Audio Watermarking

The **audiowmark** program can add watermarks to audio files and extract previously embedded watermarks from audio material. The usage is as follows:

```
usage: audiowmark <command> [ <args>... ]

Commands:
  * create a watermarked wav file with a message
    audiowmark add <input_wav> <watermarked_wav> <message_hex>

  * retrieve message
    audiowmark get <watermarked_wav>

  * compare watermark message with expected message
    audiowmark cmp <watermarked_wav> <message_hex>

  * generate 128-bit watermarking key, to be used with --key option
    audiowmark gen-key <key_file> [ --name <key_name> ]

Global options:
  -q, --quiet             disable information messages
  --strict                treat (minor) problems as errors

Options for get / cmp:
  --detect-speed          detect and correct replay speed difference
  --detect-speed-patient  slower, more accurate speed detection
  --json <file>           write JSON results into file

Options for add / get / cmp:
  --key <file>            load watermarking key from file
  --short <bits>          enable short payload mode
  --strength <s>          set watermark strength              [10]

  --input-format raw      use raw stream as input
  --output-format raw     use raw stream as output
  --format raw            use raw stream as input and output

The options to set the raw stream parameters (such as --raw-rate
or --raw-channels) are documented in the README file.

HLS command help can be displayed using --help-hls
```

Audiowmark Architecture

The **audiowmark** program is used to integrate (`add` command) and extract (`get` command) watermarks (messages of up to 128 bits) into/from audio files.

Internally, the program is organized as nested components, the outermost deals with file IO and command processing. The commands are implemented via various components that process the watermark, audio signal, an optional encoding key and user facing information.

Adding Watermarks

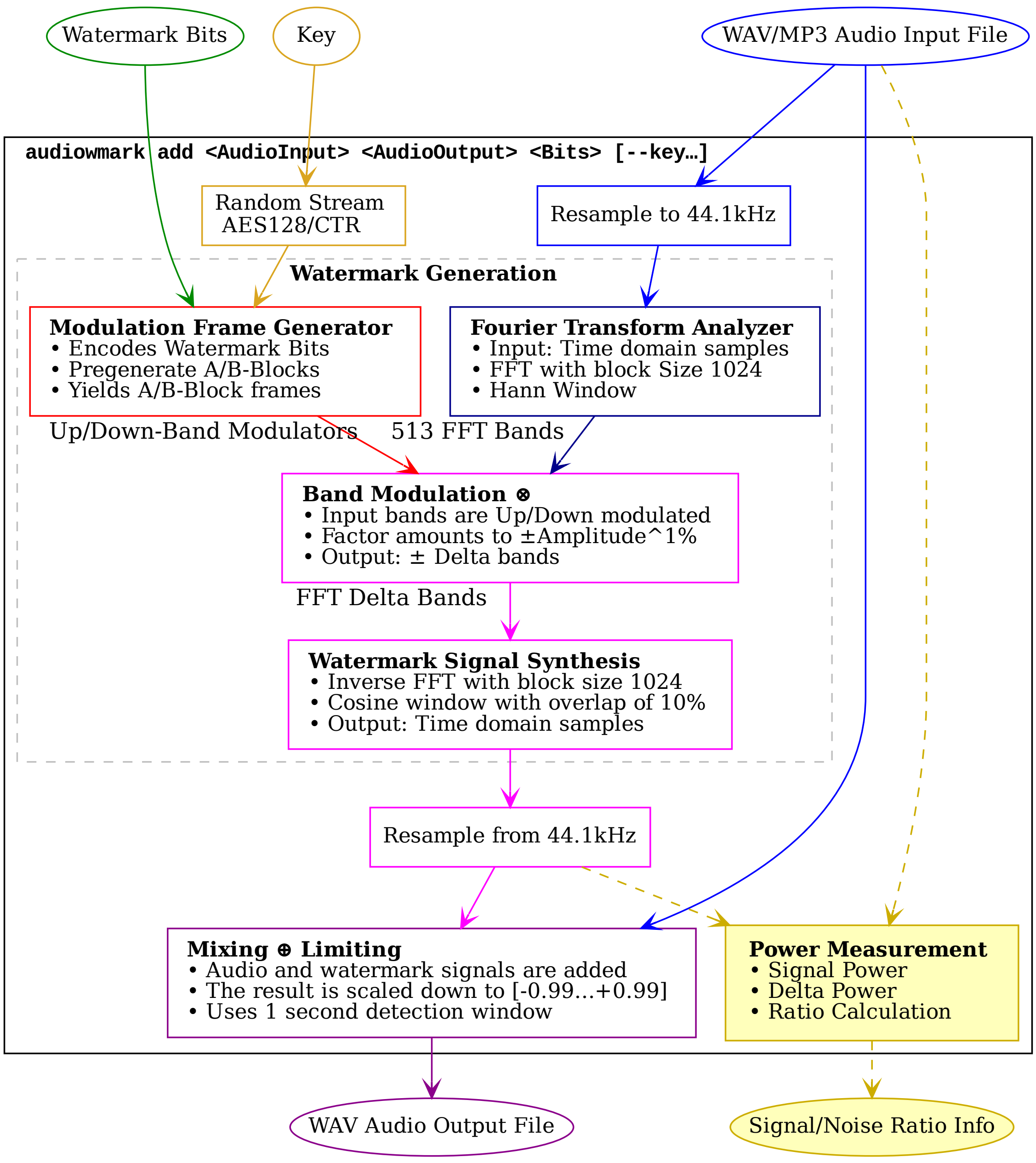
The `audiowmark add <in> <out> <bits> [--key...]` command allows adding watermarks to audio files. This command takes an audio file, a 128-bit hexadecimal watermark and an optional key as input, it combines these into a newly generated WAV file. Using the same key, the watermark bits can later be re-retrieved with the `audiowmark get` command without requiring access to the original audio input (this is called blind decoding).

By using the encoding key as input, various AES based random number streams are generated to shuffle, interleave and mix the watermark information into the audio signal.

For robust extraction and forward error correction, the watermark is encoded via convolutional codes with an order of `15` and a rate of `1/6` (similar to the communication of the Mars Pathfinder).

The expanded watermark bits are transformed into a delta spectrum at a sample rate of 44100Hz and distributed across various segments (of ca 23 millisecond lengths) of the audio signal and spread across bands above 800Hz and below 5000Hz. Based on the delta spectrum, the watermark signal can be modulated and adapted to the current segment of the input signal before the two are mixed together. To avoid clipping of the output signal, the final output stage consists of a time local limiter with ca 1 second window.

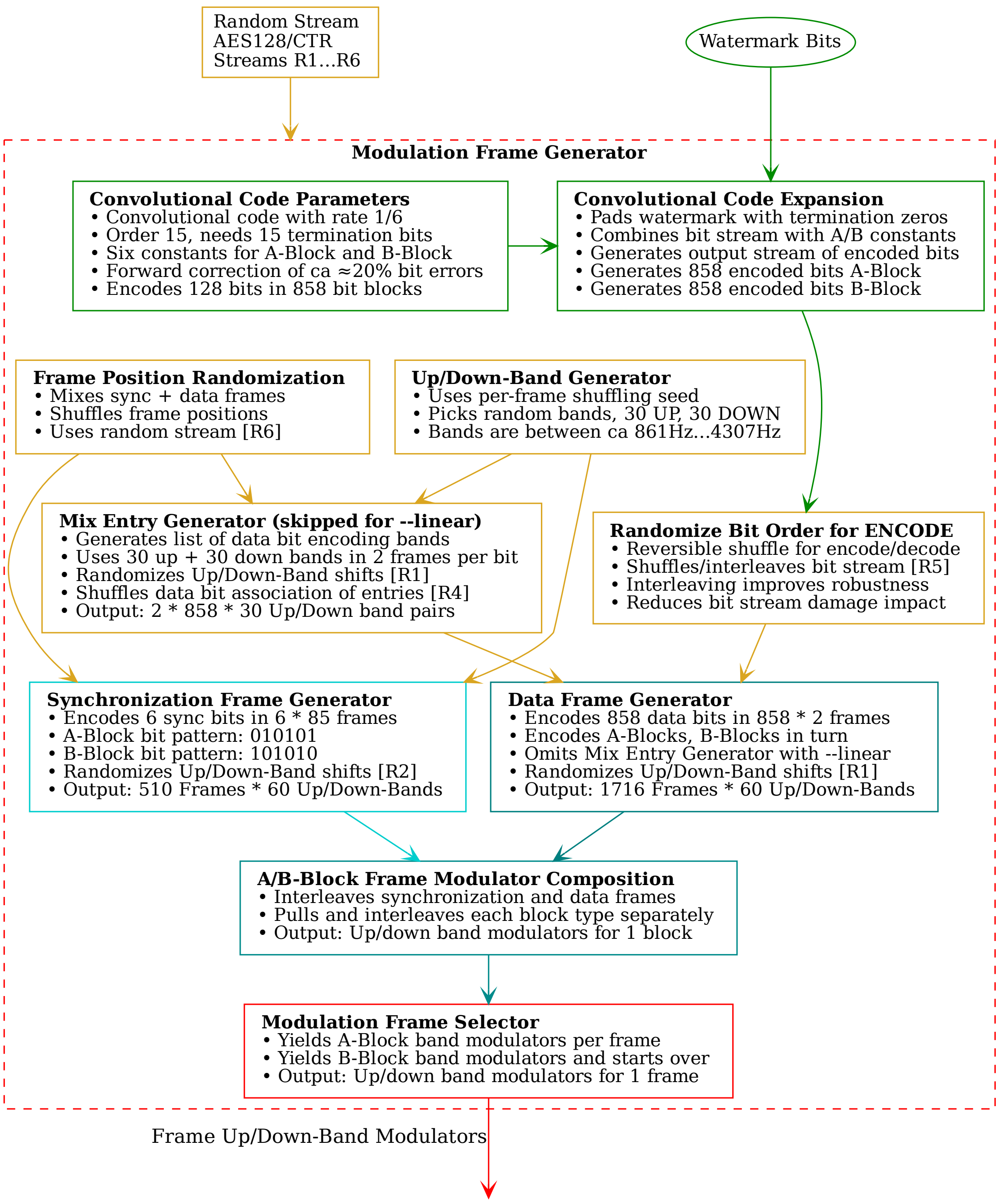
An outline of the component interactions to integrate the watermark information via delta band spectrum into the audio signal is provided in the following chart.


Watermark Bits
Key
WAV/MP3 Audio Input File
audiowmark add <AudioInput> <AudioOutput> <Bits> [--key…]
Random Stream AES128/CTR
Resample to 44.1kHz
Watermark Generation
Modulation Frame Generator
• Encodes Watermark Bits
• Pregenerate A/B-Blocks
• Yields A/B-Block frames
Fourier Transform Analyzer
• Input: Time domain samples
• FFT with block Size 1024
• Hann Window
Up/Down-Band Modulators
513 FFT Bands
Band Modulation ⊗
• Input bands are Up/Down modulated
• Factor amounts to ±Amplitude^1%
• Output: ± Delta bands
FFT Delta Bands
Watermark Signal Synthesis
• Inverse FFT with block size 1024
• Cosine window with overlap of 10%
• Output: Time domain samples
Resample from 44.1kHz
Mixing ⊕ Limiting
• Audio and watermark signals are added
• The result is scaled down to [-0.99...+0.99]
• Uses 1 second detection window
Power Measurement
• Signal Power
• Delta Power
• Ratio Calculation
WAV Audio Output File
Signal/Noise Ratio Info

At a sample frequency of 44100Hz, the audio signal used for the watermark creation is split into "Frames" of 1024 samples each, which corresponds to segments of ca 23 millisecond length. These frames are transformed from the time domain (samples) into the frequency domain (spectral bands) and vice versa to apply the watermark embedding in certain spectral bands.

Data and synchronization bits are encoded across several frames, with different levels of redundancy. In the "Modulation Frame Generator" the number of frames that compose all encoded information needed to find and extract the watermark bits are combined into two types of "Blocks".

A detailed chart of the component interactions for the Frame and Block generation in the "Modulation Frame Generator" is provided in the next chart.


Random Stream
AES128/CTR
Streams R1...R6
Watermark Bits
Modulation Frame Generator
Convolutional Code Parameters
• Convolutional code with rate 1/6
• Order 15, needs 15 termination bits
• Six constants for A-Block and B-Block
• Forward correction of ca ≈20% bit errors
• Encodes 128 bits in 858 bit blocks
Convolutional Code Expansion
• Pads watermark with termination zeros
• Combines bit stream with A/B constants
• Generates output stream of encoded bits
• Generates 858 encoded bits A-Block
• Generates 858 encoded bits B-Block
Frame Position Randomization
• Mixes sync + data frames
• Shuffles frame positions
• Uses random stream [R6]
Up/Down-Band Generator
• Uses per-frame shuffling seed
• Picks random bands, 30 UP, 30 DOWN
• Bands are between ca 861Hz...4307Hz
Mix Entry Generator (skipped for --linear)
• Generates list of data bit encoding bands
• Uses 30 up + 30 down bands in 2 frames per bit
• Randomizes Up/Down-Band shifts [R1]
• Shuffles data bit association of entries [R4]
• Output: 2 * 858 * 30 Up/Down band pairs
Randomize Bit Order for ENCODE
• Reversible shuffle for encode/decode
• Shuffles/interleaves bit stream [R5]
• Interleaving improves robustness
• Reduces bit stream damage impact
Synchronization Frame Generator
• Encodes 6 sync bits in 6 * 85 frames
• A-Block bit pattern: 010101
• B-Block bit pattern: 101010
• Randomizes Up/Down-Band shifts [R2]
• Output: 510 Frames * 60 Up/Down-Bands
Data Frame Generator
• Encodes 858 data bits in 858 * 2 frames
• Encodes A-Blocks, B-Blocks in turn
• Omits Mix Entry Generator with --linear
• Randomizes Up/Down-Band shifts [R1]
• Output: 1716 Frames * 60 Up/Down-Bands
A/B-Block Frame Modulator Composition
• Interleaves synchronization and data frames
• Pulls and interleaves each block type separately
• Output: Up/down band modulators for 1 block
Modulation Frame Selector
• Yields A-Block band modulators per frame
• Yields B-Block band modulators and starts over
• Output: Up/down band modulators for 1 frame
Frame Up/Down-Band Modulators

The watermark is encoded and embedded into the audio signal in two block types, A-Blocks and B-Blocks. The information contained in each block alone is usually sufficient to extract the watermark. However, in case of very distorted and noisy transmissions where watermark extraction from either block type fails, a combination of segments with A-Block and B-Block data may still lead to successful recovery of the original watermark.

In order to support watermark extraction from clipped excerpts of the input stream, a fixed pattern of synchronization bits is integrated into the data blocks with much higher redundancy than the data bits. The fixed pattern allows detection of the location of A-Blocks and B-Blocks as such to aid the watermark extraction.

The user provided encoding `Key` seeds an AES based pseudo random number generator in Counter Mode that is used to determine encoding places, randomize the noise introduced by the watermark and to interleave encoding for robustness. Without the key, the watermark information cannot be retrieved. Using a key is important because the implementation itself is open source, and being able to read the watermark message bits would allow an attacker to remove the watermark without degrading the audio quality.

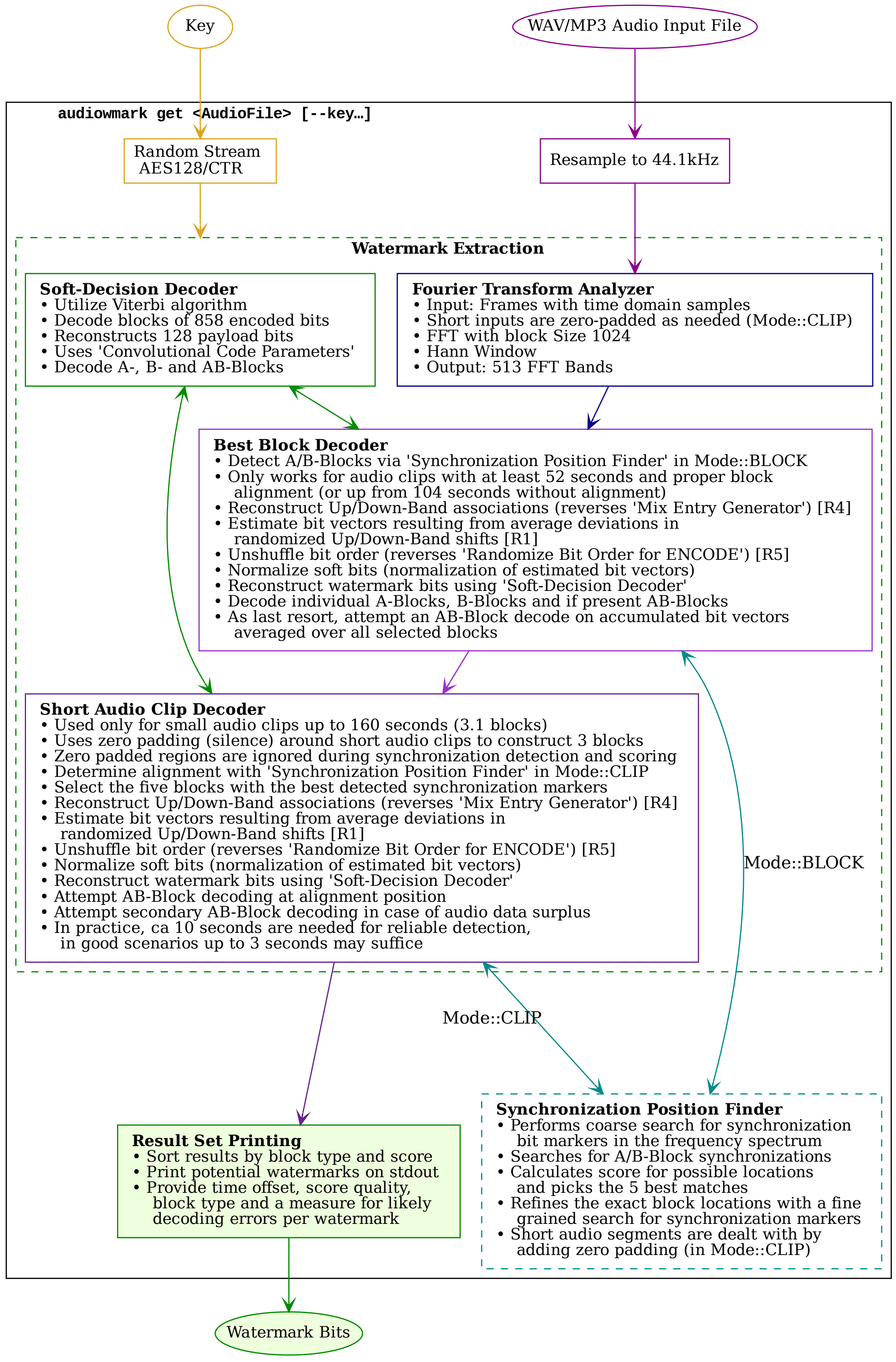
The different types of random data streams used for the distribution of the embedded watermark information are as follows:

- R1 - Used to randomizes Up/Down band shifts for watermark data bits.
- R2 - Used to randomizes Up/Down band shifts for watermark synchronization bits.
- R3 - Currently unused.
- R4 - Used to mix (shuffle) data bit associations of Up/Down bands distributed across several frames.
- R5 - Used to shuffle (interleave) the bit stream. Due to redundancy in the generated bit stream, interleaving reduces the number of adversely affected bits by bursts (holes) in transmission loss.
- R6 - Used to randomize and mix data frames with synchronization frames, this makes synchronization frames unlikely to be detectable without the encoded key.

Extracting Watermarks

The `audiowmark get <watermarked_wav> [--key...]` command extracts a watermark from an audio file. This command takes an audio file and an optional key as input. With the same key used during watermark embedding, synchronization bits are determined and searched for in the audio content. If synchronization bit matches are detected, encoded watermark information can be located, extracted and decrypted with error correction. The retrieval does not require access to the original audio input (this is called blind decoding). The detection results are produced on *stdout* with accompanying information about the location, match quality and a measure for likely decoding errors.

An outline of the component interactions to locate and extract the watermark information from the frequency spectrum in the audio signal is provided in the following charts.


Key
WAV/MP3 Audio Input File
audiowmark get <AudioFile> [--key…]
Random Stream AES128/CTR
Resample to 44.1kHz
Watermark Extraction
Soft-Decision Decoder
• Utilize Viterbi algorithm
• Decode blocks of 858 encoded bits
• Reconstructs 128 payload bits
• Uses 'Convolutional Code Parameters'
• Decode A-, B- and AB-Blocks
Fourier Transform Analyzer
• Input: Frames with time domain samples
• Short inputs are zero-padded as needed (Mode::CLIP)
• FFT with block Size 1024
• Hann Window
• Output: 513 FFT Bands
Best Block Decoder
• Detect A/B-Blocks via 'Synchronization Position Finder' in Mode::BLOCK
• Only works for audio clips with at least 52 seconds and proper block alignment (or up from 104 seconds without alignment)
• Reconstruct Up/Down-Band associations (reverses 'Mix Entry Generator') [R4]
• Estimate bit vectors resulting from average deviations in randomized Up/Down-Band shifts [R1]
• Unshuffle bit order (reverses 'Randomize Bit Order for ENCODE') [R5]
• Normalize soft bits (normalization of estimated bit vectors)
• Reconstruct watermark bits using 'Soft-Decision Decoder'
• Decode individual A-Blocks, B-Blocks and if present AB-Blocks
• As last resort, attempt an AB-Block decode on accumulated bit vectors averaged over all selected blocks
Short Audio Clip Decoder
• Used only for small audio clips up to 160 seconds (3.1 blocks)
• Uses zero padding (silence) around short audio clips to construct 3 blocks
• Zero padded regions are ignored during synchronization detection and scoring
• Determine alignment with 'Synchronization Position Finder' in Mode::CLIP
• Select the five blocks with the best detected synchronization markers
• Reconstruct Up/Down-Band associations (reverses 'Mix Entry Generator') [R4]
• Estimate bit vectors resulting from average deviations in randomized Up/Down-Band shifts [R1]
• Unshuffle bit order (reverses 'Randomize Bit Order for ENCODE') [R5]
• Normalize soft bits (normalization of estimated bit vectors)
• Reconstruct watermark bits using 'Soft-Decision Decoder'
• Attempt AB-Block decoding at alignment position
• Attempt secondary AB-Block decoding in case of audio data surplus
• In practice, ca 10 seconds are needed for reliable detection, in good scenarios up to 3 seconds may suffice
Mode::BLOCK
Mode::CLIP
Result Set Printing
• Sort results by block type and score
• Print potential watermarks on stdout
• Provide time offset, score quality, block type and a measure for likely decoding errors per watermark
Synchronization Position Finder
• Performs coarse search for synchronization bit markers in the frequency spectrum
• Searches for A/B-Block synchronizations
• Calculates score for possible locations and picks the 5 best matches
• Refines the exact block locations with a fine grained search for synchronization markers
• Short audio segments are dealt with by adding zero padding (in Mode::CLIP)
Watermark Bits

At a sample frequency of 44100Hz, a spectral analysis is performed on the audio signal and the spectrum is then searched for known synchronization markers. Upon detection of A/B-Block synchronization positions, watermark bits are extracted from known data bit locations, while making use of the embedded redundancy to make the detection more robust. Due to high redundancy and wide spread of watermark information, bits often can still be extracted from audio clips that are heavily shortened. To employ the full detection machinery to very short clips, symmetric zero padding is used to provide enough input samples (zero padded regions are ignored during scoring however).

Since detection success is directly dependent on the precise bit stream synchronization, an iterative process is used for fast approximation of synchronization locations with later refinements to yield precise results.

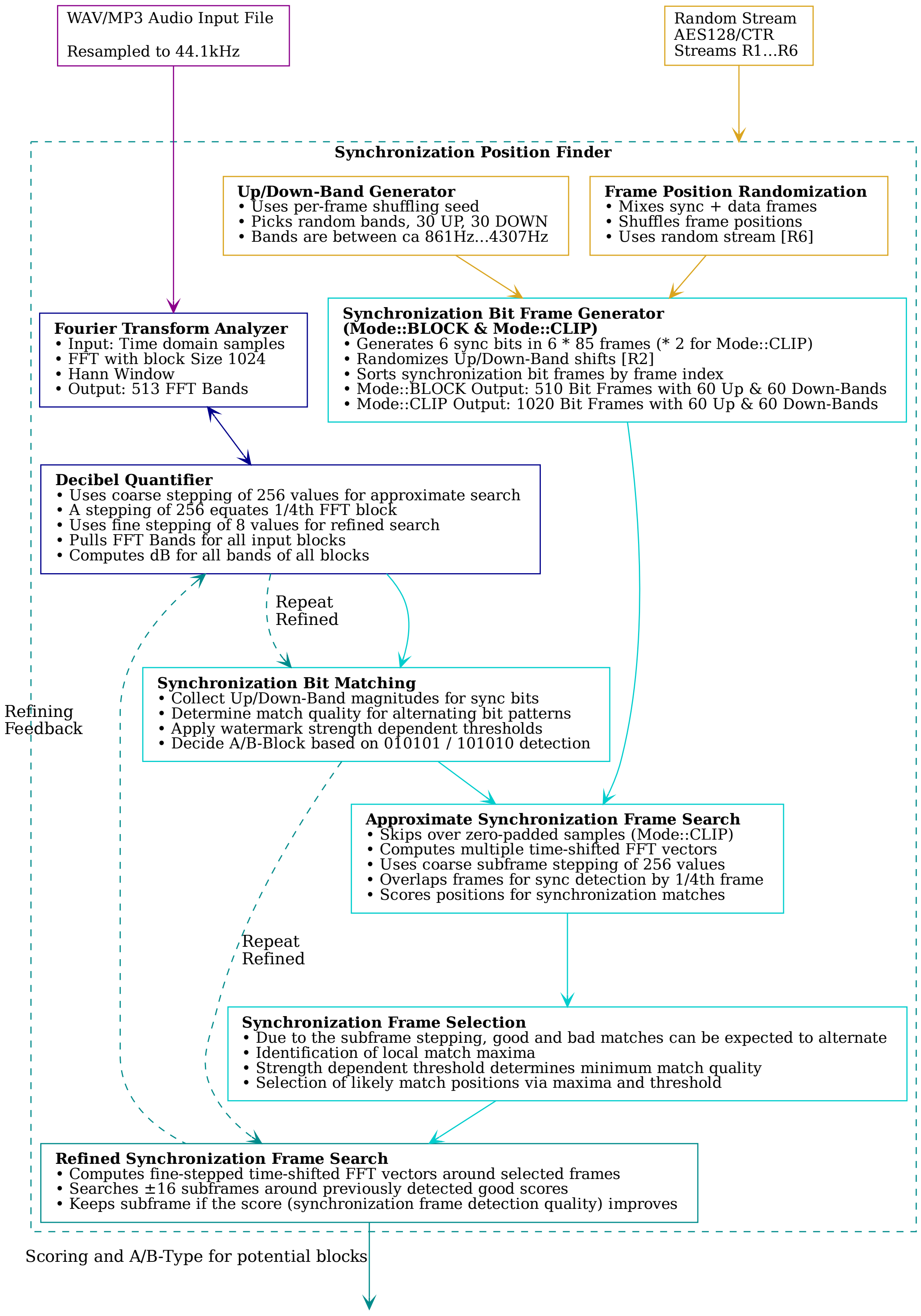
The purpose of the synchronization algorithm is to find the location of the watermark A/B blocks in the input signal. This is important because the signal may have been cropped so that the location of the blocks is not known. To be able to find the locations of the blocks, while adding the watermark, some sync bits are added to each block with relatively high redundancy. The values of these sync bits are known, for an A block they are 010101, for a B block they are 101010. The up- and down-bands used for the sync bits and offsets of all frames that belong to sync bits inside the A / B block are known and determined by the key.

To perform the actual synchronization and locate the start of an A (or B) block, two steps are performed.

- As a first step, the synchronization algorithm tests all possible positions for the start of an A (or B) block using a step size of 256 (1/4 frame size) and tries to decode the sync bits at the expected locations relative to the start of the block. Since the values and locations of the bits are known, a sync score can be computed that indicates how good the bits in the actual audio input at this position match the expected bit sequence.
- For all start locations with a significantly high sync score, in a second step the actual start position is searched by trying all different start locations near to the original match with a smaller finer step size. Again a sync score can be computed and compared to a second threshold to decide whether this is location is really likely to contain a data block. If the match is good enough the start location will be used to decode the data bits in the block.

Besides using this strategy to find "whole" data blocks, there is also a variant of the synchronization algorithm that is used if the audio signal is very short. It can find the location of the watermark even if the length of the input signal is too short to contain a complete data block. To be able to do this, the input signal is zero padded before sync detection and then the usual algorithm to find whole blocks is used.

The following chart provides the detail of the steps involved in determining the synchronization locations.


WAV/MP3 Audio Input File
Resampled to 44.1kHz
Random Stream AES128/CTR Streams R1...R6
Synchronization Position Finder
Up/Down-Band Generator
• Uses per-frame shuffling seed
• Picks random bands, 30 UP, 30 DOWN
• Bands are between ca 861Hz...4307Hz
Frame Position Randomization
• Mixes sync + data frames
• Shuffles frame positions
• Uses random stream [R6]
Fourier Transform Analyzer
• Input: Time domain samples
• FFT with block Size 1024
• Hann Window
• Output: 513 FFT Bands
Synchronization Bit Frame Generator (Mode::BLOCK & Mode::CLIP)
• Generates 6 sync bits in 6 * 85 frames (* 2 for Mode::CLIP)
• Randomizes Up/Down-Band shifts [R2]
• Sorts synchronization bit frames by frame index
• Mode::BLOCK Output: 510 Bit Frames with 60 Up & 60 Down-Bands
• Mode::CLIP Output: 1020 Bit Frames with 60 Up & 60 Down-Bands
Decibel Quantifier
• Uses coarse stepping of 256 values for approximate search
• A stepping of 256 equates 1/4th FFT block
• Uses fine stepping of 8 values for refined search
• Pulls FFT Bands for all input blocks
• Computes dB for all bands of all blocks
Repeat Refined
Refining Feedback
Synchronization Bit Matching
• Collect Up/Down-Band magnitudes for sync bits
• Determine match quality for alternating bit patterns
• Apply watermark strength dependent thresholds
• Decide A/B-Block based on 010101 / 101010 detection
Approximate Synchronization Frame Search
• Skips over zero-padded samples (Mode::CLIP)
• Computes multiple time-shifted FFT vectors
• Uses coarse subframe stepping of 256 values
• Overlaps frames for sync detection by 1/4th frame
• Scores positions for synchronization matches
Repeat Refined
Synchronization Frame Selection
• Due to the subframe stepping, good and bad matches can be expected to alternate
• Identification of local match maxima
• Strength dependent threshold determines minimum match quality
• Selection of likely match positions via maxima and threshold
Refined Synchronization Frame Search
• Computes fine-stepped time-shifted FFT vectors around selected frames
• Searches ±16 subframes around previously detected good scores
• Keeps subframe if the score (synchronization frame detection quality) improves
Scoring and A/B-Type for potential blocks

The user provided 128-Bit AES key is essential to determine spectral bands, encoding patterns, and bit locations. During decoding, the same Pseudo Random Number Generator sequences R1. . . R6 are used that facilitated watermark embedding. By using the same AES key and a cryptographically secure PRNG, the sequences are uniformly distributed and deterministically reproducible but cannot be extrapolated. This prevents watermark extraction or modification by anyone without possession of the exact encoding key.

The Patchwork Algorithm

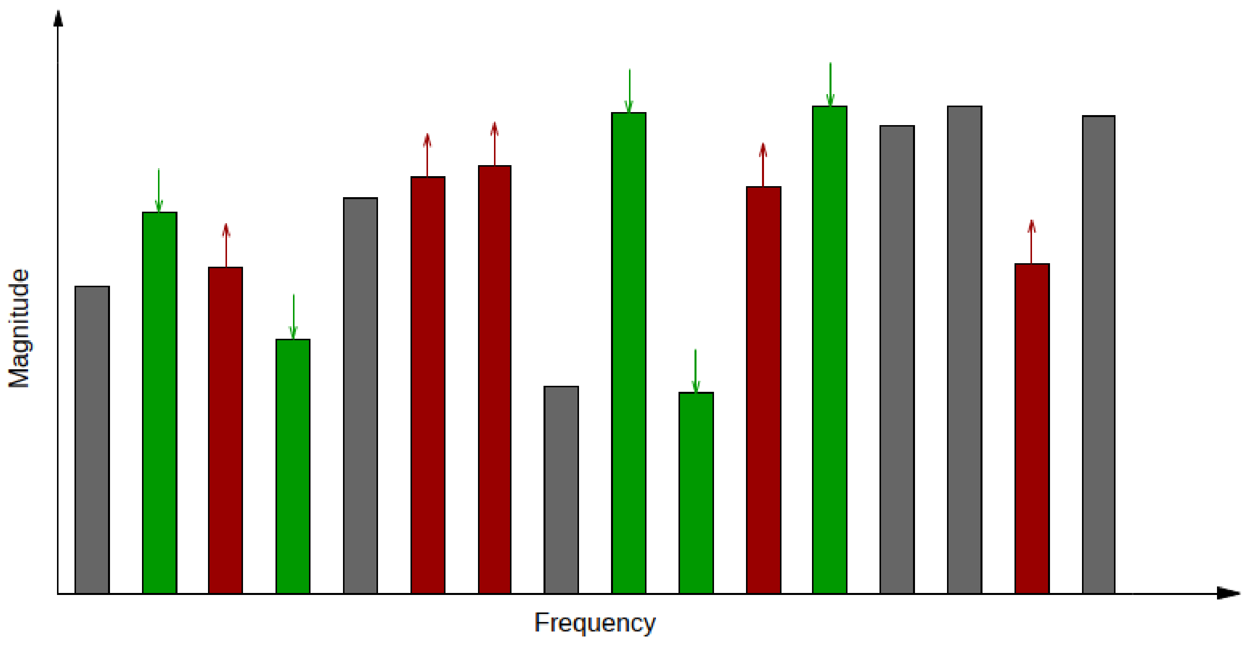


Figure 1: Example Spectrum

To store one single bit inside a spectrum, **audiowmark** uses the patchwork algorithm. From the frequency bands of the spectrum (generated by computing the FFT of one frame), two groups are choosen in the frequency range of the watermark using the pseudo random number generator. These are called up- and down-bands. In the example above, the up-bands are red and the down-bands are green. Typically there are 30 up- and 30 down-bands and the other bands do not carry information.

To embed a single bit, the following changes are made to the spectrum:

- to **store a 1 bit**, each magnitude of each up-band is increased by a small amount, and each magnitude of each down-band is decreased by a small amount (this is shown by the small arrows in the example image)
- to **store a 0 bit**, each magnitude of each up-band is decreased, and each magnitude of each up-band is increased (the opposite of the small arrows in the example image)

Since we have pseudo-randomly choosen the up- and down-bands from the spectrum, we can expect that if we sum up all values of the up-bands and sum up all values of the down-bands **before** embedding the bit, we will get a similar result (because the mean value of all spectrum bins is shared between the two).

However, since we increased all elements of the up-bands and decreased all elements of the down-bands **after embedding a 1 bit**, the sum of the up-bands should be **greater than** the sum of the down-bands.

So to decode the bit from the spectrum, we can simply use the rule

- **decode as 1 bit**, if the sum of the up-bands is greater than the sum of the down-bands
- **decode as 0 bit**, if the sum of the up-bands is smaller than the sum of the down-bands

In the actual implementation, increasing/decreasing the magnitude of the up-/down-bands is done by generating a watermark signal with the right magnitude/phase for each frame that only contains the changes. So we compute a delta spectrum, which is then passed to the IFFT, windowed and then added to the original audio, so that the sum has the desired modified spectrum magnitude.

The detection is performed on dB values of the magnitudes of the spectrum obtained from the FFT, so the sums of the dB values of up-/down-bands are computed and compared to decide whether a 0 bit or 1 bit was received.

The patchwork algorithm does not guarantee that encoding/decoding will always yield the right result at the lowest level of embedding/decoding one bit (as the difference of the up-/down-bands can be too big before embedding due to the original signal). However error correction and redundancy by embedding a bit in more than one frame makes the whole process reliable at a higher level.

There are three improvements over the basic patch work algorithm described above, which make the watermark detection more accurate:

- To use soft-decoding for the convolutional decoder, instead of deciding whether a 0 or 1 bit was received by comparing the two sums directly before decoding the convolutional code to obtain the message bits, the difference between the two sums is normalized and is used as a soft-bit input for the Viterbi algorithm.
- Instead of storing one data bit in each frame spectrum, a data bit uses up- and down-bands from different frames. This is called mix-encoding, which spreads the information of each data bit over many frames.
- As described above, the original signal can have some negative effect on the performance of the decoder, since the sum of the up-bands and the sum of the down-bands will be different even before embedding the bits. To make detection more reliable, the original signal level for each bin is estimated by taking the average value of the previous and next spectrum and subtracted before computing the sum of the up- and down-bands.

Mixing with Limiter

The input material for **audiowmark** is normalized (all samples are in the range from -1 to 1). If we simply added the watermark to the input, it could happen that this sum exceeds the range from -1 to 1 which would result in clipping. To avoid this, a limiter during mixing is used.

The limiter computes the highest peak for each one second long block. Then a linear volume envelope is constructed connecting the blocks, such that the envelope is greater or equal to the height of the peaks in each block. The typical value for really high peaks is about 1.04 for the default watermarking strength of 10.

To avoid clipping, the signal is divided by the slowly changing volume envelope. The result is somewhat similar to a lookahead peak limiter with attack of one second, and linear release of one second. Or to describe the effect more directly, if a single peak of 1.04 was produced in the watermarked signal, the limiter would slowly start decreasing the volume to 1/1.04 over the time of one second before the block that contains the peak, stay there for a while (due to the use of blocks) and afterwards slowly increase the volume again over the time of one second.

By using a limiter that works on one second blocks like this, it is possible to seek to any point in the watermark (which is required for streaming via HLS) and getting the exact same output that watermarking all previous samples would have produced, because the output of the limiter only depends on the current, previous and next one second block. So only a small context window needs to be processed when seeking.

Speed Detection

As one of the later developments, a dedicated speed detection facility has been integrated that explores the ability to extract watermarks from audio segments played back at unknown rates. In scenarios where audio has been resampled and pitched at a constant rate, synchronization markers may still be detectable by searching the audio content at varying resampled playback rates.

The `audiowmark --detect-speed` command line option attempts to detect playback rate changes compared to the original material used for embedding within 80% to 125%.

Detection of playback rate modifications is approached in several steps. First, the detector picks two short audio clips (ca 25 seconds) with high signal energy and performs multiple coarse scans while detecting <0.2% rate modulations. This rough speed estimate is improved upon with secondary scans around 1/20‰ rate modulations on an audio clip of 50 second.

Executing coarse and fine detection runs at varying resampled playback rates with multiple refining steps consumes a lot of processing resources. To speed up the detection, resampling and scanning runs are carried out on a downsampled version of the audio material (by a factor of 2) and detection runs are parallelized across all available CPU cores.

Finally, the watermark extraction is carried out on a resampled version of the audio material at the most likely detected playback rate in addition to regular watermark detection, because the detected playback rate may have been guessed wrongly.