Open Acoustic Devices

Acoustic Configuration using AudioMoth Chime

theteam@openacousticdevices.info

28th November 2020

AudioMoth is capable of receiving acoustically encoded data via its microphone. This document describes a robust protocol for this data transfer, called AudioMoth Chime, that is included in the standard AudioMoth firmware from version 1.5.0 onwards. The protocol uses data encoded within an inaudible 18 kHz carrier tone that is transmitted behind an audible tune from a computer or smartphone. This document describes the functionality that is offered in the standard firmware and related libraries, and also the technical details of how data is encoded and decoded.

1 AudioMoth Chime

AudioMoth Chime supports the transmission of two message types. One message type contains the time and timezone. The other message type contains this data with an additional 8-byte deployment ID. The former is used with persistent configurations to allow easy deployment in the field without the need to use a laptop to reconfigure the device after fitting or changing batteries (see Figure 1). The latter adds an additional deployment ID to the meta-data contained within the WAV file header (see Figure 2). This can be used by third-party upload databases to connect deployment information, such as the location and time of deployment, with the resulting recordings (e.g. the Rainforest Connection Companion App and Arbimon Platform).[1]

An AudioMoth will switch automatically into acoustic configuration mode (indicated by a constant red LED and flashing green LED) whenever the switch is moved to CUSTOM mode without the time having been set. It will remain in that state until it has successfully detected either of the two message types. This allows the the time and/or deployment ID to be set prior to recording. In addition, an option in the AudioMoth Configuration App allows the change to acoustic configuration mode to be mandated every time the switch is moved to CUSTOM whether or not the time has previously been set. This allows users to enforce the rule that the time should be updated, and/or a new deployment ID should be used, on each deployment.

An AudioMoth may also be manually switched to acoustic configuration mode at any time by moving the switch to CUSTOM mode whilst a specific tone is playing. Once in acoustic configuration mode the time and/or deployment ID may be set as above. However, in this case, if no message is detected within 30 seconds the AudioMoth will drop out of acoustic configuration mode and proceed to make recordings as normal.

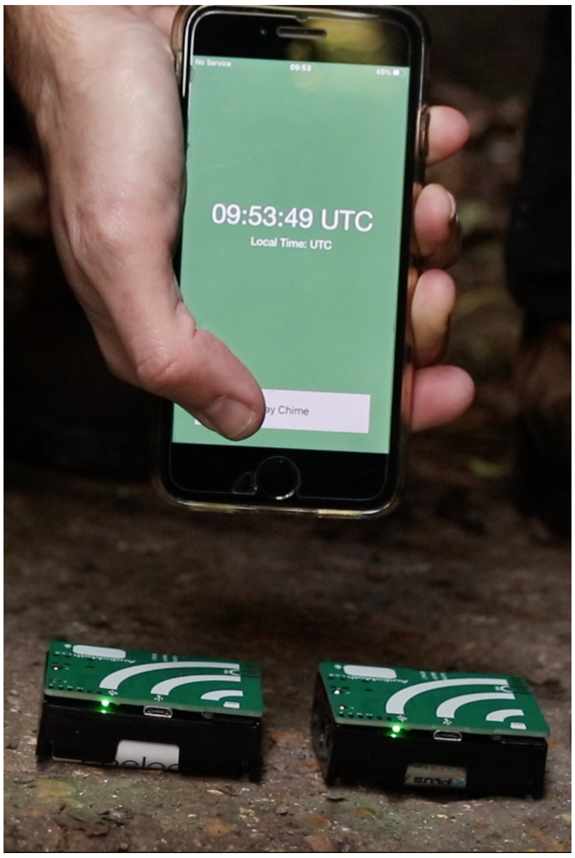
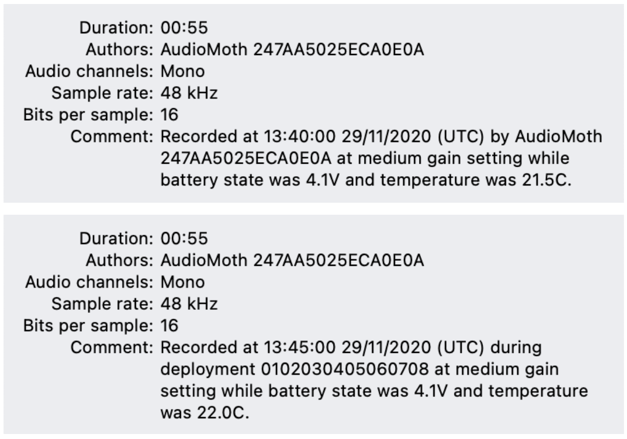


Figure 1: *A smartphone app being used to set the time on two AudioMoth prior to deployment.*

An open-source library supporting Swift, Kotlin and JavaScript is available for developers who wish to use this functionality within apps or websites.[2] The libraries provide a simple API to generate the two mes-

[1] https://www.openacousticdevices.info/rfcx

[2] https://github.com/OpenAcousticDevices/AudioMoth-Chime



```
      Duration: 00:55
       Authors: AudioMoth 247AA5025ECA0E0A
Audio channels: Mono
   Sample rate: 48 kHz
Bits per sample: 16
       Comment: Recorded at 13:40:00 29/11/2020 (UTC) by AudioMoth
                247AA5025ECA0E0A at medium gain setting while
                battery state was 4.1V and temperature was 21.5C.

      Duration: 00:55
       Authors: AudioMoth 247AA5025ECA0E0A
Audio channels: Mono
   Sample rate: 48 kHz
Bits per sample: 16
       Comment: Recorded at 13:45:00 29/11/2020 (UTC) during
                deployment 0102030405060708 at medium gain
                setting while battery state was 4.1V and temperature
                was 22.0C.
```

Figure 2: *WAV file meta-data showing the comment format when a deployment ID is used.*

sage types and to play the tone required to manually select acoustic configuration mode. The message types each use a different audible tune, composed by Martyn Harry, Professor of Music Composition at the University of Oxford, to distinguish them.[3]

Android and iOS apps that use these libraries are available from Open Acoustic Devices to support setting the time in the field.[4] Figure 1 shows an example of the use of this smartphone app to simultaneously set the time on two AudioMoth both of which are in acoustic configuration mode.

2 Implementation

AudioMoth Chime uses a custom protocol designed to allow easy generation of the encoded data on a computer or smartphone, and easy decoding on AudioMoth, whilst still being robust to background noise and interference.

2.1 Data Encoding

The protocol uses an inaudible 18 kHz carrier tone to transmit data behind an audible tune.

2.1.1 Bit Encoding

Individual bits are encoded by applying a binary phase shift to the 18 kHz tone. The time period between phase shifts encodes three symbols: a low data bit, a high data bit, and a start/stop bit. These time periods are 5 ms, 10 ms and 7.5 ms respectively. To ensure that phase shifts do not generate audible interference, an amplitude envelope is applied to each symbol with a 0.5 ms rise and fall time. Two communication speeds are supported: speed factor one and speed factor two.

[3]http://www.martynharry.com

[4]https://www.openacousticdevices.info/mobileapplications

At speed factor two all time periods (including the amplitude envelope rise and fall times) are reduced by a factor of two. Figure 3 shows an example encoding of a sequence of a low data bit, a start/stop bit, and a high data bit. The effective data transmission rate is approximately 125 - 150 bits per second at speed factor one, and 250 - 300 bits per second at speed factor two.

2.1.2 Switch Tone Encoding

The tone used to manually switch AudioMoth into acoustic configuration mode is a repeated sequence of high and low bits.

2.1.3 Byte Encoding

An individual byte can be transmitted in two ways. The simplest approach is to transmit a direct 8-bit sequence:

$$< b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7 >$$

For additional error correction a (7,4) Hamming code can be used. In this case, the lowest four bits, b_0, b_1, b_2, b_3, and the highest four bits, b_4, b_5, b_6, b_7, are encoded as two seven-bit messages with parity bits, p_l^0, p_l^1, p_l^2 and p_h^0, p_h^1, p_h^2, respectively. These are then transmitted interleaved as a 14-bit sequence:

$$< p_l^0, p_h^0, p_l^1, p_h^1, b_0, b_4, p_l^2, p_h^2, b_1, b_5, b_2, b_6, b_3, b_7 >$$

Doing so ensures that an error that might corrupt two adjacent time periods will result in single bit errors in each separate seven-bit messages meaning that both 4-bit sequences, and hence the original byte, can be recovered. The use of the (7,4) Hamming code reduces the data transmission rate by approximately half.

2.1.4 Packet Encoding

The two message types encode two different data packets. A packet consists of 16 start bits, followed by at least one data byte, followed by a little-endian 16-bit cyclic redundancy check (CRC), and then 8 stop bits. The large number of start bits is to allow the receiver to synchronise with the phase of the carrier signal prior to demodulation. This is required as the carrier signal is not present when data is not being transmitted. The data packets encoded within each message type are shown in Table 1. The default implementation uses the (7,4) Hamming code, and speed factor one, for robustness.

2.1.5 Audible Tune

The inaudible 18 kHz signal described above is added to an audible low-frequency musical tune. The audible tune provides an indication to the user that data is being transmitted and can take any form. The standard implementation uses different tunes for each

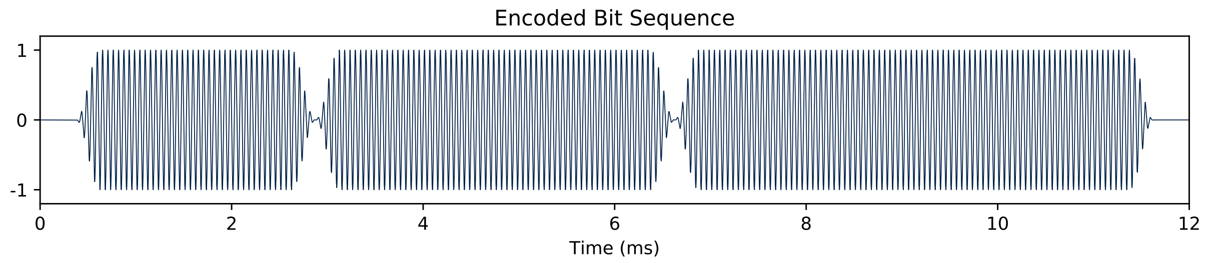


Figure 3: *Encoded sequence of a low data bit, a start/stop bit, and a high data bit.*

Time Data Packet

0 - 3	`time`	Unsigned 4-byte LE integer	Unix UTC timestamp.
4 - 5	`timezoneOffset`	Signed 2-byte LE integer	Timezone offset in minutes.

Time and Deployment ID Data Packet

0 - 3	`time`	Unsigned 4-byte LE integer	Unix UTC timestamp.
4 - 5	`timezoneOffset`	Signed 2-byte LE integer	Timezone offset in minutes.
6 - 13	`deploymentID`	8-byte array	Deployment ID.

Table 1: *Data packets used within the AudioMoth Chime message types.*

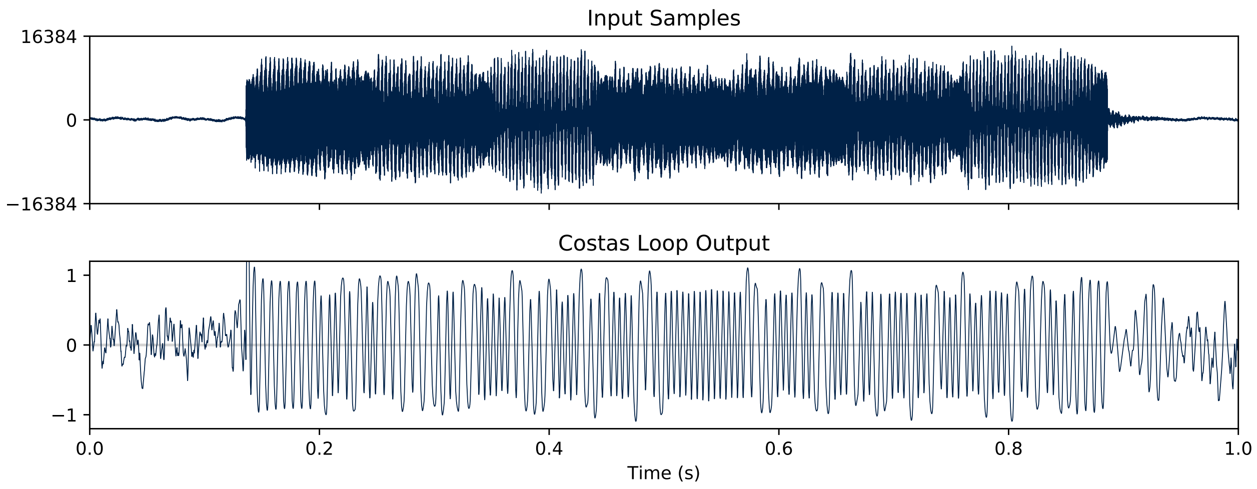


Figure 4: *Input acoustic samples and the resulting output of the Costas loop.*

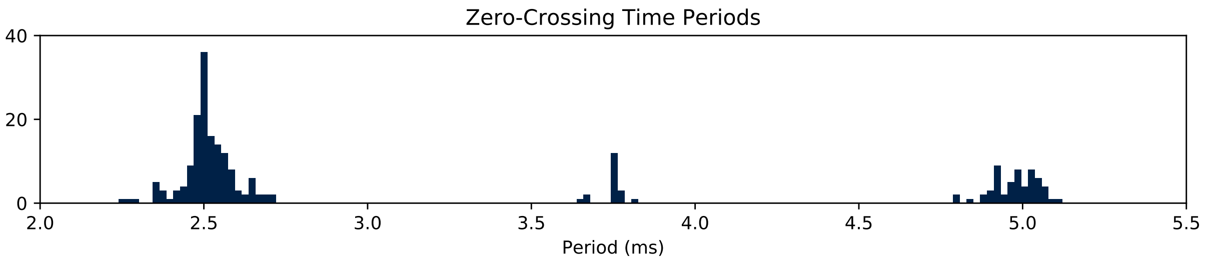


Figure 5: *Measured zero crossing time periods showing low data bits, start/stop bits, and high data bits.*

message type, and in "[note][octave]:[duration]" notation, these are given by ["C5:1", "D5:1", "E5:1", "C5:3"] and ["E♭5:1", "G5:1", "D5:1", "F#5:1", "D♭5:1", "F5:1", "C5:1", "E5:5"]. The manual switching tone uses a constant "C5" note.

2.2 Decoding

The signal described above is decoded using a firmware implementation of a Costas loop to track the phase and frequency of the original 18 kHz carrier tone.[5] Decoding is performed in real time with no buffering.

2.2.1 Input Sample Acquisition

The acoustic tone is sampled by the AudioMoth microphone at high gain setting, at a sample rate of 48 kHz with 8-times oversampling, to produce a continuous stream of signed 15-bit samples.

2.2.2 Automatic Gain Control

The received samples are passed through a first order Butterworth band-pass filter (with cut-off frequencies of 17 and 19 kHz) to extract the 18 kHz carrier tone. The resulting signal is rectified and passed to a first order Butterworth low-pass filter (with a cut-off frequency of 20 Hz at speed factor one and 40 Hz at speed factor two) to generate an amplitude reference. The output of the band-pass filter is multiplied by the reciprocal of the amplitude reference to produce a normalised signal with amplitude close with one.

2.2.3 Demodulation

The output of the previous stage is fed to a Costas loop. In each arm of the Costas loop the input signal is multiplied by an in-phase (channel 1) and a 90 degree out-of-phase (channel 2) locally-generated signal and passed through a pair of biquad low-pass filters (with a cut-off frequency of 100 Hz at speed factor one and 200 Hz at speed factor two and bandwidth of 2 octaves in both cases). These are multiplied together to generate a control signal which is used directly (without any further low-pass filtering) to adjust the frequency of the locally-generated signal; ensuring that it stays in phase with the 18 kHz carrier tone.

[5]https://en.wikipedia.org/wiki/Costas_loop

2.2.4 Decoding

The output of the low-pass filter on the in-phase channel of the Costas loop represents the phase of the original signal. Measuring the time period between zero-crossing events allows the individual symbols to be decoded. Figure 4 shows an example of the raw input signal and the resulting Costas loop output. Figure 5 shows the clear separation between low data bits, start/stop bits, and high data bits.

The firmware uses separate functions to detect the repeated high and low bits of the manual switching tone and the sequence of initial start bits, high and low data bits of the data bytes and the 16-bit CRC value, and the stop bits of the encoded data packets.[6] The start of the packet can easily be detected by waiting for the sequence of start bits, and individual bits decoded until the sequence of stop bits is detected. The two message types are distinguished by the length of the data packet.

3 Other Applications

The protocol as described here can be used for other data communication applications with minimal changes to the transmitting software and receiving firmware. For example, the protocol has been used to set a 16-byte encryption key on each AudioMoth within a citizen science deployment requiring AES encryption of recordings.[7,8] The use of the (7,4) Hamming code, and choice of speed factor one or two, can be varied depending on the packet length and the communication speed and robustness required. In trial deployments we have reliably used data packets of over 64 bytes, with speed factor two and no (7,4) Hamming code, in noisy environments. Speed factors beyond those considered here can also be implemented. However, at faster bit rates the modulation of the 18kHz carrier tone becomes audible as a background hiss.

[6]https://github.com/OpenAcousticDevices/AudioMoth-Firmware-Basic

[7]https://en.wikipedia.org/wiki/Advanced_Encryption_Standard

[8]https://greenspacehack.com/project/nature_sensing.html