APE Decoder Spec

Created by Liu Huaping

1. APE Decoder introduction(MAC)

APE is a audio lossless format file postfix, which can be decoded and outputted PCM data by Monkey's Audio Codec.
Monkey's Audio Codec web-site: http://www.monkeysaudio.com/index.html
You can get more details about the codec and C++ source codec

Monkey's Audio is a fast and easy way to compress digital music. Unlike traditional methods such as mp3, ogg, or lqt that permanently discard quality to save space, Monkey's Audio only makes *perfect*, bit-for-bit copies of your music. That means it always sounds *perfect* – exactly the same as the original. Even though the sound is perfect, it still saves a lot of space. (Think of it as a beefed-up Winzip™ for your music) The other great thing is that you can always decompress your Monkey's Audio files back to the exact, original files. That way, you'll never have to recopy your CD collection to switch formats, and you'll always be able to recreate the original music CD if something ever happens to yours.

2. MAC Theory

Digital Audio :
Sound is simply a wave, and digital audio is the digital representation of this wave. This is achieved by "sampling" the magnitude of an analog signal many times per second. This can be thought of conceptually as recording the "height" of the wave many times per second. Today's audio CD's store 44,100 samples per second. Since CD's are in stereo, they store both a left and a right value 44,100 times per second. These values are represented by 16 bit integers. Basically a WAV file is a header, followed by an array of R, L, R, L…. Since each sample takes 32 bits (16 for the left, and 16 for the right), and there are 44,100 samples per second, one second of audio takes 1,411,200 bits, or 176,400 bytes.
Lossless Compression:

1) Conversion to X,Y

The first step in lossless compression is to more efficiently model the channels L and R as some X and Y values. There is often a great deal of correlation between the L and R channels, and this can be exploited several ways, with one popular way being through the use of mid / side encoding. In this case, a mid (X) and a side (Y) value are encoded instead of a L and a R value. The mid (X) is the midpoint between the L and R channels and the side (Y) is the difference in the channels. This can be achieved:

$X = (L + R) / 2$

$Y = (L - R)$

2) Predictor

Next, the X and Y data is passed through a predictor to attempt to remove any redundancy. Basically, the goal of this stage is to make the X and Y arrays contain the smallest possible values while still

remaining decompressible. This stage is what separates one compression scheme from another. There are virtually countless ways to do this. Here is a sample using simple linear algebra:

PX and PY are the predicted X and Y; $X_{-1}$is the previous X value; X_{-2} is the X value two back

$PX = (2 * X_{-1}) - X_{-2}$

$PY = (2 * Y_{-1}) - Y_{-2}$

As an example, if X = (2, 8, 24, ?); $PX = (2 * X_{-1}) - X_{-2} = (2 * 24) - 8 = 40$

Then, these predicted values are compared with the actual value and the difference (error) is what gets sent to the next stage for encoding.

Most good predictors are adaptive, so that they adjust to how "predictable" the data currently is. For example, let's use a factor 'm' that ranges from 0 to 1024 (0 is no prediction and 1024 is full prediction). After each prediction, m is adjusted up or down depending on whether the prediction was helpful or not.. So in the previous example, what leaves the predictor is this

X = (2, 8, 24, ?)

$PX = (2 * X_{-1}) - X_{-2} = (2 * 24) - 8 = 40$

If ? = 45 and m = 512, then [Final Value] = ? - (PX * m / 1024) = 45 - (40 * m / 1024) = 45 - (40 * 512 / 1024) = 45 - 20 = 25

After this m would be adjusted upward because a higher m would have been more efficient Using different prediction equations and using multiple passes through the predictor can make a fairly substantial difference in compression level. Here is a quick list of some prediction equations as shown in the Shorten technical documentation (for different orders):

P0 = 0

$P1 = X_{-1}$

$P2 = (2 * X_{-1}) - X_{-2}$

$P3 = (3 * X_{-1}) - (3 *X_{-2}) + X_{-3}$

3) Encoding of Data / Rice coding

The goal behind audio compression is to make all of the numbers as small as possible by removing any correlation that may exist between them…once this is achieved the resulting numbers must be written to disk. One of (if not the) most efficient way to do this is with rice coding.

Why are smaller numbers better? They are better because they take less bits to represent. For example, say we want to encode this array of numbers (32 bit longs):

Base 10: 10, 14, 15, 46

or in binary

Base 2: 1010, 1110, 1111, 101110

Now obviously if we want to represent these numbers in the fewest possible bits, it would be quite inefficient to represent them each as separate longs with 32 bits apiece. That would take 128 bits, and just from looking at the same numbers represented in base two, it is obvious that there must be a better way. The ideal thing would be just to slap the four numbers together using the least bits necessary, so 1010, 1110, 1111, 101110 without the commas would be 101011101111101110. The problem here is that we don't know where one number starts and the next begins. This is where rice coding comes into play.

Rice coding is a way of using less bits to represent small numbers, while still maintaining the ability to tell one from the next. Basically it works like this:

1) You make your best guess as to how many bits a number will take, and call that k

2) Take the rightmost k bits of the number and remember what they are

3) Imagine the binary number without those rightmost k bits and look at its new value (this is the overflow that doesn't fit in k bits)

4) Use these values to encode the number… This encoded value is represented as a number of zeroes corresponding to step 3, then a terminating 1 to tell that your done sending the "overflow", then the k bits from step 2.

Let's work through our example, and try to encode the fourth number in our series 10, 14, 15, 46.

1) You make your best guess as to how many bits a number will take, and call that k: since the previous 3 numbers took 4 bits, that seems like a reasonable guess so we will set k = 4

2) Take the rightmost k bits of the number and remember what they are: The right 4 bits of 46 (101110) are 1110

3) Imagine the binary number without those rightmost k bits and look at its new value (this is the overflow that doesn't fit in k bits): When you take the 1110 away from the right of 101110 you are left with 10 or 2 (in base 10)

4) Use these values to encode the number… So, we put two 0's, followed by the terminating 1, followed by the k bits 1110…altogether we have 0011110

Now to undue this operation, we just take 0011110 and k = 4 and work our way backwards… We first see that the overflow is 2 (there are two zeroes before the terminating 1) We also see that the last four bits = 1110. So, we take the value 10 (the overflow) and the values 1110 (the k) and just do a little shifting and volah! (overflow is shifted << k bits)

Here is a little more technical and mathematical description of the same process:

Assuming some integer n is the number to encode, and k is the number of bits to encode directly.

1) sign (1 for positive, 0 for negative)

2) n / (2^k) 0's

3) terminating 1

4) k least significant bits of n

As an example, if n = 578 and k = 8: 100101000010

1) sign (1 for positive, 0 for negative) = [1]

2) n / (2^k) 0's: n / 2^k = 578 / 256 = 2 = [00]

3) terminating 1: [1]

4) k least significant bits of n: 578 = [01000010]

5) put the 1-4 together: [1][00][1][01000010] = 100101000010

During the encode process, the optimum k is determined by looking at the average value over the past however many values (16 - 128 works well), and choosing the optimum k for that average. (basically it's guessing what the next value will be, and trying to choose the most efficient k based on that) The optimum k can be calculated as [log(n) / log(2)].

3. APE Properties
   1) Channel number
      Unlike FLAC support Multi-Channels, APE only support MONO or 2 channels.

   2) APE Compression level
      APE has 5 compression levels, which can get the information form APE file Header
      Compression level == 1000 --- (fast)
      Compression level == 2000 --- (normal)
      Compression level == 3000 --- (high)
      Compression level == 4000 --- (extra high)
      Compression level == 5000 --- (insane)

      The compression level number is big, which means the compression ratio is high.

   3) APE Version
      Until now, APE has 3860, 3910, 3980, and so on. You can get the information form APE file Header.
      Such as:

```
if (CommonHeader.nVersion >= 3980)
{
	// current header format
	nRetVal = AnalyzeCurrent(pInfo);
}
else
{
	// legacy support
	nRetVal = AnalyzeOld(pInfo);
}
```

   4) APE frame and blocks(VisualOn Software Decoder)

APE has frame and block unit, but the block is the least processed unit.

A frame include lots of data block, you can divide a frame into many output buffer. Such as: #define BLOCKS_NUMBERS 1024, For Stereo, you can get 4K Bytes (1024 * 2 * 2) output PCM data one decode processing.

4. APE Bit Stream Syntax

APE Payload Format

ID3v2(Optional) + APE_DESCRIPTOR(52Bytes) + APE Header (24Bytes) + nSeekTableBytes(variable) + WavHeaderInfo(Optional) + Raw Data + ID3v1(Optional) + APE Tag(Optional)

5. VisualOn APE decoder API description

Global Data Structure

```
typedef struct{
    voAPE_DESCRIPTOR      *ape_desc;          /* APE file format descriptor structure*/
    voAPE_HEADER          *ape_hdr;           /* APE header info structure */
    voAPE_HEADER_OLD      *ape_hdr_old;       /*Version < 3980 APE header info structure*/
    voAPE_FILE_INFO       *ape_info;          /*APE file information structure */
    voCNNFState           *Xcnnf;
    voCNNFState           *Ycnnf;
    voDecodeBytes         *DecProcess;
    FrameStream           *stream_buf;
    int                   PredFlag;               /* Predictor malloc flag */
    int                   voAPED_State;
    VO_MEM_OPERATOR       *pvoMemop;
    VO_MEM_OPERATOR       voMemoprator;
    void                  *hCheck;
}voAPE_GData;
```

voAPE_DESCRIPTOR, voAPE_HEADER, voAPE_HEADER_OLD, voAPE_FILE_INFO are defined in voAPE_Header.h