wavEdit.exe

The wavEdit.exe program utilizes three additional components:

1. Octopus the integration algorithm and
2. The 32 bit version of the Mersenne Twister (http://en.wikipedia.org/wiki/Mersenne_twister) developed 1997 by Makoto Matsumoto (松本 眞) and Takuji Nishimura (西村 拓士). It is proven that the period is 2^19937-1, and a guaranteed 623-dimensional equidistribution property.
3. Libsndfile (http://www.mega-nerd.com/libsndfile/), a C library for reading and writing files containing sampled sound (such as MS Windows WAV/PCM and the Apple/SGI AIFF format) through one standard library interface by Erik de Castro Lopo (1999-2011).

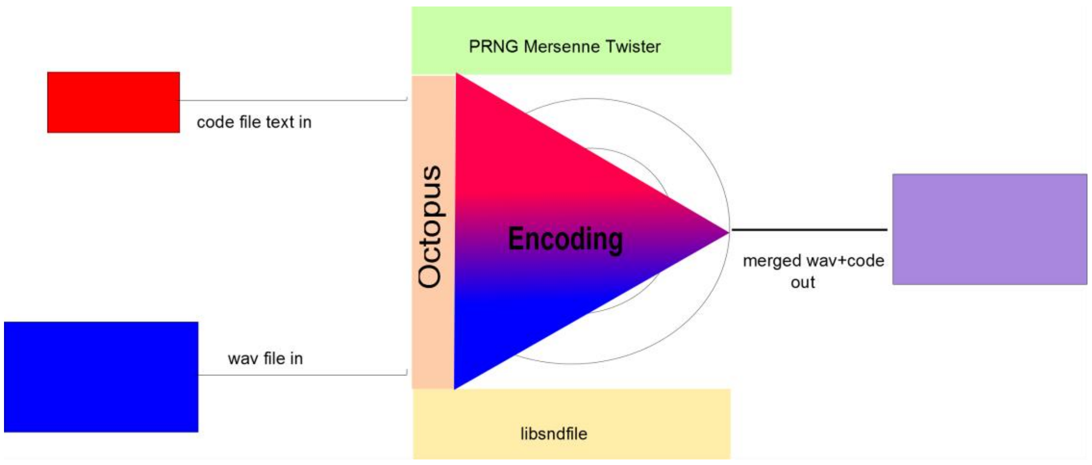
wavEdit:

Its function is both an interface to octopus and to and liberate/ease the implementation of octopus for other data types both the code-file which is momentarily only suggested to be text (see limitations below) and the receiving file which is momentarily only suggested to be plain 32 bit WAV/PCM files. Other sound encoding types have not been tested by the author due to his lack of expertise in the multimedia area. Others are welcome to adapt the interface to other data types: sound files, graphic files and video. (See also interesting possibilities in regards to streaming using CUDA in the ‘outlook’ paragraph at the end of this document.)

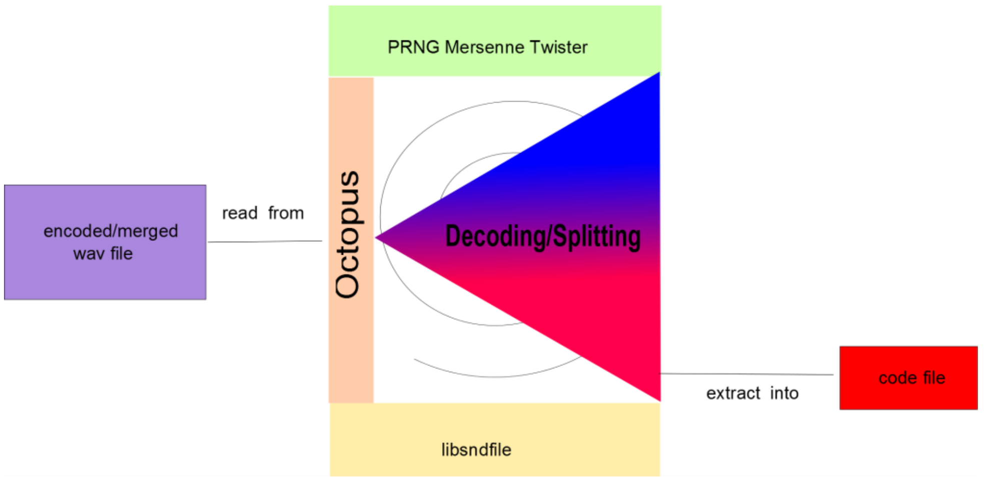
Function:

- Reads the parameters from the command line.
- Creates an instance of octopus and sets its parameters in the octopus process
- Creates input and output buffers.
- Sets the getBit and setBit function pointers into octopus. These two functions need to be adapted for each data/file type.
- Opens code file as input, wave file as input, the output file when encoding or the encoded WAV/PCM file for input and the code file for output when decoding.
- Reads the code file as strings and the input file via libsndfile and writes the encoded WAV/PCM file during encoding via libsndfile or
- Reads the WAV/PCM file and writes the extracted code file as text.
- House keeping

Encoding: Merging the code file with the WAV/PCM file into one WAV/PCM file without changing the WAV/PCM file's size:



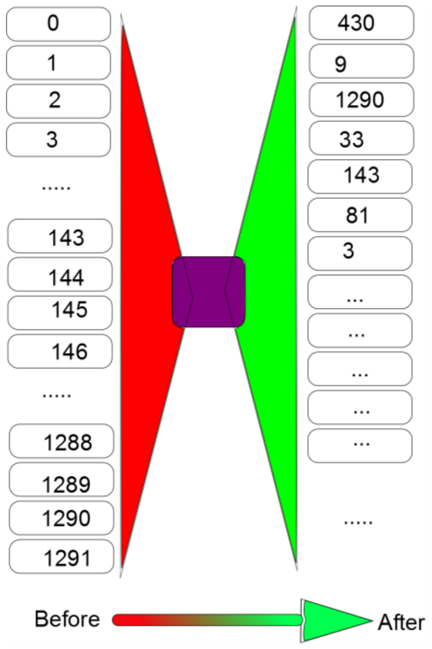
Decoding: Splitting the code text and extracting it into a text file:



Octopus:

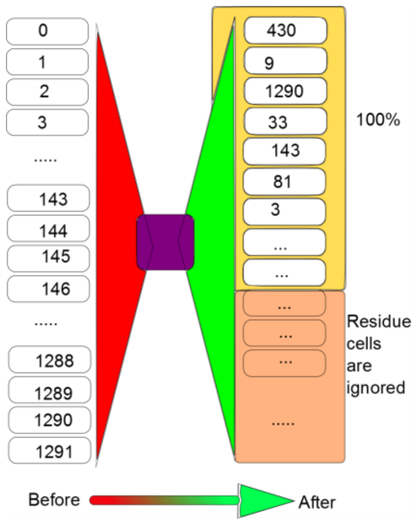
Its main feature is to pseudo-randomly set each bit of the message into the least significant bit of the sound file during encoding and retrieving it the same way to rebuild the original message.

An array calculated of the number of bits per block increased by the residue percentage is built and filled with the ordinalia of the cell in the array. The pseudo random generator scrambles them so that 0 is not in cell 0 anymore but at a randomly generated position. All cells are processed this way:



The scrambled table contains now each bit's position of the encodable text block. Function setBit sets each bit in the least significant bit of the WAV/PCM buffer.

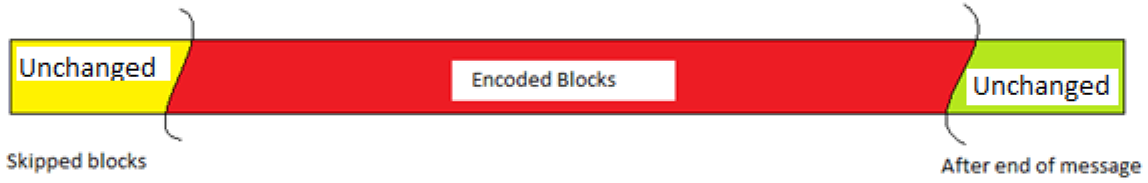
The **residue/salt parameter** builds a larger table, scrambles it but uses only the original subset leaving random bits unchanged (this affects statistical methods trying to detect if any embedding was applied on the file):



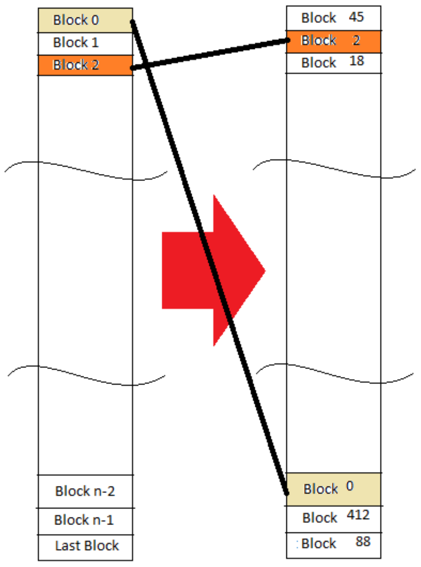
After each bit for the current block was set, this block is written into the receiving WAV/PCM file via libsndfile; the next block read from both code text and original WAV/PCM file. The scrambling repeats until the end of the text file (a terminating 0 byte is appended as EOT marker the last encoded character) has been reached. The original WAV/PCM file is from then on simply copied.

There are two additional options that could be added, but I believe that this algorithm is ‘good enough’:

An additional parameter could let the program skip N wave blocks just copying them until the encoding starts.



An additional step could scramble the blocks so that the first text block is at a random block position in the encoded file. This could be the same algorithm as above just not on bit level but on block level.



I decided against using this option since it would require random reads of blocks from the wave file and the libsndfile does for obvious reasons (no one listens to scrambled sound files) not support random reads of sound. The only alternative would be to copy the sound corpus into an intermediate file. Apply the encoding via random access and stream the modified intermediate file sequentially back into the target file. This not only complicates the code significantly, the additional step of writing manipulating and re-reading the intermediate file would also have a negative effect on throughput speed. I firmly believe that this step is not necessary since the data can -following the algorithm above- nayway not be extracted.

Outlook:

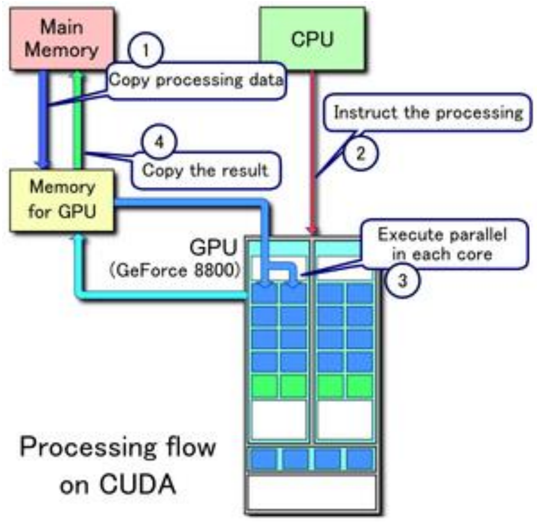
My expertise in multimedia encoding is very limited. There are uncompressed formats like WAV/PCM for sound and BMP for image data. Those can be easily used as my implementation documents.

Compressed formats may be lossless and lossy. Lossless means that the sound/image is somehow converted into a smaller file (compressed) the smaller file is stored or transmitted and the original is through an algorithm reconstituted to its original beauty. Like concentrated juices where water is extracted from the original juice and the same amount of water added later at the consumer's site to reconstitute the original amount of juice without loss of test-quality. The advantage is obviously that the same amount of goodies (may it be orange taste and vitamins or images or sound) needs less bandwidth on the transport medium (may it be a truck whose amount of load is limited or may it be a cable or wireless connection or storage amount. Any lossless format could be used here as long as the octopus encoding takes place before the compression and the decoding after the decompression (like for example RLL encoding or –to some extend- palette based compression.

The lossy formats are not usable (at least to my understanding) since the expanded data at the recipient/consumer is not identical to the producer's original data.

Video formats if lossless are perfect for large quantities of data to be hidden.

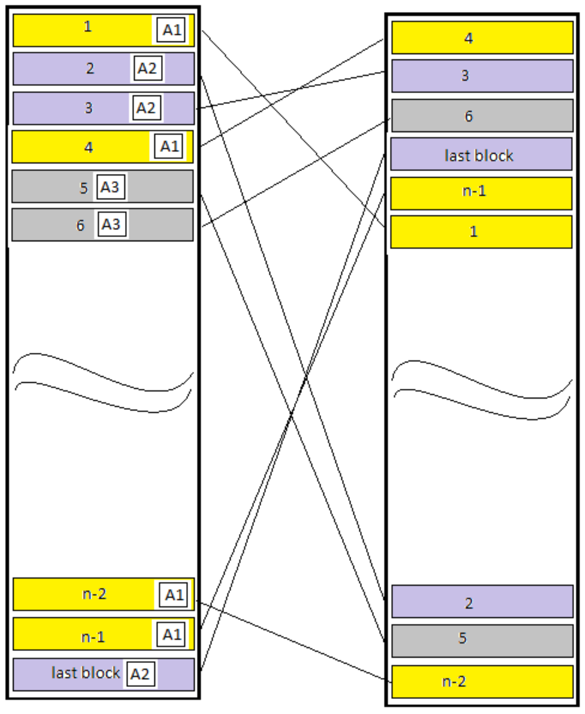
This leads to the possibility of using the multitasking capabilities of graphics cards which can process hundreds up to (yet) 1000 threads that run in parallel. CUDA (http://www.nvidia.com/object/cuda_home_new.html) / GPU accelerated computing (http://www.nvidia.com/object/what-is-gpu-computing.html) developed and supporting NVidia graphics cards is one of the development environments where the program implementation is facilitated in a C, C++, FORTRAN, Python, Ruby and Perl dialect/environment with parallel extensions.



Example of CUDA processing flow
1. Copy data from main mem to GPU mem
2. CPU instructs the process to GPU
3. GPU execute parallel in each core
4. Copy the result from GPU mem to main mem

Source: http://en.wikipedia.org/wiki/CUDA

The encoding would (to my understanding) only gain performance if the embedding of the message would use the same scrambling array multiple times since every data has to be moved first into the GPU's memory, processed and transferred back into conventional memory. Using the same scrambling table multiple time may increase the chance of hacking unless the blocks are not consecutive (back to randomly encode blocks):



'A1' … 'An' are scrambling tables different from each other, but all A1 use the same table, all A2 the same, etc.

Attack methods:

1) *Detection:*

Statistical methods won’t work; especially with a larger (> 20% residue) and because each block does use -in the original implementation- a different scrambling matrix. The file size does not change. Only having both the original and the encoded WAV/PCM to compare will show minor changes. This still does not mean that the hacker is one step closer to decoding/extracting the hidden message.

2) *Hacking:*

Even having the original and the derived version does not allow to predict later transmissions. Neither does having all three (source file, result file, code-text) solve unless the producer uses always the same 4 parameters since the next file’s embedding follows a completely different pattern. Having the source code of this program does also not help since without the 4 parameters there is no predictable pattern.

The weakest link is the Pseudo Random Generator! If the periodicity (the periodicity is the number of consecutive calls that have to be made until the chain of elements repeats itself –hence Pseudo Random Generator-) is short and a few consecutive elements in a short series can be somehow found opens the door to brute force attempts. That each block follows a different distribution is irrelevant if the series is known. This is the reason I used a PRNG with a period length of 2^19937-1. This is large enough that even having somehow retrieved a few thousand consecutive elements will not solve anything since the next message will start somewhere else in the chain of 2^19937-1 elements. The seed is kind of the entry point in that chain.

Furthermore:

Since the PRNG IS the heart of the algorithm I decided NOT to use any one of which I have no source code since there may be backdoors in the commercially available packages or included into the development environment or as loadable modules.

In short: if you don’t have the source code and compiled the source code yourself: don’t trust it!

The Mersenne Twister is fast and has a long periodicity and everyone having the source code (which is and has to be included) can verify the authenticity. This program should be compiled by the user from the source code. No precompiled or otherwise packaged implementation should be used, even if the source code is included since the compiled version is not guaranteed to be from the attached source code.

Günter Strubinsky

Omaha, the 15th Feb. 2014