *battery_state_v: float*, *recording_state:* RecordingState)

Bases: *CommentMetadata*

AudioMoth recording metadata in comment string.

Valid for versions 1.2.1, 1.2.2, 1.3.0

recording_state: *RecordingState*

Recording state of the AudioMoth.

class metamoth.metadata.**CommentMetadataV3**(*audiomoth_id: str*, *datetime: datetime*, *timezone: timezone*, *gain:* GainSetting, *comment: str*, *low_battery: bool*, *battery_state_v: float*, *recording_state:* RecordingState, *temperature_c: float*, *amplitude_threshold:* AmplitudeThreshold, *frequency_filter:* FrequencyFilter)

Bases: *CommentMetadata*

AudioMoth recording metadata in comment string.

Valid for versions 1.4.0, 1.4.1, 1.4.2, 1.4.3, 1.4.4

amplitude_threshold: *AmplitudeThreshold*

Amplitude threshold applied to the recording.

frequency_filter: *FrequencyFilter*

Frequency filter applied to the recording.

recording_state: *RecordingState*

Recording state of the AudioMoth.

temperature_c: **float**

Temperature in degrees Celsius.

class metamoth.metadata.**CommentMetadataV4**(*audiomoth_id: str*, *datetime: datetime*, *timezone: timezone*, *gain:* GainSetting, *comment: str*, *low_battery: bool*, *battery_state_v: float*, *recording_state:* RecordingState, *temperature_c: float*, *amplitude_threshold:* AmplitudeThreshold, *frequency_filter:* FrequencyFilter, *deployment_id: Optional[int]*, *external_microphone: bool*)

Bases: *CommentMetadata*

AudioMoth recording metadata in comment string.

Valid for version 1.5.0

amplitude_threshold: *AmplitudeThreshold*

Amplitude threshold applied to the recording.

deployment_id: **Optional[int]**

Deployment ID of the AudioMoth.

external_microphone: **bool**

True if an external microphone is connected to the AudioMoth.

frequency_filter: *FrequencyFilter*

Frequency filter applied to the recording.

recording_state: *RecordingState*

Recording state of the AudioMoth.

temperature_c: **float**

Temperature in degrees Celsius.

class metamoth.metadata.**CommentMetadataV5**(*audiomoth_id: str*, *datetime: datetime*, *timezone: timezone*, *gain:* GainSetting, *comment: str*, *low_battery: bool*, *battery_state_v: float*, *recording_state:* RecordingState, *temperature_c: float*, *amplitude_threshold:* AmplitudeThreshold, *frequency_filter:* FrequencyFilter, *deployment_id: Optional[str]*, *external_microphone: bool*, *minimum_trigger_duration_s: int*)

Bases: *CommentMetadata*

AudioMoth recording metadata in comment string.

Valid for versions 1.6.0, 1.7.0 and 1.7.1

amplitude_threshold: *AmplitudeThreshold*

Amplitude threshold applied to the recording.

deployment_id: **Optional[str]**

Deployment ID of the AudioMoth.

external_microphone: **bool**

True if an external microphone is connected to the AudioMoth.

frequency_filter: *FrequencyFilter*

Frequency filter applied to the recording.

minimum_trigger_duration_s: **int**

Minimum trigger duration in seconds.

recording_state: *RecordingState*

Recording state of the AudioMoth.

temperature_c: **float**

Temperature in degrees Celsius.

class metamoth.metadata.**CommentMetadataV6**(*audiomoth_id: str*, *datetime: datetime*, *timezone: timezone*, *gain:* GainSetting, *comment: str*, *low_battery: bool*, *battery_state_v: float*, *recording_state: Optional[*RecordingState*] = RecordingState.RECORDING_OKAY*, *temperature_c: Optional[float] = None*, *amplitude_threshold: Optional[*AmplitudeThreshold*] = None*, *frequency_filter: Optional[*FrequencyFilter*] = None*, *deployment_id: Optional[int] = None*, *external_microphone: bool = False*, *minimum_trigger_duration_s: Optional[int] = None*, *frequency_trigger: Optional[*FrequencyTrigger*] = None*)

Bases: *CommentMetadata*

AudioMoth recording metadata in comment string.

Valid for versions 1.8.0 and 1.8.1

amplitude_threshold: Optional[*AmplitudeThreshold*] = None

Amplitude threshold applied to the recording.

deployment_id: Optional[int] = None

Deployment ID of the AudioMoth.

external_microphone: bool = False

True if an external microphone is connected to the AudioMoth.

frequency_filter: Optional[*FrequencyFilter*] = None

Frequency filter applied to the recording.

frequency_trigger: Optional[*FrequencyTrigger*] = None

Frequency trigger metadata.

minimum_trigger_duration_s: Optional[int] = None

Minimum trigger duration in seconds.

recording_state: Optional[*RecordingState*] = 1

Recording state of the AudioMoth.

temperature_c: Optional[float] = None

Temperature in degrees Celsius.

class metamoth.metadata.**ExtraMetadata**(*path: str*, *firmware_version: str*)

Bases: object

Extra metadata.

firmware_version: str

Firmware version of the AudioMoth.

path: str

Path to the recording.

class metamoth.metadata.**FrequencyFilter**(*type:* FilterType, *higher_frequency_hz: Optional[int]*, *lower_frequency_hz: Optional[int]*)

Bases: object

Frequency filter applied to the recording.

higher_frequency_hz: Optional[int]

Higher filter frequency in Hz. None if no upper filter is applied.

lower_frequency_hz: Optional[int]

Lower filter frequency in Hz. None if no lower filter is applied.

type: *FilterType*

Filter type applied to the recording.

class metamoth.metadata.**FrequencyTrigger**(*enabled: bool*, *centre_frequency_hz: int*, *window_length_shift: int*)

Bases: object

Frequency trigger metadata.

centre_frequency_hz: int

Centre frequency in Hz.

enabled: bool

True if frequency trigger is enabled.

window_length_shift: int

Window length shift in samples.

metamoth.metadata.**assemble_metadata**(*path: str*, *media_info:* MediaInfo, *comment_metadata: dict*, *artist: Optional[str]*) → *AMMetadata*

Assemble the metadata for the recording.

Parameters

media_info
[dict] Media information dictionary.

metadata
[dict] Metadata dictionary.

artist
[str] Artist string. Can be None.

Returns

metadata
[AMMetadataV6] Metadata object.

4.1.12 metamoth.metamoth module

Main module.

metamoth.metamoth.**parse_metadata**(*path: Union[PathLike, str]*) → *AMMetadata*

Parse the metadata from an AudioMoth recording.

Parameters

path : PathLike

Returns

Metadata
Parse metadata from the recording at *path*. The metadata is returned as a `AMMetadata` object.

4.1.13 metamoth.parsing module

Functions for parsing the comment string of AudioMoth recordings.

exception metamoth.parsing.**MessageFormatError**

Bases: Exception

Exception raised when the message format is not correct.

metamoth.parsing.**db_to_amplitude**(*db_value: float*) → int

Convert dB values to amplitude threshold value.

metamoth.parsing.**parse_comment**(*comment: str*) → dict

Parse the comment string into a dictionary of metadata.

Parameters

comment : str

Returns

metadata : dict

metamoth.parsing.**parse_comment_version_1_0**(*comment: str*) → *CommentMetadataV1*

Parse the comment string of 1.0 firmware.

Parameters

comment
[str] The comment string.

Returns

metadata : dict

metamoth.parsing.**parse_comment_version_1_0_1**(*comment: str*) → *CommentMetadataV1*

Parse the comment string of 1.0.1 firmware.

Also valid for version 1.1.0.

Parameters

comment : str

Returns

metadata: CommentMetadataV1

metamoth.parsing.**parse_comment_version_1_2_0**(*comment: str*) → *CommentMetadataV1*

Parse the comment string of 1.2.0 firmware.

Parameters

comment : str

Returns

metadata: CommentMetadataV1

metamoth.parsing.**parse_comment_version_1_2_1**(*comment: str*) → *CommentMetadataV2*

Parse the comment string of 1.2.1 firmware.

Parameters

comment : str

Returns

metadata: CommentMetadataV2

metamoth.parsing.**parse_comment_version_1_2_2**(*comment: str*) → *CommentMetadataV2*

Parse the comment string of 1.2.2 firmware.

Also valid for version 1.3.0.

Parameters

comment : str

Returns

metadata: CommentMetadataV2

metamoth.parsing.**parse_comment_version_1_4_0**(*comment: str*) → *CommentMetadataV3*

Parse the comment string of 1.4.0 firmware.

Also valid for version 1.4.1.

Parameters

comment : str

Returns

metadata: CommentMetadataV3

metamoth.parsing.**parse_comment_version_1_4_2**(*comment: str*) → *CommentMetadataV3*

Parse the comment string of 1.4.2 firmware.

Also valid for versions 1.4.3 and 1.4.4.

Parameters

comment : str

Returns

metadata: CommentMetadataV3

metamoth.parsing.**parse_comment_version_1_6_0**(*comment: str*) → *CommentMetadataV5*

Parse the comment string of 1.6.0 firmware.

Parameters

comment : str

Returns

metadata: CommentMetadataV5

metamoth.parsing.**percentage_to_amplitude**(*percentage: float*) → int

Convert percentage values to amplitude threshold value.

CHAPTER

FIVE

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

5.1 Types of Contributions

5.1.1 Test Data

Test data is always welcome. We are looking for recordings of different AM firmware versions and different configurations.

If you have a AudioMoth recordings that you would like to share, please open an issue and attach the file. All files will be licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

5.1.2 Report Bugs

Report bugs at https://github.com/mbsantiago/metamoth/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.1.3 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

5.1.4 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

5.1.5 Write Documentation

metamoth could always use more documentation, whether as part of the official metamoth docs, in docstrings, or even on the web in blog posts, articles, and such.

5.1.6 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/mbsantiago/metamoth/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.2 Get Started!

Ready to contribute? Here's how to set up *metamoth* for local development.

1. Fork the *metamoth* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/metamoth.git
```

3. Install your local copy into a virtualenv. Using venv is recommended:

```
$ cd metamoth/
$ python3 -m venv env
$ source env/bin/activate
$ pip install -e .
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. Install the development requirements:

```
$ pip install -r requirements-dev.txt
```

6. When you're done making changes, check that your changes pass the linting and unit tests, including testing other Python versions with tox. We use make to run the tests:

```
$ make lint
$ make test-all
```

7. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

8. Submit a pull request through the GitHub website.

5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.8, 3.9, 3.10 and 3.11.

CHAPTER

SIX

CREDITS

6.1 Development Lead

- Santiago Martinez Balvanera <santiago.balvanera.20@ucl.ac.uk>

6.2 Contributors

None yet. Why not be the first?

CHAPTER

SEVEN

HISTORY

7.1 0.1.0 (2023-02-08)

- First release on PyPI.

7.2 1.0.0 (2023-02-12)

- Restructured code to make it easier to add new firmware support.
- Added support for AudioMoth firmware version 1.0 - 1.3.0.
- Updated documentation.
- Changed the returned metadata object to include all possible fields.
- Added functions to generate AudioMoth comments in the format of all existing firmware versions.

CHAPTER

EIGHT

FIRMWARE HISTORY

AudioMoth has gone through several firmware revisions. The current version is 1.8.2. The firmware code is open source and available on GitHub.

AudioMoth metadata changes over firmware versions. Both the data fields included and the comment structure vary. The following table shows the metadata fields for each firmware version.

Table 1: AudioMoth Metadata

Version	1.0	1.0.1	1.1.0	1.2.0	1.2.1	1.2.2	1.3.0	1.4.0	1.4.1	1.4.2	1.4.3	1.4.4	1.5.0	1.6.0	1.7.0	1.7.1	1.8.0	1.8.1
Device ID																		
Date																		
Time																		
Gain																		
Battery State																		
Timezone																		
Recording Cancelled																		
Artist																		
Temperature																		
Amplitude Threshold																		
Filter Type																		
Higher Filter Freq																		
Lower Filter Freq																		
Deployment ID																		
External Microphone																		
Trigger Duration																		

This table only shows when certain fields were added. Often, the field content is formated differently, or the whole comment structure is different.

8.1 Code Snippets

Here you can find the code snippets for each firmware version. The code snippets are taken from the `setHeaderComment` function in the `main.c` file. This function sets the comment field in the WAV header.

8.1.1 Version 1.0

Code

```
void setHeaderComment(uint32_t currentTime, uint8_t *serialNumber, uint32_t gain) {

    time_t rawtime = currentTime;

    struct tm *time = gmtime(&rawtime);

    char *comment = wavHeader.icmt.comment;

    AM_batteryState_t batteryState = AudioMoth_getBatteryState();

    sprintf(comment, "Recorded at %02d:%02d:%02d %02d/%02d/%04d by AudioMoth %08X%08X at␣
↪gain setting %d while battery state was ",
            time->tm_hour, time->tm_min, time->tm_sec, time->tm_mday, 1 + time->tm_mon,␣
↪1900 + time->tm_year,
            (unsigned int)(serialNumber + 8), (unsigned int)serialNumber, (unsigned␣
↪int)gain);

    comment += 104;

    if (batteryState == AM_BATTERY_LOW) {

        sprintf(comment, "< 3.6V");

    } else if (batteryState >= AM_BATTERY_FULL) {

        sprintf(comment, "> 5.0V");

    } else {

        batteryState += 35;

        int tens = batteryState / 10;
        int units = batteryState - 10 * tens;

        sprintf(comment, "%01d.%02dV", tens, units);

    }

}
```

8.1.2 Version 1.0.1

Diff

```
@@ -8,11 +8,11 @@

     AM_batteryState_t batteryState = AudioMoth_getBatteryState();

-    sprintf(comment, "Recorded at %02d:%02d:%02d %02d/%02d/%04d by AudioMoth %08X%08X␣
↪at gain setting %d while battery state was ",
+    sprintf(comment, "Recorded at %02d:%02d:%02d %02d/%02d/%04d (UTC) by AudioMoth %08X
↪%08X at gain setting %d while battery state was ",
             time->tm_hour, time->tm_min, time->tm_sec, time->tm_mday, 1 + time->tm_mon,␣
↪1900 + time->tm_year,
             (unsigned int)(serialNumber + 8), (unsigned int)serialNumber, (unsigned␣
↪int)gain);

-    comment += 104;
+    comment += 110;

     if (batteryState == AM_BATTERY_LOW) {

@@ -26,10 +26,7 @@

         batteryState += 35;

-        int tens = batteryState / 10;
-        int units = batteryState - 10 * tens;
-
-        sprintf(comment, "%01d.%02dV", tens, units);
+        sprintf(comment, "%01d.%01dV", batteryState / 10, batteryState % 10);

     }
```

Code

```
void setHeaderComment(uint32_t currentTime, uint8_t *serialNumber, uint32_t gain) {

    time_t rawtime = currentTime;

    struct tm *time = gmtime(&rawtime);

    char *comment = wavHeader.icmt.comment;

    AM_batteryState_t batteryState = AudioMoth_getBatteryState();

    sprintf(comment, "Recorded at %02d:%02d:%02d %02d/%02d/%04d (UTC) by AudioMoth %08X
↪%08X at gain setting %d while battery state was ",
            time->tm_hour, time->tm_min, time->tm_sec, time->tm_mday, 1 + time->tm_mon,␣
↪1900 + time->tm_year,
            (unsigned int)(serialNumber + 8), (unsigned int)serialNumber, (unsigned␣
```

(continues on next page)

(continued from previous page)

```
↪int)gain);

    comment += 110;

    if (batteryState == AM_BATTERY_LOW) {

        sprintf(comment, "< 3.6V");

    } else if (batteryState >= AM_BATTERY_FULL) {

        sprintf(comment, "> 5.0V");

    } else {

        batteryState += 35;

        sprintf(comment, "%01d.%01dV", batteryState / 10, batteryState % 10);

    }

}
```

8.1.3 Version 1.1.0

No changes were introduces in this version.

8.1.4 Version 1.2.0

Diff

```
@@ -1,6 +1,6 @@
-void setHeaderComment(uint32_t currentTime, uint8_t *serialNumber, uint32_t gain) {
+void setHeaderComment(uint32_t currentTime, int8_t timezone, uint8_t *serialNumber,␣
↪uint32_t gain) {

-    time_t rawtime = currentTime;
+    time_t rawtime = currentTime + timezone * SECONDS_IN_HOUR;

     struct tm *time = gmtime(&rawtime);

@@ -8,25 +8,35 @@

     AM_batteryState_t batteryState = AudioMoth_getBatteryState();

-    sprintf(comment, "Recorded at %02d:%02d:%02d %02d/%02d/%04d (UTC) by AudioMoth %08X
↪%08X at gain setting %d while battery state was ",
-            time->tm_hour, time->tm_min, time->tm_sec, time->tm_mday, 1 + time->tm_mon,␣
↪1900 + time->tm_year,
-            (unsigned int)(serialNumber + 8), (unsigned int)serialNumber, (unsigned␣
```

(continues on next page)

(continued from previous page)

```
↪int)gain);
+    sprintf(comment, "Recorded at %02d:%02d:%02d %02d/%02d/%04d (UTC", time->tm_hour,␣
↪time->tm_min, time->tm_sec, time->tm_mday, 1 + time->tm_mon, 1900 + time->tm_year);

-    comment += 110;
+    comment += 36;
+
+    if (timezone < 0) sprintf(comment, "%d", timezone);
+
+    if (timezone > 0) sprintf(comment, "+%d", timezone);
+
+    if (timezone < 0 || timezone > 0) comment += 2;
+
+    if (timezone < -9 || timezone > 9) comment += 1;
+
+    sprintf(comment, ") by AudioMoth %08X%08X at gain setting %d while battery state␣
↪was ", (unsigned int)*((uint32_t*)serialNumber + 1), (unsigned int)*((uint32_
↪t*)serialNumber), (unsigned int)gain);
+
+    comment += 74;

     if (batteryState == AM_BATTERY_LOW) {

-        sprintf(comment, "< 3.6V");
+        sprintf(comment, "< 3.6V.");

     } else if (batteryState >= AM_BATTERY_FULL) {

-        sprintf(comment, "> 5.0V");
+        sprintf(comment, "> 5.0V.");

     } else {

         batteryState += 35;

-        sprintf(comment, "%01d.%01dV", batteryState / 10, batteryState % 10);
+        sprintf(comment, "%01d.%01dV.", batteryState / 10, batteryState % 10);

     }
```

Code

```
void setHeaderComment(uint32_t currentTime, int8_t timezone, uint8_t *serialNumber,␣
↪uint32_t gain) {

    time_t rawtime = currentTime + timezone * SECONDS_IN_HOUR;

    struct tm *time = gmtime(&rawtime);

    char *comment = wavHeader.icmt.comment;
```

(continues on next page)

(continued from previous page)

```
    AM_batteryState_t batteryState = AudioMoth_getBatteryState();

    sprintf(comment, "Recorded at %02d:%02d:%02d %02d/%02d/%04d (UTC", time->tm_hour,␣
↪time->tm_min, time->tm_sec, time->tm_mday, 1 + time->tm_mon, 1900 + time->tm_year);

    comment += 36;

    if (timezone < 0) sprintf(comment, "%d", timezone);

    if (timezone > 0) sprintf(comment, "+%d", timezone);

    if (timezone < 0 || timezone > 0) comment += 2;

    if (timezone < -9 || timezone > 9) comment += 1;

    sprintf(comment, ") by AudioMoth %08X%08X at gain setting %d while battery state was
↪", (unsigned int)*((uint32_t*)serialNumber + 1), (unsigned int)*((uint32_
↪t*)serialNumber), (unsigned int)gain);

    comment += 74;

    if (batteryState == AM_BATTERY_LOW) {

        sprintf(comment, "< 3.6V.");

    } else if (batteryState >= AM_BATTERY_FULL) {

        sprintf(comment, "> 5.0V.");

    } else {

        batteryState += 35;

        sprintf(comment, "%01d.%01dV.", batteryState / 10, batteryState % 10);

    }

}
```

8.1.5 Version 1.2.1

Diff

```
@@ -1,12 +1,18 @@
-void setHeaderComment(uint32_t currentTime, int8_t timezone, uint8_t *serialNumber,␣
↪uint32_t gain) {
+void setHeaderComment(uint32_t currentTime, int8_t timezone, uint8_t *serialNumber,␣
↪uint32_t gain, AM_batteryState_t batteryState, bool batteryVoltageLow, bool␣
↪switchPositionChanged) {
```

(continues on next page)

(continued from previous page)

```
     time_t rawtime = currentTime + timezone * SECONDS_IN_HOUR;

     struct tm *time = gmtime(&rawtime);

-    char *comment = wavHeader.icmt.comment;
+    /* Format artist field */
+
+    char *artist = wavHeader.iart.artist;
+
+    sprintf(artist, "AudioMoth %08X%08X", (unsigned int)*((uint32_t*)serialNumber + 1),␣
↪(unsigned int)*((uint32_t*)serialNumber));

-    AM_batteryState_t batteryState = AudioMoth_getBatteryState();
+    /* Format comment field */
+
+    char *comment = wavHeader.icmt.comment;

     sprintf(comment, "Recorded at %02d:%02d:%02d %02d/%02d/%04d (UTC", time->tm_hour,␣
↪time->tm_min, time->tm_sec, time->tm_mday, 1 + time->tm_mon, 1900 + time->tm_year);

@@ -20,17 +26,21 @@

     if (timezone < -9 || timezone > 9) comment += 1;

-    sprintf(comment, ") by AudioMoth %08X%08X at gain setting %d while battery state␣
↪was ", (unsigned int)*((uint32_t*)serialNumber + 1), (unsigned int)*((uint32_
↪t*)serialNumber), (unsigned int)gain);
+    sprintf(comment, ") by %s at gain setting %d while battery state was ", artist,␣
↪(unsigned int)gain);

     comment += 74;

     if (batteryState == AM_BATTERY_LOW) {

-        sprintf(comment, "< 3.6V.");
+        sprintf(comment, "less than 3.6V.");
+
+        comment += 15;

     } else if (batteryState >= AM_BATTERY_FULL) {

-        sprintf(comment, "> 5.0V.");
+        sprintf(comment, "greater than 4.9V.");
+
+        comment += 18;

     } else {

@@ -38,6 +48,26 @@

         sprintf(comment, "%01d.%01dV.", batteryState / 10, batteryState % 10);
```

(continues on next page)

(continued from previous page)

```
+        comment += 5;
+
+    }
+
+    if (batteryVoltageLow || switchPositionChanged) {
+
+        sprintf(comment, " Recording cancelled before completion due to ");
+
+        comment += 46;
+
+        if (batteryVoltageLow) {
+
+            sprintf(comment, "low battery voltage.");
+
+        } else if (switchPositionChanged) {
+
+            sprintf(comment, "change of switch position.");
+
+        }
+
     }

 }
```

Code

```
void setHeaderComment(uint32_t currentTime, int8_t timezone, uint8_t *serialNumber,␣
↪uint32_t gain, AM_batteryState_t batteryState, bool batteryVoltageLow, bool␣
↪switchPositionChanged) {

    time_t rawtime = currentTime + timezone * SECONDS_IN_HOUR;

    struct tm *time = gmtime(&rawtime);

    /* Format artist field */

    char *artist = wavHeader.iart.artist;

    sprintf(artist, "AudioMoth %08X%08X", (unsigned int)*((uint32_t*)serialNumber + 1),␣
↪(unsigned int)*((uint32_t*)serialNumber));

    /* Format comment field */

    char *comment = wavHeader.icmt.comment;

    sprintf(comment, "Recorded at %02d:%02d:%02d %02d/%02d/%04d (UTC", time->tm_hour,␣
↪time->tm_min, time->tm_sec, time->tm_mday, 1 + time->tm_mon, 1900 + time->tm_year);

    comment += 36;
```

(continues on next page)

(continued from previous page)

```
    if (timezone < 0) sprintf(comment, "%d", timezone);

    if (timezone > 0) sprintf(comment, "+%d", timezone);

    if (timezone < 0 || timezone > 0) comment += 2;

    if (timezone < -9 || timezone > 9) comment += 1;

    sprintf(comment, ") by %s at gain setting %d while battery state was ", artist,␣
↪(unsigned int)gain);

    comment += 74;

    if (batteryState == AM_BATTERY_LOW) {

        sprintf(comment, "less than 3.6V.");

        comment += 15;

    } else if (batteryState >= AM_BATTERY_FULL) {

        sprintf(comment, "greater than 4.9V.");

        comment += 18;

    } else {

        batteryState += 35;

        sprintf(comment, "%01d.%01dV.", batteryState / 10, batteryState % 10);

        comment += 5;

    }

    if (batteryVoltageLow || switchPositionChanged) {

        sprintf(comment, " Recording cancelled before completion due to ");

        comment += 46;

        if (batteryVoltageLow) {

            sprintf(comment, "low battery voltage.");

        } else if (switchPositionChanged) {

            sprintf(comment, "change of switch position.");

        }
```

(continues on next page)

(continued from previous page)

```
    }

}
```

8.1.6 Version 1.2.2

Diff

```
@@ -1,6 +1,6 @@
-void setHeaderComment(uint32_t currentTime, int8_t timezone, uint8_t *serialNumber,␣
↪uint32_t gain, AM_batteryState_t batteryState, bool batteryVoltageLow, bool␣
↪switchPositionChanged) {
+void setHeaderComment(uint32_t currentTime, int8_t timezoneHours, int8_t␣
↪timezoneMinutes, uint8_t *serialNumber, uint32_t gain, AM_batteryState_t batteryState,␣
↪bool batteryVoltageLow, bool switchPositionChanged) {

-    time_t rawtime = currentTime + timezone * SECONDS_IN_HOUR;
+    time_t rawtime = currentTime + timezoneHours * SECONDS_IN_HOUR + timezoneMinutes *␣
↪SECONDS_IN_MINUTE;

     struct tm *time = gmtime(&rawtime);

@@ -18,13 +18,19 @@

     comment += 36;

-    if (timezone < 0) sprintf(comment, "%d", timezone);
+    if (timezoneHours < 0) sprintf(comment, "%d", timezoneHours);

-    if (timezone > 0) sprintf(comment, "+%d", timezone);
+    if (timezoneHours > 0) sprintf(comment, "+%d", timezoneHours);

-    if (timezone < 0 || timezone > 0) comment += 2;
+    if (timezoneHours < 0 || timezoneHours > 0) comment += 2;

-    if (timezone < -9 || timezone > 9) comment += 1;
+    if (timezoneHours < -9 || timezoneHours > 9) comment += 1;
+
+    if (timezoneMinutes < 0) sprintf(comment, ":%2d", -timezoneMinutes);
+
+    if (timezoneMinutes > 0) sprintf(comment, ":%2d", timezoneMinutes);
+
+    if (timezoneMinutes < 0 || timezoneMinutes > 0) comment += 3;

     sprintf(comment, ") by %s at gain setting %d while battery state was ", artist,␣
↪(unsigned int)gain);
```

Code

```
void setHeaderComment(uint32_t currentTime, int8_t timezoneHours, int8_t timezoneMinutes,
↪ uint8_t *serialNumber, uint32_t gain, AM_batteryState_t batteryState, bool␣
↪batteryVoltageLow, bool switchPositionChanged) {

    time_t rawtime = currentTime + timezoneHours * SECONDS_IN_HOUR + timezoneMinutes *␣
↪SECONDS_IN_MINUTE;

    struct tm *time = gmtime(&rawtime);

    /* Format artist field */

    char *artist = wavHeader.iart.artist;

    sprintf(artist, "AudioMoth %08X%08X", (unsigned int)*((uint32_t*)serialNumber + 1),␣
↪(unsigned int)*((uint32_t*)serialNumber));

    /* Format comment field */

    char *comment = wavHeader.icmt.comment;

    sprintf(comment, "Recorded at %02d:%02d:%02d %02d/%02d/%04d (UTC", time->tm_hour,␣
↪time->tm_min, time->tm_sec, time->tm_mday, 1 + time->tm_mon, 1900 + time->tm_year);

    comment += 36;

    if (timezoneHours < 0) sprintf(comment, "%d", timezoneHours);

    if (timezoneHours > 0) sprintf(comment, "+%d", timezoneHours);

    if (timezoneHours < 0 || timezoneHours > 0) comment += 2;

    if (timezoneHours < -9 || timezoneHours > 9) comment += 1;

    if (timezoneMinutes < 0) sprintf(comment, ":%2d", -timezoneMinutes);

    if (timezoneMinutes > 0) sprintf(comment, ":%2d", timezoneMinutes);

    if (timezoneMinutes < 0 || timezoneMinutes > 0) comment += 3;

    sprintf(comment, ") by %s at gain setting %d while battery state was ", artist,␣
↪(unsigned int)gain);

    comment += 74;

    if (batteryState == AM_BATTERY_LOW) {

        sprintf(comment, "less than 3.6V.");

        comment += 15;

    } else if (batteryState >= AM_BATTERY_FULL) {
```

(continues on next page)

(continued from previous page)

```
        sprintf(comment, "greater than 4.9V.");
        comment += 18;
    } else {
        batteryState += 35;
        sprintf(comment, "%01d.%01dV.", batteryState / 10, batteryState % 10);
        comment += 5;
    }
    if (batteryVoltageLow || switchPositionChanged) {
        sprintf(comment, " Recording cancelled before completion due to ");
        comment += 46;
        if (batteryVoltageLow) {
            sprintf(comment, "low battery voltage.");
        } else if (switchPositionChanged) {
            sprintf(comment, "change of switch position.");
        }
    }
}
```

8.1.7 Version 1.3.0

No changes in this version.

8.1.8 Version 1.4.0

Diff

```
@@ -1,4 +1,4 @@
-void setHeaderComment(uint32_t currentTime, int8_t timezoneHours, int8_t␣
↪timezoneMinutes, uint8_t *serialNumber, uint32_t gain, AM_batteryState_t batteryState,␣
↪bool batteryVoltageLow, bool switchPositionChanged) {
+static void setHeaderComment(wavHeader_t *wavHeader, uint32_t currentTime, int8_t␣
↪timezoneHours, int8_t timezoneMinutes, uint8_t *serialNumber, uint32_t gain, AM_
```

(continues on next page)

(continued from previous page)

```
↪extendedBatteryState_t extendedBatteryState, int32_t temperature, bool␣
↪supplyVoltageLow, bool switchPositionChanged, uint32_t amplitudeThreshold, AM_
↪filterType_t filterType, uint32_t lowerFilterFreq, uint32_t higherFilterFreq) {

    time_t rawtime = currentTime + timezoneHours * SECONDS_IN_HOUR + timezoneMinutes *␣
↪SECONDS_IN_MINUTE;

@@ -6,71 +6,93 @@

     /* Format artist field */

-    char *artist = wavHeader.iart.artist;
+    char *artist = wavHeader->iart.artist;

     sprintf(artist, "AudioMoth %08X%08X", (unsigned int)*((uint32_t*)serialNumber + 1),␣
↪(unsigned int)*((uint32_t*)serialNumber));

     /* Format comment field */

-    char *comment = wavHeader.icmt.comment;
+    char *comment = wavHeader->icmt.comment;

-    sprintf(comment, "Recorded at %02d:%02d:%02d %02d/%02d/%04d (UTC", time->tm_hour,␣
↪time->tm_min, time->tm_sec, time->tm_mday, 1 + time->tm_mon, 1900 + time->tm_year);
+    comment += sprintf(comment, "Recorded at %02d:%02d:%02d %02d/%02d/%04d (UTC", time->
↪tm_hour, time->tm_min, time->tm_sec, time->tm_mday, 1 + time->tm_mon, 1900 + time->tm_
↪year);

-    comment += 36;
+    if (timezoneHours < 0) {

-    if (timezoneHours < 0) sprintf(comment, "%d", timezoneHours);
+        comment += sprintf(comment, "%d", timezoneHours);

-    if (timezoneHours > 0) sprintf(comment, "+%d", timezoneHours);
+    } else if (timezoneHours > 0) {

-    if (timezoneHours < 0 || timezoneHours > 0) comment += 2;
+        comment += sprintf(comment, "+%d", timezoneHours);

-    if (timezoneHours < -9 || timezoneHours > 9) comment += 1;
+    } else {

-    if (timezoneMinutes < 0) sprintf(comment, ":%2d", -timezoneMinutes);
+        if (timezoneMinutes < 0) comment += sprintf(comment, "-%d", timezoneHours);

-    if (timezoneMinutes > 0) sprintf(comment, ":%2d", timezoneMinutes);
+        if (timezoneMinutes > 0) comment += sprintf(comment, "+%d", timezoneHours);

-    if (timezoneMinutes < 0 || timezoneMinutes > 0) comment += 3;
+    }
```

(continues on next page)

(continued from previous page)

```
-    sprintf(comment, ") by %s at gain setting %d while battery state was ", artist,␣
↪(unsigned int)gain);
+    if (timezoneMinutes < 0) comment += sprintf(comment, ":%02d", -timezoneMinutes);

-    comment += 74;
+    if (timezoneMinutes > 0) comment += sprintf(comment, ":%02d", timezoneMinutes);

-    if (batteryState == AM_BATTERY_LOW) {
+    static char *gainSettings[5] = {"low", "low-medium", "medium", "medium-high", "high
↪"};

-        sprintf(comment, "less than 3.6V.");
+    comment +=  sprintf(comment, ") by %s at %s gain setting while battery state was ",␣
↪artist, gainSettings[gain]);

-        comment += 15;
+    if (extendedBatteryState == AM_EXT_BAT_LOW) {

-    } else if (batteryState >= AM_BATTERY_FULL) {
+        comment += sprintf(comment, "less than 2.5V");

-        sprintf(comment, "greater than 4.9V.");
+    } else if (extendedBatteryState >= AM_EXT_BAT_FULL) {

-        comment += 18;
+        comment += sprintf(comment, "greater than 4.9V");

     } else {

-        batteryState += 35;
+        uint32_t batteryVoltage =  extendedBatteryState + AM_EXT_BAT_STATE_OFFSET / AM_
↪BATTERY_STATE_INCREMENT;
+
+        comment += sprintf(comment, "%01d.%01dV", (unsigned int)batteryVoltage / 10,␣
↪(unsigned int)batteryVoltage % 10);
+
+    }
+
+    char *sign = temperature < 0 ? "-" : "";
+
+    uint32_t temperatureInDecidegrees = ROUNDED_DIV(ABS(temperature), 100);
+
+    comment += sprintf(comment, " and temperature was %s%d.%dC.", sign, (unsigned␣
↪int)temperatureInDecidegrees / 10, (unsigned int)temperatureInDecidegrees % 10);

-        sprintf(comment, "%01d.%01dV.", batteryState / 10, batteryState % 10);
+    if (amplitudeThreshold > 0) {

-        comment += 5;
+        comment += sprintf(comment, " Amplitude threshold was %d.", (unsigned␣
↪int)amplitudeThreshold);
```

(continues on next page)

(continued from previous page)

```
     }

-    if (batteryVoltageLow || switchPositionChanged) {
+    if (filterType == LOW_PASS_FILTER) {
+
+        comment += sprintf(comment, " Low-pass filter applied with cut-off frequency of
↪%01d.%01dkHz.", (unsigned int)higherFilterFreq / 10, (unsigned int)higherFilterFreq %␣
↪10);
+
+    } else if (filterType == BAND_PASS_FILTER) {
+
+        comment += sprintf(comment, " Band-pass filter applied with cut-off frequencies␣
↪of %01d.%01dkHz and %01d.%01dkHz.", (unsigned int)lowerFilterFreq / 10, (unsigned␣
↪int)lowerFilterFreq % 10, (unsigned int)higherFilterFreq / 10, (unsigned␣
↪int)higherFilterFreq % 10);
+
+    } else if (filterType == HIGH_PASS_FILTER) {
+
+        comment += sprintf(comment, " High-pass filter applied with cut-off frequency␣
↪of %01d.%01dkHz.", (unsigned int)lowerFilterFreq / 10, (unsigned int)lowerFilterFreq %␣
↪10);
+
+    }

-        sprintf(comment, " Recording cancelled before completion due to ");
+    if (supplyVoltageLow || switchPositionChanged) {

-        comment += 46;
+        comment += sprintf(comment, " Recording cancelled before completion due to ");

-        if (batteryVoltageLow) {
+        if (supplyVoltageLow) {

-            sprintf(comment, "low battery voltage.");
+            comment += sprintf(comment, "low voltage.");

         } else if (switchPositionChanged) {

-            sprintf(comment, "change of switch position.");
+            comment += sprintf(comment, "change of switch position.");

         }
```

Code

```
static void setHeaderComment(wavHeader_t *wavHeader, uint32_t currentTime, int8_t␣
↪timezoneHours, int8_t timezoneMinutes, uint8_t *serialNumber, uint32_t gain, AM_
↪extendedBatteryState_t extendedBatteryState, int32_t temperature, bool␣
↪supplyVoltageLow, bool switchPositionChanged, uint32_t amplitudeThreshold, AM_
↪filterType_t filterType, uint32_t lowerFilterFreq, uint32_t higherFilterFreq) {

    time_t rawtime = currentTime + timezoneHours * SECONDS_IN_HOUR + timezoneMinutes *␣
↪SECONDS_IN_MINUTE;

    struct tm *time = gmtime(&rawtime);

    /* Format artist field */

    char *artist = wavHeader->iart.artist;

    sprintf(artist, "AudioMoth %08X%08X", (unsigned int)*((uint32_t*)serialNumber + 1),␣
↪(unsigned int)*((uint32_t*)serialNumber));

    /* Format comment field */

    char *comment = wavHeader->icmt.comment;

    comment += sprintf(comment, "Recorded at %02d:%02d:%02d %02d/%02d/%04d (UTC", time->
↪tm_hour, time->tm_min, time->tm_sec, time->tm_mday, 1 + time->tm_mon, 1900 + time->tm_
↪year);

    if (timezoneHours < 0) {

        comment += sprintf(comment, "%d", timezoneHours);

    } else if (timezoneHours > 0) {

        comment += sprintf(comment, "+%d", timezoneHours);

    } else {

        if (timezoneMinutes < 0) comment += sprintf(comment, "-%d", timezoneHours);

        if (timezoneMinutes > 0) comment += sprintf(comment, "+%d", timezoneHours);

    }

    if (timezoneMinutes < 0) comment += sprintf(comment, ":%02d", -timezoneMinutes);

    if (timezoneMinutes > 0) comment += sprintf(comment, ":%02d", timezoneMinutes);

    static char *gainSettings[5] = {"low", "low-medium", "medium", "medium-high", "high"}
↪;

    comment +=  sprintf(comment, ") by %s at %s gain setting while battery state was ",␣
↪artist, gainSettings[gain]);
```

(continues on next page)

(continued from previous page)

```
    if (extendedBatteryState == AM_EXT_BAT_LOW) {

        comment += sprintf(comment, "less than 2.5V");

    } else if (extendedBatteryState >= AM_EXT_BAT_FULL) {

        comment += sprintf(comment, "greater than 4.9V");

    } else {

        uint32_t batteryVoltage =  extendedBatteryState + AM_EXT_BAT_STATE_OFFSET / AM_
↪BATTERY_STATE_INCREMENT;

        comment += sprintf(comment, "%01d.%01dV", (unsigned int)batteryVoltage / 10,␣
↪(unsigned int)batteryVoltage % 10);

    }

    char *sign = temperature < 0 ? "-" : "";

    uint32_t temperatureInDecidegrees = ROUNDED_DIV(ABS(temperature), 100);

    comment += sprintf(comment, " and temperature was %s%d.%dC.", sign, (unsigned␣
↪int)temperatureInDecidegrees / 10, (unsigned int)temperatureInDecidegrees % 10);

    if (amplitudeThreshold > 0) {

        comment += sprintf(comment, " Amplitude threshold was %d.", (unsigned␣
↪int)amplitudeThreshold);

    }

    if (filterType == LOW_PASS_FILTER) {

        comment += sprintf(comment, " Low-pass filter applied with cut-off frequency of
↪%01d.%01dkHz.", (unsigned int)higherFilterFreq / 10, (unsigned int)higherFilterFreq %␣
↪10);

    } else if (filterType == BAND_PASS_FILTER) {

        comment += sprintf(comment, " Band-pass filter applied with cut-off frequencies␣
↪of %01d.%01dkHz and %01d.%01dkHz.", (unsigned int)lowerFilterFreq / 10, (unsigned␣
↪int)lowerFilterFreq % 10, (unsigned int)higherFilterFreq / 10, (unsigned␣
↪int)higherFilterFreq % 10);

    } else if (filterType == HIGH_PASS_FILTER) {

        comment += sprintf(comment, " High-pass filter applied with cut-off frequency of
↪%01d.%01dkHz.", (unsigned int)lowerFilterFreq / 10, (unsigned int)lowerFilterFreq %␣
↪10);
```

(continues on next page)

(continued from previous page)

```
    }

    if (supplyVoltageLow || switchPositionChanged) {

        comment += sprintf(comment, " Recording cancelled before completion due to ");

        if (supplyVoltageLow) {

            comment += sprintf(comment, "low voltage.");

        } else if (switchPositionChanged) {

            comment += sprintf(comment, "change of switch position.");

        }

    }

}
```

8.1.9 Version 1.4.1

No changes in this version.

8.1.10 Version 1.4.2

Diff

```
@@ -1,4 +1,4 @@
-static void setHeaderComment(wavHeader_t *wavHeader, uint32_t currentTime, int8_t␣
↪timezoneHours, int8_t timezoneMinutes, uint8_t *serialNumber, uint32_t gain, AM_
↪extendedBatteryState_t extendedBatteryState, int32_t temperature, bool␣
↪supplyVoltageLow, bool switchPositionChanged, uint32_t amplitudeThreshold, AM_
↪filterType_t filterType, uint32_t lowerFilterFreq, uint32_t higherFilterFreq) {
+static void setHeaderComment(wavHeader_t *wavHeader, uint32_t currentTime, int8_t␣
↪timezoneHours, int8_t timezoneMinutes, uint8_t *serialNumber, uint32_t gain, AM_
↪extendedBatteryState_t extendedBatteryState, int32_t temperature, bool␣
↪switchPositionChanged, bool supplyVoltageLow, bool fileSizeLimited, uint32_t␣
↪amplitudeThreshold, AM_filterType_t filterType, uint32_t lowerFilterFreq, uint32_t␣
↪higherFilterFreq) {

     time_t rawtime = currentTime + timezoneHours * SECONDS_IN_HOUR + timezoneMinutes *␣
↪SECONDS_IN_MINUTE;

@@ -82,17 +82,21 @@

     }

-    if (supplyVoltageLow || switchPositionChanged) {
```

(continues on next page)

(continued from previous page)

```
+    if (supplyVoltageLow || switchPositionChanged || fileSizeLimited) {

         comment += sprintf(comment, " Recording cancelled before completion due to ");

-        if (supplyVoltageLow) {
+        if (switchPositionChanged) {
+
+            comment += sprintf(comment, "change of switch position.");
+
+        } else if (supplyVoltageLow) {

             comment += sprintf(comment, "low voltage.");

-        } else if (switchPositionChanged) {
+        } else if (fileSizeLimited) {

-            comment += sprintf(comment, "change of switch position.");
+            comment += sprintf(comment, "file size limit.");

         }
```

Code

```
static void setHeaderComment(wavHeader_t *wavHeader, uint32_t currentTime, int8_t␣
↪timezoneHours, int8_t timezoneMinutes, uint8_t *serialNumber, uint32_t gain, AM_
↪extendedBatteryState_t extendedBatteryState, int32_t temperature, bool␣
↪switchPositionChanged, bool supplyVoltageLow, bool fileSizeLimited, uint32_t␣
↪amplitudeThreshold, AM_filterType_t filterType, uint32_t lowerFilterFreq, uint32_t␣
↪higherFilterFreq) {

    time_t rawtime = currentTime + timezoneHours * SECONDS_IN_HOUR + timezoneMinutes *␣
↪SECONDS_IN_MINUTE;

    struct tm *time = gmtime(&rawtime);

    /* Format artist field */

    char *artist = wavHeader->iart.artist;

    sprintf(artist, "AudioMoth %08X%08X", (unsigned int)*((uint32_t*)serialNumber + 1),␣
↪(unsigned int)*((uint32_t*)serialNumber));

    /* Format comment field */

    char *comment = wavHeader->icmt.comment;

    comment += sprintf(comment, "Recorded at %02d:%02d:%02d %02d/%02d/%04d (UTC", time->
↪tm_hour, time->tm_min, time->tm_sec, time->tm_mday, 1 + time->tm_mon, 1900 + time->tm_
↪year);
```

(continues on next page)

(continued from previous page)

```
    if (timezoneHours < 0) {

        comment += sprintf(comment, "%d", timezoneHours);

    } else if (timezoneHours > 0) {

        comment += sprintf(comment, "+%d", timezoneHours);

    } else {

        if (timezoneMinutes < 0) comment += sprintf(comment, "-%d", timezoneHours);

        if (timezoneMinutes > 0) comment += sprintf(comment, "+%d", timezoneHours);

    }

    if (timezoneMinutes < 0) comment += sprintf(comment, ":%02d", -timezoneMinutes);

    if (timezoneMinutes > 0) comment += sprintf(comment, ":%02d", timezoneMinutes);

    static char *gainSettings[5] = {"low", "low-medium", "medium", "medium-high", "high"}
↪;

    comment +=  sprintf(comment, ") by %s at %s gain setting while battery state was ",␣
↪artist, gainSettings[gain]);

    if (extendedBatteryState == AM_EXT_BAT_LOW) {

        comment += sprintf(comment, "less than 2.5V");

    } else if (extendedBatteryState >= AM_EXT_BAT_FULL) {

        comment += sprintf(comment, "greater than 4.9V");

    } else {

        uint32_t batteryVoltage =  extendedBatteryState + AM_EXT_BAT_STATE_OFFSET / AM_
↪BATTERY_STATE_INCREMENT;

        comment += sprintf(comment, "%01d.%01dV", (unsigned int)batteryVoltage / 10,␣
↪(unsigned int)batteryVoltage % 10);

    }

    char *sign = temperature < 0 ? "-" : "";

    uint32_t temperatureInDecidegrees = ROUNDED_DIV(ABS(temperature), 100);

    comment += sprintf(comment, " and temperature was %s%d.%dC.", sign, (unsigned␣
↪int)temperatureInDecidegrees / 10, (unsigned int)temperatureInDecidegrees % 10);

    if (amplitudeThreshold > 0) {
```

(continues on next page)

(continued from previous page)

```
        comment += sprintf(comment, " Amplitude threshold was %d.", (unsigned␣
↪int)amplitudeThreshold);

    }

    if (filterType == LOW_PASS_FILTER) {

        comment += sprintf(comment, " Low-pass filter applied with cut-off frequency of
↪%01d.%01dkHz.", (unsigned int)higherFilterFreq / 10, (unsigned int)higherFilterFreq %␣
↪10);

    } else if (filterType == BAND_PASS_FILTER) {

        comment += sprintf(comment, " Band-pass filter applied with cut-off frequencies␣
↪of %01d.%01dkHz and %01d.%01dkHz.", (unsigned int)lowerFilterFreq / 10, (unsigned␣
↪int)lowerFilterFreq % 10, (unsigned int)higherFilterFreq / 10, (unsigned␣
↪int)higherFilterFreq % 10);

    } else if (filterType == HIGH_PASS_FILTER) {

        comment += sprintf(comment, " High-pass filter applied with cut-off frequency of
↪%01d.%01dkHz.", (unsigned int)lowerFilterFreq / 10, (unsigned int)lowerFilterFreq %␣
↪10);

    }

    if (supplyVoltageLow || switchPositionChanged || fileSizeLimited) {

        comment += sprintf(comment, " Recording cancelled before completion due to ");

        if (switchPositionChanged) {

            comment += sprintf(comment, "change of switch position.");

        } else if (supplyVoltageLow) {

            comment += sprintf(comment, "low voltage.");

        } else if (fileSizeLimited) {

            comment += sprintf(comment, "file size limit.");

        }

    }

}
```

8.1.11 Version 1.4.3

No changes in this version.

8.1.12 Version 1.4.4

No changes in this version.

8.1.13 Version 1.5.0

Diff

```
@@ -1,4 +1,4 @@
-static void setHeaderComment(wavHeader_t *wavHeader, uint32_t currentTime, int8_t␣
↪timezoneHours, int8_t timezoneMinutes, uint8_t *serialNumber, uint32_t gain, AM_
↪extendedBatteryState_t extendedBatteryState, int32_t temperature, bool␣
↪switchPositionChanged, bool supplyVoltageLow, bool fileSizeLimited, uint32_t␣
↪amplitudeThreshold, AM_filterType_t filterType, uint32_t lowerFilterFreq, uint32_t␣
↪higherFilterFreq) {
+static void setHeaderComment(wavHeader_t *wavHeader, uint32_t currentTime, int8_t␣
↪timezoneHours, int8_t timezoneMinutes, uint8_t *serialNumber, uint8_t *deploymentID,␣
↪uint8_t *defaultDeploymentID, uint32_t gain, AM_extendedBatteryState_t␣
↪extendedBatteryState, int32_t temperature, bool externalMicrophone, AM_recordingState_
↪t recordingState, uint32_t amplitudeThreshold, AM_filterType_t filterType, uint32_t␣
↪lowerFilterFreq, uint32_t higherFilterFreq) {

     time_t rawtime = currentTime + timezoneHours * SECONDS_IN_HOUR + timezoneMinutes *␣
↪SECONDS_IN_MINUTE;

@@ -8,7 +8,7 @@

     char *artist = wavHeader->iart.artist;

-    sprintf(artist, "AudioMoth %08X%08X", (unsigned int)*((uint32_t*)serialNumber + 1),␣
↪(unsigned int)*((uint32_t*)serialNumber));
+    sprintf(artist, "AudioMoth " SERIAL_NUMBER, FORMAT_SERIAL_NUMBER(serialNumber));

     /* Format comment field */

@@ -36,9 +36,25 @@

     if (timezoneMinutes > 0) comment += sprintf(comment, ":%02d", timezoneMinutes);

+    if (memcmp(deploymentID, defaultDeploymentID, DEPLOYMENT_ID_LENGTH)) {
+
+        comment += sprintf(comment, ") during deployment " SERIAL_NUMBER " ", FORMAT_
↪SERIAL_NUMBER(deploymentID));
+
+    } else {
+
+        comment += sprintf(comment, ") by %s ", artist);
```

(continues on next page)

(continued from previous page)

```
+
+    }
+
+    if (externalMicrophone) {
+
+        comment += sprintf(comment, "using external microphone ");
+
+    }
+
     static char *gainSettings[5] = {"low", "low-medium", "medium", "medium-high", "high
↪"};

-    comment +=  sprintf(comment, ") by %s at %s gain setting while battery state was ",␣
↪artist, gainSettings[gain]);
+    comment += sprintf(comment, "at %s gain setting while battery state was ",␣
↪gainSettings[gain]);

     if (extendedBatteryState == AM_EXT_BAT_LOW) {

@@ -82,19 +98,23 @@

     }

-    if (supplyVoltageLow || switchPositionChanged || fileSizeLimited) {
+    if (recordingState != RECORDING_OKAY) {

         comment += sprintf(comment, " Recording cancelled before completion due to ");

-        if (switchPositionChanged) {
+        if (recordingState == MICROPHONE_CHANGED) {
+
+            comment += sprintf(comment, "microphone change.");
+
+        } else if (recordingState == SWITCH_CHANGED) {

             comment += sprintf(comment, "change of switch position.");

-        } else if (supplyVoltageLow) {
+        } else if (recordingState == SUPPLY_VOLTAGE_LOW) {

             comment += sprintf(comment, "low voltage.");

-        } else if (fileSizeLimited) {
+        } else if (recordingState == FILE_SIZE_LIMITED) {

             comment += sprintf(comment, "file size limit.");
```

Code

```
static void setHeaderComment(wavHeader_t *wavHeader, uint32_t currentTime, int8_t␣
↪timezoneHours, int8_t timezoneMinutes, uint8_t *serialNumber, uint8_t *deployment ID,␣
↪uint8_t *defaultDeploymentID, uint32_t gain, AM_extendedBatteryState_t␣
↪extendedBatteryState, int32_t temperature, bool externalMicrophone, AM_recordingState_
↪t recordingState, uint32_t amplitudeThreshold, AM_filterType_t filterType, uint32_t␣
↪lowerFilterFreq, uint32_t higherFilterFreq) {

    time_t rawtime = currentTime + timezoneHours * SECONDS_IN_HOUR + timezoneMinutes *␣
↪SECONDS_IN_MINUTE;

    struct tm *time = gmtime(&rawtime);

    /* Format artist field */

    char *artist = wavHeader->iart.artist;

    sprintf(artist, "AudioMoth " SERIAL_NUMBER, FORMAT_SERIAL_NUMBER(serialNumber));

    /* Format comment field */

    char *comment = wavHeader->icmt.comment;

    comment += sprintf(comment, "Recorded at %02d:%02d:%02d %02d/%02d/%04d (UTC", time->
↪tm_hour, time->tm_min, time->tm_sec, time->tm_mday, 1 + time->tm_mon, 1900 + time->tm_
↪year);

    if (timezoneHours < 0) {

        comment += sprintf(comment, "%d", timezoneHours);

    } else if (timezoneHours > 0) {

        comment += sprintf(comment, "+%d", timezoneHours);

    } else {

        if (timezoneMinutes < 0) comment += sprintf(comment, "-%d", timezoneHours);

        if (timezoneMinutes > 0) comment += sprintf(comment, "+%d", timezoneHours);

    }

    if (timezoneMinutes < 0) comment += sprintf(comment, ":%02d", -timezoneMinutes);

    if (timezoneMinutes > 0) comment += sprintf(comment, ":%02d", timezoneMinutes);

    if (memcmp(deployment ID, defaultDeploymentID, DEPLOYMENT_ID_LENGTH)) {

        comment += sprintf(comment, ") during deployment " SERIAL_NUMBER " ", FORMAT_
↪SERIAL_NUMBER(deployment ID));
```

(continues on next page)

(continued from previous page)

```
    } else {

        comment += sprintf(comment, ") by %s ", artist);

    }

    if (externalMicrophone) {

        comment += sprintf(comment, "using external microphone ");

    }

    static char *gainSettings[5] = {"low", "low-medium", "medium", "medium-high", "high"}
↪;

    comment += sprintf(comment, "at %s gain setting while battery state was ",␣
↪gainSettings[gain]);

    if (extendedBatteryState == AM_EXT_BAT_LOW) {

        comment += sprintf(comment, "less than 2.5V");

    } else if (extendedBatteryState >= AM_EXT_BAT_FULL) {

        comment += sprintf(comment, "greater than 4.9V");

    } else {

        uint32_t batteryVoltage =  extendedBatteryState + AM_EXT_BAT_STATE_OFFSET / AM_
↪BATTERY_STATE_INCREMENT;

        comment += sprintf(comment, "%01d.%01dV", (unsigned int)batteryVoltage / 10,␣
↪(unsigned int)batteryVoltage % 10);

    }

    char *sign = temperature < 0 ? "-" : "";

    uint32_t temperatureInDecidegrees = ROUNDED_DIV(ABS(temperature), 100);

    comment += sprintf(comment, " and temperature was %s%d.%dC.", sign, (unsigned␣
↪int)temperatureInDecidegrees / 10, (unsigned int)temperatureInDecidegrees % 10);

    if (amplitudeThreshold > 0) {

        comment += sprintf(comment, " Amplitude threshold was %d.", (unsigned␣
↪int)amplitudeThreshold);

    }

    if (filterType == LOW_PASS_FILTER) {
```

(continues on next page)

(continued from previous page)

```
        comment += sprintf(comment, " Low-pass filter applied with cut-off frequency of
↪%01d.%01dkHz.", (unsigned int)higherFilterFreq / 10, (unsigned int)higherFilterFreq %␣
↪10);

    } else if (filterType == BAND_PASS_FILTER) {

        comment += sprintf(comment, " Band-pass filter applied with cut-off frequencies␣
↪of %01d.%01dkHz and %01d.%01dkHz.", (unsigned int)lowerFilterFreq / 10, (unsigned␣
↪int)lowerFilterFreq % 10, (unsigned int)higherFilterFreq / 10, (unsigned␣
↪int)higherFilterFreq % 10);

    } else if (filterType == HIGH_PASS_FILTER) {

        comment += sprintf(comment, " High-pass filter applied with cut-off frequency of
↪%01d.%01dkHz.", (unsigned int)lowerFilterFreq / 10, (unsigned int)lowerFilterFreq %␣
↪10);

    }

    if (recordingState != RECORDING_OKAY) {

        comment += sprintf(comment, " Recording cancelled before completion due to ");

        if (recordingState == MICROPHONE_CHANGED) {

            comment += sprintf(comment, "microphone change.");

        } else if (recordingState == SWITCH_CHANGED) {

            comment += sprintf(comment, "change of switch position.");

        } else if (recordingState == SUPPLY_VOLTAGE_LOW) {

            comment += sprintf(comment, "low voltage.");

        } else if (recordingState == FILE_SIZE_LIMITED) {

            comment += sprintf(comment, "file size limit.");

        }

    }

}
```

8.1.14 Version 1.6.0

Diff

```
@@ -1,8 +1,8 @@
-static void setHeaderComment(wavHeader_t *wavHeader, uint32_t currentTime, int8_t␣
↪timezoneHours, int8_t timezoneMinutes, uint8_t *serialNumber, uint8_t *deploymentID,␣
↪uint8_t *defaultDeploymentID, uint32_t gain, AM_extendedBatteryState_t␣
↪extendedBatteryState, int32_t temperature, bool externalMicrophone, AM_recordingState_
↪t recordingState, uint32_t amplitudeThreshold, AM_filterType_t filterType, uint32_t␣
↪lowerFilterFreq, uint32_t higherFilterFreq) {
+static void setHeaderComment(wavHeader_t *wavHeader, configSettings_t *configSettings,␣
↪uint32_t currentTime, uint8_t *serialNumber, uint8_t *deploymentID, uint8_t␣
↪*defaultDeploymentID, AM_extendedBatteryState_t extendedBatteryState, int32_t␣
↪temperature, bool externalMicrophone, AM_recordingState_t recordingState, AM_
↪filterType_t filterType) {

-    time_t rawtime = currentTime + timezoneHours * SECONDS_IN_HOUR + timezoneMinutes *␣
↪SECONDS_IN_MINUTE;
+    time_t rawTime = currentTime + configSettings->timezoneHours * SECONDS_IN_HOUR +␣
↪configSettings->timezoneMinutes * SECONDS_IN_MINUTE;

-    struct tm *time = gmtime(&rawtime);
+    struct tm *time = gmtime(&rawTime);

     /* Format artist field */

@@ -16,6 +16,10 @@

     comment += sprintf(comment, "Recorded at %02d:%02d:%02d %02d/%02d/%04d (UTC", time->
↪tm_hour, time->tm_min, time->tm_sec, time->tm_mday, 1 + time->tm_mon, 1900 + time->tm_
↪year);

+    int8_t timezoneHours = configSettings->timezoneHours;
+
+    int8_t timezoneMinutes = configSettings->timezoneMinutes;
+
     if (timezoneHours < 0) {

         comment += sprintf(comment, "%d", timezoneHours);
@@ -54,7 +58,7 @@

     static char *gainSettings[5] = {"low", "low-medium", "medium", "medium-high", "high
↪"};

-    comment += sprintf(comment, "at %s gain setting while battery state was ",␣
↪gainSettings[gain]);
+    comment += sprintf(comment, "at %s gain while battery was ",␣
↪gainSettings[configSettings->gain]);

     if (extendedBatteryState == AM_EXT_BAT_LOW) {

@@ -68,7 +72,7 @@
```

(continues on next page)

(continued from previous page)

```
         uint32_t batteryVoltage =  extendedBatteryState + AM_EXT_BAT_STATE_OFFSET / AM_
↪BATTERY_STATE_INCREMENT;

-        comment += sprintf(comment, "%01d.%01dV", (unsigned int)batteryVoltage / 10,␣
↪(unsigned int)batteryVoltage % 10);
+        comment += sprintf(comment, "%01ld.%01ldV", batteryVoltage / 10, batteryVoltage
↪% 10);

     }

@@ -76,31 +80,49 @@

     uint32_t temperatureInDecidegrees = ROUNDED_DIV(ABS(temperature), 100);

-    comment += sprintf(comment, " and temperature was %s%d.%dC.", sign, (unsigned␣
↪int)temperatureInDecidegrees / 10, (unsigned int)temperatureInDecidegrees % 10);
+    comment += sprintf(comment, " and temperature was %s%ld.%ldC.", sign,␣
↪temperatureInDecidegrees / 10, temperatureInDecidegrees % 10);
+
+    bool amplitudeThresholdEnabled = configSettings->amplitudeThreshold > 0 ||␣
↪configSettings->enableAmplitudeThresholdDecibelScale || configSettings->
↪enableAmplitudeThresholdPercentageScale;
+
+    if (amplitudeThresholdEnabled) comment += sprintf(comment, " Amplitude threshold␣
↪was ");
+
+    if (configSettings->enableAmplitudeThresholdDecibelScale && configSettings->
↪enableAmplitudeThresholdPercentageScale == false) {
+
+        comment += formatDecibels(comment, configSettings->amplitudeThresholdDecibels);

-    if (amplitudeThreshold > 0) {
+    } else if (configSettings->enableAmplitudeThresholdPercentageScale &&␣
↪configSettings->enableAmplitudeThresholdDecibelScale == false) {

-        comment += sprintf(comment, " Amplitude threshold was %d.", (unsigned␣
↪int)amplitudeThreshold);
+        comment += formatPercentage(comment, configSettings->
↪amplitudeThresholdPercentageMantissa, configSettings->
↪amplitudeThresholdPercentageExponent);
+
+    } else if (amplitudeThresholdEnabled) {
+
+        comment += sprintf(comment, "%d", configSettings->amplitudeThreshold);

     }
```

Code

```
static void setHeaderComment(wavHeader_t *wavHeader, configSettings_t *configSettings,␣
↪uint32_t currentTime, uint8_t *serialNumber, uint8_t *deploymentID, uint8_t␣
↪*defaultDeploymentID, AM_extendedBatteryState_t extendedBatteryState, int32_t␣
↪temperature, bool externalMicrophone, AM_recordingState_t recordingState, AM_
↪filterType_t filterType) {

    time_t rawTime = currentTime + configSettings->timezoneHours * SECONDS_IN_HOUR +␣
↪configSettings->timezoneMinutes * SECONDS_IN_MINUTE;

    struct tm *time = gmtime(&rawTime);

    /* Format artist field */

    char *artist = wavHeader->iart.artist;

    sprintf(artist, "AudioMoth " SERIAL_NUMBER, FORMAT_SERIAL_NUMBER(serialNumber));

    /* Format comment field */

    char *comment = wavHeader->icmt.comment;

    comment += sprintf(comment, "Recorded at %02d:%02d:%02d %02d/%02d/%04d (UTC", time->
↪tm_hour, time->tm_min, time->tm_sec, time->tm_mday, 1 + time->tm_mon, 1900 + time->tm_
↪year);

    int8_t timezoneHours = configSettings->timezoneHours;

    int8_t timezoneMinutes = configSettings->timezoneMinutes;

    if (timezoneHours < 0) {

        comment += sprintf(comment, "%d", timezoneHours);

    } else if (timezoneHours > 0) {

        comment += sprintf(comment, "+%d", timezoneHours);

    } else {

        if (timezoneMinutes < 0) comment += sprintf(comment, "-%d", timezoneHours);

        if (timezoneMinutes > 0) comment += sprintf(comment, "+%d", timezoneHours);

    }

    if (timezoneMinutes < 0) comment += sprintf(comment, ":%02d", -timezoneMinutes);

    if (timezoneMinutes > 0) comment += sprintf(comment, ":%02d", timezoneMinutes);

    if (memcmp(deployment_ID, defaultDeploymentID, DEPLOYMENT_ID_LENGTH)) {
```

(continues on next page)

(continued from previous page)

```
        comment += sprintf(comment, ") during deployment " SERIAL_NUMBER " ", FORMAT_
↪SERIAL_NUMBER(deploymentID));

    } else {

        comment += sprintf(comment, ") by %s ", artist);

    }

    if (externalMicrophone) {

        comment += sprintf(comment, "using external microphone ");

    }

    static char *gainSettings[5] = {"low", "low-medium", "medium", "medium-high", "high"}
↪;

    comment += sprintf(comment, "at %s gain while battery was ",␣
↪gainSettings[configSettings->gain]);

    if (extendedBatteryState == AM_EXT_BAT_LOW) {

        comment += sprintf(comment, "less than 2.5V");

    } else if (extendedBatteryState >= AM_EXT_BAT_FULL) {

        comment += sprintf(comment, "greater than 4.9V");

    } else {

        uint32_t batteryVoltage =  extendedBatteryState + AM_EXT_BAT_STATE_OFFSET / AM_
↪BATTERY_STATE_INCREMENT;

        comment += sprintf(comment, "%01ld.%01ldV", batteryVoltage / 10, batteryVoltage
↪% 10);

    }

    char *sign = temperature < 0 ? "-" : "";

    uint32_t temperatureInDecidegrees = ROUNDED_DIV(ABS(temperature), 100);

    comment += sprintf(comment, " and temperature was %s%ld.%ldC.", sign,␣
↪temperatureInDecidegrees / 10, temperatureInDecidegrees % 10);

    bool amplitudeThresholdEnabled = configSettings->amplitudeThreshold > 0 ||␣
↪configSettings->enableAmplitudeThresholdDecibelScale || configSettings->
↪enableAmplitudeThresholdPercentageScale;

    if (amplitudeThresholdEnabled) comment += sprintf(comment, " Amplitude threshold was
↪");
```

(continues on next page)

(continued from previous page)

```
    if (configSettings->enableAmplitudeThresholdDecibelScale && configSettings->
↪enableAmplitudeThresholdPercentageScale == false) {

        comment += formatDecibels(comment, configSettings->amplitudeThresholdDecibels);

    } else if (configSettings->enableAmplitudeThresholdPercentageScale && configSettings-
↪>enableAmplitudeThresholdDecibelScale == false) {

        comment += formatPercentage(comment, configSettings->
↪amplitudeThresholdPercentageMantissa, configSettings->
↪amplitudeThresholdPercentageExponent);

    } else if (amplitudeThresholdEnabled) {

        comment += sprintf(comment, "%d", configSettings->amplitudeThreshold);

    }

    if (amplitudeThresholdEnabled) comment += sprintf(comment, " with %ds minimum␣
↪trigger duration.", configSettings->minimumTriggerDuration);

    uint16_t lowerFilterFreq = configSettings->lowerFilterFreq;

    uint16_t higherFilterFreq = configSettings->higherFilterFreq;

    if (filterType == LOW_PASS_FILTER) {

        comment += sprintf(comment, " Low-pass filter with frequency of %01d.%01dkHz␣
↪applied.", higherFilterFreq / 10, higherFilterFreq % 10);

    } else if (filterType == BAND_PASS_FILTER) {

        comment += sprintf(comment, " Band-pass filter with frequencies of %01d.%01dkHz␣
↪and %01d.%01dkHz applied.", lowerFilterFreq / 10, lowerFilterFreq % 10,␣
↪higherFilterFreq / 10, higherFilterFreq % 10);

    } else if (filterType == HIGH_PASS_FILTER) {

        comment += sprintf(comment, " High-pass filter with frequency of %01d.%01dkHz␣
↪applied.", lowerFilterFreq / 10, lowerFilterFreq % 10);

    }

    if (recordingState != RECORDING_OKAY) {

        comment += sprintf(comment, " Recording stopped due to ");

        if (recordingState == MICROPHONE_CHANGED) {

            comment += sprintf(comment, "microphone change.");
```

(continues on next page)

(continued from previous page)

```
        } else if (recordingState == SWITCH_CHANGED) {

            comment += sprintf(comment, "switch position change.");

        } else if (recordingState == SUPPLY_VOLTAGE_LOW) {

            comment += sprintf(comment, "low voltage.");

        } else if (recordingState == FILE_SIZE_LIMITED) {

            comment += sprintf(comment, "file size limit.");

        }

    }

}
```

8.1.15 Version 1.7.0

Diff

```
--- 1.6.0
+++ 1.7.0
@@ -72,7 +72,7 @@

         uint32_t batteryVoltage =  extendedBatteryState + AM_EXT_BAT_STATE_OFFSET / AM_
↪BATTERY_STATE_INCREMENT;

-        comment += sprintf(comment, "%01ld.%01ldV", batteryVoltage / 10, batteryVoltage
↪% 10);
+        comment += sprintf(comment, "%01lu.%01luV", batteryVoltage / 10, batteryVoltage
↪% 10);

     }

@@ -80,7 +80,7 @@

     uint32_t temperatureInDecidegrees = ROUNDED_DIV(ABS(temperature), 100);

-    comment += sprintf(comment, " and temperature was %s%ld.%ldC.", sign,␣
↪temperatureInDecidegrees / 10, temperatureInDecidegrees % 10);
+    comment += sprintf(comment, " and temperature was %s%lu.%luC.", sign,␣
↪temperatureInDecidegrees / 10, temperatureInDecidegrees % 10);

     bool amplitudeThresholdEnabled = configSettings->amplitudeThreshold > 0 ||␣
↪configSettings->enableAmplitudeThresholdDecibelScale || configSettings->
↪enableAmplitudeThresholdPercentageScale;

@@ -96,11 +96,11 @@
```

(continues on next page)

(continued from previous page)

```
     } else if (amplitudeThresholdEnabled) {

-        comment += sprintf(comment, "%d", configSettings->amplitudeThreshold);
+        comment += sprintf(comment, "%u", configSettings->amplitudeThreshold);

     }

-    if (amplitudeThresholdEnabled) comment += sprintf(comment, " with %ds minimum␣
↪trigger duration.", configSettings->minimumTriggerDuration);
+    if (amplitudeThresholdEnabled) comment += sprintf(comment, " with %us minimum␣
↪trigger duration.", configSettings->minimumTriggerDuration);

     uint16_t lowerFilterFreq = configSettings->lowerFilterFreq;

@@ -108,37 +108,41 @@

     if (filterType == LOW_PASS_FILTER) {

-        comment += sprintf(comment, " Low-pass filter with frequency of %01d.%01dkHz␣
↪applied.", higherFilterFreq / 10, higherFilterFreq % 10);
+        comment += sprintf(comment, " Low-pass filter with frequency of %01u.%01ukHz␣
↪applied.", higherFilterFreq / 10, higherFilterFreq % 10);

     } else if (filterType == BAND_PASS_FILTER) {

-        comment += sprintf(comment, " Band-pass filter with frequencies of %01d.%01dkHz␣
↪and %01d.%01dkHz applied.", lowerFilterFreq / 10, lowerFilterFreq % 10,␣
↪higherFilterFreq / 10, higherFilterFreq % 10);
+        comment += sprintf(comment, " Band-pass filter with frequencies of %01u.%01ukHz␣
↪and %01u.%01ukHz applied.", lowerFilterFreq / 10, lowerFilterFreq % 10,␣
↪higherFilterFreq / 10, higherFilterFreq % 10);

     } else if (filterType == HIGH_PASS_FILTER) {

-        comment += sprintf(comment, " High-pass filter with frequency of %01d.%01dkHz␣
↪applied.", lowerFilterFreq / 10, lowerFilterFreq % 10);
+        comment += sprintf(comment, " High-pass filter with frequency of %01u.%01ukHz␣
↪applied.", lowerFilterFreq / 10, lowerFilterFreq % 10);

     }

     if (recordingState != RECORDING_OKAY) {

-        comment += sprintf(comment, " Recording stopped due to ");
+        comment += sprintf(comment, " Recording stopped");

         if (recordingState == MICROPHONE_CHANGED) {

-            comment += sprintf(comment, "microphone change.");
+            comment += sprintf(comment, " due to microphone change.");
```

(continues on next page)

(continued from previous page)

```
         } else if (recordingState == SWITCH_CHANGED) {

-            comment += sprintf(comment, "switch position change.");
+            comment += sprintf(comment, " due to switch position change.");
+
+        } else if (recordingState == MAGNETIC_SWITCH) {
+
+            comment += sprintf(comment, " by magnetic switch.");

         } else if (recordingState == SUPPLY_VOLTAGE_LOW) {

-            comment += sprintf(comment, "low voltage.");
+            comment += sprintf(comment, " due to low voltage.");

         } else if (recordingState == FILE_SIZE_LIMITED) {

-            comment += sprintf(comment, "file size limit.");
+            comment += sprintf(comment, " due to file size limit.");

         }
```

Code

```
static void setHeaderComment(wavHeader_t *wavHeader, configSettings_t *configSettings,␣
↪uint32_t currentTime, uint8_t *serialNumber, uint8_t *deploymentID, uint8_t␣
↪*defaultDeploymentID, AM_extendedBatteryState_t extendedBatteryState, int32_t␣
↪temperature, bool externalMicrophone, AM_recordingState_t recordingState, AM_
↪filterType_t filterType) {

    time_t rawTime = currentTime + configSettings->timezoneHours * SECONDS_IN_HOUR +␣
↪configSettings->timezoneMinutes * SECONDS_IN_MINUTE;

    struct tm *time = gmtime(&rawTime);

    /* Format artist field */

    char *artist = wavHeader->iart.artist;

    sprintf(artist, "AudioMoth " SERIAL_NUMBER, FORMAT_SERIAL_NUMBER(serialNumber));

    /* Format comment field */

    char *comment = wavHeader->icmt.comment;

    comment += sprintf(comment, "Recorded at %02d:%02d:%02d %02d/%02d/%04d (UTC", time->
↪tm_hour, time->tm_min, time->tm_sec, time->tm_mday, 1 + time->tm_mon, 1900 + time->tm_
↪year);

    int8_t timezoneHours = configSettings->timezoneHours;
```

(continues on next page)

(continued from previous page)

```
    int8_t timezoneMinutes = configSettings->timezoneMinutes;

    if (timezoneHours < 0) {

        comment += sprintf(comment, "%d", timezoneHours);

    } else if (timezoneHours > 0) {

        comment += sprintf(comment, "+%d", timezoneHours);

    } else {

        if (timezoneMinutes < 0) comment += sprintf(comment, "-%d", timezoneHours);

        if (timezoneMinutes > 0) comment += sprintf(comment, "+%d", timezoneHours);

    }

    if (timezoneMinutes < 0) comment += sprintf(comment, ":%02d", -timezoneMinutes);

    if (timezoneMinutes > 0) comment += sprintf(comment, ":%02d", timezoneMinutes);

    if (memcmp(deploymentID, defaultDeploymentID, DEPLOYMENT_ID_LENGTH)) {

        comment += sprintf(comment, ") during deployment " SERIAL_NUMBER " ", FORMAT_
↪SERIAL_NUMBER(deploymentID));

    } else {

        comment += sprintf(comment, ") by %s ", artist);

    }

    if (externalMicrophone) {

        comment += sprintf(comment, "using external microphone ");

    }

    static char *gainSettings[5] = {"low", "low-medium", "medium", "medium-high", "high"}
↪;

    comment += sprintf(comment, "at %s gain while battery was ",␣
↪gainSettings[configSettings->gain]);

    if (extendedBatteryState == AM_EXT_BAT_LOW) {

        comment += sprintf(comment, "less than 2.5V");

    } else if (extendedBatteryState >= AM_EXT_BAT_FULL) {

        comment += sprintf(comment, "greater than 4.9V");
```

(continues on next page)

(continued from previous page)

```
    } else {

        uint32_t batteryVoltage =  extendedBatteryState + AM_EXT_BAT_STATE_OFFSET / AM_
↪BATTERY_STATE_INCREMENT;

        comment += sprintf(comment, "%01lu.%01luV", batteryVoltage / 10, batteryVoltage
↪% 10);

    }

    char *sign = temperature < 0 ? "-" : "";

    uint32_t temperatureInDecidegrees = ROUNDED_DIV(ABS(temperature), 100);

    comment += sprintf(comment, " and temperature was %s%lu.%luC.", sign,␣
↪temperatureInDecidegrees / 10, temperatureInDecidegrees % 10);

    bool amplitudeThresholdEnabled = configSettings->amplitudeThreshold > 0 ||␣
↪configSettings->enableAmplitudeThresholdDecibelScale || configSettings->
↪enableAmplitudeThresholdPercentageScale;

    if (amplitudeThresholdEnabled) comment += sprintf(comment, " Amplitude threshold was
↪");

    if (configSettings->enableAmplitudeThresholdDecibelScale && configSettings->
↪enableAmplitudeThresholdPercentageScale == false) {

        comment += formatDecibels(comment, configSettings->amplitudeThresholdDecibels);

    } else if (configSettings->enableAmplitudeThresholdPercentageScale && configSettings-
↪>enableAmplitudeThresholdDecibelScale == false) {

        comment += formatPercentage(comment, configSettings->
↪amplitudeThresholdPercentageMantissa, configSettings->
↪amplitudeThresholdPercentageExponent);

    } else if (amplitudeThresholdEnabled) {

        comment += sprintf(comment, "%u", configSettings->amplitudeThreshold);

    }

    if (amplitudeThresholdEnabled) comment += sprintf(comment, " with %us minimum␣
↪trigger duration.", configSettings->minimumTriggerDuration);

    uint16_t lowerFilterFreq = configSettings->lowerFilterFreq;

    uint16_t higherFilterFreq = configSettings->higherFilterFreq;

    if (filterType == LOW_PASS_FILTER) {
```

(continues on next page)

(continued from previous page)

```
        comment += sprintf(comment, " Low-pass filter with frequency of %01u.%01ukHz␣
↪applied.", higherFilterFreq / 10, higherFilterFreq % 10);

    } else if (filterType == BAND_PASS_FILTER) {

        comment += sprintf(comment, " Band-pass filter with frequencies of %01u.%01ukHz␣
↪and %01u.%01ukHz applied.", lowerFilterFreq / 10, lowerFilterFreq % 10,␣
↪higherFilterFreq / 10, higherFilterFreq % 10);

    } else if (filterType == HIGH_PASS_FILTER) {

        comment += sprintf(comment, " High-pass filter with frequency of %01u.%01ukHz␣
↪applied.", lowerFilterFreq / 10, lowerFilterFreq % 10);

    }

    if (recordingState != RECORDING_OKAY) {

        comment += sprintf(comment, " Recording stopped");

        if (recordingState == MICROPHONE_CHANGED) {

            comment += sprintf(comment, " due to microphone change.");

        } else if (recordingState == SWITCH_CHANGED) {

            comment += sprintf(comment, " due to switch position change.");

        } else if (recordingState == MAGNETIC_SWITCH) {

            comment += sprintf(comment, " by magnetic switch.");

        } else if (recordingState == SUPPLY_VOLTAGE_LOW) {

            comment += sprintf(comment, " due to low voltage.");

        } else if (recordingState == FILE_SIZE_LIMITED) {

            comment += sprintf(comment, " due to file size limit.");

        }

    }

}
```

8.1.16 Version 1.7.1

No changes in this version.

8.1.17 Version 1.8.0

Diff

```
--- 1.7.1.c
+++ 1.8.0.c
@@ -1,8 +1,10 @@
 static void setHeaderComment(wavHeader_t *wavHeader, configSettings_t *configSettings,␣
↪uint32_t currentTime, uint8_t *serialNumber, uint8_t *deployment ID, uint8_t␣
↪*defaultDeploymentID, AM_extendedBatteryState_t extendedBatteryState, int32_t␣
↪temperature, bool externalMicrophone, AM_recordingState_t recordingState, AM_
↪filterType_t filterType) {

+    struct tm time;
+
     time_t rawTime = currentTime + configSettings->timezoneHours * SECONDS_IN_HOUR +␣
↪configSettings->timezoneMinutes * SECONDS_IN_MINUTE;

-    struct tm *time = gmtime(&rawTime);
+    gmtime_r(&rawTime, &time);

     /* Format artist field */

@@ -14,7 +16,7 @@

     char *comment = wavHeader->icmt.comment;

-    comment += sprintf(comment, "Recorded at %02d:%02d:%02d %02d/%02d/%04d (UTC", time->
↪tm_hour, time->tm_min, time->tm_sec, time->tm_mday, 1 + time->tm_mon, 1900 + time->tm_
↪year);
+    comment += sprintf(comment, "Recorded at %02d:%02d:%02d %02d/%02d/%04d (UTC", time.
↪tm_hour, time.tm_min, time.tm_sec, time.tm_mday, 1 + time.tm_mon, 1900 + time.tm_year);

     int8_t timezoneHours = configSettings->timezoneHours;

@@ -82,26 +84,20 @@

     comment += sprintf(comment, " and temperature was %s%lu.%luC.", sign,␣
↪temperatureInDecidegrees / 10, temperatureInDecidegrees % 10);

-    bool amplitudeThresholdEnabled = configSettings->amplitudeThreshold > 0 ||␣
↪configSettings->enableAmplitudeThresholdDecibelScale || configSettings->
↪enableAmplitudeThresholdPercentageScale;
-
-    if (amplitudeThresholdEnabled) comment += sprintf(comment, " Amplitude threshold␣
↪was ");
+    bool frequencyTriggerEnabled = configSettings->enableFrequencyTrigger;
```

(continues on next page)

(continued from previous page)

```
-    if (configSettings->enableAmplitudeThresholdDecibelScale && configSettings->
↪enableAmplitudeThresholdPercentageScale == false) {
+    bool amplitudeThresholdEnabled = frequencyTriggerEnabled ? false : configSettings->
↪amplitudeThreshold > 0 || configSettings->enableAmplitudeThresholdDecibelScale ||␣
↪configSettings->enableAmplitudeThresholdPercentageScale;

-        comment += formatDecibels(comment, configSettings->amplitudeThresholdDecibels);
+    if (frequencyTriggerEnabled) {

-    } else if (configSettings->enableAmplitudeThresholdPercentageScale &&␣
↪configSettings->enableAmplitudeThresholdDecibelScale == false) {
+        comment += sprintf(comment, " Frequency trigger (%u.%ukHz and window length of
↪%u samples) threshold was ", configSettings->frequencyTriggerCentreFrequency / 10,␣
↪configSettings->frequencyTriggerCentreFrequency % 10, (0x01 << configSettings->
↪frequencyTriggerWindowLengthShift));

-        comment += formatPercentage(comment, configSettings->
↪amplitudeThresholdPercentageMantissa, configSettings->
↪amplitudeThresholdPercentageExponent);
+        comment += formatPercentage(comment, configSettings->
↪frequencyTriggerThresholdPercentageMantissa, configSettings->
↪frequencyTriggerThresholdPercentageExponent);

-    } else if (amplitudeThresholdEnabled) {
-
-        comment += sprintf(comment, "%u", configSettings->amplitudeThreshold);
+        comment += sprintf(comment, " with %us minimum trigger duration.",␣
↪configSettings->minimumTriggerDuration);

     }

-    if (amplitudeThresholdEnabled) comment += sprintf(comment, " with %us minimum␣
↪trigger duration.", configSettings->minimumTriggerDuration);
-
     uint16_t lowerFilterFreq = configSettings->lowerFilterFreq;

     uint16_t higherFilterFreq = configSettings->higherFilterFreq;
@@ -120,6 +116,28 @@

     }

+    if (amplitudeThresholdEnabled) {
+
+        comment += sprintf(comment, " Amplitude threshold was ");
+
+        if (configSettings->enableAmplitudeThresholdDecibelScale && configSettings->
↪enableAmplitudeThresholdPercentageScale == false) {
+
+            comment += formatDecibels(comment, configSettings->
↪amplitudeThresholdDecibels);
+
+        } else if (configSettings->enableAmplitudeThresholdPercentageScale &&␣
```

(continues on next page)

(continued from previous page)

```
↪configSettings->enableAmplitudeThresholdDecibelScale == false) {
+
+            comment += formatPercentage(comment, configSettings->
↪amplitudeThresholdPercentageMantissa, configSettings->
↪amplitudeThresholdPercentageExponent);
+
+        } else {
+
+            comment += sprintf(comment, "%u", configSettings->amplitudeThreshold);
+
+        }
+
+        comment += sprintf(comment, " with %us minimum trigger duration.",␣
↪configSettings->minimumTriggerDuration);
+
+    }
+
     if (recordingState != RECORDING_OKAY) {

         comment += sprintf(comment, " Recording stopped");
```

Code

```
static void setHeaderComment(wavHeader_t *wavHeader, configSettings_t *configSettings,␣
↪uint32_t currentTime, uint8_t *serialNumber, uint8_t *deploymentID, uint8_t␣
↪*defaultDeploymentID, AM_extendedBatteryState_t extendedBatteryState, int32_t␣
↪temperature, bool externalMicrophone, AM_recordingState_t recordingState, AM_
↪filterType_t filterType) {

    struct tm time;

    time_t rawTime = currentTime + configSettings->timezoneHours * SECONDS_IN_HOUR +␣
↪configSettings->timezoneMinutes * SECONDS_IN_MINUTE;

    gmtime_r(&rawTime, &time);

    /* Format artist field */

    char *artist = wavHeader->iart.artist;

    sprintf(artist, "AudioMoth " SERIAL_NUMBER, FORMAT_SERIAL_NUMBER(serialNumber));

    /* Format comment field */

    char *comment = wavHeader->icmt.comment;

    comment += sprintf(comment, "Recorded at %02d:%02d:%02d %02d/%02d/%04d (UTC", time.
↪tm_hour, time.tm_min, time.tm_sec, time.tm_mday, 1 + time.tm_mon, 1900 + time.tm_year);

    int8_t timezoneHours = configSettings->timezoneHours;
```

(continues on next page)

(continued from previous page)

```
    int8_t timezoneMinutes = configSettings->timezoneMinutes;

    if (timezoneHours < 0) {

        comment += sprintf(comment, "%d", timezoneHours);

    } else if (timezoneHours > 0) {

        comment += sprintf(comment, "+%d", timezoneHours);

    } else {

        if (timezoneMinutes < 0) comment += sprintf(comment, "-%d", timezoneHours);

        if (timezoneMinutes > 0) comment += sprintf(comment, "+%d", timezoneHours);

    }

    if (timezoneMinutes < 0) comment += sprintf(comment, ":%02d", -timezoneMinutes);

    if (timezoneMinutes > 0) comment += sprintf(comment, ":%02d", timezoneMinutes);

    if (memcmp(deploymentID, defaultDeploymentID, DEPLOYMENT_ID_LENGTH)) {

        comment += sprintf(comment, ") during deployment " SERIAL_NUMBER " ", FORMAT_
↪SERIAL_NUMBER(deploymentID));

    } else {

        comment += sprintf(comment, ") by %s ", artist);

    }

    if (externalMicrophone) {

        comment += sprintf(comment, "using external microphone ");

    }

    static char *gainSettings[5] = {"low", "low-medium", "medium", "medium-high", "high"}
↪;

    comment += sprintf(comment, "at %s gain while battery was ",␣
↪gainSettings[configSettings->gain]);

    if (extendedBatteryState == AM_EXT_BAT_LOW) {

        comment += sprintf(comment, "less than 2.5V");

    } else if (extendedBatteryState >= AM_EXT_BAT_FULL) {
```

(continues on next page)

(continued from previous page)

```
        comment += sprintf(comment, "greater than 4.9V");

    } else {

        uint32_t batteryVoltage =  extendedBatteryState + AM_EXT_BAT_STATE_OFFSET / AM_
↪BATTERY_STATE_INCREMENT;

        comment += sprintf(comment, "%01lu.%01luV", batteryVoltage / 10, batteryVoltage
↪% 10);

    }

    char *sign = temperature < 0 ? "-" : "";

    uint32_t temperatureInDecidegrees = ROUNDED_DIV(ABS(temperature), 100);

    comment += sprintf(comment, " and temperature was %s%lu.%luC.", sign,␣
↪temperatureInDecidegrees / 10, temperatureInDecidegrees % 10);

    bool frequencyTriggerEnabled = configSettings->enableFrequencyTrigger;

    bool amplitudeThresholdEnabled = frequencyTriggerEnabled ? false : configSettings->
↪amplitudeThreshold > 0 || configSettings->enableAmplitudeThresholdDecibelScale ||␣
↪configSettings->enableAmplitudeThresholdPercentageScale;

    if (frequencyTriggerEnabled) {

        comment += sprintf(comment, " Frequency trigger (%u.%ukHz and window length of
↪%u samples) threshold was ", configSettings->frequencyTriggerCentreFrequency / 10,␣
↪configSettings->frequencyTriggerCentreFrequency % 10, (0x01 << configSettings->
↪frequencyTriggerWindowLengthShift));

        comment += formatPercentage(comment, configSettings->
↪frequencyTriggerThresholdPercentageMantissa, configSettings->
↪frequencyTriggerThresholdPercentageExponent);

        comment += sprintf(comment, " with %us minimum trigger duration.",␣
↪configSettings->minimumTriggerDuration);

    }

    uint16_t lowerFilterFreq = configSettings->lowerFilterFreq;

    uint16_t higherFilterFreq = configSettings->higherFilterFreq;

    if (filterType == LOW_PASS_FILTER) {

        comment += sprintf(comment, " Low-pass filter with frequency of %01u.%01ukHz␣
↪applied.", higherFilterFreq / 10, higherFilterFreq % 10);

    } else if (filterType == BAND_PASS_FILTER) {
```

(continues on next page)

(continued from previous page)

```
        comment += sprintf(comment, " Band-pass filter with frequencies of %01u.%01ukHz␣
↪and %01u.%01ukHz applied.", lowerFilterFreq / 10, lowerFilterFreq % 10,␣
↪higherFilterFreq / 10, higherFilterFreq % 10);

    } else if (filterType == HIGH_PASS_FILTER) {

        comment += sprintf(comment, " High-pass filter with frequency of %01u.%01ukHz␣
↪applied.", lowerFilterFreq / 10, lowerFilterFreq % 10);

    }

    if (amplitudeThresholdEnabled) {

        comment += sprintf(comment, " Amplitude threshold was ");

        if (configSettings->enableAmplitudeThresholdDecibelScale && configSettings->
↪enableAmplitudeThresholdPercentageScale == false) {

            comment += formatDecibels(comment, configSettings->
↪amplitudeThresholdDecibels);

        } else if (configSettings->enableAmplitudeThresholdPercentageScale &&␣
↪configSettings->enableAmplitudeThresholdDecibelScale == false) {

            comment += formatPercentage(comment, configSettings->
↪amplitudeThresholdPercentageMantissa, configSettings->
↪amplitudeThresholdPercentageExponent);

        } else {

            comment += sprintf(comment, "%u", configSettings->amplitudeThreshold);

        }

        comment += sprintf(comment, " with %us minimum trigger duration.",␣
↪configSettings->minimumTriggerDuration);

    }

    if (recordingState != RECORDING_OKAY) {

        comment += sprintf(comment, " Recording stopped");

        if (recordingState == MICROPHONE_CHANGED) {

            comment += sprintf(comment, " due to microphone change.");

        } else if (recordingState == SWITCH_CHANGED) {

            comment += sprintf(comment, " due to switch position change.");

        } else if (recordingState == MAGNETIC_SWITCH) {
```

(continues on next page)

(continued from previous page)

```
            comment += sprintf(comment, " by magnetic switch.");
        } else if (recordingState == SUPPLY_VOLTAGE_LOW) {
            comment += sprintf(comment, " due to low voltage.");
        } else if (recordingState == FILE_SIZE_LIMITED) {
            comment += sprintf(comment, " due to file size limit.");
        }
    }
}
```

8.1.18 Version 1.8.1

No changes were made in this version.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

V

W