CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

SLAC:

AN ALGORITHM FOR LOSSLESS AUDIO COMPRESSION

Graduate Project in partial fulfillment
of the requirements for the degree
Masters of Science in Computer Science

by

Reid Woodbury Jr.

June 2006

This thesis/project of Reid Woodbury Jr. is approved:

Dr. Jeff Wiegley	Date
Dr. Deborah van Alphen	Date
Dr. Gloria Melara, Chair	Date

California State University, Northridge

TABLE OF CONTENTS

ABSTRACT

SLAC:

AN ALGORITHM FOR LOSSLESS AUDIO COMPRESSION

By

Reid Woodbury Jr.

Masters of Science in Computer Science

Lossless compression happens when a pattern can be described with symbols that are only as big as needed at a given instant to uniquely and exactly represent each value. Signals are a special case, as the number of bits needed to represent a value are often very high, and also have the property that each symbol is closely related to the next. Signal compression algorithms go an extra step to find the similarities between adjacent symbols and only store the differences between what the real value is and what some prediction scheme says the value should be.

Typically, compression algorithms examine symbols in a data stream each in whole—whether those symbols are characters, bytes, pixels, or audio samples—and then look at that symbol's neighbors for some pattern or redundancy. This paper proposes an approach that examines the data stream in larger pieces, treats those pieces as arrays of bits where each row represents one symbol, and examines the arrays column by column rather than one row at a time.

This thesis/project of Reid Woodbury Jr. is approved:

[illegible]
Dr. Jeff Wiegley

5/10/2006
Date

dvanalphen
Dr. Deborah van Alphen

5/11/06
Date

[illegible]
Dr. Gloria Melara, Chair

5-11-2006
Date

California State University, Northridge

Introduction

The consumer's need for accurate representation of audio program material is much less strict than that of professional listeners. Many algorithms have been developed to carefully model a signal and throw out less important information based on users' preferences and tastes. They are called *lossy* algorithms. There are similar algorithms for images too. We need a different type of compression algorithm to cover the case where any loss of information would be unacceptable, whether that be for technical or aesthetic reasons. These are called *lossless* algorithms.

This paper focuses on audio signals used in the entertainment industry such as music compact disks (CDs) and movie sound tracks. In general, a signal can be anything that represents something in the real world; a sonogram, radar, radio, video, or even still pictures. Lossless compression happens when a pattern can be described with symbols that are only as big as needed at a given instant to uniquely and exactly represent each value. Signals are a special case, as the number of bits needed to represent a value are often as high as 32 bits (sometimes more), rather than the typical eight used for text and executables, and also have the property that each instance (or symbol) is closely related to the next. And, where text and executables are precise, a repeat of the same character or word is exactly the same character or word. A signal can be say, a half-decibel quieter than the original (values approximately 94% of the magnitude of the original) and not be discernable from the original signal. Even trained listeners without a side-by-side comparison won't be able to detect this change. Signal compression algorithms go this extra step to find the similarities between adjacent samples and only store the differences between what the real value is and what some prediction scheme says the value should be.

These similarities can be calculated with some very complicated digital signal processing (DSP) algorithms.

The consumer marketplace has shown a desire for storing large amounts of audio. CD quality digital audio files are about 10.5 megabytes in size for every minute of audio. This means there's a need for 500 to 700 megabytes of storage for each CD in a collection. Doing the math, a portable player's 30-gigabyte hard drive can hold about 50 uncompressed CD albums. The lossy algorithms can raise this to about 500 CDs with only audio purists possibly detecting any loss of quality, or even a thousand with what many people would consider acceptable quality.

Lossless algorithms would bring this number to at least 80 albums with no loss of sound quality (no loss of information). Some reach average results as high as 150 albums, depending on the musical or audio content. There are many programs written for doing this and some will be examined here.

Typically, compression algorithms examine symbols in a data stream each in whole—whether those symbols are characters, bytes, pixels, or audio samples—and then look at that symbol's neighbors for some pattern or redundancy. This paper proposes an approach that examines the data stream in larger pieces, treats those pieces as arrays of bits where each row represents one symbol, and examines the arrays column by column—down the side—rather than one row at a time. It will be called *SLAC* for "sideways lossless audio compression."

Audio vs. Data Compression

Professional audio engineers, sound editors, or music or film sound mixers are familiar with the term compression but they use it very differently. For them, "audio compression" means that the dynamic range, the difference between the loudest and softest sounds, has been reduced. This is a critical step in the preparation of audio for the consumer. Natural sounds have a far too wide dynamic range for typical playback. Many electronic devices have been created to manipulate an electronic audio signal, each having their own characteristic *sound*. Audio compression algorithms used to act on a digitized version of an audio signal are often designed to mimic the characteristics of favored electronic compressors. That type of compression will not be covered further.

Objective

Lossless audio data compression has been an interest of this author for many years. As an understanding of the computing issues involved grew, the need to try out some of these ideas also grew. This new algorithm, SLAC, will be discussed along with the current state of lossless audio compression in general. Digital audio for consumers is the focus, but digital audio is just a special case of signal processing. The techniques tried so far, what their results seem to be, and some ideas for improving the output of the program will be covered. Of course, improvements mean smaller output files as this is about compression, and a faster running program. Techniques to maximize compression with this new view will be explored. A set of sound files was found on a web site that analyzes several lossless compression algorithms. These files will be examined with this algorithm and recompiled versions of existing algorithms, with the results extending what was found on that web site.[1]

Basic Data Compression Techniques

Quite often the most efficient way to represent a symbol for easiest access and interpretation of what that symbol means is not the most efficient in the amount of space used.[2] Today, the most common encoding of text is called UNICODE. There are many variants for different languages and dialects. For demonstration's sake we'll talk about the very similar and older ASCII (American Standard Code for Information Interchange). It has the amazing property in that it uses the lower seven bits of an 8-bit byte to encode the alphabet in… alphabetic order. This is a silly statement, but a common algorithm is sorting, and the most common way to sort text is in alphabetic order. So it behooves us to use an encoding in which the natural order corresponds to the order of the data it is to represent. ASCII further applies a pattern to the letters and numbers to make them easier, thus faster, to work with. For instance observe how letter case is represented for upper and lower case "A":

$$\begin{array}{l}\texttt{A = 01}\boxed{\texttt{0}}\texttt{00001}\\ \texttt{a = 01}\boxed{\texttt{1}}\texttt{00001}\end{array}$$

By simply masking off the bits in the box, or even the upper three bits, a sorting algorithm that is to ignore letter-case need only continue as usual. And the numeric characters (0–9) are represented so that the four most significant bits can be ignored and the remaining bits are stored in two's complement encoding, which is the same way integers are most often encoded. This way they can be interpreted as an integer with a minimum of manipulation.[3]

With real world data, files will contain much redundancy and thus, wasted space. For instance, we can easily see that files that contain only ASCII numbers would be wasting (at least) half their space, as the four most significant bits aren't needed. In regular text files there is rarely an equal number of each character so with more complex encoding techniques a variable number of bits can be used to represent each character with shorter strings of bits used for the characters that repeat most.[4] Patterns that repeat can also be marked and pointed to later as that pattern reoccurs.

A key word that comes up in much of the literature on data compression is *entropy*. The entropy E of a symbol is defined by the probability P of the symbol being used (1.0 being all the time, 0.0 being none of the time, 0.5 being half the time):

$$E = -P \log_2 P \ .$$[5]

The entropy of an alphabet is given as the sum of the entropies of the symbols in the alphabet. From this information we can do a statistical analysis of the data in a file and predict the best possible compression for the file.

Most often in the literature, the compression encoding type is considered to be either *statistical* or *dictionary*. Others group them differently.[6,7] This author considers *transforms* to not be directly part of the compressing of data but useful in preparation of the data. For instance, taking the delta-transform or relative encoding on a stream of 32-bit floats may shrink the magnitude of the values stored in those floating-point variables but they will still take up 32 bits until one of these encoding types is applied to the data.

An implementation of some sort will likely employ several of these techniques tailored to best compress a certain type of data. One standard for FAXing documents encodes it with a predetermined table of statistics for the amount of white and black space represented as run-length encoding (RLE), with these counts represented by a variable-length code.[8] The use of that predetermined table obviates the need for a document to be scanned twice on the transmitting side, or for the receiving side to build a table from the received data in order to rebuild the image. It is important to know the needs of the data before making a blanket rule of how to represent that data.

Often file types are built around an encoding. The file type TIFF (tagged image file format) has a version of the RLE and LZW (Lempel, Ziv, and Welch; described later) techniques as standard encoding options to make storage smaller. Files with these options set will, of course, take longer to read and write than files without these processes.

Run-Length Encoding

Sound, picture, and even text files often have runs of repeating characters or symbols. These can be spaces inserted to format text, a character or characters to indicate a black sky in a picture, or integer zeros for silence between songs on a CD. Simple graphic pictures with long expanses of the same color are well suited for run-length encoding. It is considered a statistical encoding method but it's described separately here because it considers only data adjacent to the current value being examined. The remainder of the file isn't used except to determine the end of the run of this particular value. The statistical occurrences of other values in the file have no impact on how the current value is represented.

The trade off with run length encoding (RLE) is that it needs some kind of marker to indicate if the next piece of data represents another run of symbols. This can be either an escape character, which also needs a way to indicate that this character is to be taken literally occasionally, or an extra bit can be added to each symbol that when set, could indicate that *this* value is the length of the run of the previous symbol. In both cases care must be taken to be sure that there are enough redundancies that the added bit on each symbol or added character doesn't actually make the result larger.

Statistical Encoding

Statistical techniques for compressing data work by replacing symbols represented by an equal-length code with symbols from a variable-length code whose lengths are inversely proportional to their probability. Variable length codes only work if they follow the *prefix property*. The prefix property holds that once a certain bit pattern has been assigned as the code of a symbol, no other code can start with that pattern. Thus, no pattern can be the *prefix* of another.[9] Ideally the shortest codes are then assigned to characters with the greatest probability. These algorithms are also referred to as "entropy encoders." The simplest variable length code to visualize is a *unary* code. A version is shown here:

```
0 = 0
1 = 10
2 = 110
3 = 1110
4 = 11110
```

… and so forth. A count of zeros followed by a terminating "1" could also be used. One good thing about this code is that we don't need to know in advance how many different values need to be encoded. This is because the numeric value indicates the count of 1's to use before we reach a zero.

But the statistics of a character matter for compression. It would be foolish to encode a `7` as `11111110` when there are no `5`'s or `6`'s needing to be represented with the new code. Right away we can see we'd be better off encoding `7` as `111110` and save the space. Even more importantly, a string of these unary bits can be assigned to any symbol, such as the number `17`, the letter `Q`, or even a string of characters like `alphabet soup`.

Several techniques have evolved and been refined over the last sixty years or so. This started with the work of Golomb, Rice, Shannon, and Fano. The most popular of these algorithms is Huffman coding, which always produces the optimal prefix-code for a given entropy.[10] Rather than drawing out the steps, as is done in many texts, there is an excellent web site with a Java applet by Woi Ang where one can watch a Huffman tree built graphically, step by step at:

`http://www.cs.auckland.ac.nz/software/AlgAnim/huffman.html#huffman_anim`.

The Huffman code is so good because it builds a proper prefix-code from the frequency of symbols in a data set. This can be built for each file to be compressed (which is time consuming) or can be a canonical code based on typical usage. A large amount of standard text for a language can be examined by the programmer and a fixed Huffman encode and decode tree can be written into an implementation of the algorithm. There is also a technique of building the encode/decode tree as data is examined. This makes for faster encoding and decoding. As the algorithm progresses through a file a tree is built and the code is modified, as data is read and the frequency of the symbols changes. As an interesting note, the Huffman algorithm will build a unary code if the statistics of the data indicate that this would make the best encoding.

An important but difficult to understand statistical technique is *arithmetic* coding. An oversimplification of this process starts by mapping the symbols to encode over the interval [0,1), with the size of the sub ranges for each symbol proportional to their frequency in the original file. When a sub range is used, it is broken into new set of sub ranges proportion to the original range of [0, 1). This continues breaking down into smaller and smaller pieces, each represented by a number of greater and greater precision. This can produce better results than Huffman encoding by representing a large

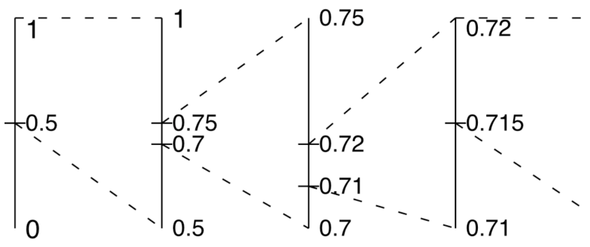


Figure 1. Recursive mapping of a range. [11]

number of input symbols, or even an entire file as one very large number, and that each symbol, in effect, will have a different number of bits representing it at different moments. This is because the input symbol is represented by the range in which the stored number falls, rather than the bits directly.

On decoding, the range in which the number appears determines what symbol it represents, as each subsequent sub range determines each subsequent symbol. Let's say the stored value is 0.718, and a set of mappings where the three most numerous characters are represented by *a*, *b*, and *c*. The symbol *a*, having a probability of 0.5, is given the range [0.5, 1); *b*, having a probability of 0.1, is given the range [0.4, 0.5); and *c*, having a probability of 0.2, is assigned the range [0.2, 0.4). Figure 1 shows how it maps the first four symbols of a set of data as *a*, *b*, *c*, *a*, etc.

Dictionary Encoding

Compression relies on files having redundancy. This redundancy need not only be based on the entropy of the characters used in a language but also on strings of characters repeating. A dictionary style algorithm will note where a pattern repeats and store a pointer to that repetition. Storing a pointer requires that the output data use more bits to represent a symbol so that we can discern when this symbol represents an original symbol or represents a series of symbols that have already been seen.

A popular algorithm (and the best until recently, see GZIP below) is LZW compression, named after its developers Lempel and Ziv, with later modifications by Welch. It works well with text and executable code.[12] It looks at files as strings of eight-bit symbols and outputs 12-bit symbols. A table with 4096 (2^{12}) positions is built as the input file is read. The first 256 output values are mapped directly from the input values. As a character is found on the input, it and subsequent characters are checked against existing entries starting at the end. If a match is found, the position of that match is written to the output rather than that character. If not, the single character is written as a 12-bit symbol and this pair is added to the table. As the table is built, the added "pair" can consist of a character, or reference to a string of characters, plus the one additional character. If the table fills up, it's cleared and a new table is started.

Lossy Compression

It should be noted that this paper is primarily about lossless compression. There is "lossy" compression, with popular forms being JPEG for pictures and MP3 for music. These algorithms' job is to find and throw out less important information to help make the resulting files smaller. They also have settings that an end user can adjust when the desired trade off between the resulting file size and the quality of the content have been determined.

JPEG

As we will examine later, a typical first step for signal compression is to increase the redundancy in the values representing a signal. A simple way to do this for audio, but not the best, is to take the running difference, or delta transform on consecutive pairs of values. A better way to correlate pairs of values, or pixels, for images when the desire is to throw out information, as in JPEG compression (Joint Photographic Experts Group), is to map each pair's values as *x* and *y* coordinates in a two-dimensional space, rotate the result 45° toward the *x* axis, then read the new *x* and *y* values. This results in the *y* values grouping around one common value, zero. As we've seen before this lends itself to better entropy encoding. It also lends itself to better lossy compression as some of the bits to represent the accuracy of these smaller magnitude values can be thrown out with a smaller effect on the reconstructed image than if we hadn't taken this step.[13] This has the visual effect of softening the edges of details in the image.

MP3

Besides the desire to store more music, widespread music sharing has increased popularity of the MP3 file type, or more accurately called "MPEG-1 Layer-3." This complex algorithm and those like this are sometimes called *perceptual compressors*. (There are many other perceptual encoding algorithms used for music and movies, and change depending on delivery method. Some are more transmission friendly, while other have a better sound quality.) It finds redundancy in material based on many psycho-acoustic studies and concepts that are used to determine what information can be thrown away, similar to the techniques used for JPEG. These algorithms break a signal into overlapping pieces, analyze the frequency content of these pieces, and store that information. This then allows the use of a psycho-acoustic technique called *masking*, as seen in Figure 2. Masking is the property that a louder tone will mask the perception of quieter tones that are close to it in frequency and time.

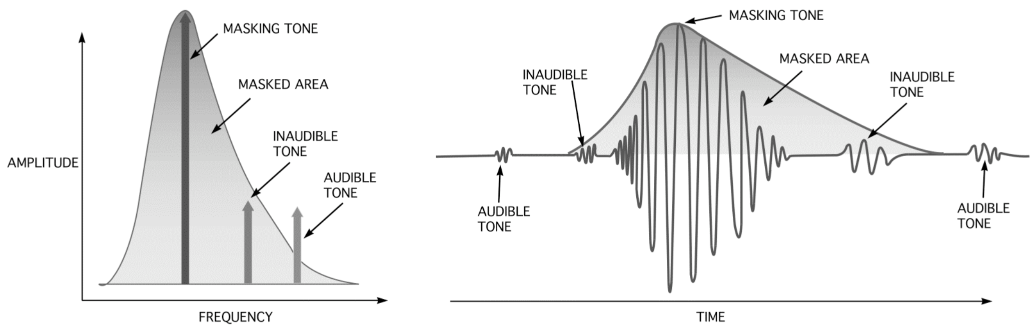


Figure 2. Effects of masking. [14]

A feature of these algorithms is that their bit rates can be set in advance, so they will only throw away the amount of data necessary to maintain that rate. They can choose to include all, part, or none of these other frequencies as needed, or to use fewer bits for marginal frequencies. These algorithms are also designed to pay more attention to (use more bits with) frequencies around 4kHz, as the human ear is more sensitive to these frequencies.

Lossless Signal Compression Programs

There are several lossless audio compression programs. Two of them, WavPack and FLAC (Free Lossless Audio Compression) were compiled locally for a proper comparison to the current incarnation of SLAC. GZIP was also tested for demonstrating the difference from those programs optimized for audio. Where information could be found, all of the signal compression algorithms use the same basic techniques (except for SLAC, of course). These can be broken down in two steps: transform and encoding. Correlation and prediction are both transform steps and both are used to shrink the magnitude of the values stored. Smaller values are more likely to have larger entropy. This is important for the encoding step. Encoding is then done with an entropy method. All programs, where information was available, use variations of Golomb-Rice or Huffman algorithms, or similar techniques of their own devising. FLAC adds the detection of long runs of absolute silence (sample values of zero) and run-length encodes these.

This excerpt draws attention to the architecture of many lossless audio codecs:

> • **Blocking.** The input is broken up into many contiguous blocks. […] The optimal size of the block is usually affected by many factors, including the sample rate, spectral characteristics over time, etc. […]
>
> • **Interchannel Decorrelation.** In the case of stereo streams, the encoder will create mid and side signals based on the [sum] and difference (respectively) of the left and right channels. The encoder will then pass the best form of the signal to the next stage.
>
> • **Prediction.** The block is passed through a prediction stage where the encoder tries to find a mathematical description (usually an approximate one) of the signal. This description is typically much smaller than the raw signal itself. Since the methods of prediction are known to both the encoder and decoder, only the parameters of the predictor need be included in the compressed stream. […]
>
> • **Residual coding.** If the predictor does not describe the signal exactly, the difference between the original signal and the predicted signal (called the error or residual signal) must be coded losslessy. If the predictor is effective, the residual signal will require fewer bits per sample than the original signal. […][15]

GZIP

Even though GZIP is not a signal compression program, per sé, it's included as a good standard program often used for other types of data. As good as LZW is and all of the struggles with copyrights that are placed on it, it turns out that for most data GZIP compresses better than LZW.[16] It's a combination of an earlier algorithm, LZ77, which uses a sliding window over the data to find duplications and redundancies, and the application of a predetermined Huffman table. The algorithm uses this window, which is up to 32Kbytes long, to search backward from the current byte for the longest match. It limits the length of the match and also takes the earliest match in order to produce the smallest reference value. This helps with the entropy and improves the effect of the Huffman algorithm. To improve execution speed, references to the past matches are placed in a hash table.[17]

Signal redundancy

In experiments, applying one of the simpler decorrelation techniques to a sound file before applying GZIP always resulted in a smaller file than using GZIP alone. This demonstrates how the requirements of compressing signals are different from other types of data, as mentioned in the abstract. If a predictive transform (correlation algorithm; discussed later) is applied to the data, the change from symbol to symbol can become very small. With the output from the delta transform, there will still be long sets of data where it will look like the data from a transform of the slightly louder data. This is also more in relation to how the data is perceived.

A program like GZIP, without a previously applied correlation transform, will see even a small moment-to-moment change in loudness as a completely different set of data and never relate it to earlier versions of an otherwise identical waveform. Dictionary methods aren't going to be suited directly to signal compression.[18] (See also "SLAC: Manipulating the Signal: Possible Transforms: Pattern Matching".)

Information in a signal is more dependent on how data in one moment is *related* to data in the next moment. Defining a "moment" is an important part of understanding a particular signal and how to compress it.

Transforms

Though transforms are not considered by this author to be a compression technique directly, transforms *are* important in the preparation of signal data before application of one of the compression techniques given above. Transforms act on how data relates to itself from one moment to the next, rather than the statistics taken and mapping done of the member values as done in non-signal compression algorithms. Study of lossless signal compression techniques shows that some transform of the data is done in each algorithm with the precise techniques kept secret in the highly competitive, closed source applications.

A transform can be as simple as a delta transform or running-difference, which means we store the difference between successive samples. We could also say "we predict this sample to be the same as the last." This works for signals as the values in successive symbols or values are closely related[19] (also see "SLAC: This Approach" in this document). A slightly more sophisticated method, described later in this document, stores the difference between the current sample and the linear prediction carried from the last two samples. An algorithm can use recursive or convolution filters, or be as complex as the Fourier Transform, or more commonly, the discrete cosine transform (DCT and variations) which is most often used in lossy algorithms.

A paper from Hewlett-Packard covers many details of what it calls *intra-channel decorrelation* and describes this part of the process this way:

> The purpose of [this] stage of the typical lossless audio coder … is to remove redundancy by decorrelating the samples within a [block]. … Most algorithms remove redundancy by some type of modified linear predictive modeling of the signal. In this approach a linear predictor is applied to the signal samples in each block resulting in a sequence of prediction error samples. The predictor parameters therefore represent the redundancy that is removed from the signal and the losslessly coded predictor parameters and prediction error together represent the signal in each block. [20]

Manipulating sound files

Programmers are less familiar with the way audio is stored than that of other files. We'll survey several here to familiarize the reader with these formats.

Sound Designer II (Sd2f)

Of all sound file formats this one is the simplest to use. It was created and made available by Digidesign, Inc. and was made public in hopes of it becoming a standard. The metadata, the data that describes how the sound data is formatted, is stored in an extra part of the file made available by Apple Computer called the *resource fork*. This leaves the *data fork* to contain only the actual audio data. If it's desired to begin reading audio from the start of the audio, the file pointer need only be put at the head of the file. Here is an excerpt from the Digidesign specification[21] document:

> Sound Designer II files store all sound samples in the data fork and all sound parameters in the resource fork. This is extremely convenient for sound data where the data fork may grow to a hundred megabytes or more. Regardless of the size of the data fork you can add, delete, and modify sound parameters at will without compacting the sound data or moving it around the disk (and extremely time consuming procedure if the file is 100 MB). In addition, you may add your own parameters to a file (as long as their resource types don't conflict with Sound Designer II's) while allowing the file to be read by both Sound Designer and your program.

The Apple resource fork can be thought of as a file system within a file. The exact position of the data from the head of the fork need not be a concern of the programmer. The OS handles reading and writing the labeled information. The labels are a combination of a four-character type code (i.e. '`Sd2f`', '`RWlc`'), a 16-bit integer ID, and an optional name string. Any four byte-values, represented by extended ASCII characters, is allowed in a four-character code, but Apple reserves the use of all lower-case letters.

The format predefines several resource types, only three of which are of interest here (and are required by the specification). The three are of type '`STR `' (note the space at the end to make four characters) and this type is used to store short strings. String ID 1000 is the sample size in bytes, string ID 1001 is the sample rate and shown as a decimal (44100.0000), and string ID 1002 is the number of channels. The use of strings makes the data human readable, and Digidesign's software always writes the resource name to the file, increasing its readability. A track from an audio CD converted to this file type would have a sample size of "2," a sample rate of "44100.0000," and the number of channels would be "2." An early Digidesign workstation actually used 20-bit samples padding the last four places with zero-bits to make a whole three bytes.

The data fork is so simply laid out that it's best to just quote from the specification document:

> The data of a Sound Designer II file is stored in Two's Complement encoding. Byte one of the data fork is the first byte of sound data. The sound data is organized as interleaved samples (if more then one channel) of either 8 or 16 [or 24] bit samples depending on the value of the 'sample-size' STR resource (see below).
>
> For example, a standard 16 bit stereo file would be organized as follows:
>
> | | |
> |---|---|
> | Left | Channel sample #1 |
> | Right | Channel sample #1 |
> | Left | Channel sample #2 |
> | Right | Channel sample #2 |
> | Left | Channel sample #3 |
> | Right | Channel sample #3 |
> | etc... | |

Audio Interchange Format (AIFF)

Apple originally designed AIFF to be used for moving information between applications and platforms but thought it useful and flexible enough for use directly by an application. It's based on a standard developed by Electronic Arts[22]. Only the Macintosh file system uses the two fork file type (*data* and *resource*) so all information is arranged and stored in the data fork of this file type. The complete specification is available from Apple and many other sources. The design of the format is so clever and flexible that it's worth study in itself.

The file structure is based on a generic type called a *chunk*, which the documentation describes with a C-like language thus:

```
typedef char[4] ID;
typedef struct {
    ID     ckID;         // chunk ID
    long   ckDataSize;   // chunk Size
    char   ckData[];     // data in variable sized array
} Chunk;
```

The member called `ckID` of type `ID` is a four-character code. This is convenient for humans and machines as four characters can be a useful abbreviation and a machine can read it as one native 32-bit value. The member called `ckDataSize` indicates the number of bytes in the array `ckData`. This size does not include the size of the members `ckID` and `ckDataSize`, which totals eight. The specification for a chunk can include other nested chunks and the chunk type can be unioned with any other structure of data the programmer wishes to code. Chunks can be in any order, so the first eight bytes need to be used as-is so a program can determine what the data is and how big it is, and calculate where the next chunk starts (if not end-of-file).

The AIFF file specification makes some alterations to this nested chunk system by defining a special outermost chunk called the *form chunk* or *container chunk* and uses the layout:

```
typedef struct {
    ID     ckID;       // always 'FORM'
    long   ckSize;     // always size of file minus 8
    ID     formType;   // always either 'AIFF' or 'AIFC'
    char   chunks [];
} ContainerChunk;
```

The `chunks` member is where the list of other chunks and nested chunks are placed. The three required chunks are called the Version Chunk, the Common Chunk, and the Sound Data Chunk. Each have the four-character IDs of ‘`FVER`’, ‘`COMM`’ and ‘`SSND`’ respectively, and can have only one of each. The Version Chunk was added on the change from the AIFF to AIFC type so that future changes could be made to the file structure without having to change the four-character file type code again (from AIFC to something else). Their structures look like this:

```
typedef struct {
  ID            ckID;      // ‘FVER’ for AIFC files
  long          ckDataSize; // always 4
  unsigned long timestamp; // 0xA2805140 (version as date)
}FormatVersionChunk;
```

for the version information,

```
typedef struct {
  ID            ckID;      // always ‘COMM’
  long          ckSize;    // always 18 for AIFF
  short         numChannels;
  unsigned long numSampleFrames; // == samples/channel
  short         sampleSize;      // bits per sample
  extended      sampleRate;      // roughly a 10 byte float
     // below added for AIFC, makes struct variable sized
  ID       compressionType; // registered 4-char code
  pstring  compressionName; // human-readable type name
}CommonChunk;
```

to hold the audio data format, and

```
typedef struct {
  ID            ckID;   // always ‘SSND’
  long          ckSize; // the size of soundData + 8
  unsigned long offset; // rarely used...
  unsigned long blockSize; // ...set to zero
  unsigned char soundData[];
}SoundDataChunk;
```

to hold the sound data itself.

The sound data in the `soundData` member is in the same arrangement as the sound data in the “data fork” of the Sound Designer II file type above. Note that AIFF stores the depth of each sample in bits where the Sound Designer II type uses bytes. If a non-multiple of eight bits is used for a sample zero bits are added on the right (LSBs) to pad it to the next 8-bit boundary.

The file type AIFC extends the AIFF type to include compressed audio data[23]. A program needs to check the compression type `ID` in the new Common Chunk to see if it understands this compression type, if any. (The compression type could be set to '`none`'.) Compression type codes should be registered with Apple to be sure there is no conflict with other codecs.

Resource Interchange Format (WAVE)

This format was created jointly by Microsoft Corporation and IBM Corporation, and is very similar to the AIFF format as it is also based on the original Electronic Arts Interchange File Format.[24] The primary difference between this format and AIFF is that a WAVE file uses the Intel byte order and AIFF uses the Motorola byte order to represent numbers and audio samples. The four-character codes used to describe the different chunks are also different from the AIFF format and the structure and nesting of the various chunks is different.

The first four bytes of a WAVE file are '`RIFF`' and stand for Resource Interchange File Format. The specification also has a file type of '`RIFX`' where the bytes are stored in Motorola byte order. Then the next four bytes make an unsigned integer indicating the number of bytes in the rest of the file. The WAVE "Format Chunk" corresponds to the AIFF "Common Chunk," and the WAVE "WAVE Data Chunk" corresponds to the AIFF "Sound Data Chunk."

ID3

The ID3 tagging system is a standardized and flexible way of storing information about an audio file within itself to determine its origin and contents. The information may be technical information, such as equalization curves, as well as related meta information, such as title, performer, copyright etc.[25] Use of these tags has become very common in files created by consumer music librarian programs like Apple's iTunes. Professional software for creating MP3 files also often has fields for entering this additional information. It was designed so that it could be added to any audio file and merely be ignored if the program reading the file doesn't understand it. Even thought the AIFF file format already has space for textual information, ID3 tagging was added by creating an ID3 chunk type '`ID3 `' and placing all the ID3 tags there.

ID3 tags are mentioned here because they are important additional information for identifying a file. This information occupies such a small percentage of the file that they do not need to be addressed as part of the compression. In fact, they should be left as is to help with further identifying the contents of a compressed file without decompressing or decoding first.

SLAC

Audio Editing & The Shape of the Waveform

The launch of cheaper, professional sound processing cards for personal computers in the early ’90’s was a tremendous advantage for smaller music and film production facilities, and for individual freelancers and hobbyists. Processing power and creative flexibility could now reach the hands of the average user. Professional quality audio here means that an audio channel, as an electrical signal, is sampled or measured 44,100 times a second and 16 bits is used to store each sample as a two’s complement, signed integer. This is still the format of the standard audio CD. Before that, digital audio in personal computers was typically sampled and stored in 8-bit samples (one byte per sample) taken at between 8000 and 22,000 times a second.

From that point on, *seeing* the waveform of the audio signal by using software to access the card and manipulate the audio files was as common as—perhaps more common than—*hearing* the sound itself (see Figure 3). It was apparent immediately to this user that there was lots of white space above and below the peaks of the waveform, or zeros, used in representing a sound. An understanding of how audio is represented in digital form soon followed and with that a sense that there must be a better way to represent the data with storage requirements being so high. Figure 3 shows almost 1.5 megabytes of data in its approximately 16 second length.

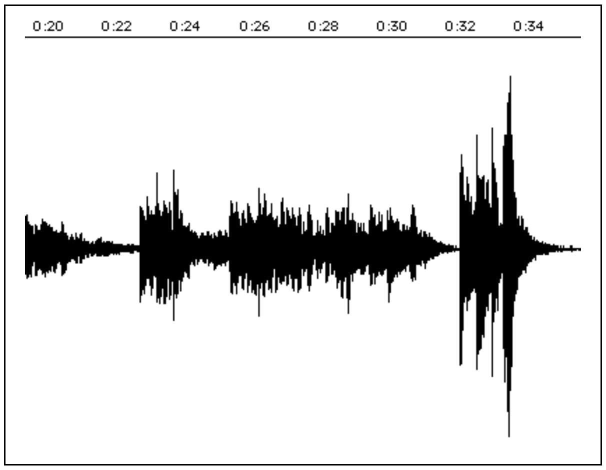


Figure 3. Portion of an audio file shown in audio editing software.

It became part of the sound editors' wisdom to not bother compressing digital audio with off-the-shelf compression programs (such as StuffIt) as one would with other computer data when archiving or backing up a project. Space reductions of 50% are considered a typical average for most data, with text files reducing to as little as 10% of their original size. Sound files, on the other hand, typically only compressed to 95% of their original size… hardly worth the time to compress and decompress the file which were generally very large and time consuming to handle to begin with.

It should be noted that that ratio of 95% was found in a casual test performed in 1994 with StuffIt, a generally free compression program for the Macintosh computer platform. It can be seen in Appendix E that the compression ratios are better now. Also note that the current version of `gzip` is used in that table and it gives better results than StuffIt.

Occasionally sound files would be examined with a hex-editor and patterns would be noticed in the data. Quite often every other byte was seen to be zero and these were noted to be quiet passages. This author's casual study revealed that standard compression programs are tailored for text, thus the very small desirable results for text files, and not working as well on data where, for instance, every other byte could repeat.

At the time, it was assumed that the values in an audio file were mostly random with some weighting to smaller magnitude numbers. But the values in an audio file aren't random in their placement except for the sound of some disbursement of noise. As stated earlier, one sample of audio is generally closely related to the value next to it. An idea struck this author early on to transform the audio file by saving only the difference between pairs of samples, often called a delta-transform or difference engine. Applying StuffIt to this transformed file resulted in a compressed file that was about 70% of the original size (even better with `gzip`, see Appendix E). Later, the technique one author calls "delta encoding"[26] was found and is often used as part of a signal compression algorithm.

Class work for this degree introduced me to several computing concepts and algorithms that otherwise would have been considered too difficult. Comp 222 Computer Organization introduced Huffman encoding and showed how it can be used to assign shorter bit patterns to more common symbols. This also encouraged me to look again at a Macintosh ToolBox (system API) routine called PackBits, which does run-length encoding on a sequence of bytes (not bits as the name implies) to see if something could be done with that.

By just looking at the bits in an audio stream, it was decided that beyond the most significant bits staying the same for a large number of samples, less significant bits seemed to change at random, or at best, there would not be enough weight given to a subset of symbols to favor shorter strings of bits over longer ones. Study of others' attempts show that an audio signal or other signals can be manipulated to favor smaller values. Where this is good for symbol-by-symbol encoding, it also increases the length of a run of the same bit in the left columns, which kept run length encoding as a more favored approach for this author. For a while, doing something like a Huffman encoding had been set aside until research for this paper revealed that most of the algorithms use a version of Huffman encoding.

After playing with some ideas in January of 2005 it was decided to "Google" for more information and a very good website was found where Robin Whittle[27] and another[28] compare in great detail several lossless audio compression programs, several of which have source code available (FLAC, WavPack, Monky's Audio, True Audio). Whittle made the original WAVE files he used for testing available on his web site so it was decided to also use those files in testing this algorithm. FLAC and WavPack were chosen for local testing. Reasonable comparisons can be made and conclusions drawn without repeating all of the tests with the other algorithms by extrapolating from the chart on Whittle's web site.

This approach

Compression of digitized signals in general, whether lossy or lossless, takes advantage of the feature where one sample or one pixel is closely related to its neighbors. Suppose a signal has been digitized into signed 16-bit integers. We can visualize this as an array of bits where each row is a sample. Say there's an array of samples starting at some value (see Table 1a with runs of 2, 4, and 8 in boxes) and count up one for each sample. Notice that starting on the right column we can see that the bits alternate every other row, then have run lengths of two in the second column, and have run lengths of

	. . .
0000000010100000	0000001000111110
0000000010100001	0000001010001000
0000000010100010	0000001001101001
0000000010100011	0000001011010111
0000000010100100	0000001011011110
0000000010100101	0000001010110011
0000000010100110	0000001100001101
0000000010100111	0000001110001010
0000000010101000	0000001101110011
	. . .
(a)	*(b)*

Table 1. The numbers 160–168 (a) and excerpt from audio file (b), both in binary.

four in the third. You’ll notice that most of the columns are correlated; that is, they have the same value throughout.

Notice this similar behavior in the short audio selection in Table 1b. The most significant bits are on the left and the least on the right. The bits in a column, primarily the left, repeat vertically. Intuitively, it can be seen here that runs of zero-bits or one-bits in a column are very common, so simple run-length encoding would be the most appropriate to use. Quiet passages in the program material or even the space between peaks of a waveform have lots of repeats. (Table 1b shows data from a quiet passage.)

The question becomes what is the best way to represent this data. Varying the number of columns examined at a time is considered. Each grouping requiring its own variation of the data structures.

It was taken *a priori* to start with the bits examined in pairs: 16th & 15th column first (leftmost, most significant bit is the 16th), then the 14th & 13th, etc. Other quantities and combinations of bits should be examined and tested. Two at a time were chosen after much thought (not experimentation, see Table 4 on page 28) as it seemed to be the most efficient way to group information.

Research on this paper led to the discovery that SLAC's method of examining the bits in a data set is similar to taking the bit-planes of an image. Different planes of bits in an image also show varying amount of correlation.[29] SLAC could easily be adapted to the lossless compression of images too.

The data types

Standard types

Over the decades, computers have had several groupings of bits used to represent some kind a symbol like a Roman character or an Arabic number, but now having a byte be eight bits as the core to represent one of these symbols is the standard. (The Unicode format has changed that, but this is a tangential issue.) From there grouping bytes in powers of two are used to create larger symbols. In the C programming language there is the "short integer" (or "short", 16 bits) and "long integer" (or "long", 32 bits). There's also the "long long" at 64 bits in length that won't be used here.

Additions for audio files

A convention for naming larger data types in audio files by Digidesign, Inc. will be used here. These types are variable in size, so these sizes are stored as part of a sound file to define what kind of data is stored and how it is to be interpreted. These terms are also used to define other sized pieces and types of data in various places such as audio CDs and electronic music keyboards. Clarity is needed here to avoid confusion with these similar uses of the same terms.[30]

One slice or one instance of time of one channel of audio is called a "sample" and is typically one, two, or three bytes in size and interpreted as a signed integer in two's complement form. Other types can be used for a sample such as floats and longs but if

$$\begin{array}{l}
\ldots \\
\texttt{00000110010011110000011110010101} \\
\boxed{\texttt{00000111}}\texttt{101111000000101111001000} \\
\texttt{00001010101110010000111010001101} \\
\boxed{\texttt{0000111100001101}}\texttt{0000111000001000} \\
\texttt{00010010010000000000110100100010} \\
\texttt{00010011111000100000110001111110} \\
\boxed{\texttt{00010011100111110000110001100111}} \\
\texttt{00010010011011110000110100001100} \\
\texttt{00010010011010010000110011110011} \\
\ldots
\end{array}$$

Table 2. Excerpt of audio bit stream with boxes around a byte, sample, and frame, respectively.

used at all they are for storage as the hardware to convert an analog signal to digital is generally converting it to integers of no more than 24 bits. Older personal computer hardware typically used 8-bit samples, the CD (Compact Disk) uses 16-bit samples, and pro-audio workstations (typically) use 24-bit samples.

A "frame" is one time-slice across all channels. Stereo, or two channels, will have 2 samples per frame and quadraphonic sound would have four samples per frame. The channel specifications for film sound of "5.1" and "7.1" would need six and eight samples per frame, respectively. A "7.1" channel, 24-bit recording would need 24 bytes per frame.

Data types for this algorithm

Two more data types will be used in this algorithm: a "block" and a "chunk". A block will be a maximum of 2^{15} frames in size and will remain this size throughout a file except for the end of the file where the size of the last chunk is the number of frames in the file modulo the chunk size specified for the algorithm. The reason for 2^{15} will be come apparent below.

```
typedef struct zeroRepeat {
   short  usePattern:1;


   short  count:15;
} zeroRepeat;   // 0x
```

```
typedef struct bitRepeat {
   Byte  usePattern:1;
   Byte  useLiteral:1;
   Byte  pattern:2;
   Byte  count:4;
} bitRepeat; // 10
```

```
typedef struct bitLiteral {
   Byte  usePattern:1;
   Byte  useLiteral:1;

   Byte  count:6;
   Byte  bits[64];
} bitLiteral;   // 11
```

Table 3. The three chunk structures shown without extended comments.

The chunk will be one to 65 bytes long (see Table 3). These are pieces of encoded audio, each chunk representing some variable amount of audio data. The names were chosen because a "block" seems to suggest a uniform size like a "concrete block" and a "chunk" suggests a variable size like a "chunk of chocolate". This is a different use of the term "chunk" from the Electronic Arts specification used at the basis for the AIFF and WAVE file types.

The chunk type is a union of three structures. In order to decode the data the first two bits of the byte are used to determine which of the following structures to use. Since the structures would never be used to indicate a run of zero, every value is offset by one like this:

```
00000000 = 1
00000001 = 2
00000010 = 3
00000011 = 4
```

… etc.

0x - Zero Repeat Chunk

This structure is used to store the count where a run from 17 to 2^{15} zero-bit pairs have been found in this pair of columns through this block of frames. The first bit of the chunk union being a zero, counting left to right, indicates the next 15 bits are to be used as the count of zero bit pairs:

```
typedef struct zeroRepeat {
   short usePattern:1; // must be 0
   short count:15;     // 32768 possible values, or 2^15
} zeroRepeat;          // size = 2 bytes
```

This is quite common with quieter passages of sound. A “short” can be used instead of this structure as values of 2^{15} or less (remember, values are offset by one) have the left most bit set to zero.

10 - Bit Repeat Chunk

A run of 4 or 5 to 16 bit pairs have been found in this pair of columns. (The “4 *or* 5” is explained in “bit literal”.) This can be any of the four combinations of two bits, including ‘00’.

The first bit of the chunk union being a 1 means to use this pattern, a 0-bit in the second position means to repeat this pattern, and the third and fourth bits store the pattern itself. The last four bits make the repeat count (maximum 2^4). The bits of the pattern to repeat are stored where they are as they will have to be moved in most cases to be placed back into position on decoding. Then the last four bits representing the count don’t have to be shifted to be used; the other bits can just be masked off:

```
typedef struct bitRepeat {
   Byte  usePattern:1; // must be 1
   Byte  useLiteral:1; // must be 0
   Byte  pattern:2;    // 00, 01, 10, 11
   Byte  count:4;      // 16 possible values, 2^4
} bitRepeat;           // size = 1 byte
```

11 - Bit Literal Chunk

The run of 1 to 64 *bytes* are to be used as is, no repetitions. Ones in the first two bits of the chunk union indicate this storage type. The last six bits are used as the count (1 to 2^6). Each byte will then hold the bit pairs in a pair of columns for four samples. A “repeat run” of four will be stored in a “bit literal” if we’re currently filling that data type and it’s not already full rather than switch to another chunk type. Since each byte can store four pairs of bits, the number of samples represented can be as high as 256. This data type could potentially add 128 *bytes* to represent a pair of columns literally as opposed to storing this data with out any counting.

```
typedef struct bitLiteral {
   Byte  usePattern:1; // must be 1
   Byte  useLiteral:1; // must be 1
   Byte  count:6;      // 64 possible values, 2^64
   Byte  bits[64];     // each stores 4 pairs of bits
} bitLiteral;          // size range from 2 to 65 bytes
```

Manipulating the Signal

These are transformations to put similar patterns of bytes/bits next to each other in a recognizable pattern for run length encoding. Research is showing that the real trick to doing lossless audio compression is to do some kind of manipulation to skew the data into a form that lends itself to patterns that are meaningful to the known lossless compression algorithms.

Complement and Rotate

It can be seen here *a priori* that runs of 0-bits in a column can be made to be by far the most common, so run-length encoding would be the most appropriate to use. It is easy to coerce the long runs of 1-bits in columns crossing negative sample values into zeros by taking the one’s-complement of all but the sign bit of those negative sample values, thus guaranteeing that quiet passages in the program material or even the space between peaks of a waveform will have lots of zeros (see Table 4).

Taking the one’s-complement of a sample, except for the sign bit, keeps most of the upper bits from changing when there are only a few bits changing across zero, then left rotate by one. This puts the sign bit, which is about as active as the least significant

Normal bit representation in excerpt of audio in file.		Same data with one's-complement and rotate.
. . .		. . .
0000010101010000		0000101010100000
0000010101001100		0000101010011000
0000010010100001		0000100101000010
0000001011111100		0000010111111000
0000001010100011		0000010101000110
0000000010110111		0000000101101110
1111111100101010	➔	0000000110101011
1111110010110101		0000011010010101
1111110101010001		0000010101011101
1111110111011100		0000010001000111
1111111011011100		0000001001000111
0000000010101010		0000000101010100
1111111100110111		0000000110010001
0000000111101110		0000001111011100
. . .		. . .

Table 4. One's-compliment except sign of negative values, then rotate.

bits (LSBs) with the LSBs. Remember the signal is constantly changing and taking the running difference has the effect of hiding slow changing values leaving only the quickly changing values.

Once a block of frames residual code has been found by some correlation algorithm it may be desirable to find the maximum number of bits used in the samples. This number of bits can be encoded at the start of the output block, five bits give us a maximum of 32 values, rather than using up 16 bits for every pair of columns of zero bits.

Correlation and Prediction

Initially, only applying a delta transform was used in this algorithm. It's easy to understand and runs very quickly. The difference between samples makes more of the numbers smaller. This could potentially add another bit, but this would only happen for sections with very loud (high amplitude) high frequency program material. The worst cases for CD audio could be 32,767 – (-32,768) = 65,535 and (-32,768) – 32,767 = -65,535 where each would take 17 bits to represent accurately. On the other hand this is very easy to calculate as simple subtraction of neighboring pairs of numbers is needed while encoding and addition while decoding.

After some research WavPack was found to use, at its fastest setting, a slightly more involved equation. It's still very fast and its results are given in Appendix E. The equation is the difference of the current value minus the last value (stopping here would be only taking the delta) plus the previous delta:

$$out_n = in_n - \left(in_{n-1} + \left(in_{n-1} - in_{n-2}\right)\right).$$

So, this means we are estimating that the current value will be as much different from the last value as the last value was different from the one before it. The algorithm performs the transform in place in memory by reading the buffer from the end to the beginning, substituting zeros for the values when the index needs to extend before the beginning of the buffer.

Correlation between the channels of a stereo pair must also be considered. Primarily monophonic program material will be very similar in both channels and therefore considered to have redundant information. Resolving this requires taking the sum and the difference of the two channels. After processing, primarily monophonic material will be strong in the new sum channel, and the difference channel will be very quiet, therefore containing more and longer runs of zeros. Each block is tested to see if it needs to be left in *LR* (left-right) form or put in *SD* (sum-difference) form. Normal stereo files will either be unchanged or improve the compression possibilities by taking the *SD* of the audio file. Taking the *SD* will hurt the compression results for program material that is primarily heavy in one channel.

This could also potentially add another bit to the sample. This will happen for signals that go above $\pm 2^{15}$ in each of the channels for (16-bit samples). To return back to left and right channels, sum the data in *SD* for the left channel and take the difference for the right, the output must be divided by two which can be done quickly by an arithmetic bit-shift to the right. In these equations *S* is the sum signal, *D* is difference with

$$S = L + R,$$

$$D = L - R;$$

so going back,

$$L = \frac{(L + R) + (L - R)}{2},$$

$$R = \frac{(L+R)-(L-R)}{2},$$

which is shown with the original left and right substituted for *S* and *D*.

Encoding

Data is loaded in blocks of 2^{15} frames because this is the maximum value that can be represented with 15 bits. Thus the maximum number of zero-bits in a column and the reason for the design of the "zero repeat" chunk type. Since each correlation transform can require another bit to properly represent the data, all samples are promoted to longs so as to not over flow the storage of those numbers. (Eight-bit samples need only be promoted to shorts, the next native type larger.) If, say the 17th and 18th bits remain zero in a 16-bit sample after the transforms the only cost is the time to count the pattern and two bytes in the output file representing that this column of bits is all zeros. The majority of values being smaller far outweigh the occasional value that needs more bits to be accurately represented.

Next, one or more decorrelation algorithms are applied to the block and a flag is set in the output stream. Each block can have a different algorithm applied. Part of the analysis is to determine which algorithm or combinations of algorithms will produce the best results. The program tests if interchannel decorrelation is necessary. The decode step is faster as it only needs to choose the algorithm indicated by the flag. The last step before encoding is to one's-complement each sample is and rotate it to put the sign bit on the right, as shown before in Table 4.

It was also considered that this file type could be used for streaming audio, thus the need to identify the start of a new block without having to start at the beginning of the file. It was determined that three sequential zero bytes would never occur as part of the algorithm and this is used to signal the start of a block. It should be noted that streaming broadcasting software requires audio to be encoded at a constant bit rate, which this is not.

For each block in the file, write three 0-bytes, then one byte with the correlation flags. Next save the size of the block (the last block might be short) as a short integer. Then for each pair of bit columns in the bit array, step through the entire block as indicated starting with the second state in Figure 4.

The working of the state machine in Figure 4 can be described like this: Scan for matching bits by marking where we are and grab a copy of the current pair of bits, then count the number of frames this pattern repeats in this pair of columns. If the repetition pattern is two zero-bits and there are more than 16 in a row, then save to the output file the count of zero-pairs in the "zero repeat" structure, which is the same as using a (positive) short integer. Runs of other bit patterns longer than 16 will have to be stored in

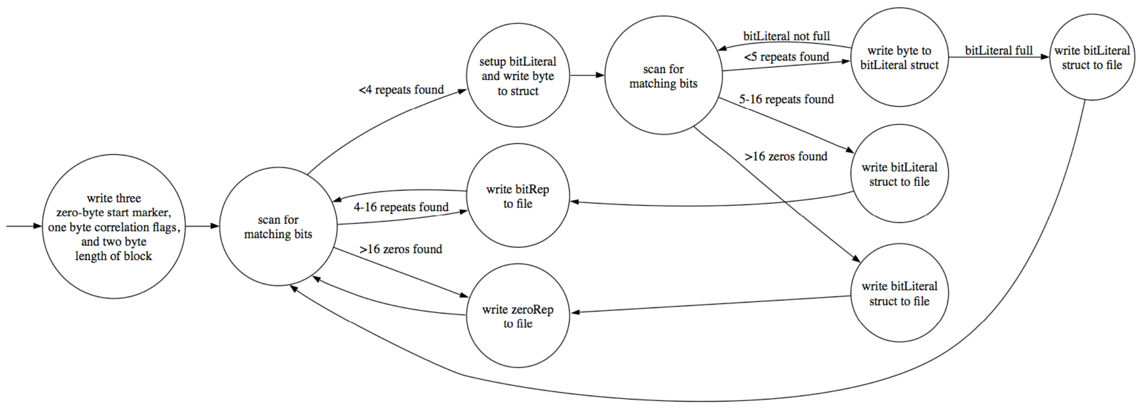

write three zero-byte start marker, one byte correlation flags, and two byte length of block
scan for matching bits
<4 repeats found
4-16 repeats found
>16 zeros found
setup bitLiteral and write byte to struct
write bitRep to file
write zeroRep to file
scan for matching bits
bitLiteral not full
<5 repeats found
5-16 repeats found
>16 zeros found
write byte to bitLiteral struct
bitLiteral full
write bitLiteral struct to file
write bitLiteral struct to file
write bitLiteral struct to file

Figure 4. State diagram for encode algorithm.

multiple "bit repeat" structures, but not if there are 17 to 19 in the count. This will become clear below.

Other than long runs of zero-bit-pairs, repetitions of any pattern of pairs of bits with counts of four or five to 16 will be stored in the "bit repeat" structure. If runs are longer than 16 a new "bit repeat" structure will need to be used. This structure will never be used to store counts of less than four. So if the count modulo 16 is less than four then that amount will be stored in the next data type.

If the run is three or fewer, then four pairs of bits are copied from the next four frames using the "bit literal" structure. When the structure is full, or a scan indicates a different structure should be used next, it is written to the output buffer. Repeating patterns of only four pairs of bits are also written to this structure—if it already exists—because switching to the "bit repeat" structure also only takes one byte giving no space advantage. It could also require the immediate restarting of the "bit literal" which has as much as an extra byte for every 8 bits saved, adding potentially 1/3 more to the size of the output than an unprocessed file, in the worst case.

Code for saving the "bit literal" structure needs to be just after the pattern counting is done. If a run of three or fewer is found it's either added to the existing "bit literal" structure or a new "bit literal" structure is set up. If there is a long enough repeating pattern found and a "bit literal" structure is in use then that "bit literal" structure is saved to the output buffer before writing the count of the current pattern to the output file as a "bit repeat" or "zero repeat" structure.

Decoding

The compressed data can be read from the file in any size piece at a time. The current Macintosh file system (this programmer's system of choice) is most efficient if file reads are in 4k-byte chunks so the compressed file is read in multiples of that. The decoding of the data can proceed one byte at a time using the form of a state machine. The first two bits of each byte are used to determine to which state we advance. The current state is held in a variable that is tested by a switch-case construct. Not shown is a counter that returns the machine back to state one. The numbers in Figure 5 actually correspond to the byte number in the block we are examining. The transition labeled "10" goes back to the same state because the code to handle this condition is placed directly where it's detected without changing the `state` variable. Actually changing states may be less confusing and may produce just as good compiled code given the current state of optimizing compilers.

The frame counter is tested to see if that pair of columns in the bit array have been completely set for that block to indicate when we move to the next pair. When all the pairs of columns are done, the entire block is written to the output file and the state ma-

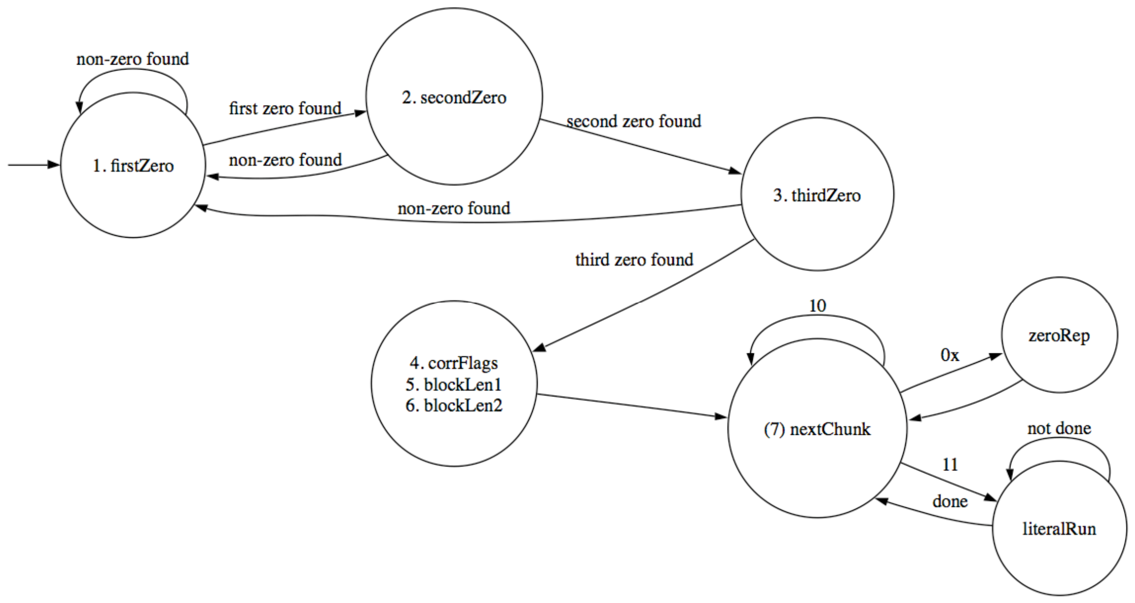


Figure 5. State diagram for decode algorithm.

chine is told to restart or exit depending on whether there is more data in the compressed file.

The machine states are identified by what we are looking for, so at the very beginning the state is set to "firstZero" as we are looking for the first zero byte of three that identifies the start of a block. When found it switches to the state "secondZero". If not found it keeps looking (this could be a stream rather than the start of a file) by switching back to looking for the first zero-byte that will mark the start of a block. The next byte contains the flags for what correlation algorithms were used, then two bytes are used as the short integer that describes the length of the block and are copied to the frame size variable in the two states "blockLen1" and "blockLen2". When that is done we set the state to "nextChunk" and start interpreting the chunks.

Why is the length of a block read in two steps? This is because there is a chance that the first byte of the short integer may be the last byte in the input buffer filled from the last time the compressed file was read. The algorithm needs to be prepared to wait for the input buffer to be refilled before reading the next byte.

The state “nextChunk” signals we’re looking for the next chunk to interpret. Now, if the first bit is a zero—the byte masked to show only the first two bits be visible can have a 0x00 or 0x40—then we grab this byte as the high byte of a short and set the state to “zeroRep” to indicate we’re looking for the second byte of the zero-repeat structure. Once that’s found we switch back to “nextChunk”.

While in “nextChunk”, if the first two bits are “10” (0x80) then the next two bits are saved as the pattern to repeat and that pattern is repeated the number of times saved in the last four bits.

Lastly, if the first two bits are “11” (0xC0) then we put the last six bits into a counter to show how many of the next bytes to write out explicitly and switch to state “literalRun”. We stay in “literalRun” until the counter has been decremented to zero and we then switch back to “nextChunk”.

The output buffer could be overrun when the bitLiteral structure is written to it. A non-multiple of four frames could have been counted by the zeroRepeat or the bitRepeat structures. So even if there’s only one literal pair of bits to write it will write four pairs. It is easier to code and faster running to declare a buffer with three extra frames at the end than to constantly check for the overrun.

The state diagram is re-started by counters that are not shown or exited completely when the end of file flag is found by the function reading the file.

Storage format

The file format for storage of the output of this algorithm is based on the AIFC file format. The file format also allows for any type of compression to be used and can be any standard type or new type. It’s up to an application as to whether it can handle that type or not. Meta data beyond sample rate, size, and number of channels can be copied in total to an AIFF style chunk. This prevents the need to understand the meaning of all the other metadata when coming from non-AIFF files. Of course if the original file is any type of AIFF or AIFC file then it makes sense to just copy all meta-data to the new file without change. Copying from the Sound Designer II format would require some special effort, as metadata is stored in the Macintosh resource fork.

Analysis

One audio CD was chosen for testing because of its familiarity to the author and its variety of selections. It's a demonstration CD put out by the company Brüel & Kjær in 1989 to show off their brand of microphones. This author has used this CD extensively for subjective testing of other parts of audio systems. The album includes a variety of pop, jazz, and many classical selections, along with sound effects and test signals. It was found to be useful that all of the selections were recorded with microphones from the same manufacturer, all of which are either a small or large diaphragm model of the same design. Where musical and performance styles vary widely the sound quality remains consistent from cut to cut.

The web site "Lossless audio compression"[31] provided the inspiration to follow through with this lossless compression idea. On referring to it recently, it was discovered that Whittle had added the audio he'd used in his testing, making it available to download as WAVE files. This solved the problem of making comparisons to work he'd already done. Both sets of audio were used in testing as this writer wanted to see results on audio that he was familiar with and wanted to use data to make meaningful comparisons to Whittle's work. Whittle's web page also refers to test files which he did not include for download and the B&K CD has suitable selections on it. On that site Whittle gives a short description of his sound files including their source with reference numbers and average volume level.

The time needed to process a file was timed with a shell script that was set up to step through them. Originally the intention was to use StuffIt instead of `gzip` to compare with this author's previous experience of compression of digital audio. There is no com-

```
!/bin/sh
or i in `ls "$1" | sed -e 's/ /:/g'`
o
  i=`echo $i | sed -e 's/:/ /g'`
  echo $i
  #time -p nice -n -20 cp "$1$i" "$2"
  #time -p nice -n -20 gzip -c "$1$i" > "$2/$i.gz"
  #time -p nice -n -20 gunzip "$1$i"
  #time -p nice -n -20 slac "$1$i" "$2"
  #time -p nice -n -20 slacd "$1$i" "$2"
  #time -p nice -n -20 slacwp "$1$i" "$2"
  #time -p nice -n -20 slac -d "$1$i" "$2"
  #time -p nice -n -20 flac -0 --totally-silent "$1$i"
  #time -p nice -n -20 flac -8 --totally-silent "$1$i"
  #time -p nice -n -20 flac -d --totally-silent "$1$i"
  #time -p nice -n -20 wavpack -f "$1$i"
  #time -p nice -n -20 wavpack -h "$1$i"
  time -p nice -n -20 wvunpack "$1$i"
  echo ""
one
```

Table 5. Shell script used for testing.

mand line version of StuffIt and it was desired to keep the timing framework consistent between the tested algorithms.

In the shell script in Table 5 the UNIX `sed` utility in the second line is there to change spaces into colons because the Macintosh file system allows file names with spaces and disallows colons, but a shell script's "for" statement uses spaces as a delimiter between items to process and will break a path in the wrong place. The fourth line changes the colon back to a space so the desired process can find the named file. The script takes source and destination directory paths as parameters. Both SLAC and `cp` need source and destination parameters as they both either need a new name for the file or a destination directory. `gzip` options were used to create new files from the original sound files rather than deleting the originals.

It was the desire of the writer to use open source algorithms since compiler optimization could be made similar across all programs thus eliminating one variable in the behavior of the programs. Both FLAC and WavPack were compiled from source code on a 1.67GHz PowerBook running MacOS X 10.4.5 (`gcc` 4.01). The algorithms were run on that laptop with the sound files read from and written to the laptop's internal hard drive. A working version of this author's algorithm was also compiled with `gcc` so it could also be used in the same timing script as the other algorithms. Primary work and debugging was done in Metrowerks Codewarrior, but it's not clear how well code optimization compares between Codewarrior and `gcc`, so `gcc` was used to keep down the possible variations. Macintosh ToolBox calls were adapted to the UNIX programming style rather than taking the time to rewrite them.

Timing was found to be very inconsistent between runs of any of the algorithms, varying upwards of 20% from run to run. It was determined that the cause of this was the fact that the host OS is a preemptive multi-tasking OS that could interrupt the algorithm at any time for other housekeeping. The OS is also known to have algorithms that cache and optimize executables[32] so one pass through the set of files was repeated in reverse order to check for that influence. Nothing beyond the earlier noted variation in processing time was noticed.

Later tests were re-run with the UNIX utility called `nice` to give priority to the tested program. Negative numbers for the "-n" option mean to be less nice to other running programs, giving priority to the current program. The most stable performance from run to run was found when each program was given a "nice" value of -20 and the shell script executed from a new shell started with a "nice" value of -10:

```
sudo nice -n -10 sh
```

with most performance time averages staying within 10% of each other.

Table of results

The results shown in Appendix E need some explanation. All files were in AIFF format with their sizes shown in bytes. The program `cp` is the UNIX utility to copy files.

It’s here to show the overhead a plain copy imposes. Time is always in seconds and every column labeled “encode” or “decode” is the time in seconds to encode or decode the file. The “t/M” column is the time normalized to process one minute of audio with this equation

$$n = t/(s/10584000),$$

where n is the normalized time, t is the total time to process on that file, and s is the original file’s unprocessed size. This size was chosen because half (or all, in the case of `cp`) of each process moves the original amount of data. The unlabeled columns after the “encode” and “decode” columns are the normalized times for those processes, too.

The main column group titles are mostly self-explanatory. “Delta” refers to the delta transform. In the script in Table 5 the programs `slac`, `slacd` and `slacwp` correspond in the tables below to `slac` with no transforms, `slac` with delta, and `slac` with WavPack’s simplest decorrelation transform. The rest show their command line options, generally run twice using the fastest setting and then the strongest setting. The very last foreshortened columns are WavPack’s strongest option with the files done in reverse order to check the effect of the OS caching commands.

Screen shots

Since this approach to signal compression was discovered by visually looking at the shape of the signal, views familiar to sound editors are shown in Appendix A.

Waveform

The term “waveform” is used for a time versus amplitude view of a sound file. The images here are screen shots of each selection of audio displayed in a ProTools “session” file. The x-axis is measured in time and is set to show the entire length of the file no matter what the actual length of the file is. The timing marks were also captured for reference and perspective. It's assumed from experience with this software that each screen pixel width is an average (not peak) of the signal's positive values and average of its negative values for the length of time the pixel spans.

Spectrum

The term "spectrum" is used for a frequency versus amplitude view of a file. The spectrum for each file was taken over only about ten seconds of program material at what was thought to be a representative section of the whole file. The motivation for this was as a time saver and also knowing the utility taking the spectrum would show the maximum values, so only a section with a strong signal was needed.

The software is a demonstration version of the Waves Audio Ltd. software plug-in called "PAZ" and it was set to take RMS values. It is thought that as the RMS value for a signal is considered to more closely represent it's perceived loudness[33] and that this correlates to compression results. This is because a single peak in the signal could be construed as being representative of the program material, but one loud peak in the signal would have little effect on the compression of an otherwise quiet passage.

Stereo correlation

This part of the Waves plug-in is a more elegant version of what audio engineers did by taking a feed of the left and right program material and connecting it to the y and x inputs, respectively, of an oscilloscope. Sensitivity was adjusted to show the typical "ball-of-string" pattern, depending on the program material, but typical for reverberant material. A vertical line indicates left-channel only program material and a horizontal line indicates right-channel only program material. Program material that is identical in both channels is shown as a line with a slope of positive one. A well-rounded ball-of-string pattern means that the program's material is well spread out in space and will seem to surround the listener.

The plug-in modifies this in several aesthetic ways but the information is the same. First, it tilts the display so that monophonic or center panned program material appears as a vertical line. Left and right program material show as slopes of negative and positive one, respectively. The scale of the display has been made to be logarithmic rather than normal linear view on an oscilloscope. It also has a peak hold function and averages the signal causing it to look more like a spiked ball rather than a wad of string.

In the following screen shots the vertical grey band in the images in the left column are a selected region of approximately ten seconds in length. This region was then used to generate the frequency response and stereo correlation images on the right.

Conclusion

The results are encouraging but the open source programs work much better for compression. SLAC *does* seem to have the edge on speed, performing better than the otherwise faster program, WavPack, with the same correlation algorithm. The future experiments mentioned below seem to be warranted.

Timing information was later gathered from the original Codewarrior compiled version of SLAC and was found to be comparable to the `gcc` compiled code. The effort to compile versions of the other algorithms locally in addition to SLAC may not have been necessary.

Future Experiments

Possible Transforms

Rather that just using delta encoding a more sophisticated approach could be used. It must be kept in mind that different transforms may add a bit or more to the resolution of the signal.

Some other transforms include linear predictive coding, wavelets, and discrete cosine transform. All of these have floating point and integer versions. The literature contains many and more sophisticated methods. Something simple may be more important than more data efficient algorithms as speed of processing is also a consideration when applying hardware to encode and decode the data.

All techniques need to predict a value only from previous values, or at least in the same direction as they are decoded. This way a transformed value can be derived from known values.

Pattern Matching

A better way of applying a running difference or delta to the signal might be to recognize a fundamental (as in harmonic fundamental) frequency and subtract the previous cycle from the current. This is similar to the LZW algorithm except that an exact match shouldn't be sought. It might be made to match all but the lower two or three bits and store a pointer to the pattern and the residual of the difference. The problems with this are determining what the fundamental frequency for a block is and in allowing for changes in that frequency.

Another possibility is to use a version of a dictionary method like LZW but again, to only match to within a few least-significant bits, then code the reference chunk and residual bits of the current set of symbols minus the reference chunk.

Splines

Several past points, or samples, can be used to determine a value based on spline curves. Earlier testing by this writer for splines to be used in decimation and interpolation algorithms showed that splines aren't good at representing signals, but the idea is to get closer to the actual next value with the simplest possible calculations. One of the spline techniques may prove to deliver a small residual code with a minimum of calculations.

Tunable Filter

A block can be tested for its dominant frequency. Then a band-pass filter can be tuned to that frequency and its output used to predict the next value with the difference from the actual value being stored. It may be useful to pair this with the output of the delta engine.

For that matter, a simple FFT (fast Fourier Transform) could be passed over a block of data and a handful of frequencies with their intensities could be encoded at the start of a block. This can be thought of as using a small amount of storage to indicate some redundancy. These frequencies are then used in the decorrelation step.

1-bit slices

In this first version of the algorithm it was assumed that the columns of bits would be best examined in pairs. The sign bit (most significant) is bit-rotated to the right (least significant) position so it is grouped with bits that will be changing approximately as often, leaving the leftmost bits to be changing the least.

After discovering the degree of importance that correlation is to the success of a lossless compression algorithm, and that there are many stronger algorithms than the delta transform, it seems that examining only one column of bits at a time may prove to be better. This would require only two structures and obviate the need of the rotate step in the complement and rotate step. While encoding, the algorithm can scan the sign bit column first and then continue on with the other columns in any order.

The "zero repeat" chunk would be used as is—but, of course, it would be only referring to a run of one column in the bit array rather than two, and would continue to refer to counts of 17 to 2^{15} zeros. The other would be a modified version of the "bit literal" chunk. The first bit set to 1 would indicate that this structure is to be used and the next seven bits used to indicate the number of the next bytes to interpret literally. The structure would look like this:

```
typedef struct bitLiteralOne {
   Byte usePattern:1; // must be 1
   Byte count:7;      // 128 possible values
   Byte bits[64];     // each stores 8 bits from column
} bitLiteral;         // size ranges from 2 to 129 bytes
```

Now each instance of the new "bit literal" structure can hold twice the number of bits as the 2-bit "bit literal" structure. A situation where *two* entire columns of bits must be encoded literally would now only cause an extra 64 bytes to be inserted. This is another thing that may make this structure better: output of the current encode algorithm using the two bit structure often has many consecutive "bit literals."

Lossy compression

A lossy form of this same algorithm could apply a certain amount of slew rate limiting, by limiting loud high frequency signals. This would let the running difference values stay smaller, but may be an objectionable form of data reduction.

Even better, use the example set in WavPack where two files are generated after the encoding. One file can be used by itself as lossy playback, or with the proper decoder, play both files where the second contains the residual error information, thus reconstructing the original signal completely.

Lossless with Another Lossy

As an experiment, convert a full fidelity, AIFF file to some lossy format. Then convert that small lossy file to a new AIFF file. Now subtract, sample by sample, the new transcoded AIFF file (with all its inaccuracies from passing through the lossy encoding) from the original file and call the new file the "residual" file. Next apply one of these compression techniques to the residual file. The combination of the sizes of the *lossy* file and the *residual* file may be the smallest lossless result yet. These two files could be woven together into one file.

Other Reading

Stuart, J. Robert “Coding High Quality Digital Audio” (Meridian Audio Ltd).

Rowe, Robert *Machine Musicianship* (Massachusetts Institute of Technology 2001).

Chamberlin, Hal *Musical Applications of Microprocessors, 2nd Ed.* (Hayden Books 1987).

Rorabaugh, C. Gritton *DSP Primer* (McGraw-Hill 1999).

Costa, Max H. M. and Malvar, Henrique S. “Efficient Run-Length Encoding of Binary Sources with Unknown Statistics” (Microsoft Research Microsoft Corporation 2003).

Endnotes & Bibliography

---

[1] Whittle, Robin, “Lossless Compression of Audio” http://www.firstpr.com.au/audiocomp/lossless/ 1998–2005.

[2] Smith, Steven W. *The Scientist and Engineer's Guide to Digital Signal Processing*, 2nd Ed. (California Technical Publishing 1999): 481.

[3] Smith: 69.

[4] Salomon, David *A Guide to Data Compression Methods* (Springer-Verlag New York, Inc. 2002): 1–2.

[5] Salomon: 9.

[6] Salomon: 3.

[7] Smith: 483–494.

[8] Salomon: 32–40.

[9] Salomon, David *Data Compression: The Complete Reference* (Springer-Verlag New York, Inc. 2000): 47.

[10] Wayner, Peter *Compression Algorithms for Real Programmers* (Morgan Kaufmann 2000): 19.

[11] Salomon: 117.

[12] Smith: 488.

[13] Salomon: 243–247.

[14] Dipert, Brian “Digital audio breaks the sound barrier” *EDN Magazine* July 20, 2000, pp. 74–75

[15] Airola, Cangialosi, & Dimino “PRESTO - Preservation Technologies for Euorpean Broadcast Archives” (PRESTO Consortium 2002) 10–11.

[16] Salomon: 202.

[17] Salomon: 205.

[18] Salomon: 642.

[19] Smith: 487.

[20] Hans, Mat; Schafer, Ronald W. “Lossless Compression of Digital Audio” (Hewlett-Packard Nov. 1999) 6–11.

[21] “Sound Designer I and Sound Designer II File Formats” (Digidesign. Inc. 1989) 4.

[22] ‘Audio Interchange File Format: “AIFF”: A Standard for Sampled Sound Files Version 1.3’ (Apple Computer, Inc. 1989) 1.

[23] “Audio Interchange File Format AIFF-C: A revision to include compresssed audio data” (Apple Computer, Inc. 1991) *i*.

[24] “Multimedia Programming Interface and Data Specifications 1.0” (IBM Corporation and Microsoft Corporation 1991) 10–37.

[25] “ID3 tag version 2.3.0” (ID3.org 1999).

[26] Smith: 486–488.

[27] Whittle.

[28] “Performance comparison of lossless audio compressors” http://members.home.nl/w.speek/comparison.htm 2005

[29] Salomon: 235–237.

[30] “Global Music Resource - Digital Audio Formats and Streaming Audio Formats” http://www.globalmusicresource.com/basics/basics3.html 2003.

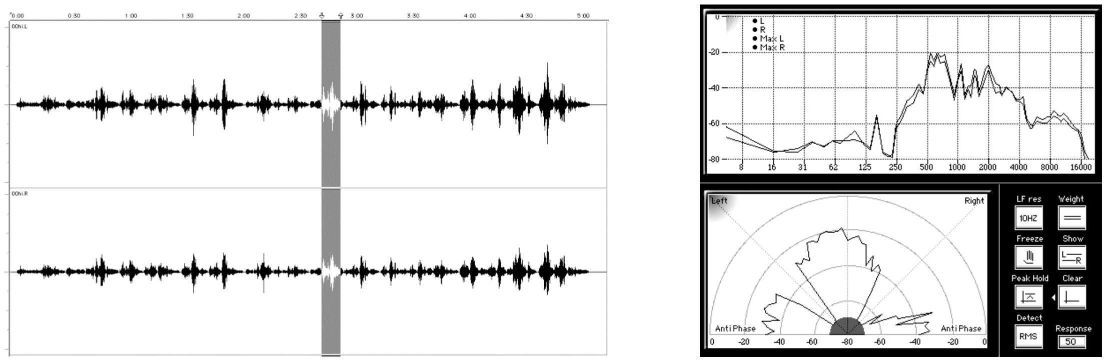
[31] Whittle.

[32] Singh, Amit “Making An Operating System Faster” http://www.kernelthread.com/mac/apme/optimizations/ 2005.

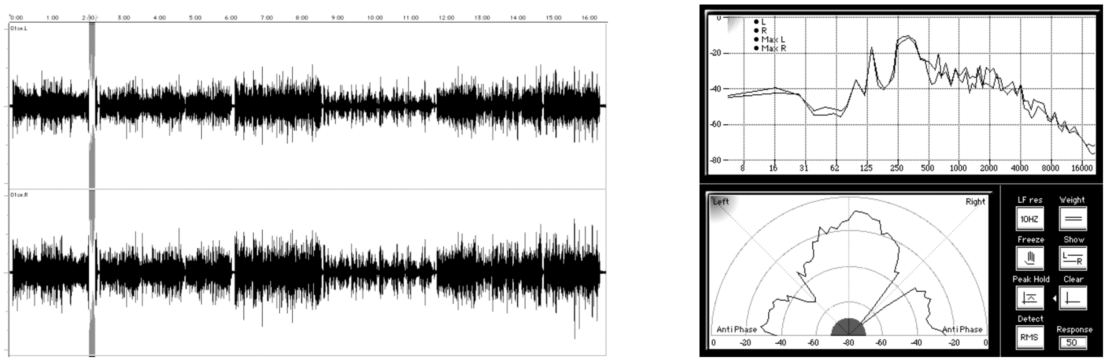
[33] Smithers, Brian “Meter Matters” (Electronic Musician 1/2003) *3*.

Appendices

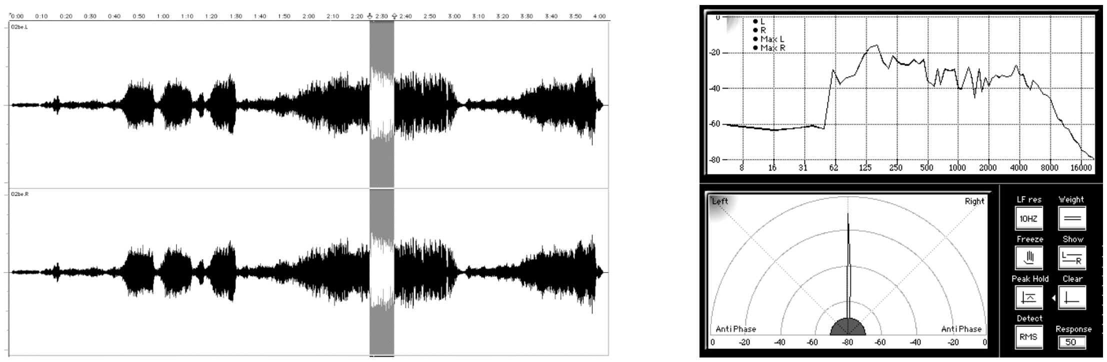
A Screen shots



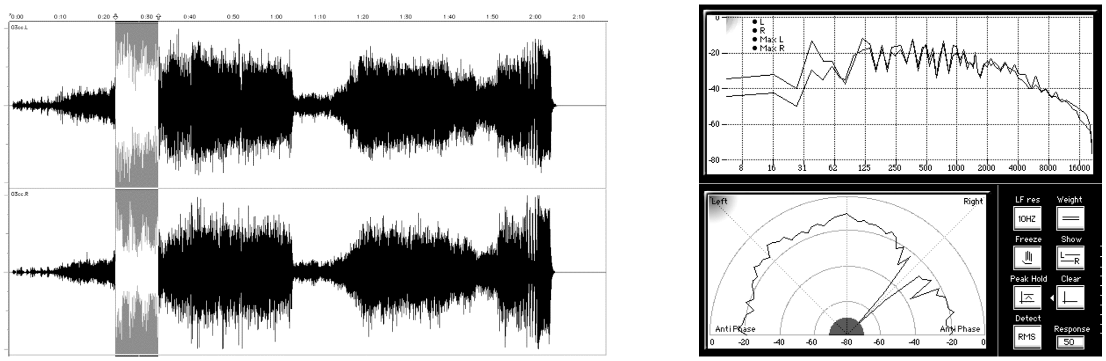
00HI - choral, women’s voices



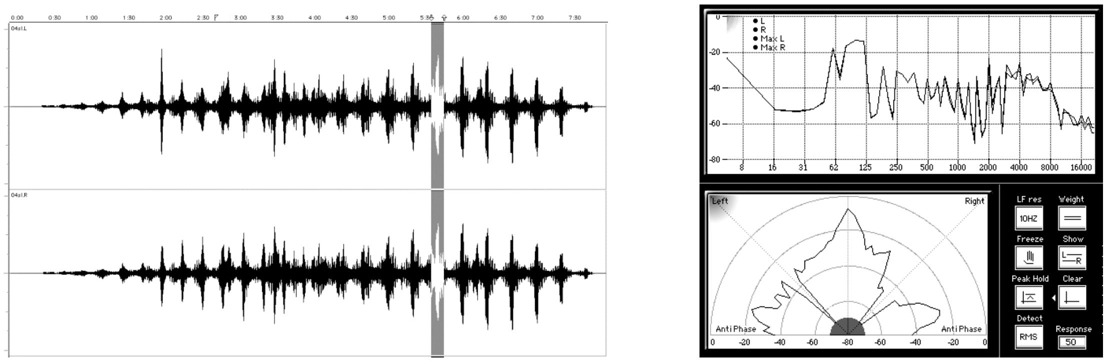
01CE - solo cello, Bach suite



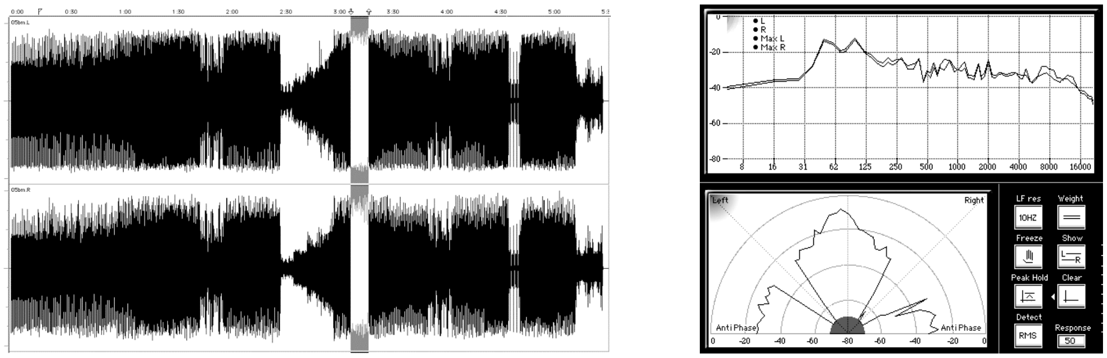
02BE - orchestra symphony, multiple tempos, monophonic



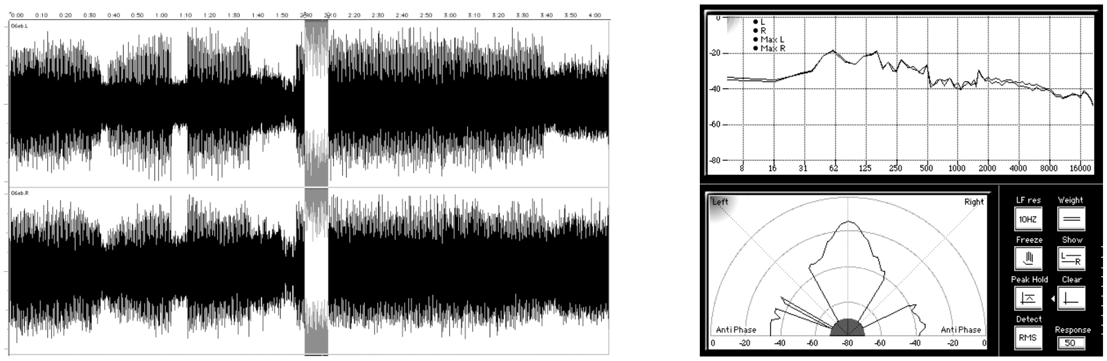
03CC - orchestra ballet, loud and fast



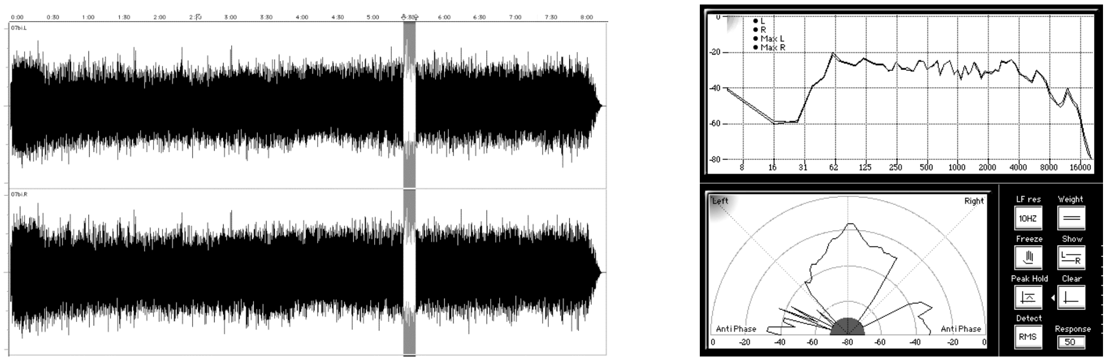
04SL - software synthesis



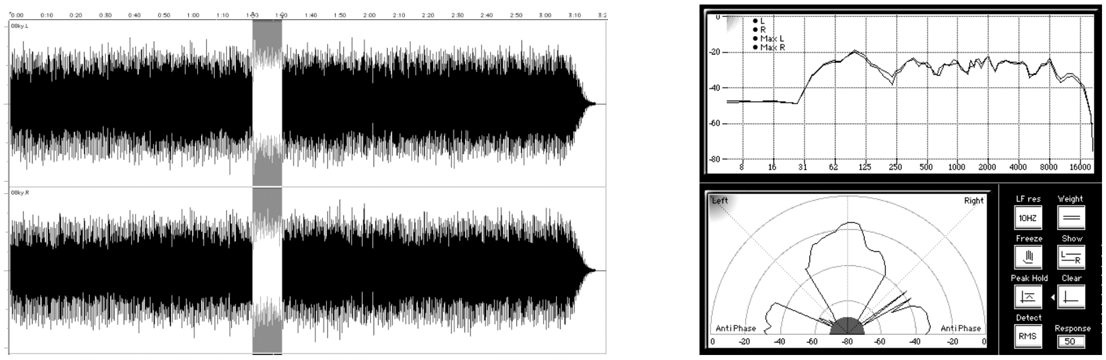
05BM - club techno



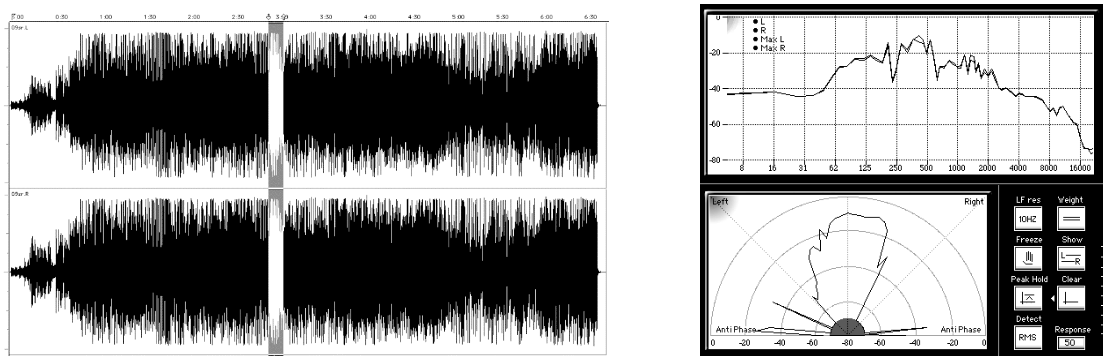
06EB - trance techno



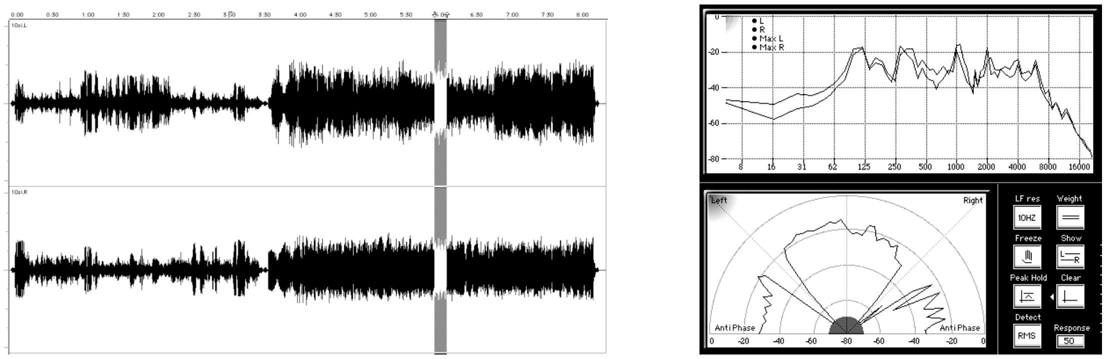
07BI - rock, heavy beat



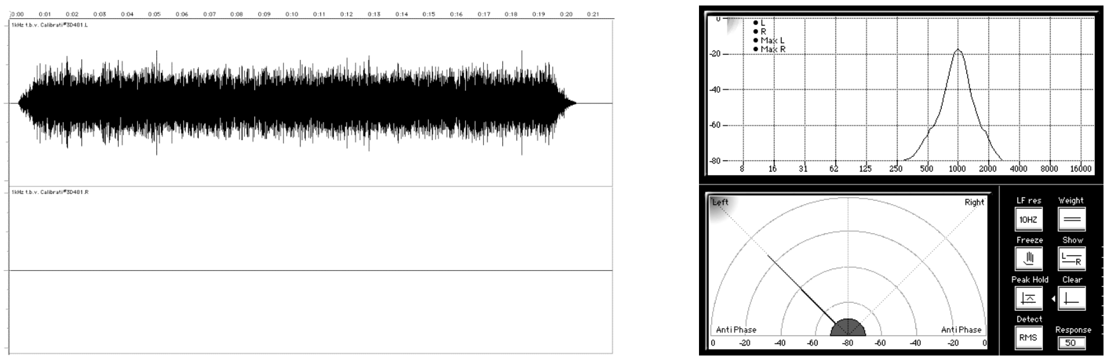
08KY - pop, heavy beat



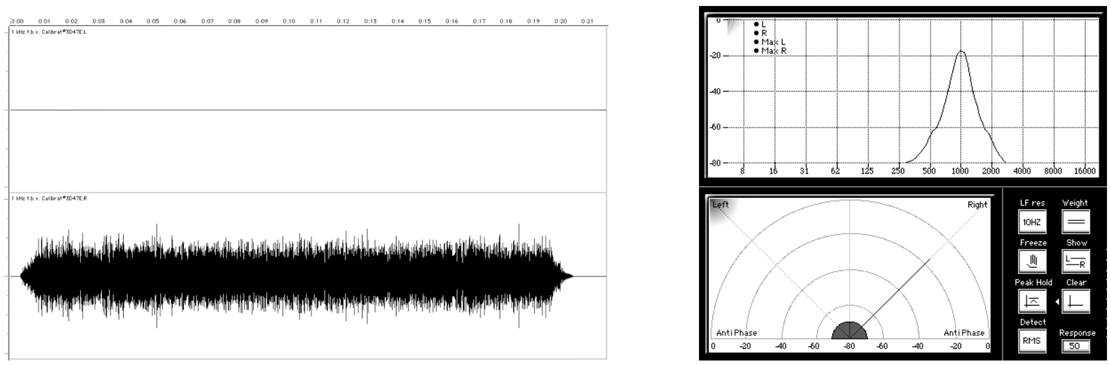
09SR - Indian classical, sitar and tabla



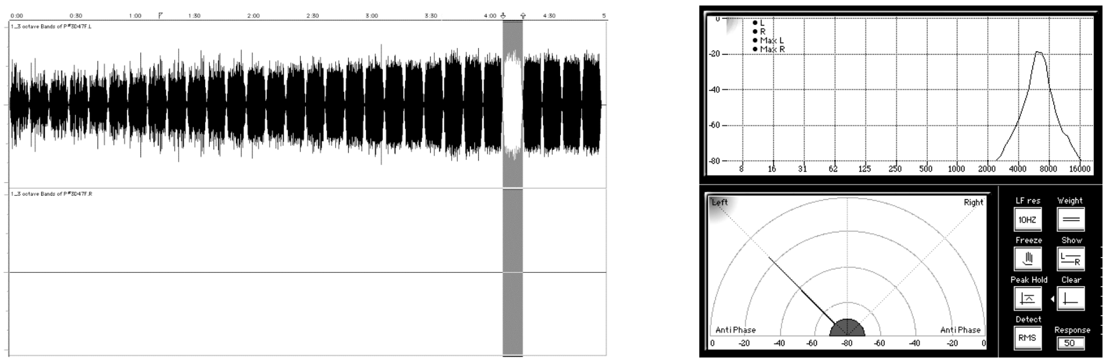
10SI - Indian classical, mandolin and mridangam



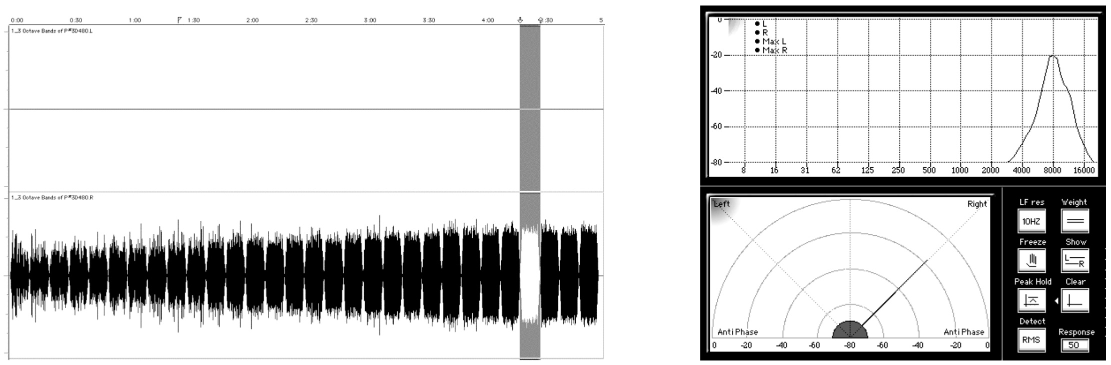
1 kHz t.b.v. calibration - left channel



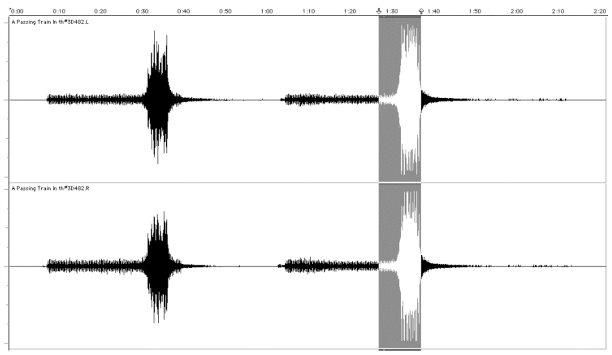
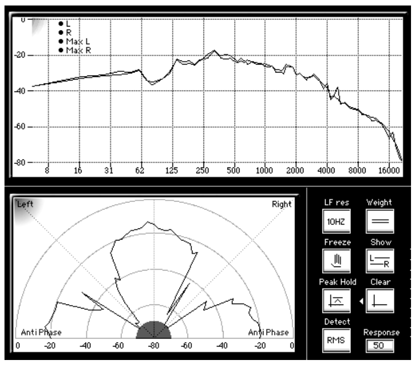
1 kHz t.b.v. calibration - right channel

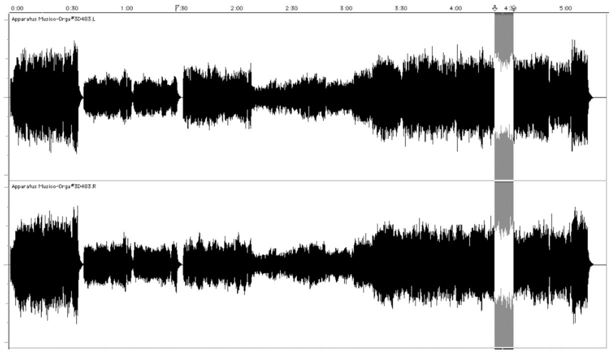
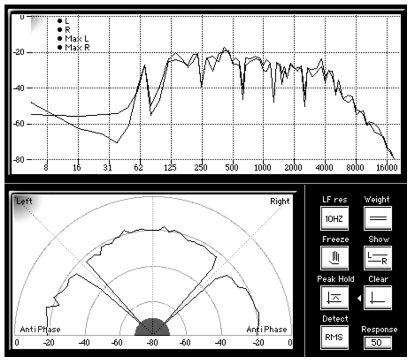


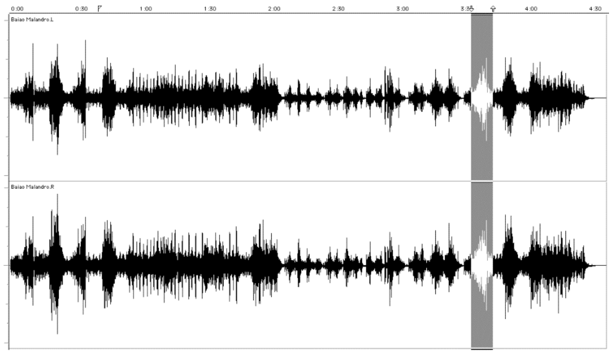
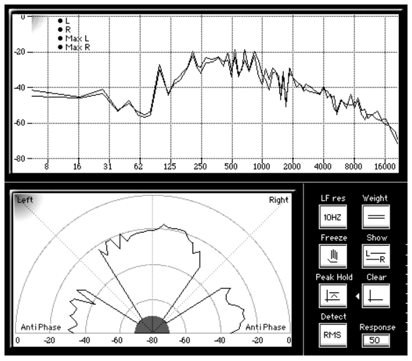
1/3 octave bands of pink noise - left channel

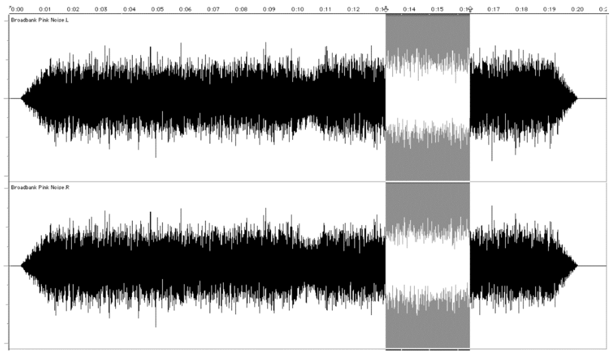
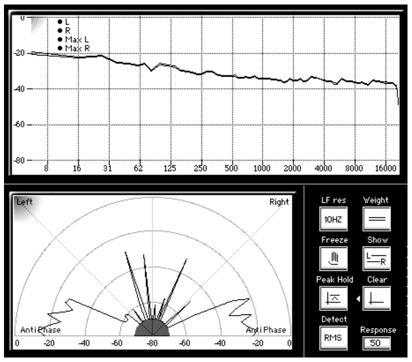


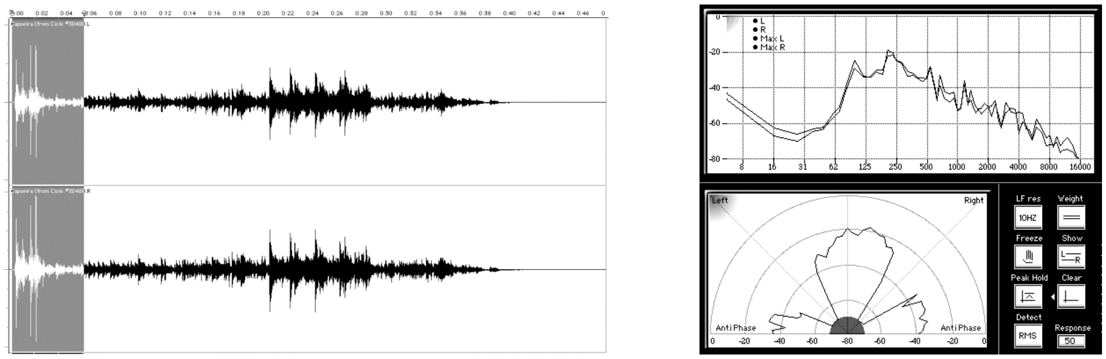
1/3 octave bands of pink noise - right channel



A Passing Train In the Quiet Dutch Farmlands - with crossing bell

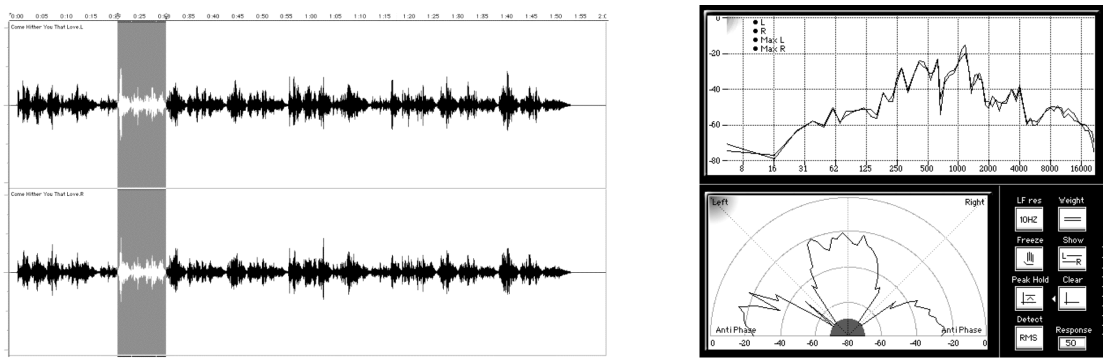


Apparatus Musico-Organisticus, Toccata 1a - pipe organ



Baiao Malandro - classical guitar duet

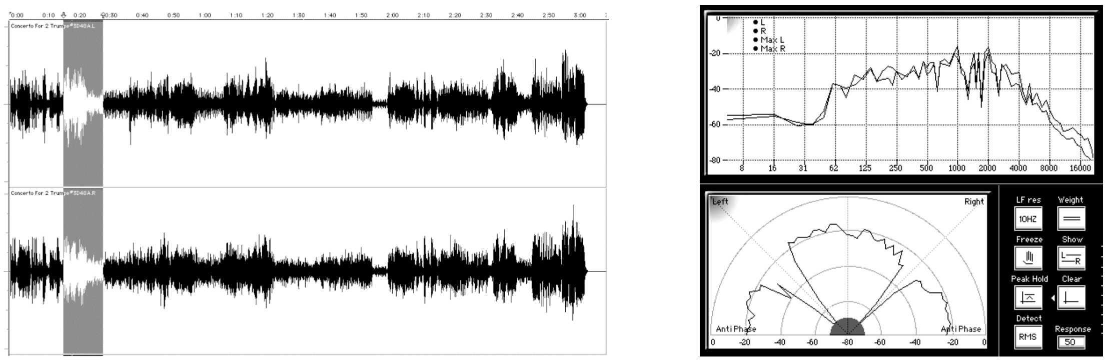


Broadband Pink Noise - second half is out-of-phase



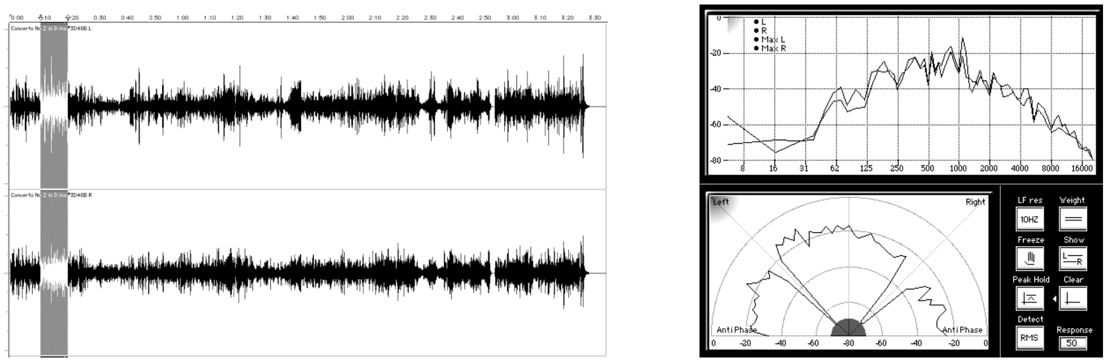
Capoeira (from Ciclo Nordestino No 3) - classical guitar duet, quiet and rhythmic



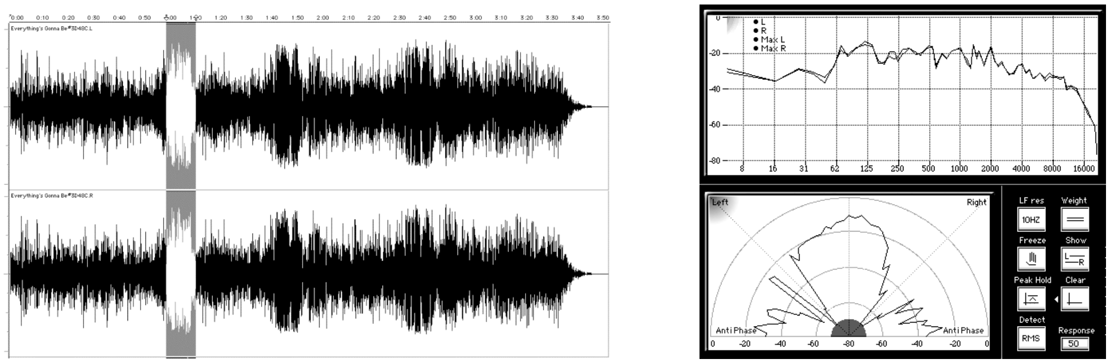
Come Hither You That Love - quiet lute and soprano

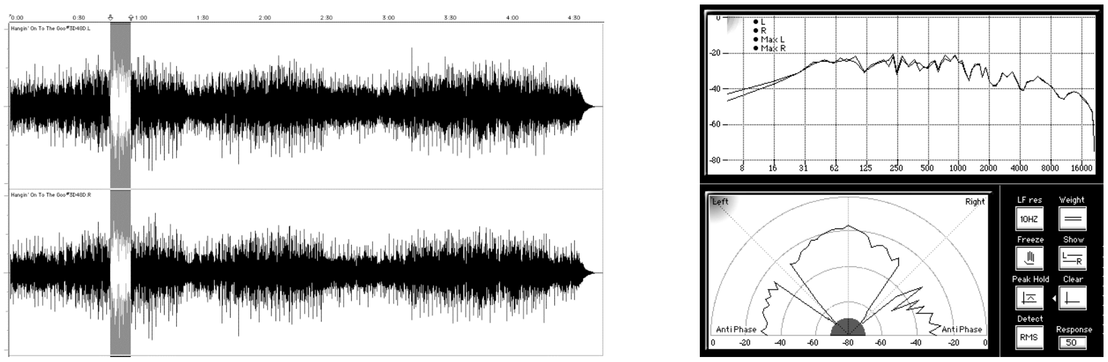


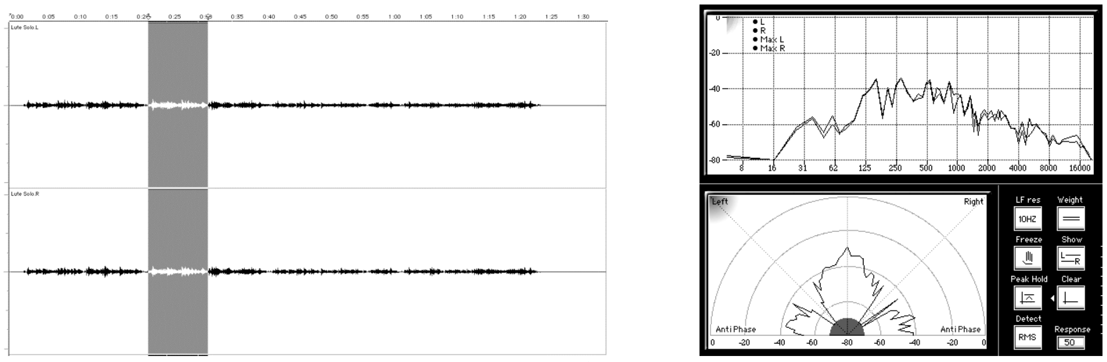
Concerto For 2 Trumpets & Strings, RV 537, C Major, Allegro - baroque

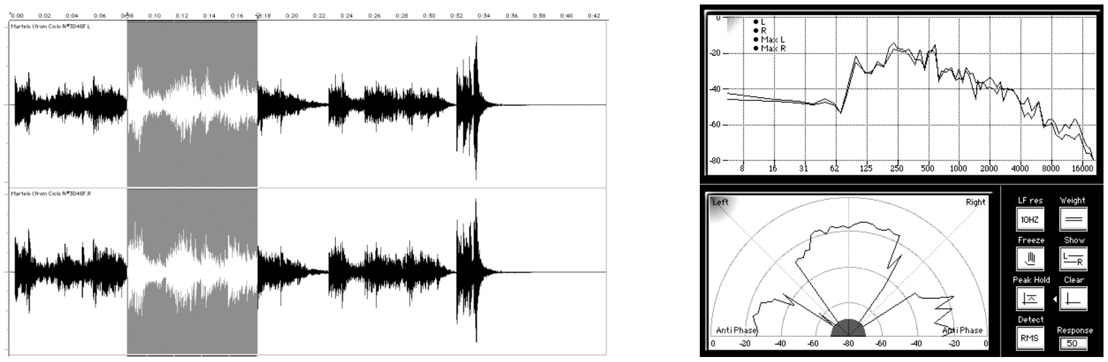


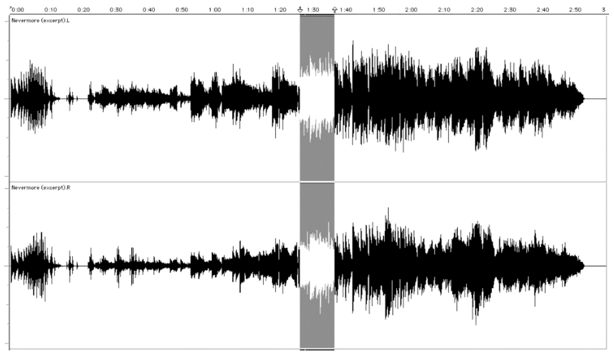
Concerto No 2 in D major, Allegro - baroque flute, harpsichord, and strings

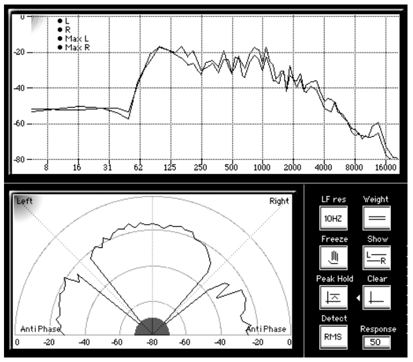

Everything's Gonna Be All Right - electric guitar and woman vocalist


Hangin' On To The Good Times - light pop with male vocalist and heavy bass

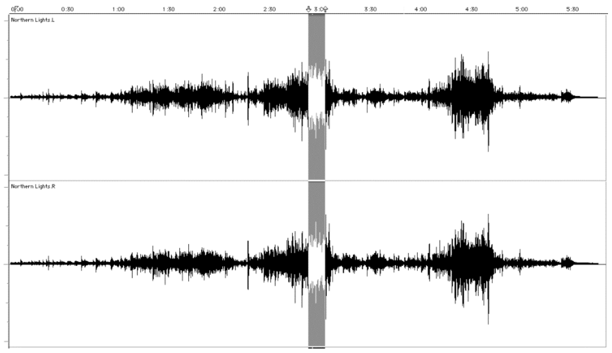

Lute Solo

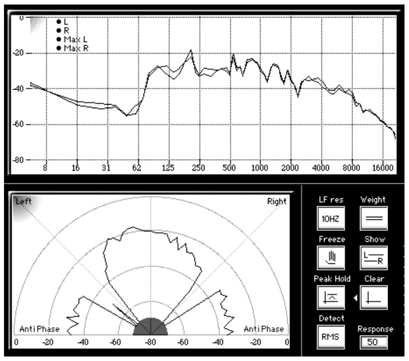

Martelo (from Ciclo Nordestino No. 1) - fast classical guitar duet



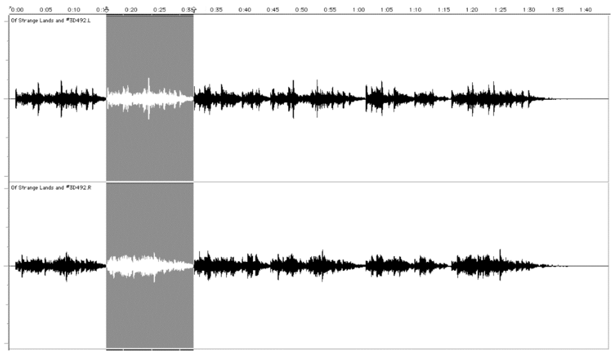
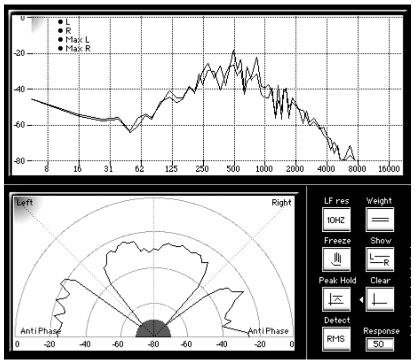


Nevermore (excerpt) - keyboard generated sampled piano, flute, and cello

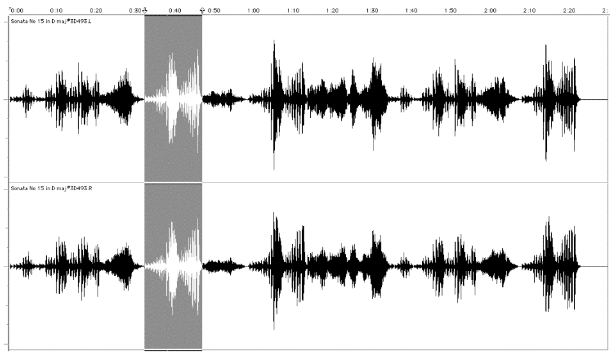
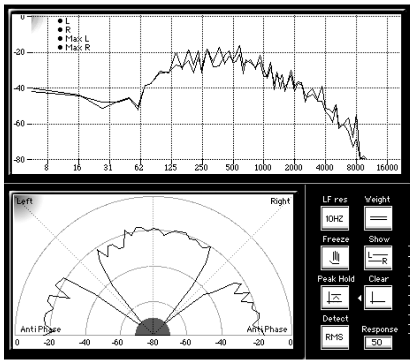




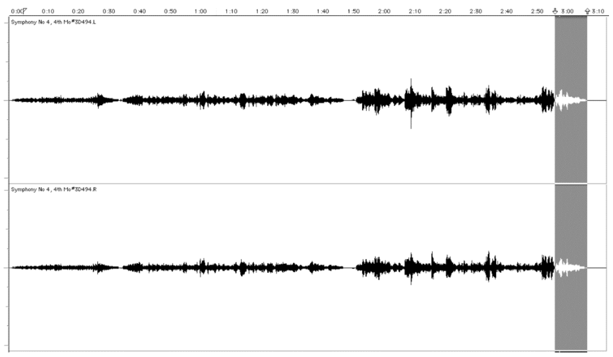
Northern Lights - quiet electric instruments and male vocalist

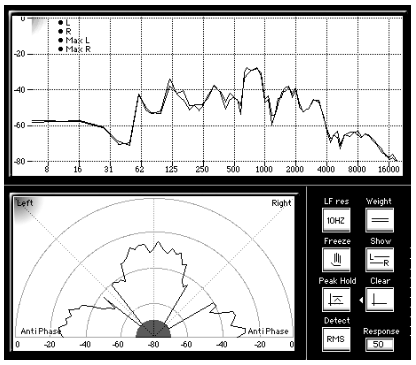



Of Strange Lands and People, Scenes From Childhood, Op. 15 - gentle classical piano solo

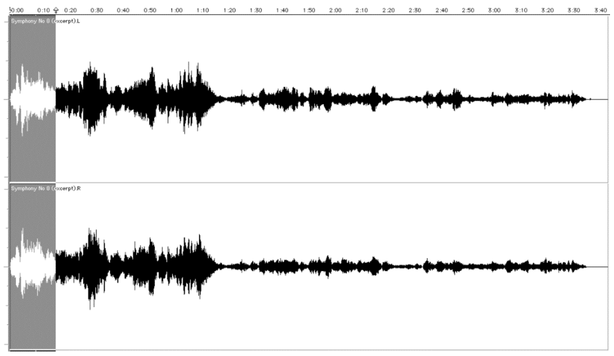



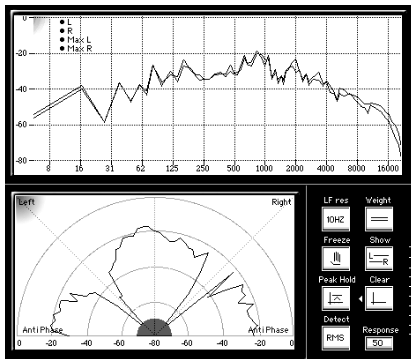
Sonata No 15 in D major, Op. 28 “Pastoral”, Scherzo-Allegro Vivace - dynamic classical piano solo





Symphony No 4, 4th Movement (excerpt) - quiet orchestra with female soloist





Symphony No 8 (excerpt) - loud organ, choir, large orchestra; quiets and adds soloists

B Header for 2-bit scan

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/*
These are the structures used to gather 2 bits at a time.
*/

#ifndef  _H_RWlc2bit
#define  _H_RWlc2bit
#pragma once

const SInt32 chunkMult    = 8 * 4096;  //  =32768, should be 4k aligned for best performance on OSX

const SInt32 maxBlock = 32768;         //  2^15; This is the number used to govern the maximum size
                                       //      of the "block".

//  All values are converted to a long int before examination.
//const long columnMask   = 0b00000000000000000000000000000011;
const SInt32 columnMask   = 3;
const SInt32 maskSize = 2;

#pragma options align=mac68k

typedef struct zeroRepeat {
    UInt16   usePattern:1;    //  0 here means to use this whole structure (don't use pattern)
    UInt16   count:15;        //  amount to use '00'; 0 means one, 1 means two, 2 means three, etc., max=32768
} zeroRepeat;                 //  Counts of <=16 will never be used. Those will fit in bitRepeat.

typedef struct bitRepeat {
    Byte     usePattern:1;    //  1 here means to use this whole structure (use a pattern)
    Byte     useLiteral:1;    //  0 here means to use the next two bits as pattern
    Byte     pattern:2;       //  bit pattern to repeat; 00 01 10 11
    Byte     count:4;         //  amount to use pattern; 0 means one, 1 means two, 2 means three, etc., max=16
} bitRepeat;

//typedef struct bitLiteralData {
//  Byte     p1:2;
//  Byte     p2:2;
//  Byte     p3:2;
//  Byte     p4:2;
//} bitLiteralData;       //  CAN'T USE IN STRUCT BELOW. Compiler pads struct boundary to even address!!!!

typedef struct bitLiteral {
    Byte     usePattern:1;    //  1 here means to use this whole structure (must be a one, use pattern)
    Byte     useLiteral:1;    //  1 here means to use these next members (must be a one, spell out each literally)
    Byte     count:6;         //  number of literal bytes (4 pairs of bits); 0 means one, 1 means two, 2 means three, etc.
    Byte     bits[64];        //  up to 64 of these bytes, corresponding to pairs from 256 samples
} bitLiteral;

#pragma options align=reset

//  Flag definitions set true for correlating process applied to block.
const char   deltaFlag    = 0x01 << 0;
const char   sumDifFlag   = 0x01 << 1;
const char   wpFlag       = 0x01 << 2;
const char   anotherFlag  = 0x01 << 3;
//  certainly more to come...

#endif   //  _H_RWlc2bit
```

C Encode

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////
void
RWlcEncode::Process  ( FSSpec &inSrcSpec, FSSpec &inDestSpec )
{
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////
    //  set up source struct and open source file
    SoundFile    sourceFile ( inSrcSpec );
    sourceFile.Open(fsRdPerm);

    //  set up destination structure
    SoundFile   destFile(inDestSpec);

    //  check for enough space on destination drive,
    //      return with error if not (set flag)
    FileInfoPB       srcInfo ( inSrcSpec );
    if ( !destFile.IsSpaceAvailable(srcInfo.GetSize() + 102400) )  // 100k free space
    {
        mDiskFull = true;
        SysBeep(0);
        return;
    }

    //  create destination file, same type as source, with QuickTime Player as the creator
    destFile.CreateAndOpen('TVOD', saveAsType,
                        sourceFile.SampleSize(), sourceFile.SampleRate(), sourceFile.Channels(), 'RWlc' );

    long sChans  = sourceFile.Channels(); //  copied out for readability

#if TEXT_OUTPUT
    unsigned long    debugCount   = 0;
    unsigned long    lastIndex    = 0;
    bool             bLitLastSaved    = false;
    unsigned long    bLitDebugCount   = 0;
    char             theBitString[64];

    ofstream textFile;
    textFile.open("RWlc.txt", ios::ate);
#endif

    ////////////////////////////////////////////////////////////////////////////////////////////////////////////
    //  calculate buffer sizes for reading from sources and write to destination
    SInt32   framesRead;
    SInt8    *bufferIn    = new SInt8[maxBlock * sourceFile.FrameSize()];    //  does this automatically clear the space?
    SInt32   *bufferLR    = new SInt32[maxBlock * sChans];
    SInt32   *bufferSD    = new SInt32[maxBlock * sChans];

    //  Worst case says output buffer needs to be 65/64ths (+1.5%) of the input
    //      plus 12.5% (per bit for 8-bit) for the correlating algorithms.
    SInt32   outSize      = maxBlock * sourceFile.FrameSize() * 2;
    Byte *bufferOut   = new Byte[outSize];

    ////////////////////////////////////////////////////////////////////////////////////////////////////////////
    //  while we can read one block of source data, run process, and write to file
    while ( (framesRead = sourceFile.ReadFrames((void *)bufferIn, maxBlock)) != 0 )
    {
#if TEXT_OUTPUT
        textFile << "\nNew block started (#" << (SInt32)debugCount++ << ").\n";
#endif

        SInt32   *bufferLong  = bufferLR;  //  This will point to one of the above depending on the outcome of SumDiff.

        ////////////////////////////////////////////////////////////////////////////////////////////////////////////
        //  convert all to 32 bit ints (for now even 8-bit files which only need to be up converted to 16-bit)
        //      This is to make room for any additional bit places generated by our correlation algorithms
        //      20-bit samples have the right most 4 bits set to zero. Shift so the smallest magnitude is 1.
        SInt32 sl;
        SInt32 ib=0;
        switch (sourceFile.SampleSize())
        {
            case 8:
            for(sl=0; sl<(framesRead * sChans); sl++)
                bufferLong[sl] = bufferIn[sl];
            break;

            case 16:
            SInt16 *shortCast = (SInt16 *) bufferIn;
```

```
                for(sl=0; sl<(framesRead * sChans); sl++)
                    bufferLong[sl] = shortCast[sl];
                break;

                case 20:
                for(sl=0; sl<(framesRead * sChans); sl++)
                {
                    //  take the long pointed to by 'ib', then arithmetic shift to the right (keeps only the left 2.5 bytes)
                    bufferLong[sl] = *((SInt32 *)(bufferIn+ib)) >> 12;
                    ib += 3;
                }
                break;

                case 24:
                for(sl=0; sl<(framesRead * sChans); sl++)
                {
                    //  take the long pointed to by 'ib', then arithmetic shift to the right (keeps only the left 3 bytes)
                    bufferLong[sl] = *((SInt32 *)(bufferIn+ib)) >> 8;
                    ib += 3;
                }
                break;

                default:
                //  some error report
                break;
            }

            //  This sets the number of bits to examine minus the size of the mask.
            //      Each massaging of the stream adds a bit significance to each sample.
            //      We don't need to examine all 32 bits of the temp buffer. The extra bits are just there to catch the overflow.
            long maskLimit = sourceFile.SampleSize();

            //  clear output buffer so bit-wise OR's work
            for (SInt32 b=0; b<outSize; b+=8)
                *(UInt64*)&bufferOut[b]   = 0LL;   //  This could leave up to the last 3 bytes uncleared.


            //  put a 3 zero-byte marker for the start of the block, this pattern never occurs as part of the compression
            //*(UInt32*)&bufferOut[0] = 0L;    //  Actually write 4 zeros. This insures the forth byte is cleared for the flags.
            //  this was cleared above.


            ////////////////////////////////////////////////////////////////////////////////////////////////////////////
            //  Correlating algorithms!

            //  take running difference on each channel
//          for ( SInt32 d = (framesRead*sChans)-1; d >= sChans; d-- )
//              bufferLong[d] -= bufferLong[d-sChans];
//          bufferOut[3] |= deltaFlag;
//          maskLimit++;

            //  difference from the last difference plus the last value (like wavpack 'fast')
            for ( SInt32 d = (framesRead*sChans)-1; d >= sChans*2; d-- )
                bufferLong[d] -= (2*bufferLong[d-sChans]) - bufferLong[d-(sChans*2)];
            for ( SInt32 d = (sChans*2)-1; d >= sChans; d-- )
                bufferLong[d] -= bufferLong[d-sChans];
            bufferOut[3] |= wpFlag;
            maskLimit++;

            //  Test the sum and difference of stereo files to see if taking this computation will shrink the files.
            if ( sChans == 2 )
            {
                //UInt64 magLR = 0, magSD = 0;

                for ( UInt32 s = 0; s < framesRead * sChans; s += 2 )
                {
                    bufferSD[s]      = bufferLR[s] + bufferLR[s+1];
                    bufferSD[s+1]    = bufferLR[s] - bufferLR[s+1];
                    magLR    += bufferLR[s] < 0   ? -bufferLR[s]   : bufferLR[s];
                    magLR    += bufferLR[s+1] < 0 ? -bufferLR[s+1] : bufferLR[s+1];
                    magSD    += bufferSD[s] < 0   ? -bufferLR[s]   : bufferLR[s];
                    magSD    += bufferSD[s+1] < 0 ? -bufferLR[s+1] : bufferLR[s+1];
                }

                if (magSD < magLR)
                {
                    bufferLong   = bufferSD;      //  change to point to data in bufferSD
                    bufferOut[3] |= sumDifFlag;
                    maskLimit++;
                }
```

```
        }


        //////////////////////////////////////////////////////////////////////////////////////////////////////////
        //  Complement and rotate
        for ( SInt32 cr = 0; cr < framesRead * sChans; cr++ )
            bufferLong[cr]   = (bufferLong[cr] & 0x80000000) ? ~(bufferLong[cr] << 1) : bufferLong[cr] << 1;
        //  shift pads right bit with a 0, comp set it to 1
        //  This is the only transform that doesn't add another bit to the sample.

        //  Before starting to encode the data test to be sure the mask limit is not greater than 30.
        if (maskLimit > 30)
            throw (-1);

        //  put the size of the target block
        *(SInt16*)&bufferOut[4]   = framesRead-1;
#if TEXT_OUTPUT
        textFile << BitString(&bufferOut[0], 2, theBitString) << "\tBlock Size: " << framesRead << endl;
#endif

        long bOutIndex    = 6;

        //  for each channel
        for (long ch=0; ch<sChans; ch++ ) //  for each channel
        {
#if TEXT_OUTPUT
            textFile << "New channel started (" << (ch?"right":"left") << ").\n";
#endif
            for ( SInt32 maskPosition = 0; maskPosition < maskLimit; maskPosition += maskSize )
            {
#if TEXT_OUTPUT
                textFile << "Mask position (" << maskPosition << ").\n";
#endif
                SInt32   frmIndex = 0; //  source sample index
                UInt32   bitMask      = columnMask << maskPosition;

                bitLiteral       bLit;
                bLit.usePattern      = 1; //  these two are always 1 for this data type
                bLit.useLiteral      = 1;

                bool     prevLiteralAvailable = false; //  was last chunk a literal or one of the repeaters

                bitRepeat    bRep;
                bRep.usePattern = 1;
                bRep.useLiteral  = 0;

                //for (long bc=0; bc<64; bc++)
                //  bLit.bits[bc] = 0;        //  set pattern to save to all zeros

                //////////////////////////////////////////////////////////////////////////////////////////////////////////
                //  scan these pairs of bits in all the samples of this channel
                while ( frmIndex < framesRead )
                {
                    //  grabbing 32 bits saves doing a bit shift at every compare
                    SInt32   firstSample = bufferLong[(frmIndex * sChans)+ch] & bitMask;
                    SInt32   pCount = 1;  //  pattern count (one found so far)

                    //  check for repeating bits
                    while ( ((bufferLong[((frmIndex+pCount)*sChans)+ch] & bitMask) == firstSample)
                                && (pCount < framesRead-frmIndex) )
                        pCount++;

                    //  write out literal if we are switching to a repeater or if literal is full
                    if ( ((bLit.count == 63) || (pCount > 4)) && prevLiteralAvailable)
                    {
                        ::BlockMove(&bLit, &bufferOut[bOutIndex], bLit.count+2);
#if TEXT_OUTPUT
                        textFile << BitString(&bufferOut[bOutIndex], 1, theBitString) << "\t\t" << bLit.count+1 << "\n";
                        for (SInt32 tf=1; tf<bLit.count+2; tf++)
                            textFile << "\t" << BitString(&bufferOut[bOutIndex+tf], 1, theBitString) << "\n";
#endif
                        bOutIndex += (bLit.count+2);
                        prevLiteralAvailable = false;

                        //for (long bc=0; bc<64; bc++)
                        //  bLit.bits[bc] = 0;   //  clear bLit space
                    }

                    //  save the tally of bits in the proper (smallest) format
                    if (pCount > 16 && firstSample == 0)  //  only '00' can have more than 16 to a run
                    {
                        *(SInt16*)&bufferOut[bOutIndex] = (SInt16) pCount-1;    //  same as zeroRepeat data type
#if TEXT_OUTPUT
                        textFile << BitString(&bufferOut[bOutIndex], 2, theBitString) << "\t" << pCount << "\n";
#endif
                        bOutIndex += 2;
                        frmIndex += pCount;
                        pCount = 0;
                    }
                    else if ( pCount > 4 || (pCount==4 && !prevLiteralAvailable) )
                    {
                        bRep.pattern = firstSample >> maskPosition;
                        pCount          = pCount>16 ? 16 : pCount;    //  limit run of pattern other than '00' to max 16
                        bRep.count      = pCount-1;

                        bufferOut[bOutIndex] = *((Byte*)&bRep);
#if TEXT_OUTPUT
                        textFile << BitString(&bufferOut[bOutIndex], 1, theBitString) << "\t\t" << pCount << "\n";
#endif
                        bOutIndex++;
                        frmIndex += pCount;
                        pCount = 0;
                    }
                    else//  else just save the next 4 literal pairs
                    {
                        if (prevLiteralAvailable)
                            bLit.count++;
                        else
                        {
                            bLit.count              = 0;
                            prevLiteralAvailable = true;
                        }

                        bLit.bits[bLit.count]  = (Byte)((bufferLong[(frmIndex*sChans)+ch] & bitMask) >> maskPosition) << 6;
                        frmIndex++;
                        bLit.bits[bLit.count] |= (Byte)((bufferLong[(frmIndex*sChans)+ch] & bitMask) >> maskPosition) << 4;
                        frmIndex++;
                        bLit.bits[bLit.count] |= (Byte)((bufferLong[(frmIndex*sChans)+ch] & bitMask) >> maskPosition) << 2;
                        frmIndex++;
                        bLit.bits[bLit.count] |= (Byte)((bufferLong[(frmIndex*sChans)+ch] & bitMask) >> maskPosition);
                        frmIndex++;

                        pCount = 0;
                    }
                }

                if (prevLiteralAvailable) //  copy remaining literal to output buffer
                {
                    ::BlockMove(&bLit, &bufferOut[bOutIndex], bLit.count+2);
#if TEXT_OUTPUT
                    textFile << BitString(&bufferOut[bOutIndex], 1, theBitString) << "\n";
                    for (SInt32 tf=1; tf<bLit.count+2; tf++)
                        textFile << "\t" << BitString(&bufferOut[bOutIndex+tf], 1, theBitString) << "\n";
#endif
                    bOutIndex += (bLit.count+2);
                }
            }
        }

        destFile.PutBytes(bufferOut, bOutIndex);
    }
    destFile.UpdateHeader(sourceFile.GetFrameMarker());    //  make the size the same as what we just read

    delete bufferIn;
    delete bufferLR;
    delete bufferSD;
    delete bufferOut;
}

#if TEXT_OUTPUT
char* BitString(void *inData, unsigned long inSize, char *ioStr)    // returns ioStr as a C string
{
    Bytemask= 0b10000000;
    Byte*inCast  = (Byte *) inData;

    for (unsigned long b = 0; b<inSize; b++)
    {
        for(unsigned long bt = 0; bt<8; bt++)
        {
            ioStr[(b*8)+bt] = (inCast[b] & (mask >> bt)) ? '1' : '0';
        }
    }
    ioStr[inSize*8] = 0;
    return ioStr;
}
#endif
```

D Decode

```
///////////////////////////////////////////////////////////////////////////////////////////////////////////////////
void
RWlcDecode::Process  ( FSSpec &inSrcSpec, FSSpec &inDestSpec )
{
////////////////////////////////////////////////////////////////////////  file management    //////////////////
    //  set up source struct and open source file
    SoundFile    sourceFile ( inSrcSpec );
    sourceFile.Open(fsRdPerm);

    //  must be compressed with my algorithm
    if (sourceFile.Format() != 'RWlc')
    {
        SysBeep(0);
        return;
    }

    //  set up destination class
    SoundFile   destFile(inDestSpec);

    //  check for enough space on destination drive,
    //      return with error if not (set flag)
    FileInfoPB   srcInfo ( inSrcSpec );
    if ( !destFile.IsSpaceAvailable(srcInfo.GetSize() + 102400) )  // 100k free space
    {
        mDiskFull = true;
        SysBeep(0);
        return;
    }

    //  create destination file, same type as source, with QuickTime Player as the creator
    destFile.CreateAndOpen('TVOD', saveAsType, sourceFile.SampleSize(), sourceFile.SampleRate(), sourceFile.Channels() );

//////////////////////////////////////////////////////////////////////    set up memory      ////////////////////////
    //  declare buffers and counters
    SInt32   bytesRead;
    SInt32   framesInBlock;

    SInt32   sChans   = sourceFile.Channels(); //  copied out for readability

    UInt8    *bufferIn    = new UInt8[chunkMult];
    SInt32   *bufferLong  = new SInt32[(chunkMult * sChans)+4];            //  plus space for bitLiteral to write past end
    SInt8    *bufferOut   = new SInt8[chunkMult * sourceFile.FrameSize()];

    SInt32   zeroRepeatRun;
    SInt32   literalCount;

    //  initialize values
    SInt32   frmIndex     = 0;
    states   state        = firstZero;
    SInt32   ch           = 0;
    UInt8    correlation  = 0;

    for (SInt32 cl=0; cl<chunkMult * sChans; cl+=2)
        *(SInt64*)&bufferLong[cl] = 0LL;

    //  the mask starts with the 2 bits on the right and moves left
    UInt32   maskPosition;
    UInt32   maskLimit;


/////////////////////////////////////////////////////////////////////     start conversion   ////////////////////
    //  read source data, run process, and write to file
    while ( (bytesRead = sourceFile.ReadBytes((void *)bufferIn, chunkMult)) != 0 ) // THIS COULD READ PAST THE AUDIO DATA
    {
        //  for each byte in bufferIn
        for ( SInt32 b=0; b<bytesRead; b++ )
        {
            switch ( state )
            {
                case firstZero:
                if ( bufferIn[b] == 0 )
                    state    = secondZero;
                //  else still looking for firstZero, something was wrong
                //      (ie. we're not at the start of a block for streaming)
                break;
```

```
case secondZero:
if ( bufferIn[b] == 0 )
    state    = thirdZero;
else
    state    = firstZero; //  if not, start over looking for first zero
break;


case thirdZero:
if ( bufferIn[b] == 0 )
    state    = corrFlags; //  we've found the three zero bytes, this is the start of a block
else
    state    = firstZero; //  bad data, start looking for start of block again
break;


case corrFlags:
//  copy correlation flags
correlation  = bufferIn[b];

//  the mask starts with the 2 bits on the right and moves left
maskPosition = 0L;
maskLimit    = destFile.SampleSize();

//  each correlation algorithm requires one more bit
for ( Byte m=0x01; m>0x00; m<<=1 )
    if (correlation & m)
        maskLimit++;
state    = blockLen1;
break;


case blockLen1:
framesInBlock    = ((SInt32)bufferIn[b]) << 8;     //  copy high-byte of short int (clears lower byte)
state    = blockLen2;
break;


case blockLen2:
framesInBlock    |= ((SInt32)bufferIn[b]);         //  copy low-byte of short
framesInBlock++; //  plus one to make up for offset
state    = nextChunk;
break;


case nextChunk:
switch ( bufferIn[b] & 0b11000000 )                //  view only top two bits
{
    case 0b00000000: //  00
    case 0b01000000: //  01
    zeroRepeatRun    = ((SInt32)bufferIn[b]) << 8; //  copy high-byte of short
    state    = zeroRep;
    break;

    case 0b10000000: //  10, do run of pattern now
    //for ( long p=0; p<=((bitRepeat*)&bufferIn[b])->count; p++ )   //  p<=count is same as p<count+1
    SInt32   theBits = ((SInt32)(bufferIn[b] & 0b00110000) >> 4) << maskPosition;
    for ( SInt32 p=0; p<=(bufferIn[b] & 0b00001111); p++ ) //  p<=count is same as p<count+1
    {
        bufferLong[(frmIndex*sChans)+ch] |= theBits;
        frmIndex++;
    }

    state    = nextChunk;
    break;

    case 0b11000000:
    literalCount = bufferIn[b] & 0b00111111;       //  same as ((bitLiteral*)&bufferIn[b])->count
    state    = literalRun;
    break;
}
break;


case zeroRep:
zeroRepeatRun    |= ((SInt32)bufferIn[b]);         //  copy low-byte of short
zeroRepeatRun++;
state    = nextChunk;
```

```
        //  we don't need to write the zeros, they're already there
        //  just move up the frame count
        frmIndex += zeroRepeatRun;
        break;


        case literalRun:
        //  copy out literal data
        bufferLong[(frmIndex*sChans)+ch]  |= ((SInt32)( bufferIn[b]               >> 6)) << maskPosition;
        frmIndex++;
        bufferLong[(frmIndex*sChans)+ch]  |= ((SInt32)((bufferIn[b] & 0b00110000) >> 4)) << maskPosition;
        frmIndex++;
        bufferLong[(frmIndex*sChans)+ch]  |= ((SInt32)((bufferIn[b] & 0b00001100) >> 2)) << maskPosition;
        frmIndex++;
        bufferLong[(frmIndex*sChans)+ch]  |= ((SInt32) (bufferIn[b] & 0b00000011)      ) << maskPosition;
        frmIndex++;

        if ( literalCount == 0 )
            state    = nextChunk;
        literalCount--;
        break;
    }


    //  test to change to new bit column
    if ( frmIndex >= framesInBlock /*&& state != blockLen1*/)
    {
        frmIndex = 0;
        maskPosition += maskSize;

        //  test for new channel
        if ( maskPosition >= maskLimit )
        {
            maskPosition = 0L;
            ch++;

            //  If all channels are done then we're at the end of a block.
            if ( ch >= sChans )
            {
                ch = 0;

                //  Rotate and complement.
                //  if left most byte is a 1 then this will produce an error (bit?)
                for ( SInt32 cr=0; cr<framesInBlock*sChans; cr++)
                    bufferLong[cr]   = (bufferLong[cr] & 1L) ? ~(bufferLong[cr] >> 1) : bufferLong[cr] >> 1;

                //  Go through de-correlation algorithms in reverse order from the encoding.

                //  Sum-Diff (only for 2 channel signals [for now]).
                if (correlation & sumDifFlag)
                {
                    switch (sChans)
                    {
                        case 2:
                        //  do process in place (same buffer)
                        for ( SInt32 sd=0; sd<framesInBlock*sChans; sd+=2) //  (s)um-(d)iff
                        {
                            SInt32 sum   = bufferLong[sd];
                            SInt32 diff  = bufferLong[sd+1];
                            bufferLong[sd]   = (sum + diff) >> 1; //  divide by 2
                            bufferLong[sd+1] = (sum - diff) >> 1;
                        }
                        break;

                        default: //  other combinations of channels might be tried for 3 or more channels
                        break;
                    }
                }

                //  Delta.
                if (correlation & deltaFlag)
                    for ( SInt32 id=sChans; id<framesInBlock*sChans; id++) //  (i)ndex (d)elta
                        bufferLong[id]   += bufferLong[id-sChans];

                SInt32 sl;
                SInt32 ib=0;
                switch (sourceFile.SampleSize())
                {
                    case 8:
                    for(sl=0; sl<(framesInBlock * sChans); sl++)
```

```
                                bufferOut[sl] = bufferLong[sl];
                            break;

                            case 16:
                            short *shortCast = (short *) bufferOut;
                            for(sl=0; sl<(framesInBlock * sChans); sl++)
                                shortCast[sl] = bufferLong[sl];
                            break;

                            case 20:
                            for(sl=0; sl<(framesInBlock * sChans); sl++)
                            {
                                bufferLong[sl]   <<= 4;
                                Byte*longByte    = (Byte  *) &bufferLong[sl];
                                bufferOut[ib++]  = longByte[1];       //  zero, the high byte is not used
                                bufferOut[ib++]  = longByte[2];
                                bufferOut[ib++]  = longByte[3];
                            }
                            break;

                            case 24:
                            for(sl=0; sl<(framesInBlock * sChans); sl++)
                            {
                                Byte*longByte    = (Byte  *) &bufferLong[sl];
                                bufferOut[ib++]  = longByte[1];       //  zero, the high byte is not used
                                bufferOut[ib++]  = longByte[2];
                                bufferOut[ib++]  = longByte[3];
                            }
                            break;

                            default:
                            //  some error report
                            break;
                        }

                        destFile.WriteFrames(bufferOut, framesInBlock);

                        //  clear buffer (write zeros) for next block
                        for (SInt32 cl=0; cl<chunkMult * sChans; cl+=2)
                            *(SInt64*)&bufferLong[cl] = 0LL;

                        state   = firstZero;
                    }
                }
            }
        }
    }

    delete bufferIn;
    delete bufferLong;
    delete bufferOut;
}
```

E Results

file		cp		gzip only						gzip w/ delta					
name	size	time	t/M	size	%	encode		decode		size	%	encode		decode	
00hi.aif Choral	55918906	1.84	0.35	36888938	66.0	11.94	2.26	6.41	1.21	28594286	51.1	17.87	3.38	5.66	1.07
01ce.aif Solo cello	177312682	7.91	0.47	155205456	87.5	28.9	1.73	22.34	1.33	118959287	67.1	42.38	2.53	20.74	1.24
02be.aif Orchestra	43582666	1.48	0.36	35594753	81.7	8.55	2.08	5.13	1.25	30061992	69.0	9.23	2.24	4.74	1.15
03cc.aif Ballet	24395050	0.59	0.26	20857970	85.5	4.37	1.90	2.87	1.25	17807626	73.0	5.13	2.23	2.93	1.27
04sl.aif Software synthesis	85029610	3.54	0.44	67665617	79.6	16.68	2.08	10.41	1.30	51037353	60.0	21.15	2.63	9.09	1.13
05bm.aif Club techno	59136442	1.89	0.34	55418957	93.7	11.42	2.04	7.32	1.31	50434803	85.3	11.61	2.08	6.3	1.13
06eb.aif Rampant trance techno	43998970	1.21	0.29	40950540	93.1	8.71	2.10	2.35	0.57	35629858	81.0	9.48	2.28	4.92	1.18
07bi.aif Rock	88853962	3.46	0.41	81328151	91.5	17.2	2.05	3.89	0.46	72540016	81.6	19.08	2.27	9.92	1.18
08ky.aif Pop	35896330	1.03	0.30	33169246	92.4	7.25	2.14	1.28	0.38	30907717	86.1	7.07	2.08	4	1.18
09sr.aif Indian classical 1	71693770	3.29	0.49	67314627	93.9	14.09	2.08	3.5	0.52	55139377	76.9	15.82	2.34	7.78	1.15
10si.aif Indian classical 2	89411386	3.61	0.43	78853909	88.2	17.87	2.12	4.81	0.57	71570366	80.0	19.7	2.33	9.87	1.17
1 kHz t.b.v. Calibration- Left Channel.aif	3880854	0.12	0.33	2191314	56.5	1.72	4.69	0.29	0.79	1769781	45.6	1.7	4.64	0.22	0.60
1 kHz t.b.v. Calibration- Right Channel.aif	3883064	0.08	0.22	2233364	57.5	1.65	4.50	0.28	0.76	1822309	46.9	1.68	4.58	0.33	0.90
1_3 Octave Bands of Pink Noise.aif	53451468	1.73	0.34	30567068	57.2	21.82	4.32	5.83	1.15	22127746	41.4	17.28	3.42	4.83	0.96
1_3 octave Bands of Pink Noise- Left Channel.aif	53451454	1.95	0.39	30557093	57.2	21.77	4.31	5.75	1.14	22116928	41.4	17.35	3.44	4.79	0.95
A Passing Train In the Quiet Dutch Farmlands???.a	25316894	0.30	0.13	16826555	66.5	5.14	2.15	2.84	1.19	12557140	49.6	6.39	2.67	2.1	0.88
Apparatus Musico-Organisticus, Toccata 1a.aif	57685084	2.00	0.37	50367829	87.3	11.17	2.05	7.39	1.36	43914047	76.1	13.21	2.42	6.43	1.18
Baiao Malandro.aif	49217864	2.23	0.48	41742152	84.8	10.12	2.18	2.28	0.49	31319824	63.6	12.45	2.68	5.36	1.15
Broadbank Pink Noise.aif	3765444	0.08	0.22	3319749	88.2	0.77	2.16	0.23	0.65	3217274	85.4	0.81	2.28	0.36	1.01
Capoeira (from Ciclo Nordestino No 3).aif	8563564	0.13	0.16	6223917	72.7	1.69	2.09	0.46	0.57	4496600	52.5	2.3	2.84	0.81	1.00
Come Hither You That Love.aif	21523102	0.27	0.13	17340610	80.6	4.45	2.19	0.77	0.38	14164229	65.8	5.47	2.69	2.22	1.09
Concerto For 2 Trumpets & Strings, RV 537, C Maj	33694726	1.59	0.50	29207135	86.7	6.63	2.08	1.59	0.50	25096246	74.5	7.84	2.46	3.59	1.13
Concerto No 2 in D major, Allegro.aif	38281084	1.13	0.31	32754992	85.6	7.68	2.12	1.3	0.36	26890786	70.2	9.22	2.55	4.25	1.18
Everything's Gonna Be All Right.aif	41545622	1.24	0.32	37282997	89.7	8.32	2.12	1.42	0.36	30921099	74.4	9.47	2.41	4.47	1.14
Hangin' On To The Good Times.aif	50805440	4.95	1.03	45637738	89.8	9.8	2.04	3.01	0.63	37989128	74.8	11.66	2.43	5.61	1.17
Lute Solo.aif	16760254	1.59	1.00	11448268	68.3	3.44	2.17	1.09	0.69	8044680	48.0	4.79	3.02	1.58	1.00
Martelo (from Ciclo Nordestino No. 1).aif	7674506	0.66	0.91	5907548	77.0	1.54	2.12	0.41	0.57	4525428	59.0	1.91	2.63	0.73	1.01
Nevermore (excerpt).aif	32107062	3.45	1.14	26468462	82.4	6.42	2.12	1.08	0.36	22248495	69.3	8.14	2.68	3.36	1.11
Northern Lights.aif	62624250	6.73	1.14	49307363	78.7	12.89	2.18	3.01	0.51	36578243	58.4	16.68	2.82	6.08	1.03
Of Strange Lands and People, Scenes From Childh	18611324	1.70	0.97	14463127	77.7	3.72	2.12	0.81	0.46	10620759	57.1	4.96	2.82	1.78	1.01
Sonata No 15 in D major, Op. 28 _Pastoral_, Scher	26815208	2.65	1.05	21798999	81.3	5.31	2.10	0.95	0.37	16169031	60.3	6.77	2.67	2.55	1.01
Symphony No 4, 4th Movement (excerpt).aif	34482630	3.67	1.13	26018813	75.5	7.31	2.24	1.25	0.38	19958713	57.9	9.29	2.85	3.45	1.06
Symphony No 8 (excerpt).aif	39868698	4.30	1.14	32241100	80.9	8.12	2.16	1.35	0.36	26258917	65.9	10.21	2.71	4.3	1.14
TOTAL	1.459E+09	72.40	0.53	1.207E+09	82.7	308.46	2.24	113.7	0.82	985490084	67.5	358.1	2.60	155.15	1.13

Table 6a. Results for cp, gzip only, and gzip with delta encoding.

file		slac alone						slac w/delta					
name	size	size	%	encode		decode		size	%	encode		decode	
00hi.aif Choral	55918906	35703339	63.8	7.13	1.35	5.67	1.07	30255451	54.1	6.29	1.19	5.28	1.00
01ce.aif Solo cello	177312682	130909484	73.8	22.38	1.34	19.03	1.14	99449984	56.1	20.08	1.20	17.89	1.07
02be.aif Orchestra	43582666	27736777	63.6	5.23	1.27	4.48	1.09	23804354	54.6	4.89	1.19	4.12	1.00
03cc.aif Ballet	24395050	19634100	80.5	3.2	1.39	2.61	1.13	16566626	67.9	2.92	1.27	2.55	1.11
04sl.aif Software synthesis	85029610	57779407	68.0	10.51	1.31	8.78	1.09	46481490	54.7	9.87	1.23	8.25	1.03
05bm.aif Club techno	59136442	52251841	88.4	8.48	1.52	6.83	1.22	47630496	80.5	7.9	1.41	7.11	1.27
06eb.aif Rampant trance techno	43998970	37812467	85.9	5.77	1.39	4.93	1.19	33617040	76.4	5.7	1.37	4.91	1.18
07bi.aif Rock	88853962	76253636	85.8	12.25	1.46	10.07	1.20	67812148	76.3	11.88	1.42	10.03	1.19
08ky.aif Pop	35896330	32362702	90.2	5.17	1.52	4.09	1.21	30067607	83.8	5.08	1.50	4.14	1.22
09sr.aif Indian classical 1	71693770	58937748	82.2	9.09	1.34	8.06	1.19	46982316	65.5	8.6	1.27	7.72	1.14
10si.aif Indian classical 2	89411386	74315469	83.1	12.9	1.53	9.81	1.16	65520794	73.3	12.01	1.42	9.76	1.16
1 kHz t.b.v. Calibration- Left Channel.aif	3880854	1564683	40.3	0.49	1.34	0.19	0.52	1238264	31.9	0.47	1.28	0.2	0.55
1 kHz t.b.v. Calibration- Right Channel.aif	3883064	1616759	41.6	0.51	1.39	0.2	0.55	1300958	33.5	0.47	1.28	0.19	0.52
1_3 Octave Bands of Pink Noise.aif	53451468	20759070	38.8	5.34	1.06	4.53	0.90	16328474	30.5	4.84	0.96	4.34	0.86
1_3 octave Bands of Pink Noise- Left Channel.aif	53451454	20750671	38.8	5.05	1.00	4.11	0.81	16316850	30.5	4.59	0.91	4.23	0.84
A Passing Train In the Quiet Dutch Farmlands???.a	25316894	14235630	56.2	3.13	1.31	2.28	0.95	11057384	43.7	2.57	1.07	2.09	0.87
Apparatus Musico-Organisticus, Toccata 1a.aif	57685084	48827675	84.6	8.26	1.52	6.27	1.15	41440645	71.8	7.5	1.38	6.36	1.17
Baiao Malandro.aif	49217864	34908972	70.9	6.27	1.35	5.07	1.09	26165276	53.2	5.71	1.23	4.87	1.05
Broadbank Pink Noise.aif	3765444	3264771	86.7	0.49	1.38	0.29	0.82	2977349	79.1	0.51	1.43	0.27	0.76
Capoeira (from Ciclo Nordestino No 3).aif	8563564	5158282	60.2	1.26	1.56	0.79	0.98	3770374	44.0	0.95	1.17	0.6	0.74
Come Hither You That Love.aif	21523102	15204020	70.6	2.64	1.30	2.06	1.01	12348747	57.4	2.35	1.16	1.99	0.98
Concerto For 2 Trumpets & Strings, RV 537, C Maj	33694726	26323419	78.1	4.61	1.45	3.68	1.16	22363043	66.4	4.21	1.32	3.54	1.11
Concerto No 2 in D major, Allegro.aif	38281084	28753728	75.1	5.09	1.41	4.15	1.15	23413383	61.2	4.7	1.30	3.87	1.07
Everything's Gonna Be All Right.aif	41545622	32789813	78.9	5.55	1.41	4.35	1.11	27767607	66.8	5.15	1.31	4.38	1.12
Hangin' On To The Good Times.aif	50805440	40470915	79.7	7.19	1.50	5.41	1.13	34082321	67.1	6.38	1.33	4.98	1.04
Lute Solo.aif	16760254	9301722	55.5	1.94	1.23	1.44	0.91	7208986	43.0	1.66	1.05	1.28	0.81
Martelo (from Ciclo Nordestino No. 1).aif	7674506	5073713	66.1	1.1	1.52	0.64	0.88	3822768	49.8	0.87	1.20	0.54	0.74
Nevermore (excerpt).aif	32107062	25075791	78.1	4.24	1.40	3.55	1.17	20455885	63.7	3.93	1.30	3.1	1.02
Northern Lights.aif	62624250	41603524	66.4	8.35	1.41	6.62	1.12	32965446	52.6	6.87	1.16	5.76	0.97
Of Strange Lands and People, Scenes From Childh	18611324	11967849	64.3	2.16	1.23	1.91	1.09	8518538	45.8	2.05	1.17	1.45	0.82
Sonata No 15 in D major, Op. 28 _Pastoral_, Scher	26815208	18100951	67.5	3.35	1.32	2.74	1.08	13228385	49.3	2.99	1.18	2.19	0.86
Symphony No 4, 4th Movement (excerpt).aif	34482630	22141275	64.2	4.28	1.31	3.29	1.01	17762170	51.5	3.81	1.17	3.06	0.94
Symphony No 8 (excerpt).aif	39868698	28433988	71.3	5.05	1.34	3.95	1.05	23016877	57.7	4.6	1.22	3.57	0.95
TOTAL	1.459E+09	1.06E+09	72.6	188.46	1.37	151.88	1.10	875738036	60.0	172.4	1.25	144.62	1.05

Table 6b. Results for SLAC with no decorrelation, and with delta encoding.

file		slac w/wpFast						flac -0					
name	size	size	%	encode		decode		size	%	encode		decode	
00hi.aif Choral	55918906	29051026	52.0	6.34	1.20	5.47	1.04	21047970	37.6	4.05	0.77	5.29	1.00
01ce.aif Solo cello	177312682	89236238	50.3	18.85	1.13	16.87	1.01	75200532	42.4	12.29	0.73	18.05	1.08
02be.aif Orchestra	43582666	23343243	53.6	4.97	1.21	3.98	0.97	24376447	55.9	3.4	0.83	5.07	1.23
03cc.aif Ballet	24395050	15965050	65.4	2.95	1.28	2.45	1.06	14235912	58.4	2.8	1.21	2.58	1.12
04sl.aif Software synthesis	85029610	43664922	51.4	9.26	1.15	7.53	0.94	36325332	42.7	8.5	1.06	8.62	1.07
05bm.aif Club techno	59136442	47726370	80.7	7.72	1.38	6.82	1.22	43949202	74.3	7.18	1.29	7.61	1.36
06eb.aif Rampant trance techno	43998970	33547446	76.2	5.59	1.34	4.74	1.14	30367169	69.0	5.52	1.33	5.35	1.29
07bi.aif Rock	88853962	65331051	73.5	11.38	1.36	9.44	1.12	58032630	65.3	10.59	1.26	10.82	1.29
08ky.aif Pop	35896330	29784524	83.0	4.84	1.43	4.28	1.26	26820677	74.7	4.44	1.31	4.51	1.33
09sr.aif Indian classical 1	71693770	40904081	57.1	7.97	1.18	7.03	1.04	38656259	53.9	7.89	1.16	7.93	1.17
10si.aif Indian classical 2	89411386	60250542	67.4	11.16	1.32	8.77	1.04	51308077	57.4	10.36	1.23	9.9	1.17
1 kHz t.b.v. Calibration- Left Channel.aif	3880854	940130	24.2	0.47	1.28	0.16	0.44	667733	17.2	0.43	1.17	0.36	0.98
1 kHz t.b.v. Calibration- Right Channel.aif	3883064	1009092	26.0	0.42	1.14	0.17	0.46	708914	18.3	0.53	1.44	0.36	0.98
1_3 Octave Bands of Pink Noise.aif	53451468	14045228	26.3	4.77	0.94	3.71	0.73	11658078	21.8	4.36	0.86	4.45	0.88
1_3 octave Bands of Pink Noise- Left Channel.aif	53451454	14032276	26.3	4.6	0.91	3.79	0.75	11651042	21.8	4.23	0.84	4.01	0.79
A Passing Train In the Quiet Dutch Farmlands???.a	25316894	10540021	41.6	2.54	1.06	1.91	0.80	8877209	35.1	2.66	1.11	2.44	1.02
Apparatus Musico-Organisticus, Toccata 1a.aif	57685084	37664922	65.3	6.91	1.27	5.68	1.04	32508059	56.4	6.69	1.23	6.13	1.12
Baiao Malandro.aif	49217864	21482996	43.6	5.12	1.10	4.06	0.87	18245407	37.1	4.97	1.07	4.79	1.03
Broadbank Pink Noise.aif	3765444	3108558	82.6	0.63	1.77	0.29	0.82	2982954	79.2	0.61	1.71	0.44	1.24
Capoeira (from Ciclo Nordestino No 3).aif	8563564	3120670	36.4	0.83	1.03	0.47	0.58	2682263	31.3	0.92	1.14	0.76	0.94
Come Hither You That Love.aif	21523102	11610634	53.9	2.33	1.15	1.86	0.91	9924867	46.1	2.37	1.17	2.19	1.08
Concerto For 2 Trumpets & Strings, RV 537, C Maj	33694726	19846807	58.9	3.82	1.20	3.1	0.97	16399931	48.7	3.59	1.13	3.56	1.12
Concerto No 2 in D major, Allegro.aif	38281084	20794102	54.3	4.15	1.15	3.45	0.95	17275305	45.1	4.17	1.15	3.8	1.05
Everything's Gonna Be All Right.aif	41545622	26917461	64.8	5.03	1.28	4.19	1.07	24384017	58.7	4.76	1.21	4.6	1.17
Hangin' On To The Good Times.aif	50805440	32863438	64.7	6.46	1.35	4.82	1.00	28817311	56.7	5.55	1.16	5.98	1.25
Lute Solo.aif	16760254	6609844	39.4	1.78	1.12	1.19	0.75	5679886	33.9	1.69	1.07	1.61	1.02
Martelo (from Ciclo Nordestino No. 1).aif	7674506	3223969	42.0	0.71	0.98	0.53	0.73	2805063	36.6	0.87	1.20	0.85	1.17
Nevermore (excerpt).aif	32107062	17654616	55.0	3.85	1.27	2.84	0.94	13801949	43.0	3.35	1.10	3.3	1.09
Northern Lights.aif	62624250	31366390	50.1	6.82	1.15	5.46	0.92	26395824	42.1	6.65	1.12	6.32	1.07
Of Strange Lands and People, Scenes From Childh	18611324	6566405	35.3	1.79	1.02	1.31	0.74	5395618	29.0	1.89	1.07	1.69	0.96
Sonata No 15 in D major, Op. 28 _Pastoral_, Scher	26815208	10960762	40.9	2.63	1.04	2.03	0.80	8544867	31.9	2.66	1.05	2.22	0.88
Symphony No 4, 4th Movement (excerpt).aif	34482630	16318208	47.3	3.66	1.12	2.86	0.88	13604149	39.5	3.56	1.09	3.23	0.99
Symphony No 8 (excerpt).aif	39868698	20507520	51.4	4.33	1.15	3.46	0.92	17459269	43.8	4.19	1.11	4.23	1.12
TOTAL	1.459E+09	809988542	55.5	164.68	1.19	134.72	0.98	700789922	48.0	147.72	1.07	153.05	1.11

Table 6c. Results for SLAC with WavPack fast encoding and FLAC with option "-0".

file		flac -8						WavPack -f					
name	size	size	%	encode		decode		size	%	encode		decode	
00hi.aif Choral	55918906	19781507	35.4	39.75	7.52	5.17	0.98	20625476	36.9	6.48	1.23	5.91	1.12
01ce.aif Solo cello	177312682	73732170	41.6	126.59	7.56	17.16	1.02	73095180	41.2	20.46	1.22	19.04	1.14
02be.aif Orchestra	43582666	17835781	40.9	31.05	7.54	4.41	1.07	18061710	41.4	4.99	1.21	5.13	1.25
03cc.aif Ballet	24395050	13538417	55.5	17.63	7.65	2.7	1.17	13729824	56.3	3.09	1.34	3.18	1.38
04sl.aif Software synthesis	85029610	34691660	40.8	60.3	7.51	8.98	1.12	36537236	43.0	9.77	1.22	9.26	1.15
05bm.aif Club techno	59136442	40256556	68.1	43.11	7.72	7.3	1.31	41265946	69.8	8.2	1.47	7.99	1.43
06eb.aif Rampant trance techno	43998970	29419852	66.9	32.24	7.76	5.44	1.31	29431580	66.9	5.65	1.36	5.79	1.39
07bi.aif Rock	88853962	52726291	59.3	64.38	7.67	10.01	1.19	53929020	60.7	11.08	1.32	11.38	1.36
08ky.aif Pop	35896330	25174854	70.1	26.11	7.70	4.71	1.39	25434892	70.9	4.78	1.41	4.88	1.44
09sr.aif Indian classical 1	71693770	33231706	46.4	51.54	7.61	7.22	1.07	32377318	45.2	9.67	1.43	8.3	1.23
10si.aif Indian classical 2	89411386	46166424	51.6	64.16	7.59	9.72	1.15	46324118	51.8	10.8	1.28	10.43	1.23
1 kHz t.b.v. Calibration- Left Channel.aif	3880854	655037	16.9	2.3	6.27	0.37	1.01	1233968	31.8	0.67	1.83	0.45	1.23
1 kHz t.b.v. Calibration- Right Channel.aif	3883064	696794	17.9	2.3	6.27	0.38	1.04	1246190	32.1	0.64	1.74	0.48	1.31
1_3 Octave Bands of Pink Noise.aif	53451468	9810047	18.4	28.26	5.60	4.22	0.84	20387766	38.1	5.96	1.18	5.5	1.09
1_3 octave Bands of Pink Noise- Left Channel.aif	53451454	9795553	18.3	27.92	5.53	3.63	0.72	20395024	38.2	6.19	1.23	5.45	1.08
A Passing Train In the Quiet Dutch Farmlands???.a	25316894	8345075	33.0	17.11	7.15	2.12	0.89	8543938	33.7	3.63	1.52	2.56	1.07
Apparatus Musico-Organisticus, Toccata 1a.aif	57685084	30898128	53.6	39.99	7.34	6.25	1.15	30348078	52.6	7.14	1.31	7.11	1.30
Baiao Malandro.aif	49217864	17984231	36.5	33.63	7.23	4.5	0.97	17781630	36.1	5.53	1.19	4.92	1.06
Broadbank Pink Noise.aif	3765444	2552663	67.8	2.83	7.95	0.52	1.46	2617764	69.5	0.61	1.71	0.51	1.43
Capoeira (from Ciclo Nordestino No 3).aif	8563564	2648615	30.9	5.7	7.04	0.8	0.99	2633968	30.8	1.02	1.26	0.97	1.20
Come Hither You That Love.aif	21523102	9568497	44.5	14.65	7.20	2.26	1.11	9691908	45.0	2.66	1.31	2.48	1.22
Concerto For 2 Trumpets & Strings, RV 537, C Maj	33694726	15913373	47.2	23.39	7.35	3.79	1.19	15734670	46.7	4.17	1.31	4.23	1.33
Concerto No 2 in D major, Allegro.aif	38281084	16853072	44.0	26.79	7.41	3.9	1.08	16587168	43.3	4.57	1.26	4.33	1.20
Everything's Gonna Be All Right.aif	41545622	22468194	54.1	28.89	7.36	4.66	1.19	23025398	55.4	5.25	1.34	5.12	1.30
Hangin' On To The Good Times.aif	50805440	27792107	54.7	35.78	7.45	5.5	1.15	28035000	55.2	6.28	1.31	6.55	1.36
Lute Solo.aif	16760254	5489756	32.8	11.25	7.10	1.47	0.93	5602944	33.4	1.98	1.25	1.71	1.08
Martelo (from Ciclo Nordestino No. 1).aif	7674506	2776668	36.2	5	6.90	0.74	1.02	2763548	36.0	0.88	1.21	0.77	1.06
Nevermore (excerpt).aif	32107062	13329995	41.5	21.88	7.21	2.89	0.95	13022718	40.6	3.89	1.28	3.39	1.12
Northern Lights.aif	62624250	24987809	39.9	43.23	7.31	6.63	1.12	25354640	40.5	7.18	1.21	6.6	1.12
Of Strange Lands and People, Scenes From Childh	18611324	5245224	28.2	12.35	7.02	1.54	0.88	5282460	28.4	2.09	1.19	1.89	1.07
Sonata No 15 in D major, Op. 28 _Pastoral_, Scher	26815208	8134639	30.3	18.23	7.20	2.18	0.86	8022964	29.9	2.79	1.10	2.53	1.00
Symphony No 4, 4th Movement (excerpt).aif	34482630	12887097	37.4	23.43	7.19	3.27	1.00	13172258	38.2	3.92	1.20	3.7	1.14
Symphony No 8 (excerpt).aif	39868698	16697146	41.9	27.22	7.23	4.37	1.16	16871912	42.3	4.8	1.27	4.24	1.13
TOTAL	1.459E+09	652084938	44.7	1009	7.32	148.81	1.08	679168214	46.5	176.82	1.28	166.78	1.21

Table 6d. Results for FLAC with option "-8" and WavPack fast setting.

file		WavPack -h							
name	size	size	%	encode		decode		enc (ls -r)	
00hi.aif Choral	55918906	20219622	36.2	14.18	2.68	8.22	1.56	16.77	3.17
01ce.aif Solo cello	177312682	71092468	40.1	32.37	1.93	26.63	1.59	36.14	2.16
02be.aif Orchestra	43582666	16844800	38.7	7.98	1.94	6.68	1.62	9.45	2.29
03cc.aif Ballet	24395050	13197188	54.1	4.72	2.05	4.21	1.83	5.61	2.43
04sl.aif Software synthesis	85029610	34201148	40.2	15.57	1.94	12.54	1.56	17.59	2.19
05bm.aif Club techno	59136442	39218860	66.3	11.54	2.07	9.85	1.76	12.7	2.27
06eb.aif Rampant trance techno	43998970	28620610	65.0	8.48	2.04	7.54	1.81	9.81	2.36
07bi.aif Rock	88853962	50473824	56.8	17.03	2.03	13.92	1.66	19.04	2.27
08ky.aif Pop	35896330	24403400	68.0	7.04	2.08	6.3	1.86	8.51	2.51
09sr.aif Indian classical 1	71693770	31453468	43.9	13.39	1.98	11	1.62	17.45	2.58
10si.aif Indian classical 2	89411386	44480598	49.7	17.17	2.03	14.01	1.66	18.81	2.23
1 kHz t.b.v. Calibration- Left Channel.aif	3880854	974642	25.1	0.95	2.59	0.7	1.91		
1 kHz t.b.v. Calibration- Right Channel.aif	3883064	997094	25.7	0.92	2.51	0.7	1.91		
1_3 Octave Bands of Pink Noise.aif	53451468	16742916	31.3	9.67	1.91	8.04	1.59		
1_3 octave Bands of Pink Noise- Left Channel.aif	53451454	16732726	31.3	9.72	1.92	7.73	1.53		
A Passing Train In the Quiet Dutch Farmlands???.a	25316894	8138458	32.1	4.66	1.95	3.74	1.56		
Apparatus Musico-Organisticus, Toccata 1a.aif	57685084	29298970	50.8	10.67	1.96	8.91	1.63		
Baiao Malandro.aif	49217864	17381370	35.3	9.48	2.04	7.34	1.58		
Broadbank Pink Noise.aif	3765444	2392636	63.5	0.94	2.64	0.71	2.00		
Capoeira (from Ciclo Nordestino No 3).aif	8563564	2566926	30.0	1.66	2.05	1.37	1.69		
Come Hither You That Love.aif	21523102	9372132	43.5	4.08	2.01	3.39	1.67		
Concerto For 2 Trumpets & Strings, RV 537, C Maj	33694726	15240956	45.2	6.43	2.02	5.09	1.60		
Concerto No 2 in D major, Allegro.aif	38281084	16143934	42.2	7.36	2.03	5.9	1.63		
Everything's Gonna Be All Right.aif	41545622	21611908	52.0	7.96	2.03	6.47	1.65		
Hangin' On To The Good Times.aif	50805440	26244744	51.7	9.76	2.03	8.2	1.71		
Lute Solo.aif	16760254	5393968	32.2	3.2	2.02	2.54	1.60		
Martelo (from Ciclo Nordestino No. 1).aif	7674506	2687014	35.0	1.51	2.08	1.16	1.60		
Nevermore (excerpt).aif	32107062	12383110	38.6	6.19	2.04	4.86	1.60		
Northern Lights.aif	62624250	24114800	38.5	11.45	1.94	9.58	1.62		
Of Strange Lands and People, Scenes From Childh	18611324	5011376	26.9	3.62	2.06	2.76	1.57		
Sonata No 15 in D major, Op. 28 _Pastoral_, Schei	26815208	7646346	28.5	4.83	1.91	3.92	1.55		
Symphony No 4, 4th Movement (excerpt).aif	34482630	12683546	36.8	6.58	2.02	5.38	1.65		
Symphony No 8 (excerpt).aif	39868698	16349608	41.0	7.36	1.95	6.14	1.63		
TOTAL	1.459E+09	644315166	44.2	278.47	2.02	225.53	1.64		

Table 6e. Results for WavPack high compression setting and files in reverse order.