Audio Watermarking Tools 5 (AWT5)

of www.audiowatermarking.com

Technology Guide

AWT5 version 0.01.06

October 2021

Contents

1 Introduction

Audio Watermarking Tools 5 (AWT5) of www.audiowatermarking.com is an audio trigger and watermarking solution specially designed for reliable far-field over-the-air digital signaling. AWT5 applications include "second screen" synchronization, interactivity and interaction via sound, "kick-shopping", remote triggering, events initiation, and other applications implying transmission of digital codes over-the-air through acoustic sound waves. AWT5 is based on a unique and fully proprietary technique of hiding data inside acoustic audio content. AWT5 codes reliably withstand transmission over the air (from loudspeakers to microphone), lossy audio coding (e.g. MP3, Youtube) and room reverberations (e.g. living rooms, halls). AWT5 triggering occurs with high timing accuracy.

AWT5 is a set of software utilities for embedding (and retrieving) digital watermarks carrying arbitrary digital data within (from) audio files and audio streams. The watermarks are imperceptibly embedded directly into the acoustic content so that they cannot be removed without damaging the original audio quality. The tools are distributed as a package of console utilities running on Microsoft Windows, Apple Mac OS X and Linux/*nix systems; Intel x86, x64 and ARM versions are available.

AWT5 package consists of an encoder (watermark embedding tool), a decoder (watermark extraction tool) and this document.

Encoder and decoder are console (terminal) applications controlled by parameters in the command line. The encoder embeds a watermark payload, a byte data, into an audio file or stream. The decoder detects whether a given audio stream/file (or its fragment) is watermarked and extracts the watermark if it has been found.

AWT5 SDK (Software Development Kit) for all major desktop and mobile platforms is available upon request. AWT5 SDK implements comprehensive watermark embedding and extraction functionality, including audio disk files and in-memory processing functionality, as well as live streams (buffer-by-buffer) operations.

2 General information

2.1 AWT5 at a glance

AWT5 algorithm allows embedding watermarks with arbitrary digital data (usually, few bytes long binary identifiers) into an audio stream and retrieving them accordingly. The algorithm implements a so-called "blind watermarking" approach in the sense that the original (not watermarked) audio stream is not needed to extract the watermark from the watermarked stream. The watermark is extracted directly from the watermarked audio stream or its fragment.

The watermarking algorithm works in the time-frequency domain in several narrow frequency sub-bands and implements a sophisticated proprietary embedding technique. The encoding process takes a binary watermark payload and embeds it into the audio signal within the audible frequency range by altering sub-band amplitudes and “embossing” digital data into the signal spectrum. The algorithm is designed to preserve signal quality on one hand while making the embedded data resistant to signal transformations such as lossy coding and room reverberations.

The algorithm can be applied to practically all kinds of audio data. Typical examples: music (pop, jazz, classics, rock), speech recordings, audiobooks, podcasts, musical instrument samples, etc.

AWT5 console encoder and decoder tools operate with wave PCM (.wav) audio files 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 channels[1].

Supported watermark payload size is from 1 to 12 bytes (subject to limitations in different AWT5 packages). Recommended watermark payload size ensuring uncompromised robustness usually is up to 4 bytes.

Each particular copy of AWT5 binaries with a particular Serial Number (SN) contains a unique numeric identifier used during encoding to scramble watermark payload. This security feature prevents one AWT5 user (with one SN) from extracting watermarks from watermarked files created by another AWT5 user (with another SN number).

[1] 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.

2.2 AWT5 applications

AWT5 is a universal, scalable and flexible data-hiding solution addressing different applications and use-cases from the following areas:

- digital licensing and digital delivery for audio & music distribution, selling audiobooks, music promo sharing, etc.
- monitoring of airplay (e.g. commercials, music, etc.) playing on TV, satellite, FM radio, etc.
- Interactive and event-based experiences, signaling and second screen synchronization via sound

Watermarks can be encoded and extracted offline (from files) or by means of live stream encoding/scanning.

2.3 Watermark robustness and aural transparency

On robustness

The proposed watermarking scheme demonstrates very high robustness against a majority of audio transformations, including the following:

- transmission over-the-air (OTA), i.e. sound traveling from loudspeaker to microphone even in highly reverberant environments
- lossy transcoding using MP3, MPEG-4, AAC, OPUS and other audio codecs, including multipass transcoding at very low bitrates
- time-stretching (speed variation)
- signal cropping, cutting
- sample rate conversion, amplitude re-quantization
- effect processing, from a simple EQ to an extreme dynamic range compression, reverberation, echo, spectral effects, etc.
- waveform distortions such as limiting, clipping, slope manipulation, gain control
- A/D - D/A conversion
- Transmission over radio waves (e.g. FM)

On transparency

With default parameters, the proposed watermarking algorithm demonstrates practically undistinguishable watermarking which is transparent to an average listener with audio equipment of any quality on the majority of audio content. For the sake of truth, it should be noted that like with any other real-world technology, there are examples of particular audio samples that may reveal some watermarking artifacts compared to the original non-watermarked audio; however, in such specific cases, these artifacts are rather minor and may be noticeable only to the experienced listener. Depending on the target needs, the user may adjust encoding parameters to achieve an optimal balance between aural transparency and robustness.

2.4 How the algorithm works

The watermarking algorithm works in the time-frequency domain in several narrow frequency sub-bands and implements a sophisticated proprietary embedding technique. A brief high-level description of the watermarking algorithm implemented in AWT5 is provided below:

- the watermark payload (a byte data) is converted into a watermark data packet containing encrypted binary watermark payload accompanied with an error correction code
- the source audio stream is decomposed into several frequency band signals within a selected (user-adjustable) carrier region of the audio spectrum; the number of bands used to carry the watermark payload depends on algorithm parameters and payload length; frequency boundaries of the carrier region are adjustable via algorithm parameters
- each frequency band is divided into blocks of a certain length
- each frequency band of the carrier signal is associated with the corresponding bit of the watermark payload data packet assigned to this signal block
- watermark encoding is done by altering signal amplitudes in the carrier bands using a sophisticated technique, and only during transients (explained below)
- the watermark is repeated throughout the entire audio stream as many times as the duration and dynamics of the audio stream permits
- the encoded (watermarked) output audio spectrum is then synthesized by combining modified carrier band signals back into the full band output signal.

The term “signal transient” or simply a “transient” is a short form of a term known in audio signal processing as “signal attack transient” which has no unique definition, but is usually used to describe a short-duration signal interval representing non-harmonic and high energy attack phase of a sound source, or, in other words, a short-duration signal interval (usually < 50 ms) demonstrating rapidly increasing energy and fast evolution of the sound signal short-time spectrum. Human speech and music sounds are typical examples of dynamically evolving wide-band acoustic content with rapidly changing peaks and dips. Speech sibilants (e.g. [s] like in “sip”, [z] like in “zip”, [ʃ] like in “ship”), non-sibilant fricatives ([f] like in “fine”, [θ] like in “thing”, [v] like in “vine”, etc.) and music beats (drums, cymbals) are particular examples of natural signals having wide spectrum and fast attacks.

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

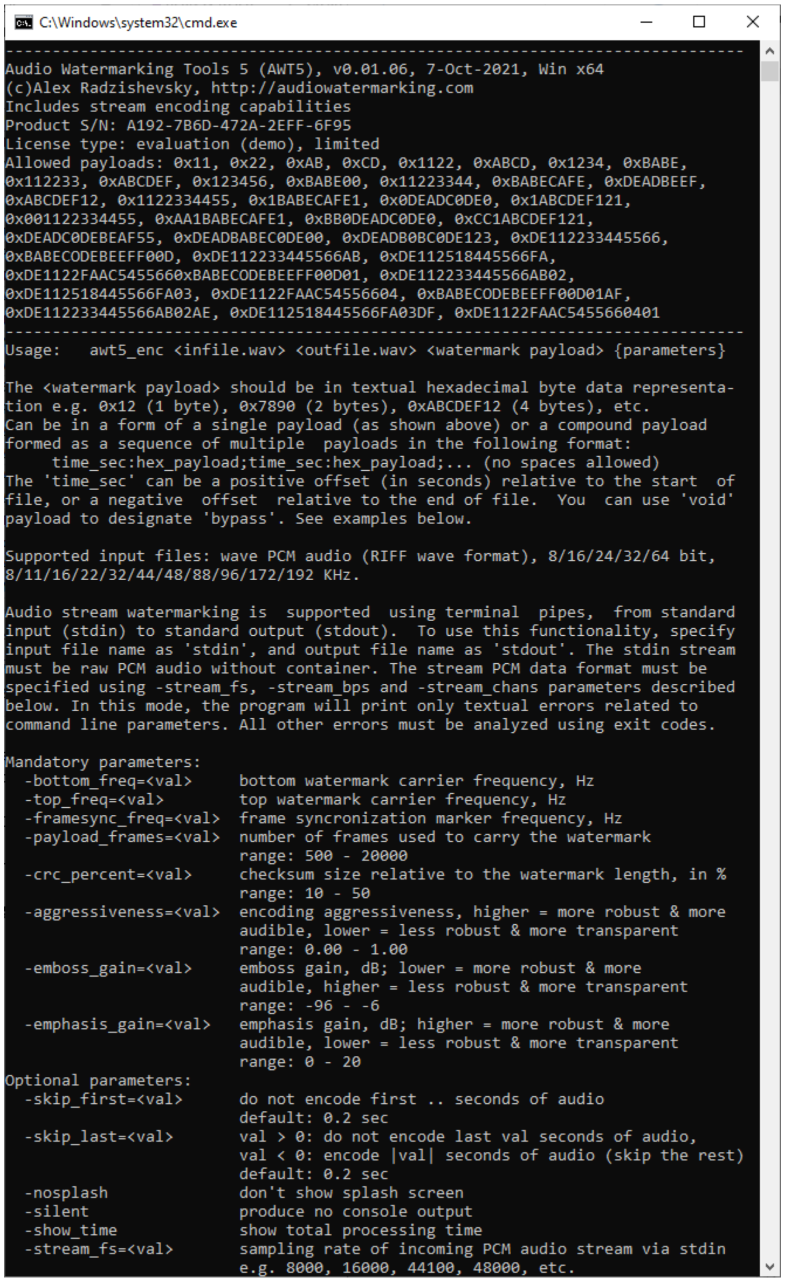
The number of copies of the watermark payload embedded into the audio stream is proportional to its duration and significantly depends on the signal dynamics. As explained above, encoding of watermark payload data is done only on signal fronts (attack). Thus, a signal with low dynamics may carry fewer watermark copies than a signal with high dynamics. For example, slow classical music is expected to carry fewer watermark copies than speech recording or modern dance music with beats.

On the decoding stage, the decoder performs the same frequency band decomposition steps as at the encoding stage. 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 of successful watermark extraction. Since the entire process is based on collecting statistics, at least several watermark copies must be contained in the analyzed fragment to allow successful extraction. The algorithm may not be able to extract a watermark from an audio segment containing only 1-2 copies of the watermark if the signal is distorted. This, however, does not prevent the decoder from reporting watermarks with high timing accuracy relative to the audio stream timeline. Since the statistics aggregation is based on a history buffer, each occurrence of a watermark is detected and reported with high timing precision (typically < 100ms).

3 Encoder

3.1 Using the encoder

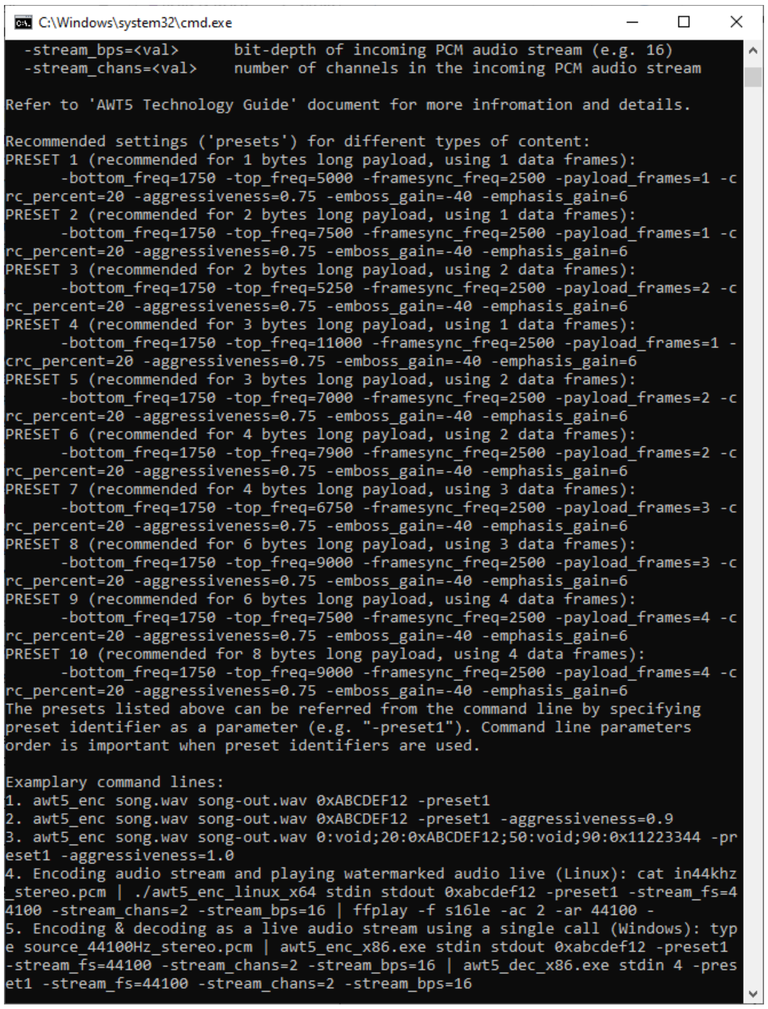
The encoder embeds a watermark payload into an audio stream or an audio file. The usage screen is shown when the encoder is invoked from the command line without parameters.



C:\Windows\system32\cmd.exe

```
-----------------------------------------------------------------------------
Audio Watermarking Tools 5 (AWT5), v0.01.06, 7-Oct-2021, Win x64
(c)Alex Radzishevsky, http://audiowatermarking.com
Includes stream encoding capabilities
Product S/N: A192-7B6D-472A-2EFF-6F95
License type: evaluation (demo), limited
Allowed payloads: 0x11, 0x22, 0xAB, 0xCD, 0x1122, 0xABCD, 0x1234, 0xBABE,
0x112233, 0xABCDEF, 0x123456, 0xBABE00, 0x11223344, 0xBABECAFE, 0xDEADBEEF,
0xABCDEF12, 0x1122334455, 0x1BABECAFE1, 0x0DEADC0DE0, 0x1ABCDEF121,
0x001122334455, 0xAA1BABECAFE1, 0xBB0DEADC0DE0, 0xCC1ABCDEF121,
0xDEADC0DEBEAF55, 0xDEADBABEC0DE00, 0xDEADB0BC0DE123, 0xDE112233445566,
0xBABECODEBEEFF00D, 0xDE112233445566AB, 0xDE112518445566FA,
0xDE1122FAAC5455660xBABECODEBEEFF00D01, 0xDE112233445566AB02,
0xDE112518445566FA03, 0xDE1122FAAC54556604, 0xBABECODEBEEFF00D01AF,
0xDE112233445566AB02AE, 0xDE112518445566FA03DF, 0xDE1122FAAC5455660401
-----------------------------------------------------------------------------
Usage:   awt5_enc <infile.wav> <outfile.wav> <watermark payload> {parameters}

The <watermark payload> should be in textual hexadecimal byte data representa-
tion e.g. 0x12 (1 byte), 0x7890 (2 bytes), 0xABCDEF12 (4 bytes), etc.
Can be in a form of a single payload (as shown above) or a compound payload
formed as a sequence of multiple  payloads in the following format:
     time_sec:hex_payload;time_sec:hex_payload;... (no spaces allowed)
The 'time_sec' can be a positive offset (in seconds) relative to the start  of
file, or a negative  offset  relative to the end of file.  You  can use 'void'
payload to designate 'bypass'. See examples below.

Supported input files: wave PCM audio (RIFF wave format), 8/16/24/32/64 bit,
8/11/16/22/32/44/48/88/96/172/192 KHz.

Audio stream watermarking is  supported  using terminal  pipes,  from standard
input (stdin) to standard output (stdout).  To use this functionality, specify
input file name as 'stdin', and output file name as 'stdout'. The stdin stream
must be raw PCM audio without container. The stream PCM data format must be
specified using -stream_fs, -stream_bps and -stream_chans parameters described
below. In this mode, the program will print only textual errors related to
command line parameters. All other errors must be analyzed using exit codes.

Mandatory parameters:
  -bottom_freq=<val>      bottom watermark carrier frequency, Hz
  -top_freq=<val>         top watermark carrier frequency, Hz
  -framesync_freq=<val>   frame syncronization marker frequency, Hz
  -payload_frames=<val>   number of frames used to carry the watermark
                          range: 500 - 20000
  -crc_percent=<val>      checksum size relative to the watermark length, in %
                          range: 10 - 50
  -aggressiveness=<val>   encoding aggressiveness, higher = more robust & more
                          audible, lower = less robust & more transparent
                          range: 0.00 - 1.00
  -emboss_gain=<val>      emboss gain, dB; lower = more robust & more
                          audible, higher = less robust & more transparent
                          range: -96 - -6
  -emphasis_gain=<val>    emphasis gain, dB; higher = more robust & more
                          audible, lower = less robust & more transparent
                          range: 0 - 20
Optional parameters:
  -skip_first=<val>       do not encode first .. seconds of audio
                          default: 0.2 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: 0.2 sec
  -nosplash               don't show splash screen
  -silent                 produce no console output
  -show_time              show total processing time
  -stream_fs=<val>        sampling rate of incoming PCM audio stream via stdin
                          e.g. 8000, 16000, 44100, 48000, etc.
```

Figure 1. Encoder usage screen



```
  -stream_bps=<val>      bit-depth of incoming PCM audio stream (e.g. 16)
  -stream_chans=<val>    number of channels in the incoming PCM audio stream

Refer to 'AWT5 Technology Guide' document for more infromation and details.

Recommended settings ('presets') for different types of content:
PRESET 1 (recommended for 1 bytes long payload, using 1 data frames):
      -bottom_freq=1750 -top_freq=5000 -framesync_freq=2500 -payload_frames=1 -c
rc_percent=20 -aggressiveness=0.75 -emboss_gain=-40 -emphasis_gain=6
PRESET 2 (recommended for 2 bytes long payload, using 1 data frames):
      -bottom_freq=1750 -top_freq=7500 -framesync_freq=2500 -payload_frames=1 -c
rc_percent=20 -aggressiveness=0.75 -emboss_gain=-40 -emphasis_gain=6
PRESET 3 (recommended for 2 bytes long payload, using 2 data frames):
      -bottom_freq=1750 -top_freq=5250 -framesync_freq=2500 -payload_frames=2 -c
rc_percent=20 -aggressiveness=0.75 -emboss_gain=-40 -emphasis_gain=6
PRESET 4 (recommended for 3 bytes long payload, using 1 data frames):
      -bottom_freq=1750 -top_freq=11000 -framesync_freq=2500 -payload_frames=1 -
crc_percent=20 -aggressiveness=0.75 -emboss_gain=-40 -emphasis_gain=6
PRESET 5 (recommended for 3 bytes long payload, using 2 data frames):
      -bottom_freq=1750 -top_freq=7000 -framesync_freq=2500 -payload_frames=2 -c
rc_percent=20 -aggressiveness=0.75 -emboss_gain=-40 -emphasis_gain=6
PRESET 6 (recommended for 4 bytes long payload, using 2 data frames):
      -bottom_freq=1750 -top_freq=7900 -framesync_freq=2500 -payload_frames=2 -c
rc_percent=20 -aggressiveness=0.75 -emboss_gain=-40 -emphasis_gain=6
PRESET 7 (recommended for 4 bytes long payload, using 3 data frames):
      -bottom_freq=1750 -top_freq=6750 -framesync_freq=2500 -payload_frames=3 -c
rc_percent=20 -aggressiveness=0.75 -emboss_gain=-40 -emphasis_gain=6
PRESET 8 (recommended for 6 bytes long payload, using 3 data frames):
      -bottom_freq=1750 -top_freq=9000 -framesync_freq=2500 -payload_frames=3 -c
rc_percent=20 -aggressiveness=0.75 -emboss_gain=-40 -emphasis_gain=6
PRESET 9 (recommended for 6 bytes long payload, using 4 data frames):
      -bottom_freq=1750 -top_freq=7500 -framesync_freq=2500 -payload_frames=4 -c
rc_percent=20 -aggressiveness=0.75 -emboss_gain=-40 -emphasis_gain=6
PRESET 10 (recommended for 8 bytes long payload, using 4 data frames):
      -bottom_freq=1750 -top_freq=9000 -framesync_freq=2500 -payload_frames=4 -c
rc_percent=20 -aggressiveness=0.75 -emboss_gain=-40 -emphasis_gain=6
The presets listed above can be referred from the command line by specifying
preset identifier as a parameter (e.g. "-preset1"). Command line parameters
order is important when preset identifiers are used.

Examplary command lines:
1. awt5_enc song.wav song-out.wav 0xABCDEF12 -preset1
2. awt5_enc song.wav song-out.wav 0xABCDEF12 -preset1 -aggressiveness=0.9
3. awt5_enc song.wav song-out.wav 0:void;20:0xABCDEF12;50:void;90:0x11223344 -pr
eset1 -aggressiveness=1.0
4. Encoding audio stream and playing watermarked audio live (Linux): cat in44khz
_stereo.pcm | ./awt5_enc_linux_x64 stdin stdout 0xabcdef12 -preset1 -stream_fs=4
4100 -stream_chans=2 -stream_bps=16 | ffplay -f s16le -ac 2 -ar 44100 -
5. Encoding & decoding as a live audio stream using a single call (Windows): typ
e source_44100Hz_stereo.pcm | awt5_enc_x86.exe stdin stdout 0xabcdef12 -preset1
-stream_fs=44100 -stream_chans=2 -stream_bps=16 | awt5_dec_x86.exe stdin 4 -pres
et1 -stream_fs=44100 -stream_chans=2 -stream_bps=16
```

Figure 2. Encoder usage screen (cont.)

In order to encode an audio file, the following parameters must be specified:

1. input audio file name (or “stdin” for audio pipe stream)
2. output audio file name (or “stdout” for audio pipe stream)
3. watermark payload in the hexadecimal form e.g. 0xABCDEF12 (refer to page 30 for more information on the watermark payload form)
4. mandatory algorithm settings: bottom & top frequency of the carrier band (-bottom_freq, -top_freq), number of data frames (-payload_frames), frame-sync data band frequency (-framesync_freq), size of checksum (-crc_percent) and encoding aggressiveness.

Other encoding parameters are optional.

You can pick algorithm settings from one of the recommended sets of settings (“presets”) listed below.

```
PRESET 1 (recommended for 1 byte long payload, using 1 data frame):
      -bottom_freq=1750 -top_freq=5000 -framesync_freq=2500 -payload_frames=1 -crc_percent=20 -
aggressiveness=0.75

PRESET 2 (recommended for 2 bytes long payload, using 1 data frame):
      -bottom_freq=1750 -top_freq=7500 -framesync_freq=2500 -payload_frames=1 -crc_percent=20 -
aggressiveness=0.75

PRESET 3 (recommended for 2 bytes long payload, using 2 data frames):
      -bottom_freq=1750 -top_freq=5250 -framesync_freq=2500 -payload_frames=2 -crc_percent=20 -
aggressiveness=0.75

PRESET 4 (recommended for 3 bytes long payload, using 1 data frames):
      -bottom_freq=1750 -top_freq=11000 -framesync_freq=2500 -payload_frames=1 -crc_percent=20 -
aggressiveness=0.75

PRESET 5 (recommended for 3 bytes long payload, using 2 data frames):
      -bottom_freq=1750 -top_freq=7000 -framesync_freq=2500 -payload_frames=2 -crc_percent=20 -
aggressiveness=0.75

PRESET 6 (recommended for 4 bytes long payload, using 2 data frames):
      -bottom_freq=1750 -top_freq=7900 -framesync_freq=2500 -payload_frames=2 -crc_percent=20 -
aggressiveness=0.75

PRESET 7 (recommended for 4 bytes long payload, using 3 data frames):
      -bottom_freq=1750 -top_freq=6750 -framesync_freq=2500 -payload_frames=3 -crc_percent=20 -
aggressiveness=0.75

PRESET 8 (recommended for 6 bytes long payload, using 3 data frames):
      -bottom_freq=1750 -top_freq=9000 -framesync_freq=2500 -payload_frames=3 -crc_percent=20 -
aggressiveness=0.75

PRESET 9 (recommended for 6 bytes long payload, using 4 data frames):
      -bottom_freq=1750 -top_freq=7500 -framesync_freq=2500 -payload_frames=4 -crc_percent=20 -
aggressiveness=0.75

PRESET 10 (recommended for 8 bytes long payload, using 4 data frames):
      -bottom_freq=1750 -top_freq=9000 -framesync_freq=2500 -payload_frames=4 -crc_percent=20 -
aggressiveness=0.75
```

These presets are also listed in the encoder usage screen. The presets are explained in section 3.5.

NOTE: the presets can be referred from the command line by specifying the preset identifier as a parameter (e.g. "-preset1"). Command-line parameters order is important when preset identifiers are used. If you want to modify a preset setting, it should be specified after the preset identifier.

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 channels[2].

The watermark payload should be specified in a hexadecimal form, i.e. using “0”-“9” and “A”-“F” characters (for example, “0xFE21” which represents 2 bytes long payload). The supported length of the watermark payload is from 1 to 7 bytes. Recommended watermark payload sizes are 1-4 bytes. For more information about the watermark payload form, please refer to page 30.

Please note: The maximal watermark payload size supported by a particular copy of the encoder depends on the version of AWT5 package that you license. It can be: 1 byte max, 2 bytes max, 4 bytes max or unlimited.

If you are using demo version of the AWT5 package, the only watermarks that can be embedded are:

- 1 byte: 0x11, 0x22, 0xAB, 0xCD
- 2 bytes: 0x1122, 0xABCD, 0x1234, 0xBABE
- 3 bytes: 0x112233, 0xABCDEF, 0x123456, 0xBABE00
- 4 bytes: 0x11223344, 0xBABECAFE, 0xDEADBEEF, 0xABCDEF12
- 5 bytes: 0x1122334455, 0x1BABECAFE1, 0x0DEADC0DE0, 0x1ABCDEF121
- 6 bytes: 0x001122334455, 0xAA1BABECAFE1, 0xBB0DEADC0DE0, 0xCC1ABCDEF121
- 7 bytes: 0xDEADC0DEBEAF55, 0xDEADBABEC0DE00, 0xDEADB0BC0DE123, 0xDE112233445566
- 8 bytes: 0xBABECODEBEEFF00D, 0xDE112233445566AB, 0xDE112518445566FA, 0xDE1122FAAC545566
- 9 bytes: 0xBABECODEBEEFF00D01, 0xDE112233445566AB02, 0xDE112518445566FA03, 0xDE1122FAAC54556604
- 10 bytes: 0xBABECODEBEEFF00D01AF, 0xDE112233445566AB02AE, 0xDE112518445566FA03DF, 0xDE1122FAAC5455660401.

Despite this limitation, the demo package offers enough flexibility to perform comprehensive performance tests (including acoustic performance, execution speed, robustness and so on) to let you decide whether AWT5 suits your needs before licensing the fully functional version.

Here is an example of running the encoder with audio files on a disk. The input file is “music.wav”, the output (watermarked) file is “music-wtr.wav”, the watermark payload is “0xABCDEF12” (4 bytes), and the other settings correspond to those of the Preset 6:

```
awt5_enc music.wav music-wtr.wav 0xABCDEF12 -bottom_freq=1750 -top_freq=7900 -framesync_freq=2500 -
payload_frames=2 -crc_percent=20 -aggressiveness=0.75
```

[2] 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.

In this example, the specified watermark payload is embedded (repeated) throughout the entire audio file, as many times as the audio file duration and the audio dynamics permit.

The identical result can be obtained by referring to the algorithm parameters using the corresponding preset number:

```
awt5_enc music.wav music-wtr.wav 0xABCDEF12 -preset6
```

The watermark payload can also be specified in a compound form, i.e. as a sequence of different payloads that are to be embedded at different time locations within the audio stream. In such a case, each payload must be accompanied by the time where the payload should start. Such sequence should be specified in the following form:

```
time_sec:hex_payload;time_sec:hex_payload;...
```

No spaces are allowed in the compound payload, the semicolon ";" is used as a delimiter. The 'time_sec' can be a positive offset (in seconds) relative to the start of the file, or a negative offset relative to the end of file. You can use 'void' payload to designate bypass (no watermarking). Example of running the encoder with a sequence of payloads:

```
awt5_enc music.wav music-wtr.wav 10:0xABCDEF12;25.5:0x11223344;50:0xABCDEF12 -preset1
```

3.2 Encoder parameters and settings in detail

The following settings control the watermarking algorithm:

- bottom & top frequency of the watermark carrier band (**-bottom_freq, -top_freq**),
- frequency carrying frame-sync data (**-framesync_freq**)
- number of frames used to carry the watermark payload (**-payload_frames**),
- size of checksum (**-crc_percent**)
- encoding aggressiveness (**-aggressiveness**)

-bottom_freq, -top_freq specify the audio frequency region used to carry the watermark payload. The settings are flexible; however, the carrier band must be defined in an audible frequency range for the watermarks to be detectable. The default/recommended carrier band is between 1.5 – 7 kHz. It is recommended to set the bottom carrier frequency not too low to keep the watermarks inaudible. Signal alterations in frequencies below 1000 Hz may lead to noticeable artifacts. The carrier frequency range should not spread too high because such watermarks may become vulnerable to bandwidth limiting caused by lossy coders and broadcasting channels.

The actual raw watermark payload length depends on the user-defined watermark payload plus the added CRC code. The size of the CRC code is adjustable by the user using **-crc_percent**. For example, with 24-bit (3 bytes) watermark payload and 25% CRC, the total raw watermark payload is 24 + 24*25/100 = 30 bit. A longer CRC leads to more reliable watermark extraction with a lower false detection rate. On the other hand, longer raw watermark payload length may degrade watermark detection due to increased bit extraction errors, especially with highly distorted or corrupted signals. Thus, optimal CRC lengths are somewhere between 15% and 40%.

The raw watermark payload can be split over multiple data-frames specified by **-payload_frames**. The watermark data length for each single data frame is calculated as:

$$\frac{\text{raw watermark payload length}}{\text{number of data frames}}.$$

For example, if the number of data frames is set to 1, the entire raw watermark payload is encoded within a single data frame spread over the carrier frequency band. With the number of data frames set to 2, the raw watermark data at each frame is half of the total raw watermark payload length. It means that for successful watermark extraction, it is required to collect and decode two data frames. Multi-frame payloads require additional frame-sync information to be embedded along with the watermark payload. The **-framesync_freq** setting defines the frequency at which the frame-sync data is embedded. This setting must fall inside the carrier bandwidth i.e. in between **-bottom_freq** and **-top_freq.**

The algorithm settings described above allow fine-tuning the watermark embedding process to find a suitable trade-off between watermark carrier bandwidth used to carry the watermark and watermark data span over the chosen carrier bandwidth for each specific watermark payload size. With a given watermark payload (e.g. 3 bytes), one can decide to fit it into a single data frame with a larger carrier bandwidth or multiple data frames using a narrower data carrier.

Section 3.3 lists minimal & recommended watermark carrier signal bandwidth in audio frequency domain as a function of watermark payload length, CRC % and number of watermark data frames.

The **-aggressiveness** setting impacts the number of watermark copies embedded into the signal. The setting controls the encoder’s sensitivity to signal fronts (attacks). As explained in Section 2.4, to withstand reverberations, watermark embedding events occur on signal fronts (attacks) only. The aggressiveness setting tunes the encoder’s attack detector sensitivity. Larger aggressiveness values lead to a relaxation of requirements to signal attack, which leads to an increased amount of watermark embeddings within a given audio signal. Lower aggressiveness values result in fewer watermark copies but even more transparent (inaudible) watermarking.

The **-emboss_gain** and **-emphasis_gain** settings control the strength of watermark imprinting (embossment) into the signal spectrum. With the default values (-40 dB and 6 dB correspondingly), the encoding process produces “holes” and “bumps” in the spectrum on transients. It makes the watermarks robust to over-the-air transmission, but may impact transparency if too wide carrier bandwidth is used, or if the carrier bandwidth starts (**-bottom_freq**) too low. In applications implying only moderate signal distortions on its way from the encoder to the decoder, more relaxed values for the gains can be used. For example, -emboss_gain=-20, emphasis_gain=0, or even -emboss_gain=-12, emphasis_gain=0. Proper balance between the embossment strength and audio transparency is to be found on application case-by-case basis.

AWT5 encoder and decoder offer multiple predefined presets of settings covering some typical use-cases, but the user can experiment with the settings himself to find the most optimal and balanced set of parameters for his specific needs.

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

```
awt5_enc music.wav music-wtr.wav 0xABCDEF12 -bottom_freq=1750 -top_freq=7000 -framesync_freq=2500 -
payload_frames=2 -crc_percent=20 -aggressiveness=0.75
```

```
awt5_enc music.wav music-wtr.wav 0xABCDEF12 -bottom_freq=1000 -top_freq=6000 -framesync_freq=3000 -
payload_frames=1 -crc_percent=30 -aggressiveness=0.1
```

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

-skip_first= / **-skip_last=** specifies the number of seconds from the start/end of the input audio that the encoder should skip (i.e. left untouched, not watermarked). This feature can be used to allow smooth cross-fades between two or more consequent fragments of a continuous audio stream. If specified **-skip_last** value is negative, it forces the encoder to watermark only the specified duration of audio (instead of skipping the same duration at the end). Note: **-skip_first** / **-skip_last** cannot be used in combination with compound payload.

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

–nosplash prevents showing program information on the screen

–show_time forces the encoder to show processing time

–stream_fs specifies sampling rate of the incoming PCM audio stream received from stdin. Acceptable values: 192000, 176400, 96000, 88200, 48000, 44100, 32000, 22050, 16000, 11025, 8000. The parameter should be used only in streaming mode (stdin-to-stdout processing).

–stream_bps specifies the bit-depth of the incoming PCM audio stream samples (bits per sample). Acceptable values: 16. The parameter should be used only in streaming mode (stdin-to-stdout processing).

–stream_chans specifies the number of audio channels in the incoming PCM audio stream. Acceptable values: 1 - 32. The parameter should be used only in streaming mode (stdin-to-stdout processing).

3.3 Minimal frequency bandwidth and multi-layer watermarking

The table below lists minimal & recommended watermark carrier signal bandwidth in audio frequency domain as a function of watermark payload length, CRC % and number of watermark data frames. The bandwidth is shown in Hz. In terms of the AWT5 algorithm, the bandwidth is defined by the difference between the values of the **-top_freq** and the **-bottom_freq** values.

Each cell in the table below includes two numbers delimited with the "/" symbol. The first (smaller) number is the minimum required bandwidth, and the second (larger) number is a recommended bandwidth providing robust watermarking.

For a given watermark payload length and CRC %, there is a trade-off between the required bandwidth and the number of data frames used to carry the watermark. Generally, it is recommended to use fewer data frames to carry watermarks.

Payload length, bytes	**CRC, %**	**Watermark data frames**					
		1	**2**	**3**	**4**	**5**	**6**
1	10	2250 / 3000	2000 / 2500	2500 / 3250	2500 / 3250	4000 / 5000	4000 / 5000
	20	2500 / 3250	2000 / 2500	2500 / 3250	2500 / 3250	4000 / 5000	4000 / 5000
	30	2500 / 3250	2000 / 2500	2500 / 3250	2500 / 3250	4000 / 5000	4000 / 5000
	40	2750 / 3500	1750 / 2250	2500 / 3250	2500 / 3250	3250 / 4250	4000 / 5000
2	10	4750 / 6000	2500 / 3250	2250 / 3000	2250 / 3000	2750 / 3500	3250 / 4250
	20	5000 / 6250	2750 / 3500	2250 / 3000	2250 / 3000	2750 / 3500	2750 / 3500
	30	5500 / 7000	3000 / 3750	2250 / 3000	2250 / 3000	2500 / 3250	2750 / 3500
	40	5750 / 7250	3000 / 3750	2500 / 3250	2250 / 3000	2500 / 3250	2750 / 3500
3	10	6750 / 8500	3500 / 4500	2750 / 3500	2250 / 3000	2500 / 3250	2500 / 3250
	20	7750 / 9750	4250 / 5500	3000 / 3750	2500 / 3250	2500 / 3250	2500 / 3250
	30	8250 / 10500	4500 / 5750	3250 / 4250	2500 / 3250	2500 / 3250	2500 / 3250
	40	9000 / 11250	4750 / 6000	3500 / 4500	2750 / 3500	2500 / 3250	2500 / 3250
4	10	9250 / 11750	5000 / 6250	3500 / 4500	2750 / 3500	2500 / 3250	2500 / 3250
	20	10000 / 12500	5250 / 6750	4000 / 5000	3000 / 3750	2750 / 3500	2500 / 3250
	30	-	5750 / 7250	4250 / 5500	3250 / 4250	3000 / 3750	2500 / 3250
	40	-	6250 / 8000	4500 / 5750	3500 / 4500	3000 / 3750	2750 / 3500
5	10	-	6000 / 7500	4500 / 5750	3250 / 4250	3000 / 3750	2750 / 3500
	20	-	6500 / 8250	4750 / 6000	3500 / 4500	3250 / 4250	2750 / 3500
	30	-	7000 / 8750	5250 / 6750	4000 / 5000	3500 / 4500	3000 / 3750
	40	-	7750 / 9750	5500 / 7000	4250 / 5500	4000 / 5000	3250 / 4250
6	10	-	7250 / 9250	5250 / 6750	4250 / 5500	3500 / 4500	3000 / 3750
	20	-	8000 / 10000	5750 / 7250	4500 / 5750	4000 / 5000	3250 / 4250
	30	-	8500 / 10750	6000 / 7500	4750 / 6000	4250 / 5500	3500 / 4500
	40	-	9250 / 11750	6500 / 8250	5000 / 6250	4500 / 5750	4000 / 5000
7	10	-	8500 / 10750	6000 / 7500	4750 / 6000	4250 / 5500	3500 / 4500
	20	-	9250 / 11750	6500 / 8250	5000 / 6250	4500 / 5750	4000 / 5000
	30	-	10000 / 12500	7000 / 8750	5500 / 7000	4750 / 6000	4250 / 5500
	40	-	-	7250 / 9250	5750 / 7250	5000 / 6250	4250 / 5500
8	10	-	9500 / 12000	6750 / 8500	5250 / 6750	4500 / 5750	4000 / 5000
	20	-	-	7250 / 9250	5750 / 7250	5000 / 6250	4250 / 5500
	30	-	-	8000 / 10000	6000 / 7500	5250 / 6750	4500 / 5750
	40	-	-	8500 / 10750	6500 / 8250	5500 / 7000	4750 / 6000
9	10	-	-	7750 / 9750	5750 / 7250	5000 / 6250	4500 / 5750
	20	-	-	8250 / 10500	6250 / 8000	5500 / 7000	4750 / 6000
	30	-	-	9000 / 11250	6750 / 8500	5750 / 7250	5000 / 6250
	40	-	-	9500 / 12000	7250 / 9250	6250 / 8000	5250 / 6750

10	10	-	-	8500 / 10750	6250 / 8000	5500 / 7000	4750 / 6000
	20	-	-	9000 / 11250	6750 / 8500	6000 / 7500	5000 / 6250
	30	-	-	9750 / 12250	7250 / 9250	6250 / 8000	5500 / 7000
	40	-	-	-	8000 / 10000	6750 / 8500	5750 / 7250
11	10	-	-	9250 / 11750	7000 / 8750	6000 / 7500	5250 / 6750
	20	-	-	10000 / 12500	7750 / 9750	6500 / 8250	5500 / 7000
	30	-	-	-	8250 / 10500	6750 / 8500	5750 / 7250
	40	-	-	-	8750 / 11000	7250 / 9250	6250 / 8000
12	10	-	-	10000 / 12500	7750 / 9750	6500 / 8250	5500 / 7000
	20	-	-	-	8250 / 10500	6750 / 8500	6000 / 7500
	30	-	-	-	9000 / 11250	7250 / 9250	6250 / 8000
	40	-	-	-	9500 / 12000	8000 / 10000	6750 / 8500

For example, for a 3-bytes long watermark payload and 20% CRC, a minimum bandwidth of 7750 Hz is required if the watermark is stored in a single data frame, while the recommended bandwidth is 9750. If two data frames carry the same watermark, the bandwidth reduces to 4250 Hz (minimal) and 5500 (recommended).

The user can select carrier band frequency region according to his needs. The same bandwidth of 5000 Hz can be obtained with **-top_freq=2000** & **-bottom_freq=7000** and also with **-top_freq=4000** & **-bottom_freq=9000.**

This flexibly configurable watermark carrier allows embedding multiple independent watermarks withing the same audio signal. Here is an example of dual-layer watermarking using 3-bytes long watermark payload:

Layer 1:

```
awt5_enc music.wav music-wtr.wav 0xABCDEF -bottom_freq=1750 -top_freq=6000 -framesync_freq=2500 -
payload_frames=2 -crc_percent=20 -aggressiveness=0.75
```

Layer 2:

```
awt5_enc music.wav music-wtr.wav 0x123456 -bottom_freq=6500 -top_freq=11750 -framesync_freq=7500 -
payload_frames=2 -crc_percent=20 -aggressiveness=0.75
```

3.4 Live audio stream encoding in command line

The encoder can process live audio streams directly from command line / console. It can take PCM audio stream from the standard input (stdin) and send the encoded (watermarked) audio stream into the standard output (stdout). To use this functionality, specify the input file name as 'stdin' and the output file name as ‘stdout’. The stdin stream must be raw PCM audio without a container. The stream PCM data format must be specified using **-stream_fs**, **-stream_bps** and **-stream_chans** parameters described below. The example below shows how to start live watermark encoding in the Linux environment:

```
cat in44khz_stereo.pcm | ./awt5_enc_linux_x64 stdin stdout 0xabcdef12 -preset1 -stream_fs=44100 -
stream_chans=2 -stream_bps=16 | ffplay -f s16le -ac 2 -ar 44100 -
```

The same for Windows:

```
type in44khz_stereo.pcm | ./awt5_enc_linux_x64 stdin stdout 0xabcdef12 -preset1 -stream_fs=44100 -
stream_chans=2 -stream_bps=16 | ffplay -f s16le -ac 2 -ar 44100 -
```

Note that in this mode, the program only prints textual errors related to command line parameters. All other errors must be analyzed using exit codes.

3.5 Recommended encoding parameters (presets)

The present section provides several examples of recommended encoding parameters ("presets") for different watermark payload lengths. Each preset is a set of pre-defined, recommended parameter values optimized for different payload sizes.

The following presets are available in the encoder and the decoder:

```
PRESET 1 (recommended for 1 byte long payload, using 1 data frame):
      -bottom_freq=1750 -top_freq=5000 -framesync_freq=2500 -payload_frames=1 -crc_percent=20 -
aggressiveness=0.75 -emboss_gain=-40 -emphasis_gain=6

PRESET 2 (recommended for 2 bytes long payload, using 1 data frame):
      -bottom_freq=1750 -top_freq=7500 -framesync_freq=2500 -payload_frames=1 -crc_percent=20 -
aggressiveness=0.75 -emboss_gain=-40 -emphasis_gain=6

PRESET 3 (recommended for 2 bytes long payload, using 2 data frames):
      -bottom_freq=1750 -top_freq=5250 -framesync_freq=2500 -payload_frames=2 -crc_percent=20 -
aggressiveness=0.75 -emboss_gain=-40 -emphasis_gain=6

PRESET 4 (recommended for 3 bytes long payload, using 1 data frames):
      -bottom_freq=1750 -top_freq=11000 -framesync_freq=2500 -payload_frames=1 -crc_percent=20 -
aggressiveness=0.75 -emboss_gain=-40 -emphasis_gain=6

PRESET 5 (recommended for 3 bytes long payload, using 2 data frames):
      -bottom_freq=1750 -top_freq=7000 -framesync_freq=2500 -payload_frames=2 -crc_percent=20 -
aggressiveness=0.75 -emboss_gain=-40 -emphasis_gain=6

PRESET 6 (recommended for 4 bytes long payload, using 2 data frames):
      -bottom_freq=1750 -top_freq=7900 -framesync_freq=2500 -payload_frames=2 -crc_percent=20 -
aggressiveness=0.75 -emboss_gain=-40 -emphasis_gain=6
```

```
PRESET 7 (recommended for 4 bytes long payload, using 3 data frames):
      -bottom_freq=1750 -top_freq=6750 -framesync_freq=2500 -payload_frames=3 -crc_percent=20 -
aggressiveness=0.75 -emboss_gain=-40 -emphasis_gain=6

PRESET 8 (recommended for 6 bytes long payload, using 3 data frames):
      -bottom_freq=1750 -top_freq=9000 -framesync_freq=2500 -payload_frames=3 -crc_percent=20 -
aggressiveness=0.75 -emboss_gain=-40 -emphasis_gain=6

PRESET 9 (recommended for 6 bytes long payload, using 4 data frames):
      -bottom_freq=1750 -top_freq=7500 -framesync_freq=2500 -payload_frames=4 -crc_percent=20 -
aggressiveness=0.75 -emboss_gain=-40 -emphasis_gain=6

PRESET 10 (recommended for 8 bytes long payload, using 4 data frames):
      -bottom_freq=1750 -top_freq=9000 -framesync_freq=2500 -payload_frames=4 -crc_percent=20 -
aggressiveness=0.75 -emboss_gain=-40 -emphasis_gain=6
```

Encoding example using 3-byte watermark payload:

```
awt5_enc music.wav music-wtr.wav 0xABCDEF -bottom_freq=1750 -top_freq=11000 -framesync_freq=2500 -
payload_frames=1 -crc_percent=20 -aggressiveness=0.75
```

or using the preset alias corresponding to the same settings:

```
awt5_enc music.wav music-wtr.wav 0xABCDEF –preset4
```

Corresponding decoding:

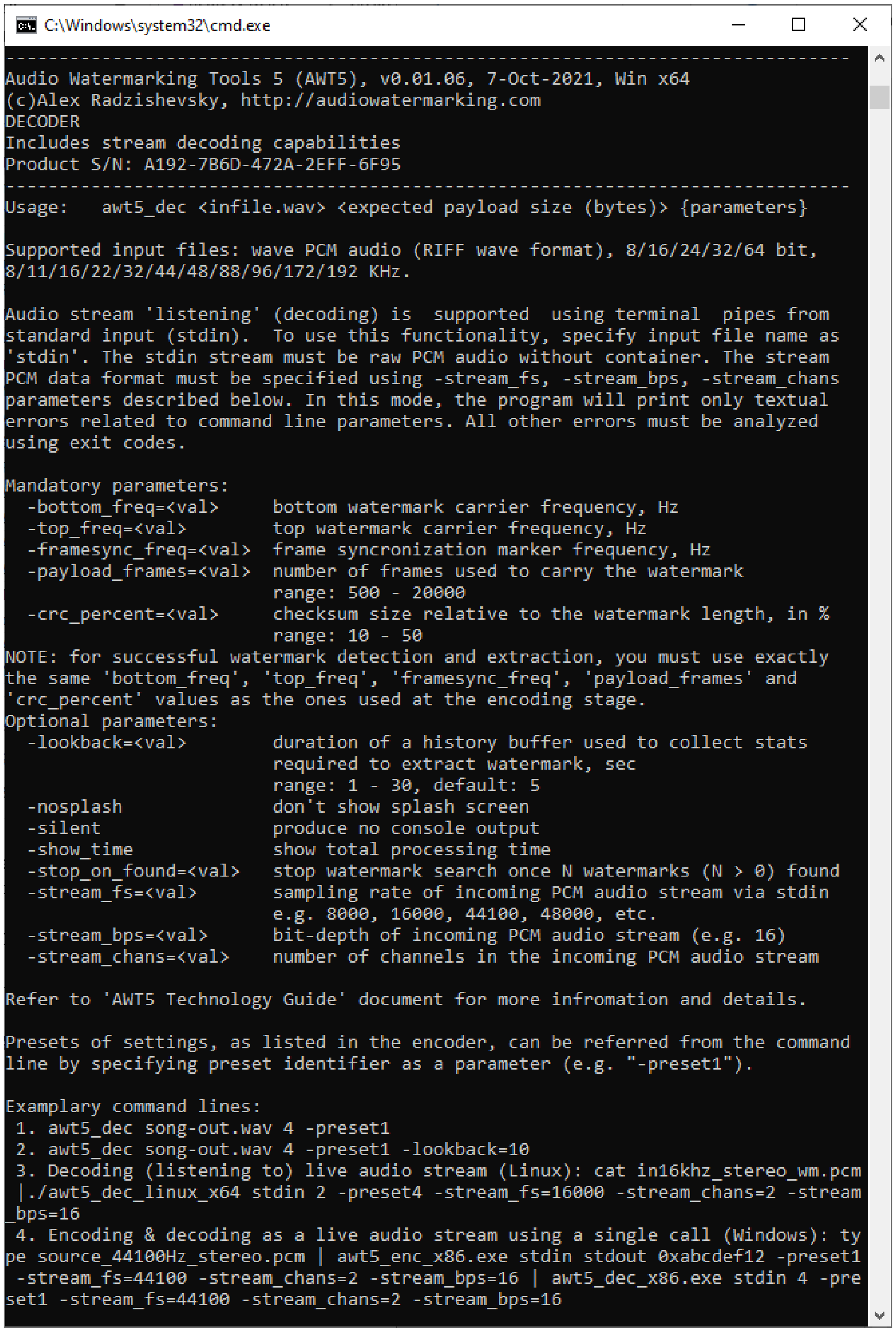
```
awt5_dec music-wtr.wav 3 -bottom_freq=1750 -top_freq=11000 -framesync_freq=2500 -payload_frames=1 -
crc_percent=20
```

or using the preset alias:

```
awt5_dec music-wtr.wav 3 –preset4
```

4 Decoder

The decoder extracts a watermark from an audio file or an audio stream. The usage screen is shown (below) when the decoder is invoked from the command line without parameters:



```
-------------------------------------------------------------------------------
Audio Watermarking Tools 5 (AWT5), v0.01.06, 7-Oct-2021, Win x64
(c)Alex Radzishevsky, http://audiowatermarking.com
DECODER
Includes stream decoding capabilities
Product S/N: A192-7B6D-472A-2EFF-6F95
-------------------------------------------------------------------------------
Usage:   awt5_dec <infile.wav> <expected payload size (bytes)> {parameters}

Supported input files: wave PCM audio (RIFF wave format), 8/16/24/32/64 bit,
8/11/16/22/32/44/48/88/96/172/192 KHz.

Audio stream 'listening' (decoding) is  supported  using terminal  pipes from
standard input (stdin).  To use this functionality, specify input file name as
'stdin'. The stdin stream must be raw PCM audio without container. The stream
PCM data format must be specified using -stream_fs, -stream_bps, -stream_chans
parameters described below. In this mode, the program will print only textual
errors related to command line parameters. All other errors must be analyzed
using exit codes.

Mandatory parameters:
  -bottom_freq=<val>     bottom watermark carrier frequency, Hz
  -top_freq=<val>        top watermark carrier frequency, Hz
  -framesync_freq=<val>  frame syncronization marker frequency, Hz
  -payload_frames=<val>  number of frames used to carry the watermark
                         range: 500 - 20000
  -crc_percent=<val>     checksum size relative to the watermark length, in %
                         range: 10 - 50
NOTE: for successful watermark detection and extraction, you must use exactly
the same 'bottom_freq', 'top_freq', 'framesync_freq', 'payload_frames' and
'crc_percent' values as the ones used at the encoding stage.
Optional parameters:
  -lookback=<val>        duration of a history buffer used to collect stats
                         required to extract watermark, sec
                         range: 1 - 30, default: 5
  -nosplash              don't show splash screen
  -silent                produce no console output
  -show_time             show total processing time
  -stop_on_found=<val>   stop watermark search once N watermarks (N > 0) found
  -stream_fs=<val>       sampling rate of incoming PCM audio stream via stdin
                         e.g. 8000, 16000, 44100, 48000, etc.
  -stream_bps=<val>      bit-depth of incoming PCM audio stream (e.g. 16)
  -stream_chans=<val>    number of channels in the incoming PCM audio stream

Refer to 'AWT5 Technology Guide' document for more infromation and details.

Presets of settings, as listed in the encoder, can be referred from the command
line by specifying preset identifier as a parameter (e.g. "-preset1").

Examplary command lines:
 1. awt5_dec song-out.wav 4 -preset1
 2. awt5_dec song-out.wav 4 -preset1 -lookback=10
 3. Decoding (listening to) live audio stream (Linux): cat in16khz_stereo_wm.pcm
 |./awt5_dec_linux_x64 stdin 2 -preset4 -stream_fs=16000 -stream_chans=2 -stream
_bps=16
 4. Encoding & decoding as a live audio stream using a single call (Windows): ty
pe source_44100Hz_stereo.pcm | awt5_enc_x86.exe stdin stdout 0xabcdef12 -preset1
 -stream_fs=44100 -stream_chans=2 -stream_bps=16 | awt5_dec_x86.exe stdin 4 -pre
set1 -stream_fs=44100 -stream_chans=2 -stream_bps=16
```

Figure 3. Decoder usage screen

IMPORTANT: To successfully detect and extract a watermark payload from an audio file/stream, you must specify the expected watermark payload size and precisely the same core algorithm settings as the ones used at the encoding stage. This includes settings for the bottom & top frequency of the carrier band (**-bottom_freq**, **-top_freq**), number of data frames (**-payload_frames**), frame-sync data band frequency (**-framesync_freq**) and size of checksum (**-crc_percent**).

The requirement to specify decoding settings identical to the ones used at the encoding stage is never an issue. You can choose a suitable payload form/size (refer to page 30) and watermarking parameters (or preset) for a specific application/use-case once, and then keep using the same settings to encode all your files/streams. For example, you can decide, once and for all, to use 3-bytes long watermark payload and some specific bottom & top frequency of the carrier band (**-bottom_freq**, **-top_freq**), number of data frames (**-payload_frames**), frame-sync data band frequency (**-framesync_freq**) and size of checksum (**-crc_percent**). This way, you will always know how the parameters for the decoder should be specified.

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

```
awt5_enc music.wav music-wtr.wav 0xABCD -bottom_freq=1750 -top_freq=7000 -framesync_freq=2500 -
payload_frames=2 -crc_percent=20 -aggressiveness=0.75
```

then you need to run the decoder as follows:

```
awt5_dec music-wtr.wav 2 -bottom_freq=1750 -top_freq=7000 -framesync_freq=2500 -payload_frames=2 -
crc_percent=20
```

Note: '2' tells the decoder that the expected watermark payload is 2 bytes long. The decoder does not require the aggressiveness setting.

Another example of encoding and corresponding decoding settings using a preset alias and 3-byte watermark payload:

```
awt5_enc music.wav music-wtr.wav 0x112233 –preset1
```

⇩

```
awt5_dec music-wtr.wav 3 –preset1
```

The decoder analyses the file/stream and prints out all watermarks found with corresponding time-stamps relative to the beginning of the file/stream. Example decoder output:

```
Input file: music-wtr.wav' [44100 Hz, 16 bit, 2 chan, 73 sec]
Processing...
[00:00:01.5] Found watermark: 0xABCDEF (0.51) @ chan 1;
[00:00:01.5] Found watermark: 0xABCDEF (0.51) @ chan 2;
[00:00:02.0] Found watermark: 0xABCDEF (0.66) @ chan 1;
[00:00:02.5] Found watermark: 0xABCDEF (0.74) @ chan 1;
[00:00:02.5] Found watermark: 0xABCDEF (0.51) @ chan 2;
[00:00:03.0] Found watermark: 0xABCDEF (0.82) @ chan 1;
```

```
[00:00:03.0] Found watermark: 0xABCDEF (0.77) @ chan 2;
[00:00:03.5] Found watermark: 0xABCDEF (0.85) @ chan 1;
[00:00:03.5] Found watermark: 0xABCDEF (0.77) @ chan 2;
[00:00:04.0] Found watermark: 0xABCDEF (0.89) @ chan 1;
[00:00:04.0] Found watermark: 0xABCDEF (0.86) @ chan 2;
[00:00:04.5] Found watermark: 0xABCDEF (0.91) @ chan 1;
[00:00:04.5] Found watermark: 0xABCDEF (0.86) @ chan 2;
[00:00:05.0] Found watermark: 0xABCDEF (0.93) @ chan 1;
[00:00:05.0] Found watermark: 0xABCDEF (0.91) @ chan 2;
[00:00:05.5] Found watermark: 0xABCDEF (0.94) @ chan 1;
[00:00:05.5] Found watermark: 0xABCDEF (0.91) @ chan 2;
[00:00:06.0] Found watermark: 0xABCDEF (0.94) @ chan 1;
[00:00:06.0] Found watermark: 0xABCDEF (0.92) @ chan 2;
[00:00:06.5] Found watermark: 0xABCDEF (0.94) @ chan 1;
[00:00:06.5] Found watermark: 0xABCDEF (0.92) @ chan 2;
[00:00:07.0] Found watermark: 0xABCDEF (0.96) @ chan 1;
[00:00:07.0] Found watermark: 0xABCDEF (0.95) @ chan 2;
[00:00:07.5] Found watermark: 0xABCDEF (0.94) @ chan 1;
[00:00:07.5] Found watermark: 0xABCDEF (0.95) @ chan 2;
[00:00:08.0] Found watermark: 0xABCDEF (0.94) @ chan 1;
[00:00:08.0] Found watermark: 0xABCDEF (0.95) @ chan 2;
[00:00:08.5] Found watermark: 0xABCDEF (0.94) @ chan 1;
[00:00:08.5] Found watermark: 0xABCDEF (0.95) @ chan 2;
[00:00:09.0] Found watermark: 0xABCDEF (0.96) @ chan 1;
[00:00:09.0] Found watermark: 0xABCDEF (0.97) @ chan 2;
[00:00:09.5] Found watermark: 0xABCDEF (0.94) @ chan 1;
[00:00:09.5] Found watermark: 0xABCDEF (0.95) @ chan 2;
[00:00:10.0] Found watermark: 0xABCDEF (0.95) @ chan 1;
[00:00:10.1] Found watermark: 0xABCDEF (0.96) @ chan 2;
```

Since the watermarking data rate provided by AWT5 is generally relatively high, especially with signals having good dynamics, it is usually enough to use only a small portion of the watermarked stream to detect and extract watermarks contained in it. Depending on signal quality, watermark extraction may take longer or shorter audio piece. If the audio stream is distorted significantly or in case the watermark payload is split into multiple data frames (**-payload_frames** setting larger than 1), you may need to use a longer portion of the audio by specifying appropriate '**-lookback=**' value for decoding because analysis of longer signals containing more copies of the watermark statistically improves extraction reliability.

The reliability is specified as a value between 0.0 and 1.0 in parenthesis succeeding the watermark. Values under 0.3 may indicate results that are not reliable enough. In such cases, it is recommended to take final decision on watermark extraction based on multiple succeeding watermarks, if returned watermark values are identical.

The decoder has additional parameters controlling its performance and output:

-stop_on_found switch forces to stop analysis (decoding) on the first found watermark. The switch can also be used as **-stop_on_found=N**, where N is integer > 0. In this case, the audio analysis stops after N found watermarks.

-lookback setting controls the size of an internal look-back buffer used to collect statistics during the audio scanning & analysis process. A larger buffer increases chances to extract watermarks even from heavily distorted recordings, but may lead to no watermark detection with audio fragments containing different watermark payloads in the same fragment.

–stream_fs specifies the sampling rate of the incoming PCM audio stream received from stdin. Acceptable values: 192000, 176400, 96000, 88200, 48000, 44100, 32000, 22050, 16000, 11025, 8000. The parameter should be used only in streaming mode (stdin-to-stdout processing).

–stream_bps specifies the bit-depth of the incoming PCM audio stream samples (bits per sample). Acceptable values are: 16. The parameter should be used only in streaming mode (stdin-to-stdout processing).

–stream_chans specifies the number of audio channels in the incoming PCM audio stream. Acceptable values: 1 - 32. The parameter should be used only in streaming mode (stdin-to-stdout processing).

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

-nosplash switch prevents showing program information on the screen

-show_time switch forces the encoder to show the processing time

4.1 Live audio stream scanning in command line

The decoder can “listen” to a live audio stream directly from command line / console. It can take PCM audio stream from the standard input (stdin) and send live watermark extraction report to the standard output (stdout). Data redirection is to be done using terminal pipes. In order to use this functionality, specify input file name as 'stdin'. The stdin stream must be raw PCM audio without container. The stream PCM data format must be specified using -stream_fs, -stream_bps and -stream_chans parameters described below.

Example below shows how to start live watermark detection and analysis from a simulated real-time audio stream in the Linux environment using pipes:

```
cat in16khz_stereo_wm.pcm |./awt5_dec_linux_x64 stdin 2 -preset1 -stream_fs=16000 -stream_chans=2 -
stream_bps=16
```

The same for Windows:

```
type in16khz_stereo_wm.pcm |./awt5_dec_linux_x64 stdin 2 -preset1 -stream_fs=16000 -stream_chans=2 -
stream_bps=16
```

Encoding and decoding in one call (Linux):

```
type source_44100Hz_stereo.pcm | awt5_enc_x86.exe stdin stdout 0xabcdef12 -preset1 -stream_fs=44100 -
stream_chans=2 -stream_bps=16 | awt5_dec_x86.exe stdin 4 -preset1 -stream_fs=44100 -stream_chans=2 -
stream_bps=16
```

Listening to microphone signal using ffmpeg and analyzing it as live stream in the real time (Linux):

```
ffmpeg -loglevel error -hide_banner -nostats -f alsa -i hw:0,0  -f s16le -acodec pcm_s16le -ar 48000 -ac 1 -
vn -sn - | ./awt5_dec_linux_x64 stdin 1 -preset1 -stream_fs=48000 -stream_chans=1 -stream_bps=16
```

(you may need to adjust the listening device ID specified by "hw:0,0" by listing the available devices using: `arecord -l` command).

Similarly, listening to microphone signal using 'arecord' and analyzing it as live stream in the real time (Linux):

```
arecord --quiet --file-type raw --channels=1 --rate=48000 --format=S16_LE | ./awt5_dec_linux_x64 stdin 1 -
preset1 -stream_fs=48000 -stream_chans=1 -stream_bps=16
```

5 AWT5 Software Development Kit (SDK)

AWT5 can be integrated into software applications using AWT5 SDK. AWT5 SDK consists of a C library file (static or shared), header file and a descriptive integration example. AWT5 SDK is provided upon request and can be compiled for virtually any OS or platform.

AWT5 API is straightforward. It provides watermark encoding and decoding functionality through low-latency buffer-by-buffer processing.

AWT5 header file listing is presented below. Comprehensive examples of AWT5 SDK usage are supplied as part of the SDK package.

```
// ========================================+===========
//
// AWT5 SDK header file
// (c) Alex Radzishevsky, 2007-2021
// http://audiowatermarking.com (.info)
//
// AWT5 0.01.05
// SDK update: 2021-09-22
//
// ===================================================


// ===================================================
//
// DO NOT EDIT THIS FILE!
//
// ===================================================


#if __cplusplus
extern "C" {
#endif

#ifndef AWT5_SDK_H
#define AWT5_SDK_H


    // Maximal printable size of watermark payload in a textual hex form (e.g. "0xABCDEF")
    // Supported maximum: "0xFFFFFFFFFFFFFF" (12 bytes) plus ending \0
    #define AWT5_MAX_PRINTABLE_PAYLOAD_BYTES    27



    // ============================================================================
    // ============================================================================
    // ============================================================================
    //
    // Audio stream encoding (watermarking)
    //
    // ============================================================================
    // ============================================================================
    // ============================================================================


    // AWT5 Encoder object declaration
    typedef struct _AWT5_ENC_OBJ AWT5_ENC_OBJ, * PAWT5_ENC_OBJ;


    // --------------------------------------------------------------------------
    // Allocates AWT5 stream encoder object and initializes it.
    // Returns (via arguments):
    //      - pointer on enc_obj_ptr (or NULL on error).
    //      - algorithmic latency - i.e. how many samples must be feed to the encoder input
    //        in order to see the first sample on the output
    //        (depends on sampling frequency)
    //      - size of one single-channel audio buffer expected by the encoding function
    //        (depends on sampling frequency)
    // Returns:
    //      0  -- on success,
    //      >0 -- on error (see AWT5 doc for error codes)
    // Notes:
    // * wm_payload is expected to end with \0 (nil)
    // * the object is for single or MULTIPLE channels of audio. The input audio samples
    //   of a multi-channel audio are expected to be deinterlieved.
    //   For multiple channels of audio, the encoder assumes the following input buffer
    //   data structure:
    //   [chan1-buf][chan2-buf] ... [chanN-buf]
    // --------------------------------------------------------------------------

    unsigned awt5_encode_stream_init(
        PAWT5_ENC_OBJ* enc_obj_ptr,     // OUT: pointer on AWT5 Encoder object
        unsigned* algorithmic_delay,    // OUT: algorithmic latency, samples
        unsigned* buffer_size,          // OUT: input/output buffer size, samples (each sample is PCM float -1.0 .. +1.0)
        unsigned fs,                    // IN: sampling frequency, Hz (e.g. 8000, 16000, 44100, etc.)
        unsigned chans,                 // IN: number of audio channels (1, 2, ...)
        char* hex_payload,              // IN: watermark payload in a textual hex form (e.g. "0xABCDEF"), must end with 0x
0 ("\0");
        unsigned bot_freq_hz,           // IN: bottom watermark carrier frequency, Hz (e.g. 2000)
        unsigned top_freq_hz,           // IN: top watermark carrier frequency, Hz (e.g. 7000)
        unsigned framesync_freq,        // IN: frame syncronization marker frequency, Hz (e.g. 3000)
        unsigned payload_frames,        // IN: number of frames used to carry the watermark (e.g. 1, 2)
        unsigned CRC_prcnt,             // IN: checksum size relative to the watermark length, in % (e.g. 20)
        float enc_aggr,                 // IN: encoding aggressiveness, higher = more robust & more audible,
                                        //     lower = less robust & more transparent (e.g. 0.75)
        unsigned emboss_gain,           // IN: emboss gain, dB (e.g. -96)
        unsigned emphasis_gain,         // IN: emphasis gain, dB (e.g. +6)
        char* license                   // IN: license obtained from AWT5 Licensing Server
                                        //     * Can be NULL for untied AWT5 versions
                                        //     * Used only with SaaS / Client-Server version of AWT5, the license is an
                                        //       array of characters
    );




    // --------------------------------------------------------------------------
    // Processes (watermarks) single audio buffer. The size of the audio buffer is assumed
    // as returned by the initialization function via 'buffer_size' (in samples)
    // multiplied by the number of audio channels (i.e. chans*buffer_size samples)
    // The PCM data is assumed to be in floats (-1.0 .. +1.0).
    // The input audio samples of a multi-channel audio are expected to be deinterlieved.
    // For multiple channels of audio, the encoder assumes the following input buffer
    // data structure:
    //   [chan1-buf][chan2-buf] ... [chanN-buf]
    // AWT5 stream encoder object must be initialized prior to this call.
    // Parameter descriptions are provided below;
    // Returns:
    //      0  -- on success,
    //      >0 -- on error (see AWT5 doc for error codes)
    // --------------------------------------------------------------------------

    int awt5_encode_stream_buffer(
        PAWT5_ENC_OBJ enc_obj,          // IN: pointer on initialized AWT5 Encoder object
        float* input_buffer,            // IN: input audio (each sample is PCM float -1.0 .. +1.0)
        float* output_buffer,           // OUT: output audio (each sample is PCM float -1.0 .. +1.0)
        unsigned* embedding_occurred    // OUT: flag, indicates whether watermark embedding occurred on the frame or not
    );
    // ---------------------------------------------------------------------------
    // Changes/sets watermark payload in already initialized encoder object.
    // Allows to change watermark payload on the fly during live stream watermarking
    // The watermark payload in a textual hex form (e.g. "0xABCDEF"), must end with 0x0 ("\0");
    // Parameter descriptions are provided below;
    // Returns:
    //      0  -- on success,
    //      >0 -- on error (see AWT5 doc for error codes)
    // ---------------------------------------------------------------------------

    unsigned awt5_encode_stream_set_payload(
        PAWT5_ENC_OBJ enc_obj,          // IN: pointer on initialized AWT5 Encoder object
        char* hex_payload,              // IN: watermark payload in a textual hex form (e.g. "0xABCDEF"), must end with 0x
0 ("\0");
        char* license                   // IN: license obtained from AWT5 Licensing Server
                                        //     * Can be NULL for untied AWT5 versions
                                        //     * Used only with SaaS / Client-Server version of AWT5, the license is an
                                        //       array of characters
    );


    // ---------------------------------------------------------------------------
    // Enables/disables processing (watermarking) bypass.
    // If bypass is disabled (bypass=0), awt5_encode_stream_buffer() call outputs watermarked
    // output (normal working state). If bypass is enabled (bypass=1), awt5_encode_stream_buffer()
    // call outputs not watermarked audio data.
    // Encoder object must be initialized.
    // Parameter descriptions are provided below;
    // Returns:
    //      0  -- on success,
    //      >0 -- on error (see AWT5 doc for error codes)
    // ---------------------------------------------------------------------------

    unsigned awt5_encode_stream_set_bypass(
        PAWT5_ENC_OBJ enc_obj,          // IN: pointer on initialized AWT5 Encoder object
        unsigned bypass                 // IN: 1 -- for bypass (no embedding), 0 -- for processing (embedding)
    );


    // ---------------------------------------------------------------------------
    // Kills AWT5 stream encoder object and frees all allocated memory
    // Returns: nothing
    // ---------------------------------------------------------------------------

    void awt5_encode_stream_kill(
        PAWT5_ENC_OBJ *enc_obj
    );




    // ============================================================================
    // ============================================================================
    // ============================================================================
    //
    // Audio stream decoding (watermark extraction)
    //
    // ============================================================================
    // ============================================================================
    // ============================================================================

    // AWT5 Stream Decoder object declaration
    typedef struct _AWT5_DEC_OBJ AWT5_DEC_OBJ, * PAWT5_DEC_OBJ;
    // ---------------------------------------------------------------------------
    // Initializes AWT5 Stream Decoder object.
    // Allocates memory for AWT5 Stream Decoder object which must be kept in
    // memory as long as the audio decoding function is used.
    // Intended to process a SINGLE channel audio. Multiple channels require mupltiple
    // decoder instances.
    // Parameter descriptions are provided below;
    // Returns:
    //      0  -- on success,
    //      >0 -- on error (see AWT5 doc for error codes)
    // ---------------------------------------------------------------------------

    unsigned awt5_decode_stream_init(
        PAWT5_DEC_OBJ* dec_obj_ptr,          // OUT: pointer on the decoder object (memory is allocated inside this functio
n)
        unsigned* buffer_size,               // OUT: input/output buffer size, samples (each sample is PCM float -
1.0 .. +1.0)
        unsigned wm_payload_length_bytes,    // IN: expected watermark payload length, bytes
        unsigned fs,                         // IN: sampling frequency, Hz (e.g. 8000, 16000, 44100, etc.)
        unsigned bot_freq_hz,                // IN: bottom watermark carrier frequency, Hz (e.g. 2000)
        unsigned top_freq_hz,                // IN: top watermark carrier frequency, Hz (e.g. 7000)
        unsigned framesync_freq,             // IN: frame syncronization marker frequency, Hz (e.g. 3000)
        unsigned payload_frames,             // IN: number of frames used to carry the watermark (e.g. 1, 2)
        unsigned CRC_prcnt,                  // IN: checksum size relative to the watermark length, in % (e.g. 20)
        unsigned lookback_sec,               // IN: duration of a history buffer used to collect stats required to extract
watermark, sec (e.g. 5)
        char* license                        // IN: license obtained from AWT5 Licensing Server
                                             //     * Can be NULL for untied AWT5 versions
                                             //     * Used only with SaaS / Client-
Server version of AWT5, the license is an
                                             //       array of characters
    );


    // ---------------------------------------------------------------------------
    // Performs processing of single audio buffer.
    // Performs watermark detection and extraction.
    // The size of the audio buffer is assumed as returned by the initialization
    // function via 'buffer_size' (in samples).
    // The PCM data is assumed to be in floats (-1.0 .. +1.0).
    // The input audio samples of a multi-channel audio must be processed using separate
    // decoder objects -- a dedicated decoder for each channel.
    // AWT5 decoder object must be initialized prior to this call.
    // Parameter descriptions are provided below;
    // Returns:
    //      0  -- on success,
    //      >0 -- on error (see AWT5 doc for error codes)
    // ---------------------------------------------------------------------------

    int awt5_decode_stream_buffer(
        PAWT5_DEC_OBJ dec_obj,               // IN: pointer on initialized AWT5 Decoder object
        float* input_buffer,                 // IN: input PCM audio (each sample is PCM float -1.0 .. +1.0)
        char* watermark,                     // OUT: detected watermark (hex text e.g. "0xABCDFEF12"), the memory must be p
reallocated externally;
                                             //      NULL if nothing is detected
        float* reliability                   // OUT: reliability of the detected watermark (0.0 - 1.0); mostly informationa
l field
    );


    // ---------------------------------------------------------------------------
    // Kills AWT5 Stream Decoder object and frees all memory
    // Returns no errors
    // ---------------------------------------------------------------------------

    void awt5_decode_stream_kill(
        PAWT5_DEC_OBJ *dec_obj                // IN: pointer on initialized AWT5 Decoder object
    );


    // ===========================================================================
    // ===========================================================================
    // ===========================================================================
    //
    // AWT5 SaaS/Client-Server tied version
    //
    // ===========================================================================
    // ===========================================================================
    // ===========================================================================




    // --------------------------------------------------------------------------
    // ONLY FOR SAAS/Client-Server version of AWT5.
    // Generates a "license request key" that is to be sent to the AWT Licensing
    // Server to obtain license key for AWT5 Encoder or Decoder.
    // Expects duration of audio to be processed (in seconds). The AWT5 processing
    // tool will refuse processing audio longer than requested.
    // Returns a string with the license request key.
    // Note: free()-ing the memory is on user
    // --------------------------------------------------------------------------

    char* awt5_license_request(unsigned duration_sec);




    // ===========================================================================
    // ===========================================================================
    // ===========================================================================
    //
    // Informational calls
    //
    // ===========================================================================
    // ===========================================================================
    // ===========================================================================


    // --------------------------------------------------------------------------
    // Returns AWT5 Serial Number information (text)
    // --------------------------------------------------------------------------
    void awt5_sn (char **serial_number);



    // --------------------------------------------------------------------------
    // Prints AWT5 error text into stdout
    // --------------------------------------------------------------------------
    void awt5_print_err_text(int err);




#endif

#if __cplusplus
}   // Extern C
#endif
```

6 Choosing optimal watermark payload form

AWT5 operates with a watermark payload in the hexadecimal form (e.g. 0xABCDEF12). Therefore, any information that you want to include into your watermark must be represented as byte data 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 an ASCII table of printable symbols in the Appendix: ASCII printable characters (page 42). On the other hand, the standard ASCII representation of the text symbols is not compact. AWT5 package contains a small console utility called ‘text2hex’, which can convert text into hexadecimal form (and back) using more compact representations (refer to page 32).

Choosing the 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 embed more watermark copies into a 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 a heavily distorted piece of audio.

Additionally, the chosen watermark payload form should be of fixed length. This is because for decoding, you need to specify the 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 only you -- the owner of a specific AWT5 copy having a unique Serial Number -- are able to detect/decode your watermarks; therefore, you should not worry too much about the 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”.

In light of the aspects mentioned above, it is better to keep your watermark payload as short as possible while still allowing to recognize 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 the 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, it is very informative as it contains a well recognizable abbreviation of the company name and the date.

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 a 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 billion 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 the numerical userID of the user with decimal number 12346.

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

Using ASCII codes to encode the text, in this case, is not the best idea because one text symbol will consume 8 bits (one byte). You can use a representation that is much more compact by enumerating text symbols in your 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 AWT5 package and is described in the next section.

6.1 text2hex utility

text2hex is a console application that comes as a part of AWT5 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 includes all digits and all letters as well as special characters. Using 6-bit representation saves 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 a 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 AWT5 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 AWT5 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

This section offers some useful tips, recommendations, and ready solutions for applications utilizing AWT5.

7.1 Watermarking audio track of a video file

To embed watermarked audio track into a video file, you need to extract the original audio track from the video file first, then encode the extracted audio track with AWT5 and then merge back the watermarked audio track with the video track.

To extract an audio track from a 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 AWT5.

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 latter with FAAC encoder and then merge audio and video into the output file 'out.m4v'.

7.2 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 (
        awt5_enc "%%~nxX" "outputs\%%~nxX" 0x12 -preset1
)
```

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

Place 'awt5_enc.exe' (AWT5 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.

7.3 Encoding (watermarking) live audio stream

Encoding of live audio streams using command-line encoder tool is explained in section 3.4.

7.4 Analyzing (extracting watermarks from) live audio stream

Watermark extraction from live audio streams using the command line decoder tool is explained in section 4.1.

8 Licensing terms

AWT5 licensing information is available upon request.

9 FAQ

Q: Can special "watermarking attacks" compromise the watermark robustness of AWT5? In other words, are there methods able to destroy AWT5 watermarks?

A: Yes, of course. Damaging watermarks is always possible by developing a special watermarking attack targeted at a specific watermarking technique. Therefore, the watermarking technique used in AWT5 is not attack agnostic too. However, the author believes that the proposed method is strong enough to survive most standard watermarking attacks.

Q: Does the use of this watermarking solution compromise the performance of automatic audio recognition services (such as TrackID, Echonest, etc.) being applied to the watermarked files?

A: The answer depends on every particular audio recognition technology. Since the AWT5 watermark is practically transparent for the human listener, it should be transparent for any "good" audio recognition engine as well. Therefore, AWT5 watermarking should not harm recognition results of the most popular audio recognition technologies, at least theoretically.

Q: Can one AWT5 user extract watermarks created by another AWT5 user?

A: No. Each particular copy of AWT5 binaries with a specific Serial Number (SN) contains a unique numeric identifier used during encoding to scramble watermark payload. This security feature prevents one AWT5 user (with one SN) from extracting watermarks from watermarked files created by another AWT5 user (with another SN number).

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

A: Yes, of course. Since the watermarking rate provided by AWT5 is relatively high, it is generally enough to use only a small portion of the watermarked stream to extract the watermark from it.

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. AWT5 encoder operates with wave files only and is unable to embed watermarks directly into the MP3 bitstream. Therefore, the only way to watermark MP3 files is to decode the MP3 file into a PCM wave file, watermark the obtained wave file, and encode the watermarked wave file back to MP3.

Q: Why extracted watermark reliability is not 1.0 when decoding from the encoded file directly?

A: Because watermarks are embedded by modifying the aural information of the carrier audio signal. Therefore, watermark extraction reliability depends on two main factors: degree of quality degradation of the analyzed audio stream (distortions, noises, etc.) relative to the original watermarked stream and the "aural relevance" of the carrier audio stream. Obviously, embedding a watermark into totally silent audio results in zero reliability because there is no acoustic information in silence. Therefore, the reliability depends on the contents of the carrier audio. Fast-changing, rich sound is a better carrier than a smooth sound with silent parts. For this reason, some signals provide almost 100% reliability when extracting from the encoded file directly, while others may result in lower reliability values.

Q: Can I use AWT5 to detect watermarked content in broadcast recordings?

A: Yes. AWT5 decoder allows analyzing long audio recordings and also live streams.

10 Contact information

You can contact AWT5 author via e-mail: mailbox@audiowatermarking.com

Website: www.audiowatermarking.com

11 Appendix: AWT5 error codes

ERROR (1): could not allocate memory
ERROR (2): input file (or stream) cannot be read or does not exist
ERROR (3): unrecognized command line parameter; run the program without parameters for the exact syntax
ERROR (4): specified watermark payload size exceeds limitations of this program copy
ERROR (8): input wave data should be sampled at 192000, 176400, 96000, 88200, 48000, 44100, 32000, 22050, 16000, 11025, 8000 Hz
ERROR (9): the audio stream is too short
ERROR (10): could not write the output file
ERROR (11): sample data bit-resolution is not supported
ERROR (12): uninitialized AWT5 encoder object
ERROR (15): expected payload size should be specified in bytes
ERROR (16): specified payload size exceeds supported range
ERROR (21): specified hexadecimal watermark payload is invalid
ERROR (22): source and watermarked files cannot be the same
ERROR (24): watermark payload should be specified; should be of integer number of bytes
ERROR (27): input file should be standard RIFF wave PCM
ERROR (28): another instance of this program is active, multi-instance operation is not allowed
ERROR (43): skip_first/skip_last cannot be used in combination with compound payload; re-format your compound payload using 'void' payloads or use simple payload in combination with skip_first/skip_last
ERROR (44): wrong parameter value
ERROR (48): too many watermark payloads are specified in the compound payload
ERROR (49): input file is too large for this program build
ERROR (57): stream encoding functionality is not supported in this encoder version
ERROR (58): unsupported number of audio channels
ERROR (59): can only encode from stdin to stdout, can't output into file
ERROR (60): compound watermark payloads are not supported by the streaming CLI encoder tool
ERROR (67): skip_first/skip_last settings are not supported by the streaming CLI encoder tool
ERROR (68): streaming-related CLI parameters can be used only with 'stdin' type of input
ERROR (70): stream decoding functionality is not supported by this decoder version
ERROR (71): specified digital license key validity time is wrong
ERROR (72): specified digital license key has expired
ERROR (73): proper digital license key must be supplied
ERROR (74): cannot analyze audio -- the processed audio duration exceeds the licensed duration
ERROR (75): cannot encode audio -- the encoded audio duration exceeds the licensed duration; encoder switched to bypass
ERROR (76): this version of encoder does not support compound payload
ERROR (77): cannot process -- supplied license key does not correspond to the file requested
ERROR (78): this program version cannot operate with multi-channel audio
ERROR (79): lock-file not found, program terminated
ERROR (100): unexpected error occurred
ERROR (110): uninitialized encoder config
ERROR (114): unexpected error occured: Frame size must be integer of AWT5 buffer size
ERROR (510): `bottom_freq` value is not specified or is out of range
ERROR (511): `top_freq` value is not specified or is out of range

ERROR (512): `top_freq` value must be larger than `bottom_freq` value
ERROR (513): `framesync_freq` value must be larger than `bottom_freq` and lower than `top_freq`
ERROR (514): `payload_frames` value is not specified or is out of range
ERROR (515): `crc_percent` value is not specified or is out of range
ERROR (516): `aggressiveness` value is not specified or is out of range
ERROR (517): `lookback` value is not specified or is out of range
ERROR (518): `emboss_gain` value is not specified or is out of range
ERROR (519): `emphasis_gain` value is not specified or is out of range
ERROR (521): can`t operate with this combination of payload length, CRC percent, number of payload frames -- too narrow bands; consider shortening the watermark payload, CRC or increase the number of payload frames ('payload_frames')
ERROR (522): framesync data does not fit the specified frequency range, modify the range or the `framesync_freq` value
ERROR (523): the specified frequency range of the watermark carrier is too wide
ERROR (524): the specified frequency range of the watermark carrier cannot be used at this sampling frequency
ERROR (541): can`t operate with this combination of parameter values
ERROR (542): the specified frequency range of the watermark carrier is too narrow for the given watermark payload data density, either increase the frequency range, or shorten the payload, or increase the payload frame span (`payload_frames`)
ERROR (543): INTERNAL ERROR -- illegal attack test settings
ERROR (544): the specified CRC length is too large for this watermark payload length, shorten the CRC
ERROR (551): can't change payload size in initialized encoder object, re-init the object first
ERROR (552): INTERNAL ERROR -- uninit encoder
ERROR (553): INTERNAL ERROR -- uninit encoder
ERROR (700): unexpected error
ERROR (701): unexpected error

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				