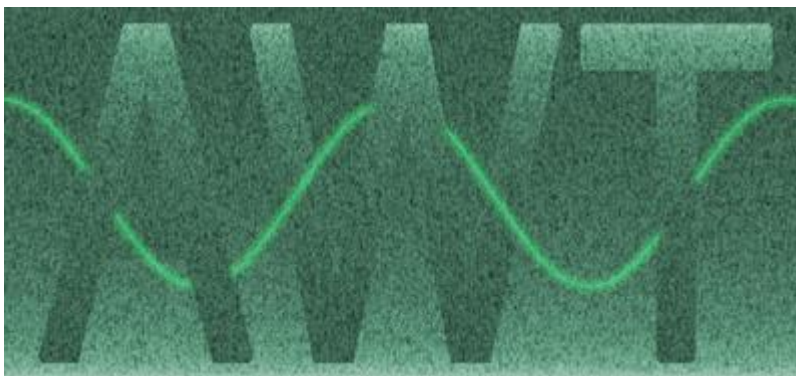




Audio Watermarking Tools 3 (AWT3) of www.audiowatermarking.info

User Guide

AWT3 version 0.10.00
September 2018

Contents

1	Introduction	3
2	General information.....	4
2.1	AWT3 at a glance.....	4
2.2	AWT3 applications	6
2.3	How the algorithm works.....	7
2.4	Watermark robustness and aural transparency	9
3	Encoder	10
4	Decoder	13
5	AWT3 Software Development Kit (SDK).....	17
5.1	AWT3 SDK Header	17
6	Choosing optimal watermark payload form	23
6.1	text2hex utility	25
7	Useful tips.....	27
7.1	Converting audio or video file into file format suitable for AWT3.....	27
7.2	Watermarking long audio files using AWT3	27
7.3	Extracting watermarks from broadcasts and long recordings	29
7.4	Watermarking audio track of a video file.....	29
7.5	Watermarking multiple files.....	29
7.6	Analyzing live audio stream	30
8	Licensing terms.....	33
9	FAQ.....	34
10	Contact information	35
11	Appendix: AWT3 error codes	36
12	Appendix: ASCII printable characters.....	37

1 Introduction

Audio Watermarking Tools 3 (AWT3) of www.audiowatermarking.info is a set of software utilities for embedding (and retrieving) digital watermarks carrying arbitrary digital data within (from) audio files and streams. The watermarks are embedded directly into the audio content and carry arbitrary digital identifiers.

The tools are distributed as a package of console file-to-file utilities and in a form of SDK (Software Development Kit) libraries running on desktop operating systems (Microsoft Windows, Apple Mac OS, Linux/*nix systems) and on mobile platforms such as Apple iOS and Google Android. AWT3 SDK allows watermarking live (real-time) audio streams, and provides disk files and in-memory processing functionality.

AWT3 functionality consists of encoder (watermark embedding tool) and decoder (watermark detection/extraction tool). These functions are implemented in a form of command-line utilities and SDK APIs.

AWT3 technology is patented¹.

¹ U.S. Patent No. 8,116,514.

2 General information

2.1 AWT3 at a glance

AWT3 technology represents a direct extension of the patented AWT2 algorithm and uses the very same core approach. The algorithm allows embedding watermarks with arbitrary digital data (usually, few bytes long binary identifiers) into audio stream and retrieve them accordingly. The original audio stream is modified in a special way by the embedding mechanism and is used as a carrier.

The main purpose of AWT3 technology is to allow reliable over-the-air signaling/triggering using sound.

Unlike AWT2, AWT3 is able to carry watermarks even within absolutely “silent” carrier audio signal, i.e. within a signal having no audible data in it (“digital silence”). This is achieved by mixing the original audio content with artificially generated tones (“beeps”) having configurable frequency and level. By default, the used tones are located in the ultra-sonic frequency region, above 18 KHz, in which the human ear is not sensitive especially when the ultra-sonic content is “masked” by audible content in lower frequencies.

The watermarking algorithm operates with audio signal in the time domain in several frequency sub-bands. Overall idea behind the algorithm consists in embedding of binary watermark payload into carrier audio signal in the time domain by time-shifting the carrier signal blocks. This watermarking approach is patented².

AWT3 encoder and decoder operate with wave PCM (.wav) audio data of almost any format — with sampling rates from 8 to 192 KHz, amplitude resolutions of 8/16/24/32/64 bits and any number of channels³. Additional audio formats (such as MP3, OGG, AMR, etc.) are supported by the decoder by means of automatic conversion using external tool, Ffmpeg (<http://ffmpeg.org>).

Supported watermark payload size is from 1 to 30 bytes (subject to limitations in different AWT3 packages). Recommended watermark payload size is 2 - 6 bytes.

Watermarking data rate depends on watermark payload length and is approximately 24 bps for 1-byte payload, 32 bps for 2-bytes payload, 37 bps for 3-bytes payload, 44 bps for 4-bytes payload, 45 bps for 5-bytes payload and 47 bps for 6-

² U.S. Patent No. 8,116,514.

³ Supported wave files: basic and WAVE_FORMAT_EXTENSIBLE, sampled at 192000, 176400, 96000, 88200, 48000, 44100, 32000, 22050, 16000, 11025, 8000 Hz, signed 16-/24-/32-bit little-endian PCM or unsigned 8-bit PCM, 32-/64-bit IEEE float.

bytes payload. The watermarking rate increases with increase of watermark payload size due to some constant overhead for each watermark copy. In terms of AWT3 technology, the data rate means a number of the same watermark contained within audio fragment of a specific duration. For example, with 3-bytes watermarks, the data rate is 37 bps, which leads to more than 6 watermark copies in 4-seconds long audio fragment ($[4\text{sec} * 37\text{bps}] / [3\text{ bytes} * 8\text{ bit}] = 6.2$).

Each particular copy of AWT3 binaries with particular Serial Number (SN) contains a unique numeric identifier that is used during encoding to scramble watermark payload. This security feature prevents one AWT3 user (with one SN) from extracting watermarks from watermarked files created by another AWT3 user (with another SN number).

If you run AWT3 on x64 machine, you might want to use AWT3 x64 binaries because of the ability of x64 machine to operate with greater amount of RAM.

2.2 AWT3 applications

AWT3 technology has been designed to provide reliable over-the-air sound signaling by means of audio watermarks. AWT3 watermarks demonstrate high robustness against over-the-air sound transducing and strong room reverberations and allow digital codes transmission on several meters distances from the sound source in real acoustic environments.

AWT3 watermarks can survive 3-5 (and more) meters distance between sound source (loudspeakers) and capturing device (microphone) in typical home environment. To achieve this goal, ultra-sonic frequencies (> 18 KHz) are used as a carrier because only high frequencies can reliably withstand strong room reverberations. AWT3 signaling is reliable even on large stadiums and cinema halls.

A typical application of AWT3 is a “kick-shopping”. For example, mini-speakers can be installed in shopping centers to allow passing by shoppers capturing “digital coupons” using their mobile devices.

2.3 How the algorithm works

Below is a brief high-level description of AWT3 watermarking algorithm.

AWT3 algorithm is fully based on the watermarking engine of AWT2 and uses the very same patented core approach.

- Watermark payload is converted into a watermarking data packet containing encrypted watermark payload and error-correction code
- The source audio stream is decomposed into several frequency sub-band signals (carrier signals), starting from user-defined carrier frequency (18 KHz by default)
- Artificial tone is added into (mixed with) each sub-band signal, the tone level is user-adjustable
- Each sub-band signal is then divided into blocks of a certain length
- Each block of the carrier sub-band signal is associated with corresponding symbol (bit) of the watermark payload data packet
- Each block of the carrier signal is then time-shifted (forwards or backwards in time) to a degree determined from inter-band signal analysis and associated with the corresponding watermark payload data packet symbol (bit) value
- The watermark is repeated throughout the entire audio stream as many times as the stream duration permits
- The output (watermarked) audio stream is then synthesized by combining modified (time-shifted) carrier sub-band signals back into full band signal
- On the last stage the output (watermarked) stream is sent to the output

For increased reliability, different statistical, security, error-correction and signal processing mechanisms are applied.

Important notes:

- number of copies of the watermark payload embedded into audio stream is proportional to its duration;
- duration of each watermark payload copy is proportional to the watermark payload length (number of bytes);
- due to a constant overhead for each copy of the watermark payload, watermarking rate increases with increase of the payload size.

On the decoding stage, the decoder performs the same frequency band decomposition steps as for the encoding. It then collects “statistics” over the analyzed fragment and looks for watermark copies in it by performing self-synchronization

attempts using sophisticated logic and heuristics based on inter-band signal analysis. The more watermark copies are contained in the analyzed fragment, the higher the probability to find proper synchronization and to successfully extract the watermark. Since the whole process is based on collecting “statistics”, at least several watermark copies must be contained in the analyzed fragment to allow the extraction. The algorithm is unable to extract watermark from audio fragment containing only 1-2 copies of the watermark.

A special “detection” mechanism, part of AWT3 SDK, allows determining the exact time-place in the audio stream where the AWT3 code transmission has started, with a precision between 100-300ms (depending on detector settings). The detector also determines time/speed variation of the carrier signal and reports estimated degree of the time/speed variation (within up to 3%).

2.4 Watermark robustness and aural transparency

AWT3 watermarks are carried by modified source audio signal with artificially generated tones added into it (mixed with it). By default, the algorithm uses ultra-sonic frequency region, starting from 18 KHz, to carry the watermarks. This allows for reliable over-the-air transmission of the watermarks, but imposes certain constraints on distribution and transmission of the watermarked signal.

Lossy audio encoders (such as MPEG-1 Layer 3 or simply “MP3”, Dolby AC3, AAC, etc.) may filter out the ultra-sonic frequency content thus completely removing AWT3 watermarks. Some encoders preserve all frequencies, but only under high bitrates (e.g. > 192 Kbps). The behavior depends on particular codec implementation and encoding parameters. It is possible to enforce some encoders to preserve all frequencies (e.g. Lame MP3 encoder has a special setting that disables low-pass filtering preceding the actual encoding). However, not all audio codecs and encoders allow such control. For the above-mentioned reasons, special attention must be paid to the way AWT3-watermarked signal is transmitted and distributed.

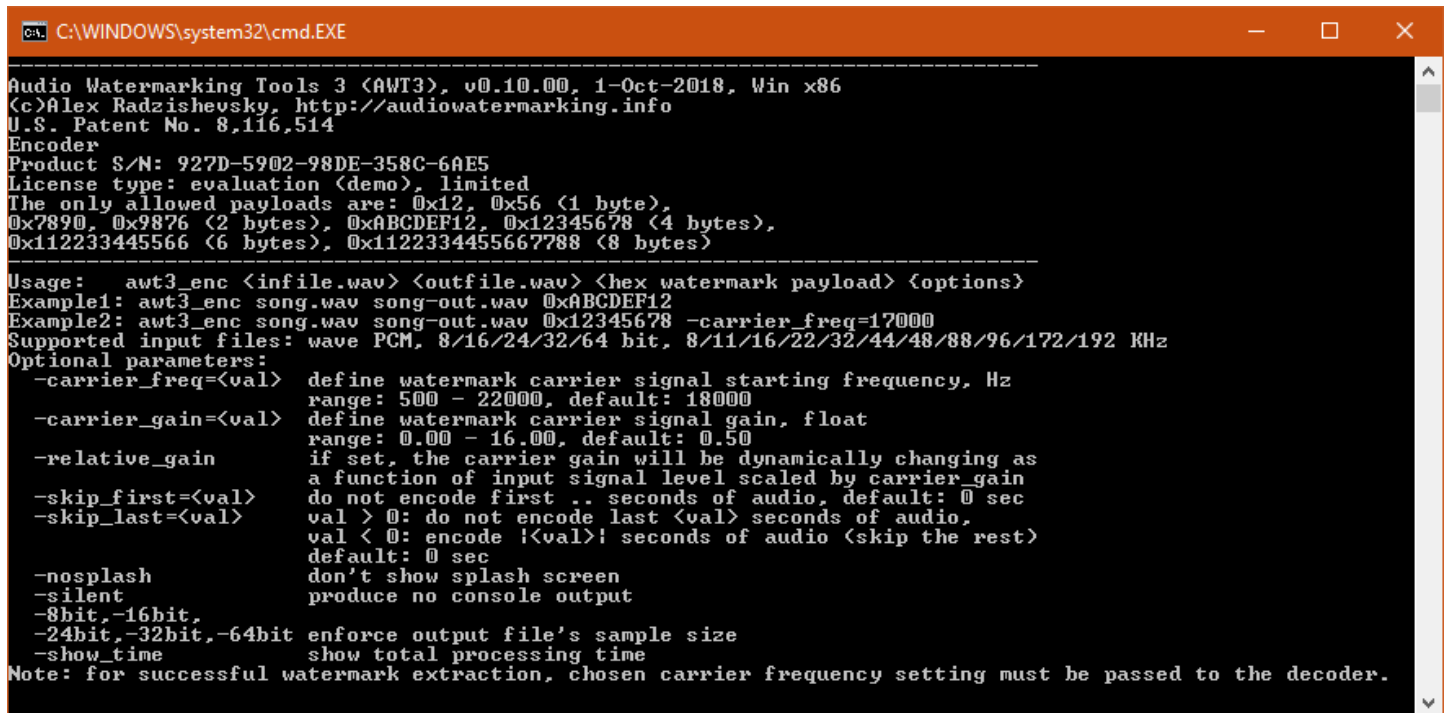
Majority of hardware audio components, such as loudspeakers and microphones, can handle transmission and reception of ultra-sonic frequencies. However, in some cases the use of very cheap components may spoil transmission or reception of AWT3 watermarks. For example, some cheap smartphones available on the market demonstrate poor microphone audio path quality preserving sound frequencies up to 12 KHz only.

By default, AWT3 carrier tones are located in the ultra-sonic frequency region, above 18 KHz, in which the human hearing is not sensitive especially when the ultra-sonic content is “masked” by audible content in lower frequencies (such as speech, music). Lowering the carrier frequency can lead to improved watermarks robustness under lossy audio compression, but may make the watermarks noticeable and even annoying for listeners.

For the above reasons, watermark carrier frequency and level must be carefully selected for each specific application and use-case.

3 Encoder

The encoder embeds watermark payload into an audio stream (audio file). The usage screen is shown (below) when the encoder is executed in the command line without parameters:



```

C:\WINDOWS\system32\cmd.EXE
-----
Audio Watermarking Tools 3 (AWT3), v0.10.00, 1-Oct-2018, Win x86
(c)Alex Radzishovsky, http://audiowatermarking.info
U.S. Patent No. 8,116,514
Encoder
Product S/N: 927D-5902-98DE-358C-6AE5
License type: evaluation (demo), limited
The only allowed payloads are: 0x12, 0x56 (1 byte),
0x7890, 0x9876 (2 bytes), 0xABCD12, 0x12345678 (4 bytes),
0x112233445566 (6 bytes), 0x1122334455667788 (8 bytes)
-----
Usage:   awt3_enc <infile.wav> <outfile.wav> <hex watermark payload> <options>
Example1: awt3_enc song.wav song-out.wav 0xABCD12
Example2: awt3_enc song.wav song-out.wav 0x12345678 -carrier_freq=17000
Supported input files: wave PCM, 8/16/24/32/64 bit, 8/11/16/22/32/44/48/88/96/172/192 KHz
Optional parameters:
-carrier_freq=<val>  define watermark carrier signal starting frequency, Hz
                      range: 500 - 22000, default: 18000
-carrier_gain=<val>  define watermark carrier signal gain, float
                      range: 0.00 - 16.00, default: 0.50
-relative_gain      if set, the carrier gain will be dynamically changing as
                      a function of input signal level scaled by carrier gain
-skip_first=<val>    do not encode first .. seconds of audio, default: 0 sec
-skip_last=<val>     val > 0: do not encode last <val> seconds of audio,
                      val < 0: encode {<val>} seconds of audio (skip the rest)
                      default: 0 sec
-nosplash           don't show splash screen
-silent            produce no console output
-8bit,-16bit,
-24bit,-32bit,-64bit enforce output file's sample size
-show_time         show total processing time
Note: for successful watermark extraction, chosen carrier frequency setting must be passed to the decoder.
  
```

In order to encode a file, the user must specify input and output file names, and watermark payload (refer to page 23 for more information about watermark payload form). Other encoding parameters are optional.

The encoder operates with wave PCM files of almost any format - with sampling rates from 8 to 192 KHz, and amplitude resolution of 8/16/24/32/64 bits and any number of channels⁴.

The watermark payload should be specified in a hexadecimal form, i.e. using "0"- "9" and "A"- "F" characters (for example, "0xFE21" which is 2 bytes). Supported length of watermarks is from 1 to 30 bytes. Recommended watermark payload sizes are from 2 to 6 bytes. For more information about watermark payload form, please refer to page 23.

⁴ Supported wave files: basic and WAVE_FORMAT_EXTENSIBLE, sampled at 192000, 176400, 96000, 88200, 48000, 44100, 32000, 22050, 16000, 11025, 8000 Hz, signed 16-/24-/32-bit little-endian PCM or unsigned 8-bit PCM, 32-/64-bit IEEE float.

Please note: Maximal watermark payload size that is supported by particular copy of the encoder depends on the version of AWT3 package that you purchase. This can be: 1 byte max, 2 bytes max, 4 bytes max or unlimited.

If you are using demo version of the AWT3 package, the only watermarks that can be embedded are: 0x12, 0x56 (1 byte), 0x7890, 0x9876 (2 bytes), 0xABCDEF12, 0x12345678 (4 bytes), 0x112233445566 (6 bytes), 0x1122334455667788 (8 bytes) and 0x00112233445566778899001122...778899 (40 bytes). Despite this limitation, the demo package offers enough flexibility to perform a variety of performance tests so that you can decide whether AWT3 suits your needs before buying a fully functional version.

Here is an example of running the encoder. The input file is “music.wav”, the output (watermarked) file is “music-wtr.wav”, and the watermark payload is “0xABCDEF12”:

```
awt3_enc music.wav music-wtr.wav 0xABCDEF12
```

In this example all algorithm parameters such as carrier frequency, carrier gain and other settings remain default. Depending on your needs, you may consider customizing these settings.

The encoder has several additional parameters controlling its performance and output:

-carrier_freq= specifies bottom limit of the frequency region that is used to carry the watermark. The carrier bandwidth (B) is 2500 Hz wide (not controllable). Specified carrier frequency should correspond to the sampling rate (Fs) of the input file. Namely, the following rule applies: $\text{carrier_freq} < F_s / 2 - B$.

-carrier_gain= specifies level of carrier tones added to the source signal in the carrier frequency region. The level is defined as fraction of maximal digital scale (0 dBFs). Namely, 0.5 = -6 dBFs, 0.25 = -12 dBFs, 2.0 = +6 dBFs, etc.

-relative_gain if specified, the level of the carrier tones will change dynamically as a function of the full-band input signal level (measured as peak amplitude envelope) scaled by the carrier gain. Namely, 1.0 means that the tones level will follow the level of the full scale input signal, 2.0 means that the tones level will be twice (+6 dB) the level of the full scale input signal.

-skip_first= / **-skip_last=** specifies a number of seconds at the start/end of the input audio that should be skipped by the encoder (i.e. left untouched, not watermarked). This feature can be used to watermark only some specific part of signal, or to allow smooth cross-fades between two or more consequent fragments of a continuous audio stream. For example, one way to watermark very long audio stream is to divide it into shorter fragments, then watermark each fragment separately, and then construct the output from the watermarked fragments by combining them together with cross-fades. If specified **-skip_last** value is negative, it forces the encoder to watermark only specified duration of audio (instead of skipping the same duration at the end).

-silent flag can be used if you do not want any console output. Only error messages are displayed in this mode (if any).

-8bit, -16bit, -24bit, -32bit, 64bit control output files data format.

-nosplash prevents showing program information on the screen

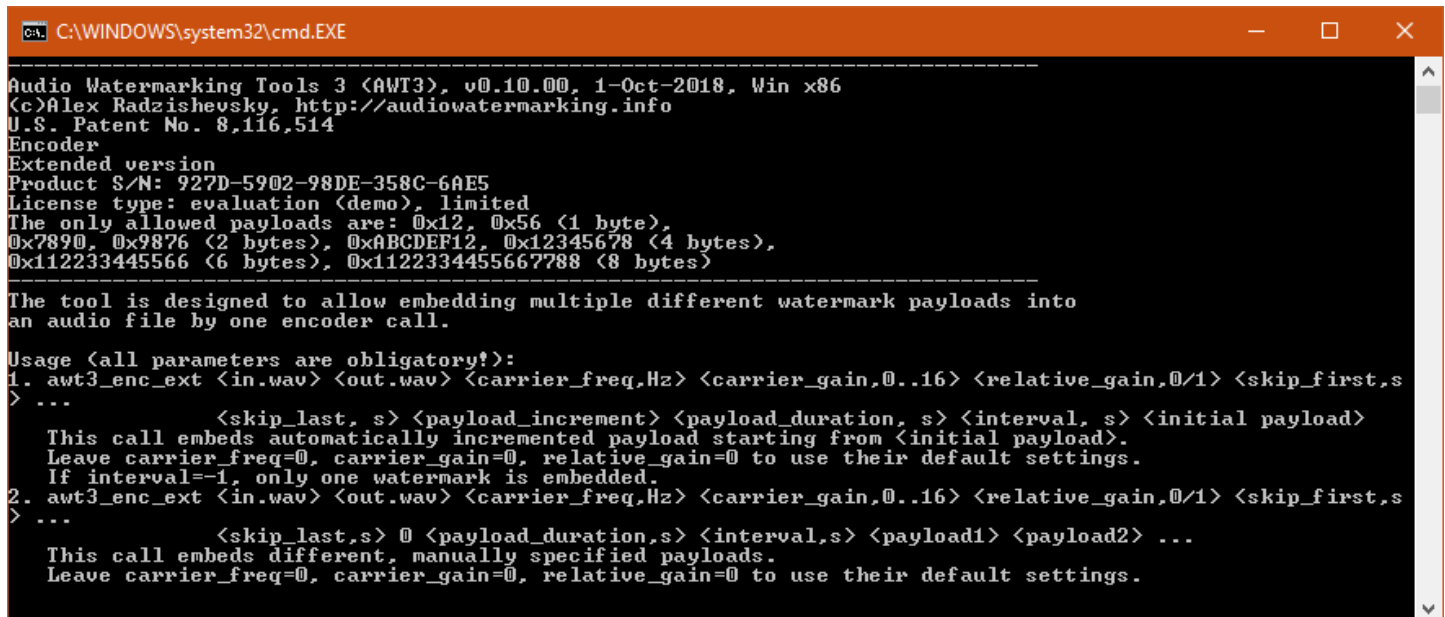
-show_time forces the encoder to show processing time

Here are additional examples of running the encoder with customized parameters:

```
awt3_enc music.wav music-wtr.wav 0xABCDEF12 -carrier_freq=19000
```

```
awt3_enc music.wav music-wtr.wav 0xABCDEF12 -carrier_freq=17000 -carrier_gain=0.25
```

There is an alternative version of the encoder tool called 'awt3_enc_ext' (extended encoder). This tool is a little less user friendly in terms of command line parameters parsing, but allows embedding multiple, altering in time, different watermark payloads into a single file. The tool's usage screen is shown below.



```
C:\WINDOWS\system32\cmd.EXE

Audio Watermarking Tools 3 (AWT3), v0.10.00, 1-Oct-2018, Win x86
(c) Alex Radziszhevsky, http://audiowatermarking.info
U.S. Patent No. 8,116,514
Encoder
Extended version
Product S/N: 927D-5902-98DE-358C-6AE5
License type: evaluation (demo), limited
The only allowed payloads are: 0x12, 0x56 (1 byte),
0x7890, 0x9876 (2 bytes), 0xABCDEF12, 0x12345678 (4 bytes),
0x112233445566 (6 bytes), 0x1122334455667788 (8 bytes)

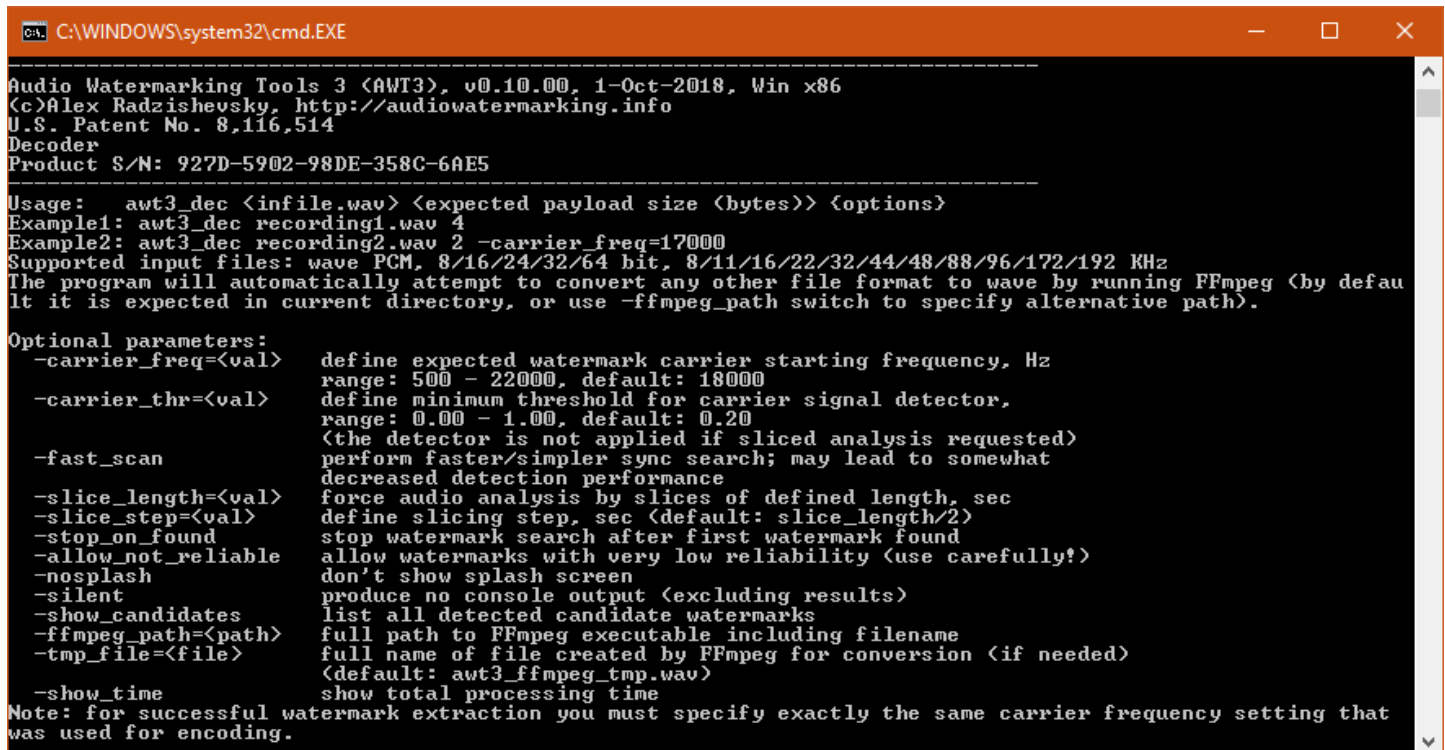
-----
The tool is designed to allow embedding multiple different watermark payloads into
an audio file by one encoder call.

Usage (all parameters are obligatory!):
1. awt3_enc_ext <in.wav> <out.wav> <carrier_freq,Hz> <carrier_gain,0..16> <relative_gain,0/1> <skip_first,s>
> ...
    <skip_last,s> <payload_increment> <payload_duration,s> <interval,s> <initial payload>
This call embeds automatically incremented payload starting from <initial payload>.
Leave carrier_freq=0, carrier_gain=0, relative_gain=0 to use their default settings.
If interval=-1, only one watermark is embedded.
2. awt3_enc_ext <in.wav> <out.wav> <carrier_freq,Hz> <carrier_gain,0..16> <relative_gain,0/1> <skip_first,s>
> ...
    <skip_last,s> 0 <payload_duration,s> <interval,s> <payload1> <payload2> ...
This call embeds different, manually specified payloads.
Leave carrier_freq=0, carrier_gain=0, relative_gain=0 to use their default settings.
```

Using the tool, you can embed a sequence of different payloads with desired intervals into single audio file.

4 Decoder

The decoder detects and extracts watermark from an audio file. The usage screen is shown (below) when the decoder is executed from the command line without parameters:



```

C:\WINDOWS\system32\cmd.EXE
-----
Audio Watermarking Tools 3 (AWT3), v0.10.00, 1-Oct-2018, Win x86
(c)Alex Radzishovsky, http://audiowatermarking.info
U.S. Patent No. 8,116,514
Decoder
Product S/N: 927D-5902-98DE-358C-6AE5
-----
Usage:  awt3_dec <infile.wav> <expected payload size (bytes)> <options>
Example1: awt3_dec recording1.wav 4
Example2: awt3_dec recording2.wav 2 -carrier_freq=17000
Supported input files: wave PCM, 8/16/24/32/64 bit, 8/11/16/22/32/44/48/88/96/172/192 KHz
The program will automatically attempt to convert any other file format to wave by running FFmpeg (by default it is expected in current directory, or use -ffmpeg_path switch to specify alternative path).

Optional parameters:
-carrier_freq=<val>    define expected watermark carrier starting frequency, Hz
                        range: 500 - 22000, default: 18000
-carrier_thr=<val>     define minimum threshold for carrier signal detector,
                        range: 0.00 - 1.00, default: 0.20
                        <the detector is not applied if sliced analysis requested>
-fast_scan            perform faster/simpler sync search; may lead to somewhat
                        decreased detection performance
-slice_length=<val>    force audio analysis by slices of defined length, sec
-slice_step=<val>       define slicing step, sec <default: slice_length/2>
-stop_on_found        stop watermark search after first watermark found
-allow_not_reliable   allow watermarks with very low reliability <use carefully!>
-nosplash            don't show splash screen
-silent              produce no console output <excluding results>
-show_candidates      list all detected candidate watermarks
-ffmpeg_path=<path>    full path to FFmpeg executable including filename
-tmp_file=<file>       full name of file created by FFmpeg for conversion <if needed>
                        <default: awt3_ffmpeg_tmp.wav>
-show_time            show total processing time
Note: for successful watermark extraction you must specify exactly the same carrier frequency setting that
was used for encoding.
  
```

IMPORTANT: To successfully detect and extract watermark payload from an audio file, you must specify expected watermark payload size and exactly the same “-carrier_freq” parameter value that were used at the encoding stage.

The requirement to specify decoding settings such as expected watermark payload size and carrier frequency is generally not an issue. You can choose suitable payload form/size (refer to page 23) and watermarking parameters for a specific application/use-case once, and then keep using the same settings to encode all your files. For example, you can decide once and for all to use 4-bytes long watermark payload with some certain carrier frequency (such as 18000 Hz, which is a default). This way you will always know how the parameters for the decoder should be specified.

For example, if the following was the command line used for encoding:

```
awt3_enc music.wav music-wtr.wav 0xABCDEF12
```

then you need to run the decoder as follows:

```
awt3_dec music-wtr.wav 4
```

'4' at the end tells to decoder that the expected watermark payload is 4 bytes long.

Another example:

```
awt3_enc music.wav music-wtr.wav 0x112233445566 --carrier_freq=19000
```



```
awt3_dec music-wtr.wav 6 --carrier_freq=19000
```

Additional example:

```
awt3_enc music.wav music-wtr.wav 0x1234 --carrier_freq=17500 --carrier_gain=0.25
```



```
awt3_dec music-wtr.wav 2 --high_carrier_freq=17500
```

Watermark detection and extraction reliability depends on degree of quality degradation of the analyzed audio stream (distortions, noises, extraneous sounds, etc.) relative to the original watermarked stream. Additionally, with ultra-sonic frequencies used to carry the watermarks (which is a default setting of AWT3 encoder), the reliability depends on the way the encoded audio signal travels to the decoder, namely, on ability of the transmission path from the encoder to the decoder to preserve ultra-sonic tones (refer to 2.4).

Since the watermark extraction process is based on collecting "statistics" over the analyzed audio fragment (refer to 2.3), at least several copies of watermark must be contained in it to allow successful watermark extraction. At the same time, AWT3 watermark data rate is relatively high (i.e. many watermark copies are contained in within audio fragment of a certain length). For this reason even relatively short audio fragment is usually enough to extract watermark. For example, 3 seconds of audio is normally enough to extract 3-bytes long watermark. Length and location of analyzed audio fragment in the input audio file can be defined using "--skip_first" and "--skip_last" parameters described below.

False positive, i.e. "successful" extraction of incorrect watermarks is possible, but is rare. One possible way to get false positive is to try extracting watermark from a watermarked stream by specifying expected watermark payload incorrectly. This confusion should be avoided.

Current version of decoder accepts almost any audio file formats via FFmpeg (<http://ffmpeg.org/>) while the native format for AWT3 decoder is wave (.wav) PCM file. It means that if specified file is not wave PCM, the decoder will automatically attempt to convert it into wave by running FFmpeg. FFmpeg executable should be located in current directory (or pointed by --ffmpeg_path command line switch). FFmpeg is distributed as a part of AWT3 package. You can also download FFmpeg binaries for Windows and Linux separately [here](#).

Useful tips on manual file formats conversion can be found in section 7.

AWT3 decoder allows analyzing long audio recordings by slices. This feature is especially useful in cases when watermarks search should be performed within recordings of broadcasts in which not the whole recording is watermarked, but only parts of it. There are two parameters controlling this mechanism: `-slice_length` and `-slice_step`. They define respectively length (duration) of audio stream slices taken for analysis, and analysis step; both should be specified in seconds.

For example, this AWT3 decoder call:

```
awt3_dec long-broadcast-rec.wav 1 -slice_length=10 -slice_step=4 -fast_scan
```

results in the following output:

```
Reading input file 'long-broadcast-rec.wav'... Done [441000 Hz, 16 bit, 1 chan, 151 sec]
Analyzing audio by slices...
[00:00.00 - 00:00.10]
[00:00.04 - 00:00.14] Found watermark: 0x12 (71%)
[00:00.08 - 00:00.18] Found watermark: 0x12 (98%)
[00:00.12 - 00:00.22] Found watermark: 0x12 (98%)
[00:00.16 - 00:00.26] Found watermark: 0x12 (77%)
[00:00.20 - 00:00.30] Found watermark: 0x12 (41%)
[00:00.24 - 00:00.34]
[00:00.28 - 00:00.38]
[00:00.32 - 00:00.42]
[00:00.36 - 00:00.46]
[00:00.40 - 00:00.50]
[00:00.44 - 00:00.54] Found watermark: 0x56 (48%)
[00:00.48 - 00:00.58] Found watermark: 0x56 (79%)
[00:00.52 - 00:01.02] Found watermark: 0x56 (97%)
[00:00.56 - 00:01.06] Found watermark: 0x56 (96%)
[00:01.00 - 00:01.10] Found watermark: 0x56 (96%)
[00:01.04 - 00:01.14] Found watermark: 0x56 (97%)
[00:01.08 - 00:01.18] Found watermark: 0x56 (59%)
[00:01.12 - 00:01.22]
[00:01.16 - 00:01.26]
[00:01.20 - 00:01.30]
[00:01.24 - 00:01.34]
[00:01.28 - 00:01.38] Found watermark: 0x12 (43%)
[00:01.32 - 00:01.42] Found watermark: 0x12 (78%)
[00:01.36 - 00:01.46] Found watermark: 0x12 (71%)
[00:01.40 - 00:01.50]
[00:01.44 - 00:01.54]
[00:01.48 - 00:01.58]
[00:01.52 - 00:02.02]
[00:01.56 - 00:02.06] Found watermark: 0x12 (49%)
[00:02.00 - 00:02.10] Found watermark: 0x12 (80%)
[00:02.04 - 00:02.14] Found watermark: 0x12 (98%)
[00:02.08 - 00:02.18] Found watermark: 0x12 (62%)
[00:02.12 - 00:02.22]
[00:02.16 - 00:02.26]
[00:02.20 - 00:02.30]
Done.
```

The decoder has additional parameters controlling its performance and output:

-carrier_freq= specifies expected bottom limit of the frequency region that is used to carry the watermark. This setting should exactly correspond to the value set during encoding process.

-carrier_thr= defines a threshold for “watermark pattern detection” mechanism executed at the very first stage of the analysis process before watermark decoding/extraction. If this threshold is set to 0.0, no pattern detection is performed and each look-up window is analyzed by the decoding engine. With non-zero values (the default setting is 0.2), each look-up window is first analyzed by the pattern detection mechanism determining shape of the spectrum energies and performing similarity test of the determined shape against etalon watermark pattern shape. If the calculated similarity value is below the threshold, further decoding/extraction is not performed. Spectral pattern detection mechanism operates significantly faster than the decoding/extraction process. Thus, the use of watermark pattern detector can significantly speed-up the decoding process by skipping the extraction stage when no appropriate watermark pattern was detected. The default threshold value of 0.2 is safe to use with over-the-air recordings using ultra-sonic carrier.

-fast_scan switch forces the decoder to perform simpler and thus faster synchronization scan at expense of some loss of reliability. The use of this switch is reliable enough on not heavily distorted audio streams.

-slice_length, -slice_step - AWT3 decoder allows analyzing audio recordings by slices. This feature is especially useful in cases when watermarks search should be performed within recordings of broadcasts in which not the whole recording is watermarked, but only parts of it. These parameters define respectively length (duration) of audio stream slices taken for analysis, and analysis step; both should be specified in seconds.

-stop_on_found switch is relevant for audio analysis by slices only. It forces to stop analysis on the first found watermark.

-allow_not_reliable switch forces the decoder to list watermarks that have low reliability (that are normally skipped and not considered during the decoding process). Use this feature with caution.

-silent switch can be used if you do not want any console output. Only error messages are displayed in this mode (if any).

-ffmpeg_path= allows to define a full path to FFmpeg executable including file name. AWT3 decoder will run FFmpeg using the specified string.

-tmp_file= allows to define full path and name of temporary file that is created by FFmpeg if conversion of the input file is necessary.

-nosplash switch prevents showing program information on the screen

-show_time switch forces the encoder to show processing time

5 AWT3 Software Development Kit (SDK)

AWT3 can be easily integrated into software applications using AWT3 SDK. AWT3 SDK consists of a C library file (static or shared), corresponding header file and descriptive integration example. AWT3 SDK is provided upon request and can be compiled for any OS.

AWT3 API is very simple. It provides watermark encoding and decoding functionality, including operations with wave files on disk, with PCM audio data located in memory, and live audio stream processing functions.

5.1 AWT3 SDK Header

AWT3 header file listing is presented below. Examples of AWT3 SDK usage are supplied as part of the SDK package.

```
// =====
// AWT3 SDK header file
// (c) Alex Radzishevsky, 2007-2018 (http://audiowatermarking.info)
//
// *** IMPORTANT NOTE: AWT3 SDK may *not* be thread-safe.
//      Please use with care.
//
// Update: Sep 2018
//
// There are 3 kinds of API functionality:
// - encoding/decoding of audio files on disk (.WAV PCM RIFF wave)
// - encoding/decoding of audio data in memory --
//   applicable when the entire audio recording is available in memory as a whole
// - encoding/decoding of audio stream -- applicable in real-time/streaming systems
//
// =====

#ifdef __cplusplus
extern "C" {
#endif

#ifdef AWT3_SDK_H
#define AWT3_SDK_H

// -----
// Returns AWT3 Serial Number information (text)
// -----
void awt3_sn (char **serial_number);

// =====
// Operations with files on disk (RIFF Wave PCM audio files)
// =====

// -----
// AWT3 encoder API function (operates with WAV file)
// Embeds watermark into audio wave file.
```

```

//      returns 0 if no error, >0 otherwise (see AWT3 doc for error codes)
// -----
int awt3_encode_disk_file (
    char *input_file_name,           // INPUT: input audio file name
    char *output_file_name,          // INPUT: output audio file name
    char *hex_payload,               // INPUT: watermarking payload in hex form (e.g. "0xABCDEF")
    float skip_first,                 // INPUT: skip (do not encode) first N seconds of audio (float)
    float skip_last,                 // INPUT: if >0 -
    skip (do not encode) last N seconds of audio (float); if <0 - encode only abs(N) seconds of audio and skip the rest
    unsigned carrier_freq,            // INPUT: carrier signal starting frequency (Hz, e.g. 18000)
    float carrier_gain,               // INPUT: carrier signal gain, float (0.0 - 16.0);
    unsigned relative_gain);          // INPUT: flag (0 or 1), defines whether the carrier_gain is absolute or relative to the input signal amplitude

// -----
// AWT3 decoder API function (operates with WAV file)
// Searches, detects and extracts watermark from audio wave file.
//      returns 0 if no error, >0 otherwise (see AWT3 doc for error codes)
// -----
int awt3_decode_disk_file (
    char *input_file_name,           // INPUT: input audio file name
    unsigned wm_key_length,           // INPUT: expected watermarking payload length
    char **watermark,                 // OUTPUT: detected watermark (hex), memory allocated inside the function, freeing is on user!
    unsigned int *watermark_reliability, // OUTPUT: watermark reliability (%)
    unsigned carrier_freq,            // INPUT: expected carrier signal starting frequency (Hz, e.g. 18000)
    unsigned fast_scan,               // INPUT: fast scan mode (0 or 1)
    unsigned allow_not_reliable,      // INPUT: allow detecting even not reliable watermarks (0 or 1)
    float carrier_thr);               // INPUT: minimal carrier threshold (0.0 - 1.0, default 0.2)

// =====
// Operations in memory (PCM data)
// =====

// -----
// AWT3 encoder API function (operates with sampled PCM audio data in memory, 1 channel)
// Embeds watermark into wave PCM audio in memory.
// Several channels should be processed separately
// The audio data should be: 1 channel, PCM, sample values format: float, allowed sample values range: [-1.0 .. +1.0]
//      returns 0 if no error, >0 otherwise (see AWT3 doc for error codes)
// -----
int awt3_encode_memory (float *float_input_pcm_audio_data, // INPUT: pointer on input audio data samples array
    float **float_output_pcm_audio_data, // OUTPUT: pointer on encoded audio data samples array
    unsigned total_samples,               // INPUT: total number of samples
    unsigned Fs,                          // INPUT: audio sampling rate (allowed values: 192000, 176400, 96000, 88200, 48000, 44100, 32000, 22050, 16000, 11025, 8000 Hz)
    char *hex_payload,                   // INPUT: watermarking payload in hex form (e.g. "0xABCDEF")
    float skip_first,                    // INPUT: skip (do not encode) first N seconds of audio (float)
    float skip_last,                     // INPUT: if >0 -
    skip (do not encode) last N seconds of audio (float); if <0 - encode only abs(N) seconds of audio and skip the rest
    unsigned carrier_freq,                // INPUT: carrier signal starting frequency (Hz)
    float carrier_gain,                    // INPUT: carrier signal gain, float (0.0 - 1.0);
    unsigned relative_gain);               // INPUT: flag (0 or 1), defines whether the carrier_gain is absolute or relative to the input signal amplitude

// -----
// AWT3 detector API function (operates with sampled PCM audio data in memory, 1 channel)
// Quickly analyzes wave PCM audio in memory and returns array of values 'carrier_variance_array' and 'time_stretch_array'.
// [*] The values in 'carrier_variance_array' indicate "probability" of AWT3 watermark presence in 100 equidistant intervals within the analyzed piece of audio
// (e.g. for audio duration of 5 seconds, each value corresponds to 50ms interval).
// Returned values are from 0.0 to 1.0. Lower values (< 0.2) indicate about no carrier signal, higher values (e.g. 1.0) indicate that the carrier is present.
// 'carrier_variance_array' is expected to be declared as float[100]. Memory allocation and freeing for the 'carrier_variance_array' is on caller!

```

```

// [*] The values 'time_stretch_array' are estimation of signal time_stretch relative to its normal speed.
// Value 1.0 means no time/speed change, values > 1.0 - slower tempo, values < 1.0 -
// faster tempo, 0.0 means no estimation performed (carrier not detected)
// The decoder can handle up to 1.0 +/- 0.05 speed variation. If the time-
// stretch is higher, the decoder won't decode watermark.
// The audio data should be: 1 channel, PCM, sample values format: double, allowed sample values range: [-1.0 .. +1.0]
// returns 0 if no error, >0 otherwise (see AWT3 doc for error codes)
// NOTE: This function is auxiliary. awt3_decode_memory() does not require using awt3_detect_memory().
// -----
int awt3_detect_memory( float *float_pcm_audio_data, // INPUT: audio data samples array
    unsigned total_samples, // INPUT: total number of samples
    unsigned Fs, // INPUT: audio sampling rate (allowed values: 192000, 176400, 96000, 88200,
48000, 44100, 32000, 22050, 16000, 11025, 8000 Hz)
    unsigned carrier_freq, // INPUT: expected carrier signal starting frequency (Hz)
    unsigned time_precision, // INPUT: can be 0 (standard), 1 or 2. Lower value (e.g. 0) leads to better '
carrier_variance_array' // reliability, but lower time-
precision (approx. 0.3 sec @ 44.1 KHz), higher value provide better
    float *carrier_variance_array, // OUTPUT: array (float[100]) with carrier signal detection values in equidist
ant intervals within the analyzed audio; must be pre-allocated by user!
    float *time_stretch_array // OUTPUT: array (float[100]) with signal time_stretch estimation values in eq
uidistant intervals within the analyzed audio; must be pre-allocated by user!
);

// -----
// AWT3 decoder API function (operates with sampled PCM audio data in memory, 1 channel)
// Searches, detects and extracts watermark from wave PCM audio in memory.
// Several channels should be processed separately
// The audio data should be: 1 channel, PCM, sample values format: double, allowed sample values range: [-1.0 .. +1.0]
// returns 0 if no error, >0 otherwise (see AWT3 doc for error codes)
// -----
int awt3_decode_memory (float *float_pcm_audio_data, // INPUT: audio data samples array
    unsigned total_samples, // INPUT: total number of samples
    unsigned Fs, // INPUT: audio sampling rate (allowed values: 192000, 176400, 96000, 88200, 4
8000, 44100, 32000, 22050, 16000, 11025, 8000 Hz)
    unsigned wm_key_length, // INPUT: expected watermarking payload length
    char **watermark, // OUTPUT: detected watermark (hex), memory allocated inside the function, free
ing is on user!
    unsigned int *watermark_reliability, // OUTPUT: watermark reliability (%)
    unsigned carrier_freq, // INPUT: expected carrier signal starting frequency (Hz)
    unsigned fast_scan, // INPUT: fast scan mode (0 or 1)
    unsigned allow_not_reliable, // INPUT: allow detecting even not reliable watermarks (0 or 1)
    float carrier_thr); // INPUT: minimal carrier threshold (0.0 - 1.0, default 0.2)

// =====
// =====
//
// Live (real-time) audio stream decoding
//
// =====
// =====

// AWT3 Stream Decoder object declaration
typedef struct _AWT3_STREAM_DEC AWT3_STREAM_DEC, *PAWT3_STREAM_DEC;

// -----
// Initializes AWT3 Stream Decoder object
// Allocates memory for AWT3 Stream Decoder object which must be kept in user
// memory as long as the audio decoding function is used.
// * Parameter descriptions are provided below;
// * Returned errors can be:
// - from the list of "AWT3 Stream Decoder errors" (negative)
// - from the list of "AWT3 SDK errors" (positive)
// - 0 if no errors

```

```

// -----
int awt3_decode_stream_init(
    PAWT3_STREAM_DEC* ptr,                // pointer on the decoder object
    float decoding_audio_slice_len_sec,
    // desired size of decoded slice (in seconds), min: 1.0, max: 10.0
    float decoding_step_sec,
    // attempt watermark decoding each .. seconds, , min: 1.0, max: 'decoding_audio_slice_len_sec'
    unsigned Fs
    // audio sampling rate (allowed values: 192000, 176400, 96000, 88200, 48000, 44100, 32000, 22050, 16000, 11025,
8000 Hz)
);

// -----
// Performs audio processing of one audio buffer.
// Collects input audio in the object and performs watermark detection and extraction each
// N seconds as specified by the 'decoding_step_sec' value of the awt3_decode_stream_init() function.
// * Parameter descriptions are provided below;
// * Returned errors can be:
//     - from the list of "AWT3 Stream Decoder errors" (negative)
//     - from the list of "AWT3 SDK errors" (positive)
//     - 0 if no errors
// -----
int awt3_decode_stream_buffer(
    PAWT3_STREAM_DEC ptr,                // IN: pointer on the decoder object
    float *input_buffer,
    // IN: pointer on user's input audio data buffer (array of float sample values)
    unsigned input_data_size,
    // IN: number of samples in the 'input buffer' (can be any amount of data, but not more than 'decoding_step_sec
')
    unsigned carrier_freq,
    // IN: expected AWT3 carrier signal starting frequency (Hz, default 18000 Hz)
    unsigned fast_scan,                // IN: fast scan mode (0 or 1)
    unsigned allow_not_reliable,
    // IN: allow detecting even not reliable watermarks (0 or 1)
    float carrier_thr,
    // IN: minimal carrier threshold (0.0 - 1.0, default 0.2)
    unsigned time_precision,           // IN: can be 0 (standard), 1 or 2. Lower value (e.g. 0) leads to better watermark detection

    // reliability with respect to the selected carrier_thr, but lower time-
precision (approx. 0.3 sec @ 44.1 KHz),

    // higher value provide better time precision, but lower reliability of the detection values.
    unsigned wm_payload_bytes,
    // IN: length of expected watermark payload, bytes
    unsigned *wm_reliability,           // OUT: reliability of the watermark, if detected
    char **wm_payload,
    // OUT: detected watermark (hex text), memory allocated inside the function, freeing is on user! NULL is nothing
is detected
    float *reverse_time_sec
    // OUT: returns time passed since the watermark carrier has started (relative to the size of single
    // time slice defined by 'decoding_audio_slice_len_sec')
);

// -----
// Kills AWT3 Stream Decoder object and frees all memory
// Returns no errors
// -----
void awt3_decode_stream_kill(
    PAWT3_STREAM_DEC* ptr                // pointer on the decoder object
);

```

```

// =====
// =====
//
// Live (real-time) audio stream encoding
//
// =====
// =====

// *** DISCLAIMER ***:
// Originally, AWT3 is not a "true" real-
// time library. It is actually a "file to file". In other words, it cannot process audio by sub-second
// long audio buffers so that watermark embedding process is being spread over sequence of small audio buffers processed b
// y separate processing
// function calls. The current implementation of the real-time encoding converts AWT3 Encoder into pseudo-
// realtime library and allows using
// AWT3 encoder in real-time systems in which delay of several seconds is allowed.
//

// AWT3 Stream Decoder object declaration
typedef struct _AWT3_STREAM_ENC AWT3_STREAM_ENC, *PAWT3_STREAM_ENC;

// -----
// Initializes AWT3 Stream Encoder envelope.
// Allocates memory for AWT3 Stream Encoder object which must be kept in user
// memory as long as audio encoding function is used.
// * Parameter descriptions are provided below;
// * Returned errors can be:
//
//         - from the list of "AWT3 Stream Encoder errors" (negative)
//         - from the list of "AWT3 SDK errors" (positive)
//         - 0 if no errors
// -----
int awt3_encode_stream_init(
    // AWT3 Stream Encoder envelope parameters
    PAWT3_STREAM_ENC* ptr,           // pointer on AWT3 Stream Encoder object
    unsigned audio_iobuffer_size_smpl, // desired size of audio buffer (in samples)
    unsigned encoding_slice_size_smpl, // desired size of encoded slice (in samples)

    // -
    must be several seconds long

    // -
    must be N times 'audio_iobuffer_size_smpl', N >= 1
    // AWT3 encoding parameters
    unsigned Fs,           // audio sampling rate (allowed values: 192000, 176400, 96000, 88200, 48000,
44100, 32000, 22050, 16000, 11025, 8000 Hz)
    char *hex_payload,     // watermarking payload in hex form (e.g. "0xABCDEF")
    double skip_first,     // skip (do not encode) first N seconds of the audio slice (float, recommend
ed value: >= 0.2)
    double skip_last,     // if >0 - skip (do not encode) last N seconds of audio (float); if <0 -
encode only abs(N) seconds of audio and skip the rest (double, recommended value: >= 0.2)
    unsigned carrier_freq, // carrier signal starting frequency (Hz, recommended value: 18000)
    double carrier_gain,   // carrier signal gain, double (0.0 - 16.0);
    unsigned relative_gain // flag (0 or 1), defines whether the carrier_gain is absolute or relative t
o the input signal amplitude
);

// -----
// Performs audio processing of one audio buffer.
// Collects input audio for encoding in the internal cyclic buffer structure.
// Encoding occurs when enough buffers are collected to process as one slice.
// Since slice size is N times audio buffer size, the function returns zero-filled
// output audio buffers for the first N calls.
// The processing delay is N times audio buffer size.
// When actual watermarked audio data starts to come, 'started' flag becomes 1.
// * Parameter descriptions are provided below;
// * Returned errors can be:

```

```

//          - from the list of "AWT3 Stream Encoder errors" (negative)
//          - from the list of "AWT3 SDK errors" (positive)
//          - 0 if no errors
// -----
int awt3_encode_stream_buffer(
    PAWT3_STREAM_ENC ptr,                // pointer on AWT3 Stream Encoder object
    float *input_buffer,                 // pointer on user's input audio data buffer
    float *output_buffer,                // pointer on output audio data buffer
    unsigned *started
    // flag: indicates whether actual watermarked data starts to arrive
);

// -----
// Defines which watermark will be used to watermark the next slice
// * Parameter descriptions are provided below;
// * Returned errors can be:
//          - from the list of "AWT3 Stream Encoder errors" (negative)
//          - from the list of "AWT3 SDK errors" (positive)
//          - 0 if no errors
// -----
int awt3_encode_stream_set_payload(
    PAWT3_STREAM_ENC ptr,                // pointer on AWT3 Stream Encoder object
    char *hex_payload
    // watermarking payload in hex form (e.g. "0xABCDEF")
);

// -----
// Kills AWT3 Stream Encoder object and frees all memory
// Returns no errors
// -----
void awt3_encode_stream_kill(
    PAWT3_STREAM_ENC* ptr                // pointer on AWT3 Stream Encoder object
);

#endif

#if __cplusplus
} // Extern C
#endif

```

6 Choosing optimal watermark payload form

AWT3 operates with watermark payload in the hexadecimal form (e.g. 0xABCDEF12). Therefore, any information that you want to include into your watermark must be represented in the hexadecimal form, i.e. using digits from 0 to 9 and letters from A to F.

If you want your watermarks to contain textual data, it should be represented in the hexadecimal form first. For this purpose it is convenient to use ASCII code that introduces numerical representation for text symbols. You can find ASCII table of printable symbols in the Appendix: ASCII printable characters (page 37). On the other hand, standard ASCII representation of the text symbols is not compact. AWT3 package contains a small console utility called 'text2hex' which can be used to convert text into hexadecimal form (and back) using more compact representations (refer to page 25).

Choosing optimal watermark payload form has two aspects. On one hand, chosen watermark payload form and its length should be suitable for including all necessary information, i.e. all the information that you want to embed using watermarks. On the other hand, chosen watermark payload form should not compromise the robustness of the watermarks, it should be as compact (short) as possible to allow embedding more watermark copies into certain duration of audio. More watermark copies lead to better statistical measures and therefore better ability of the decoder to detect/extract the watermark even from heavily distorted piece of audio.

Additionally, the chosen watermark payload form should be of fixed length. This is due to the fact that for decoding you need to specify expected watermark payload size to the decoder. It means that for a specific application, you need to choose the length of the watermark payload size once and for all.

When choosing the payload form please keep in mind that it is only you (the owner of a specific AWT3 copy with unique Serial Number) who is able to detect/decode your watermarks, therefore you should not worry too much about absolutely unambiguous meaning of the form or representation of the embedded data.

Let us consider three examples.

Example 1. You're a composer or music owner wishing to embed your ID or company name and release date into your audio files. For the purpose of this example let your company be called "Music Universe".

Taking into account the aspects described above, it is better to keep your watermark payload as short as possible, but still allowing to clearly recognizing the data contained in it. The recommendation is to embed the company name in abbreviated form, like "MusUni" (6 bytes). Writing the full name as "Music Universe" (14 bytes in ASCII representation) is a waste of payload size. The date can be stored in its pure form as year number (2 bytes), month (1 byte) and day (1 byte). However, this full date form is not optimal. Instead, you can write it as a number of days since first day of 2010. In this case 2 bytes (that is 65535 days covering 179 years) should be enough.

In conclusion, the suggested payload form for this case is:

"MusUni" (6 bytes) + date (2 bytes)

This watermark payload form is quite compact (8 bytes = 64 bits) and at the same time is very informative as it contains well recognizable abbreviation of the company name and the date. With watermarking rate of 15 bits per second (in 'normal' operating mode) this watermark form allows embedding approximately 14 copies of the payload in one minute of audio, which is good watermarking density providing very good statistical measure on watermark detection stage.

For example, for 2010, February 1st (that is 32th day of 2010) the watermark payload of the discussed form will look as follows:

0x4D7573556E690020

where 0x4D is "M", 0x75 is "u", 0x73 is "s", 0x55 is "U", 0x6E is "n", 0x69 is "i", and 0x0020 is decimal 32.

Example 2. You are running a music web-store and wish to embed unique watermarks into every distributed file. The embedded watermark should contain an identifier of the recipient user (from the user database) and your company identifier.

In this case you should decide about the form of the user's identifiers. If your web-store user database assigns unique numerical ID to each user, then you can consider embedding this numerical ID directly into the watermark payload. For example, 4-bytes ID allows identifying more than 4 billions of different users. In this case the chosen payload form is:

"MusUni" (6 bytes) + userID (4 bytes)

This watermark payload form is very compact and at the same time very informative.

For example:

0x4D7573556E690000303A

where 0x4D7573556E69 is "MusUni" (like in the previous example) and 0x0000303A is numerical userID of the user with decimal number 12346.

In 'normal' operating mode, with watermarking rate of 19 bits per second this payload (10 bytes = 80 bits) allows to embed approximately 14 copies of the watermark in one minute of audio.

Example 3. You want to embed textual data containing written name of the file recipient.

Using ASCII codes to encode text in this case is not the best idea because one text symbol will consume 8 bits (one byte). You can use representation that is much more compact by enumerating text symbols in your own way instead. For example, you can decide using only small letters (from a to z) and some more characters such as space, underscore, etc:

0 – a, 1 – b, 2 – c, ..., 25 – z, 26 – space, 27 – underscore (_), 28 – dot, 29 – comma, 30 – @, 31 – &

With this encoding scheme each symbol consumes only 5 bits. So, encoding of 20 text symbols (name) consumes only 100 bits (12 bytes) instead of 160 bits (20 bytes) when using standard ASCII code.

The described idea of compact representation of textual data is implemented in a small command line utility called 'text2hex' which comes as a part of AWT3 package and described in the next section. This functionality is also implemented in AWT3 GUI tool.

6.1 text2hex utility

text2hex is a console application that comes as a part of AWT3 package. This application allows converting text data into hexadecimal form and back. While standard ASCII representation uses 8 bit (1 byte) to represent one text symbol, text2hex utility allows using more compact 5-, 6- and 7-bit representation forms. It means that using text2hex you can store more text symbols in the same amount of bytes.

Depending on your needs you may choose one of the available encoding schemes. Below is the list of symbols available in each translation table.

```

--- Symbols available in 5 bit translation table:
  a b c d e f g h i j k l m n o p q r s t u v w x y z . ,
@ &

--- Symbols available in 6 bit translation table:
  ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; ? @
[ \ ] ^ _ a b c d e f g h i j k l m n o p q r s t u v w x y
z | ~

--- Symbols available in 7 bit translation table:
  ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < =
> ? @ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [
\ ] ^ _ ` a b c d e f g h i j k l m n o p q r s t u v w x y
z { | } ~

```

It is believed that 6-bit conversion table contains enough symbols to represent any textual information in a meaningful way. This table contains all digits and all letters as well as special symbols. By using 6-bit representation you save 2 bits on every text symbol compared to standard 8-bit ASCII representation. It means that you can store, for example, 20 text symbols in 15 bytes.

text2hex has two modes: encoding and decoding. To convert text into hexadecimal form use 'encode' mode. To convert hex string back into text use 'decode' mode.

Command line usage format of the utility is:

```
text2hex <work mode> <bits> [payload size (bytes)] "string",
```

where:

`<work mode>` is the word 'encode' or 'decode' (without the quotes)

`<bits>` is number of bits per one symbol (5, 6, or 7) according to one of the tables above

`[payload size]` (mandatory only for encoding mode) is a desired length of the output hex string. If the actual data converted into the hex form is shorter, the output is padded with zeros. This is very useful in AWT3 for keeping the same hex payload length for any text string length

`"string"` is the input text string in quotes "" (in encoding mode) or hex string (in decoding mode).

Example:

```
text2hex.exe encode 6 40 "hey! don't even think to touch my files! (c)john doe"
```

This command converts 52 symbols long string "hey! Don't even think to touch my files! (c)john doe" into 40 bytes long hexadecimal form using 6-bit representation. The output hex string is:

```
0xD453D08394E313B8059C58E0EF5363760ED383B4C76758033E094DF456D0824654B53D638394C500
```

This string can now be used with AWT3 encoder.

To convert the hex string back into text run:

```
text2hex.exe decode 6 0xD453D08394E313B8059C58E0EF5363760ED383B4C76758033E094DF456D0824654B53D638394C500
```

The output will be:

```
hey! don't even think to touch my files! (c)john doe
```

7 Useful tips

In this section you can find some useful tips, recommendations and ready solutions for applications utilizing AWT3.

7.1 Converting audio or video file into file format suitable for AWT3

Whereas AWT3 decoder supports almost all feasible file formats via FFmpeg⁵ (utilizing it for files conversion into wave PCM format), AWT3 encoder accepts only wave files. However, some applications (such as embedding watermarks into video content) may require operating with already compressed audio files (e.g. MP3, AAC, AC3, MP4, etc.). In this case it can be useful to know how to convert (or decode) any audio or video file into wave PCM sound file using FFmpeg. Below is a command line for conversion of 'input_file' into 'output_file.wav' (wave PCM).

```
ffmpeg -i input_file -y -ar 44100 -vn output_file.wav
```

The 'input_file' can be either audio or video file containing audio track.

Non-standard WAV files (having erroneous or non-standard header and thus not supported by AWT3) possibly can be corrected and converted into standard waves using a third-party utility called SoX⁶:

```
sox -D source.wav --encoding signed-integer -t wavpcm converted.wav
```

7.2 Watermarking long audio files using AWT3

In case the audio file that should be watermarked is very long (so much long, that AWT3 encoder cannot process it as a whole piece due to computer memory limitations), it is still possible to watermark it in pieces. To do so it is convenient to use a third-party utility called SoX.

Here is a Windows batch file which takes a file 'long-in.wav', slices it into shorter pieces using SoX, encodes every piece using AWT3 encoder and combines the output by merging encoded pieces together back into one long output file.

⁵ FFmpeg web-site: <http://ffmpeg.org/>

⁶ SoX web-site: <http://sox.sourceforge.net/>

```

REM === Splitting the long input file 'long-in.wav' into shorter
REM === slices (600 seconds long each)
sox long-in.wav slice.wav trim 0 600 : newfile : restart

REM === Watermarking every slice separately (note: the first and the last
REM === second of the slice are skipped for smooth transition between merged parts)
for %%X in ("slice*.wav") do (
    awt3_enc %%X wtr_%%X 0x12 -skip_first=1 -skip_last=1
)

REM === Combining the watermarked slices together back into one long watermarked output
sox wtr_slice*.wav long-out.wav

```

Using this trick, it is possible to encode hours-long audio files even on computer with very limited RAM size.

In case there is a need to watermark very long MP3 file, the watermarking process together with preparatory decoding of the MP3 file into wave and following backward encoding of the watermarked file into MP3 may be extremely time consuming. A better option in this case is to watermark only selected parts of the MP3. This can be done by splitting the MP3 file into pieces without its transcoding as a whole. The following Windows batch file demonstrates the idea. The following third-party freeware cross-platform open source tools are involved: mp3splt⁷, Lame MP3 encoder⁸.

```

REM splitting input MP3 file into two pieces: first piece (4 minutes), second piece - the rest
mp3splt.exe in.mp3 0.0 4.0 EOF

REM decoding the first piece into WAV
lame.exe --decode in_000m_00s__004m_00s.mp3 tmp.wav
del in_000m_00s__004m_00s.mp3

REM embedding watermark into the first (decoded) piece
awt3_enc.exe tmp.wav out.wav 0xABCDEF12 -skip_first=1 -skip_last=1
del tmp.wav

REM encoding watermarked piece back to MP3 at 320 Kbps bitrate
lame -b 320 out.wav in_000m_00s__004m_00s.mp3
del out.wav

REM merging all pieces back into one long MP3
mp3wrap.exe merged.mp3 in_*.mp3
del in_*.mp3

```

⁷ mp3splt homepage: <http://mp3splt.sourceforge.net>

⁸ Lame encoder homepage: <http://lame.sourceforge.net/>

7.3 Extracting watermarks from broadcasts and long recordings

AWT3 decoder allows analyzing long audio recordings by slices. This feature is especially useful in cases when watermarks search should be performed within recordings of broadcasts in which not the whole recording is watermarked, but only parts of it (e.g. particular songs in radio broadcast). There are two parameters controlling this mechanism: `-slice_length` and `-slice_step`. These parameters define respectively length (duration) of audio stream slices taken for analysis, and analysis step; both should be specified in seconds.

For example, the following AWT3 decoder call:

```
awt3_dec long-broadcast-rec.wav 1 -slice_length=5 -slice_step=3 -fast_scan
```

forces the decoder to analyze the input file “long-broadcast-rec.wav” by slices 5 seconds long each; the slices are taken every 3 seconds.

Using the `-stop_on_found` switch It forces the decoder to stop analysis on first found watermark.

Recommendations on slice durations optimal for decoding are provided in section **Error! Reference source not found.**

7.4 Watermarking audio track of a video file

In order to embed watermarked audio track into a video file, it is needed first to extract original audio track from the video file, then encode the extracted audio track with AWT3 and then merge the watermarked audio track with the video track back.

To extract audio track from video file, use FFmpeg:

```
ffmpeg -i video.m4v -y audio-track.wav
```

Extracted ‘audio-track.wav’ is a wave PCM file that can be encoded with AWT3 directly or in pieces (as described section 7.2).

To replace the original audio track in the video file with the watermarked audio track, run:

```
ffmpeg -i video.m4v -i watermarked-audio-track.wav -map 0:0 -map 1:0 -vcodec copy -acodec libfaac -ab 256k out.m4v
```

This command forces FFmpeg to take video from the file ‘video.m4v’, take audio from ‘watermarked-audio-track.wav’, encode the later with FAAC encoder and then merge audio and video into the output file ‘out.m4v’.

7.5 Watermarking multiple files

Here is a simple way to watermark all files located in a folder (in Microsoft Windows).

Create a batch file called 'encode-all.bat'. Put the following contents into it:

```
mkdir outputs
@for %%X in (*.wav) do (
    awt3_enc "%%~nxX" "outputs\%%~nxX" 0x12
)
```

Place this file in the folder with the wave files to be watermarked.

Place 'awt3_enc.exe' (AWT3 command line tool) in the same folder.

Run the batch file. It will create a sub-folder called 'outputs', watermark all wave files (.WAV) located in the current folder and place all watermarked files inside the 'outputs' sub-folder.

7.6 Analyzing live audio stream

If you need to analyze live audio stream using AWT3 watermark decoder, here is an efficient recipe on establishing live analysis of audio stream from sound card (mic or line-in) using all freeware tools on Microsoft Windows OS-based machine.

Third-party tools involved:

- SoX (Sound eXchange) – <http://sox.sourceforge.net> (used for live recording of audio from sound card)
- Blat – <http://blat.net> (command line utility that sends e-mail)

Create a folder, extract SoX and Blat packages into it. Put AWT3 decoder into the folder. You should have approximately the following files structure:

```
blat.dll
pthreadgc2.dll
zlib1.dll
awt3_dec.exe
blat.exe
sox.exe
```

Create a batch-file, call it 'AWT3Monitor-runme.bat' (without quotes). This batch continuously monitors (listens to) audio stream by invoking sound recording and watermark detection every N seconds. The contents of the batch:

```
@REM === Audio stream monitoring definitions
@SET slicelen= 16
@SET slicestep= 8

@REM === Entering infinite loop
@echo start > result.txt
```

```

@SET /a waitsec= 1 + slicestep
:Start

@REM Get current timestamp in a form:
@REM DD-MM-YYYY-HH-MM-SS-ms (e.g. 10-02-2013-21-04-46-48)
@set SAVESTAMP=%DATE:/=-%-%TIME::-=%
@set SAVESTAMP=%SAVESTAMP: =%
@set SAVESTAMP=%SAVESTAMP:;=-%
@set SAVESTAMP=%SAVESTAMP:.-=%

@REM Start recording audio slice from the 'default' sound device and analyze it using AWT3 decoder
@start /SEPARATE /MIN call _rec_and_analyze.bat %SAVESTAMP%.wav %slicelen%

@REM Wait 8 seconds before the next slice
@PING 127.0.0.1 -n %waitsec% -w 1000 > nul
@more result.txt

@goto Start

```

The batch file runs continuously in the infinite loop. You will need to close its window to stop the listening process.

You can change time delay (step) between recorded audio slices by changing the value of 'slicestep' variable (in the example above the recording is performed every 8 seconds). You can change the length of the recorded audio stream slice y changing the value of 'slicelen' variable (in the example above the slice length is set to 16 seconds). With these recording settings, the recorded slices overlap, that provides reliable monitoring of the audio stream.

Create another batch file, call it '_rec_and_analyze.bat' (without quotes). Here are the contents of this batch:

```

@REM Running SoX tool to record N sec long slice from the 'default' sound card
@sox.exe --channels 1 --rate 44100 --bits 16 -d "%1" trim 0 %2

@REM Running AWT3 decoder to analyze the recorded slice
@awt3_dec.exe "%1" 4 -silent >"%1".txt
@IF NOT EXIST "%1" goto panic

@REM storing result as text to show in the main listener loop
@SET /p res= < "%1".txt
@echo %1 - %res% > result.txt

@REM Check whether a watermark was found
more "%1".txt | find /i "No watermark"
@IF ERRORLEVEL 1 goto watermark
@IF NOT ERRORLEVEL 1 goto nowatermark

:watermark
@ECHO !Watermark detected!
@mkdir Watermarked
@copy "%1" Watermarked\
@copy "%1".txt Watermarked\

```

```
@REM Send e-mail with attached WAVE file and watermark in the subject
blat -SaveSettings -f mymail@mail.com -server mail.com -port 26 -u user -pw pswrd
blat "%1".txt -attach "%1" -to to@mail.com -i "AWT3_Analyzer" -subject "watermark found"
```

```
@goto ext
```

```
:nowatermark
@ECHO !No watermark!
@goto ext
```

```
:ext
@del "%1"
@del "%1".txt
@exit
```

```
:panic
@ECHO Could not record audio! Panic!
@echo Could not record audio > result.txt
@exit
```

This batch performs recording from the sound card input into a wave file (audio slice) and then performs watermark detection in the recorded wave using AWT3 decoder. If a watermark is detected, the recorded wave file is sent by e-mail in the attachment together with the information about detected watermark in the message body. You need to put appropriate information into mail addresses and mail server parameters in the Blat tool calling string. All watermarked waves are stored in 'watermarked' sub-folder. In case of no watermark, the recorded audio slice is deleted.

You should have the following files structure:

```
_rec_and_analyze.bat
AWT3Monitor-runme.bat
blat.dll
pthreadgc2.dll
zlib1.dll
awt3_dec.exe
blat.exe
sox.exe
```

Before running this simple monitoring system, make the desired sound card input to be the default recording audio source in your Windows OS audio mixer (in Windows 7, go to "Recording devices" tab, right-click with mouse on the desired sound card input and chose "Set as Default Device"). In the audio device properties (or in the volume mixer) set appropriate recording level to avoid audio signal clipping.

Run 'AWT3Monitor-runme.bat' to start live audio stream listening and analysis.

8 Licensing terms

AWT3 technology licensing terms depend on specific application and use-case, and therefore are subject to negotiation with AWT3 developer.

9 FAQ

Q: Can one AWT3 user extract watermarks created by another AWT3 user?

A: No. Each particular copy of AWT3 binaries with particular Serial Number (SN) contains a unique numeric identifier that is used during encoding to scramble watermark payload. This security feature prevents one AWT3 user (with one SN) from extracting watermarks from watermarked files created by another AWT3 user (with another SN number).

Q: Can I extract watermark from a small fragment of the watermarked stream?

A: Yes, of course. Since the watermarking rate provided by AWT3 is quite high, it is generally enough to use only a small portion of the watermarked stream to extract the watermark from it (3-10 seconds, depending on watermark payload size), especially when the audio is only slightly distorted compared to the source file. On the other hand, if the audio stream is significantly distorted, you might need to use longer portion or the entire available watermarked recording because analysis of longer signals improves extraction reliability.

Q: Can I watermark MP3/OGG/... files?

A: This procedure requires transcoding. In other words, you need to decode MP3-files back to wave format before watermarking them. AWT3 encoder operates with wave files only, and is unable to embed watermarks directly into MP3 bit stream. Therefore, the only way to watermark MP3 files is to decode MP3 file to wave file, then watermark the obtained wave file, and then encode the watermarked wave file back to MP3.

10 Contact information

You can contact AWT3 author Alex Radzishevsky by e-mail: mailbox@audiowatermarking.info

11 Appendix: AWT3 error codes

- 1: could not allocate memory
- 2: input file cannot be read or does not exist
- 3: unrecognized command line parameter; run the program without parameters for the exact syntax
- 4: specified watermark payload size exceeds limitations of this program copy
- 5: input file conversion error: could not run FFmpeg, or the file cannot be converted into wave, or error occurred during the file conversion
- 6: could not read converted file (created by FFmpeg)
- 7: input audio file is too short
- 8: input wave should be sampled at 192000, 176400, 96000, 88200, 48000, 44100, 32000, 22050, 16000, 11025, 8000 Hz
- 9: the audio stream is too short: less than 3 copies of the watermark fit into
- 10: could not write the output file
- 11: specified watermark payload size is too long. Supported sizes are up to [.] bytes
- 12: the watermark cannot be decoded because the expected carrier freq. region exceeds the input file audio spectrum.
- 13: the watermark cannot be encoded because the expected carrier freq. region exceeds the input file audio spectrum.
- 15: expected payload size should be specified in bytes
- 16: specified payload size exceeds supported range
- 17: specified carrier signal threshold value is not supported
- 20: name of temporary audio file for FFmpeg cannot be empty
- 21: specified hexadecimal watermark payload is invalid
- 22: source and watermarked files cannot be the same
- 23: specified carrier signal gain value is not supported
- 24: watermark payload should be specified; should be of integer number of bytes
- 25: specified carrier signal frequency value is not supported
- 26: you are using demo version of the encoder. The only allowed payloads are: 0x12, 0x56 (1 byte), 0x7890, 0x9876 (2 bytes), 0xABCDEF12, 0x12345678 (4 bytes), 0x112233445566 (6 bytes), 0x1122334455667788 (8 bytes)
- 27: input file should be standard RIFF wave PCM
- 28: incorrect slicing parameter value
- 29: cannot search watermarks with chosen slice length (too short)

12 Appendix: ASCII printable characters

Binary	Dec	Hex	Glyph	Binary	Dec	Hex	Glyph
100000	32	20		1010000	80	50	P
100001	33	21	!	1010001	81	51	Q
100010	34	22	"	1010010	82	52	R
100011	35	23	#	1010011	83	53	S
100100	36	24	\$	1010100	84	54	T
100101	37	25	%	1010101	85	55	U
100110	38	26	&	1010110	86	56	V
100111	39	27		1010111	87	57	W
101000	40	28	(	1011000	88	58	X
101001	41	29	)	1011001	89	59	Y
101010	42	2A	*	1011010	90	5A	Z
101011	43	2B	+	1011011	91	5B	[
101100	44	2C	,	1011100	92	5C	\
101101	45	2D	-	1011101	93	5D	]
101110	46	2E	.	1011110	94	5E	^
101111	47	2F	/	1011111	95	5F	_
110000	48	30	0	1100000	96	60	`
110001	49	31	1	1100001	97	61	a
110010	50	32	2	1100010	98	62	b
110011	51	33	3	1100011	99	63	c
110100	52	34	4	1100100	100	64	d
110101	53	35	5	1100101	101	65	e
110110	54	36	6	1100110	102	66	f
110111	55	37	7	1100111	103	67	g
111000	56	38	8	1101000	104	68	h
111001	57	39	9	1101001	105	69	i
111010	58	3A	:	1101010	106	6A	j
111011	59	3B	;	1101011	107	6B	k
111100	60	3C	<	1101100	108	6C	l
111101	61	3D	=	1101101	109	6D	m
111110	62	3E	>	1101110	110	6E	n
111111	63	3F	?	1101111	111	6F	o
1000000	64	40	@	1110000	112	70	p
1000001	65	41	A	1110001	113	71	q
1000010	66	42	B	1110010	114	72	r

1000011	67	43	C	1110011	115	73	s
1000100	68	44	D	1110100	116	74	t
1000101	69	45	E	1110101	117	75	u
1000110	70	46	F	1110110	118	76	v
1000111	71	47	G	1110111	119	77	w
1001000	72	48	H	1111000	120	78	x
1001001	73	49	I	1111001	121	79	y
1001010	74	4A	J	1111010	122	7A	z
1001011	75	4B	K	1111011	123	7B	{
1001100	76	4C	L	1111100	124	7C	
1001101	77	4D	M	1111101	125	7D	}
1001110	78	4E	N	1111110	126	7E	~
1001111	79	4F	O				