

Audio Formats Reference

Brian Langenberger
tuffy@users.sourceforge.net

This work is licensed under the Creative Commons Attribution-Share Alike 3.0 United States License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/3.0/us/> or send a letter to: Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.



1 October 2008

Table of Contents

Audio Formats Reference	1
Basics	2
Hexadecimal	2
Endianness	3
Character Encodings	3
PCM	4
RIFF WAVE	5
the RIFF WAVE stream	5
the fmt chunk	5
the data chunk	5
AIFF	6
the AIFF stream	6
the COMM chunk	6
the SSND chunk	6
Sun AU	7
the AU stream	7
the AU header	7
FLAC	8
the FLAC file stream	8
FLAC metadata	9
the PADDING metadata block	9
the APPLICATION metadata block	9
the SEEKTABLE metadata block	9
the STREAMINFO metadata block	9
the VORBIS_COMMENT metadata block	10
the PICTURE metadata block	10
the CUESHEET metadata block	11
FLAC decoding	12
the CONSTANT subframe	13
the VERBATIM subframe	13
the FIXED subframe	13
the LPC subframe	14
the Residual	15
Rice Encoding	16
Channels	17
Wasted bits per sample	17
FLAC encoding	18
Metadata header	18
the STREAMINFO metadata block	18
the VORBIS_COMMENT metadata block	18
the PADDING metadata block	18
Frame header	19
Channel assignment	19
Subframe header	19
the CONSTANT subframe	20
the VERBATIM subframe	20
the FIXED subframe	20
the LPC subframe	21
Windowing	22
Computing autocorrelation	23
LP coefficient calculation	24

Best order estimation	25
Best order exhaustive search	25
Quantizing coefficients	26
Calculating Residual	27
the Residual	28
the Checksums	29
CRC-8	29
CRC-16	30
Monkey's Audio	32
the Monkey's Audio stream	32
the APE Descriptor	32
the APE Header	32
the APEv2 tag	33
the APEv2 tag header/footer	34
the APEv2 flags	34
WavPack	35
the WavPack file stream	35
a WavPack block header	36
a WavPack sub-block header	37
MP3	38
the MP3 file stream	38
an MPEG frame header	39
the Xing header	40
the ID3v1 tag	41
ID3v1	41
ID3v1.1	41
the ID3v2 tag	42
the ID3v2 stream	42
ID3v2.2	42
the ID3v2.2 Header	42
an ID3v2.2 Frame	42
ID3v2.2 Frame IDs	43
the PIC Frame	44
ID3v2.3	45
the ID3v2.3 Header	45
an ID3v2.3 Frame	45
ID3v2.3 Frame IDs	46
the APIC Frame	47
ID3v2.4	48
the ID3v2.4 Header	48
an ID3v2.4 Frame	48
ID3v2.4 Frame IDs	49
the APIC Frame	50
Ogg Vorbis	51
the Ogg file stream	51
an Ogg page	51
Ogg packets	52
the Identification packet	52
the Comment packet	53
Ogg Speex	54
the Header packet	54
the Comment packet	54
Ogg FLAC	55
the Ogg FLAC file stream	55

the STREAMINFO metadata packet	55
the Metadata packets	55
M4A	56
the QuickTime file stream	56
a QuickTime atom	56
Container atoms	56
the mdhd atom	57
the mp4a atom	58
the meta atom	59
the trkn sub-atom	60
the disk sub-atom	60
Musepack	61
Musepack SV7	61
the Musepack SV7 file stream	61
the Musepack SV7 header	61
Musepack SV8	62
the Musepack SV8 file stream	62
Nut-encoded values	62
the SH packet	62
the SE packet	63
the RG packet	63
the EI packet	63
FreeDB	64
Native Protocol	64
The Disc ID	65
Initial Greeting	66
Client-Server Handshake	66
Set Protocol Level	66
Calculate Disc ID	67
Query Database	67
Read XMCD Data	68
Close Connection	69
List Genres	69
List Mirrors	70
Web Protocol	71
XMCD	72
BNF	72
ReplayGain	73
Applying ReplayGain	73
Calculating ReplayGain	74
the Equal Loudness Filter	74
the Yule Filter	75
the Butter Filter	75
a Filtering Example	76
RMS Energy Blocks	78
Statistical Processing and Calibration	78
References	79

1. Audio Formats Reference

This is my collected notes about the many audio formats and audio-related protocols implemented by Python Audio Tools. As it has outgrown its original place at the end of the manual, I've shifted all the reference information to this document.

The reason that information is here instead of a web page is because I've typeset it for print density. I like to have printed information on hand when working so that I'll have paper documents to reference, scribble notes on, and shuffle around as needed. HTML is pretty lousy for that task since one never knows what information will wind up on which page, or how many pages that information will take in total. Typesetting it by hand guarantees everything about a FLAC frame will fit on a single page, for instance. I don't want to print any more pages than I have to, after all.

1.1. Basics

This section covers the basics for parsing binary files.

1.1.1. Hexadecimal

In order to understand hexadecimal, it's important to re-familiarize oneself with decimal, which everyone reading this should be familiar with. In ordinary decimal numbers, there are a total of ten characters per digit: 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. Because there are ten, we'll call it base-10. So the number 675 is made up of the digits 6, 7 and 5 and can be calculated in the following way:

$$(6 \times 10^2) + (7 \times 10^1) + (5 \times 10^0) = 675$$

In hexadecimal, there are sixteen characters per digit: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F. A, B, C, D, E and F correspond to the decimal numbers 10, 11, 12, 13, 14 and 15, respectively. Because there are sixteen, we'll call it base-16. So the number 2A3 is made up of the digits 2, A and 3 and can be calculated in the following way:

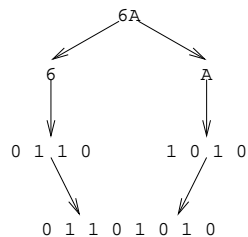
$$(2 \times 16^2) + (10 \times 16^1) + (3 \times 16^0) = 675$$

Why use hexadecimal? The reason brings us back to binary file formats, which are made up of bytes. Each byte is made up of 8 bits and can have a value from 0 to 255, in decimal. Representing a binary file in hexadecimal means a byte requires exactly two digits with values from 0 to FF. That saves us a lot of space versus trying to represent bytes in decimal.

Hexadecimal has another important property when dealing with binary data. Because each digit has 16 possible values, each hexadecimal digit represents exactly 4 bits ($16 = 2^4$).

Hexadecimal Conversion Table					
Hex	Binary	Decimal	Hex	Binary	Decimal
0	0 0 0 0	0	8	1 0 0 0	8
1	0 0 0 1	1	9	1 0 0 1	9
2	0 0 1 0	2	A	1 0 1 0	10
3	0 0 1 1	3	B	1 0 1 1	11
4	0 1 0 0	4	C	1 1 0 0	12
5	0 1 0 1	5	D	1 1 0 1	13
6	0 1 1 0	6	E	1 1 1 0	14
7	0 1 1 1	7	F	1 1 1 1	15

This makes it very easy to go back and forth between hexadecimal and binary. For instance, let's take the byte 6A.



Going from binary to hexadecimal is a simple matter of reversing the process. Combining multiple bytes into a single, larger number requires a similar method, but that brings us to endianness issues.

1.1.2. Endianness

You will need to know about endianness anytime a single value spans multiple bytes. As an example, let's take the first 16 bytes of a small RIFF WAVE file:

```
52 49 46 46 54 9b 12 00 57 41 56 45 66 6d 74 20
```

The first four bytes are the ASCII string 'RIFF' (0x52 0x49 0x46 0x46). The next four bytes are a 32-bit unsigned integer which is a size value. Reading from left to right, that value would be 0x549B1200. That's almost 1.5 gigabytes. Since this file is nowhere near that large, we're clearly not reading those bytes correctly.

The key is that RIFF WAVE files are 'little endian'. In plain English, that means we have to read in those bytes from right to left. Thus, the value is actually 0x00129B54. That's a little over 1 megabyte, which is closer to our expectations.

Remember that little endian reverses the bytes, not the hexadecimal digits. Simply reversing the string to 0x0021B945 is not correct.

1.1.3. Character Encodings

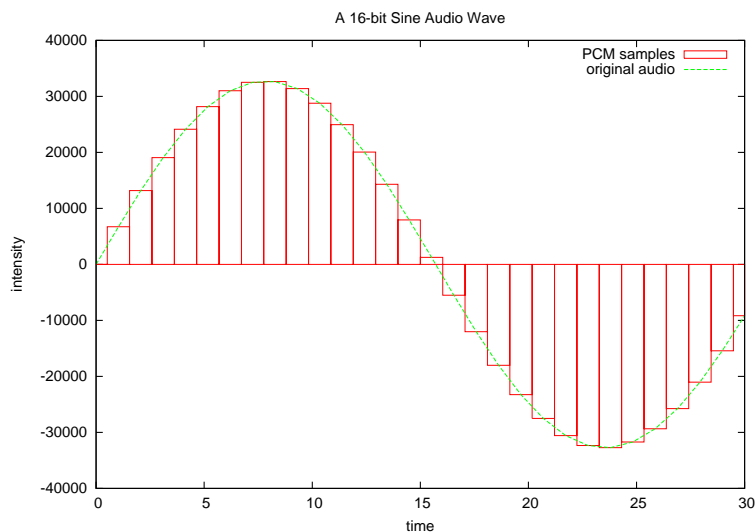
Many audio formats store metadata, which contains information about the song's name, artist, album and so forth. This information is stored as text, but it's important to know what sort of text in order to read it and display it properly.

As an example, take the simple character 'é'. In latin-1 encoding, it is stored as a single byte 0xE9. In UTF-8 encoding, it is stored as the bytes 0xC3A9. In UTF-16BE encoding, it is stored as the bytes 0x00E9.

Although decoding and encoding text is a complex subject beyond the scope of this document, you must always be aware that track metadata may not be 7-bit ASCII text and should handle it properly in whatever encoding is supported by the metadata formats. Look to your programming tools for libraries to assist in Unicode text handling.

1.1.4. PCM

Pulse-Code Modulation is a method for transforming an analog audio signal into a digital representation. It takes that signal, ‘samples’ its intensity at discrete intervals and yields a stream of signed integer values. By replaying those values to a speaker at the same speed and intensity, a close approximation of the original signal is produced.



Let’s take some example bytes from a CD-quality PCM stream:

```
1B 00 43 FF 1D 00 45 FF 1C 00 4E FF 1E 00 59 FF
```

CD-quality is 16-bit, 2 channel, 44100Hz. 16-bit means those bytes are split into 16-bit signed, little-endian samples. Therefore, our bytes are actually the integer samples:

```
27 -189 29 -187 28 -178 30 -167
```

The number of channels indicates how many speakers the signal supports. 2 channels means the samples are sent to 2 different speakers. PCM interleaves its samples, sending one sample to each channel simultaneously before moving on to the next set. In the case of 2 channels, the first sample is sent to the left speaker and the second is sent to the right speaker. So, our stream of data now looks like:

left speaker	right speaker
27	-189
29	-187
28	-178
30	-167

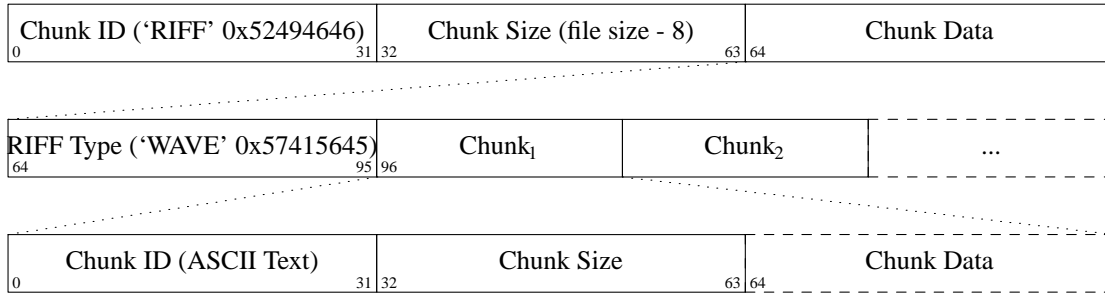
44100Hz means those pairs of samples are sent at the rate of 44100 per second. Thus, our set of samples takes precisely 1/11025th of a second when replayed.

A channel-independent block of samples is commonly referred to as a ‘frame’. In this example, we have a total of 4 PCM frames. However, the term ‘frame’ appears a lot in digital audio. It is important not to confuse a PCM frame with a CD frame (a block of audio 1/75th of a second long), an MP3 frame, a FLAC frame or any other sort of frame.

1.2. RIFF WAVE

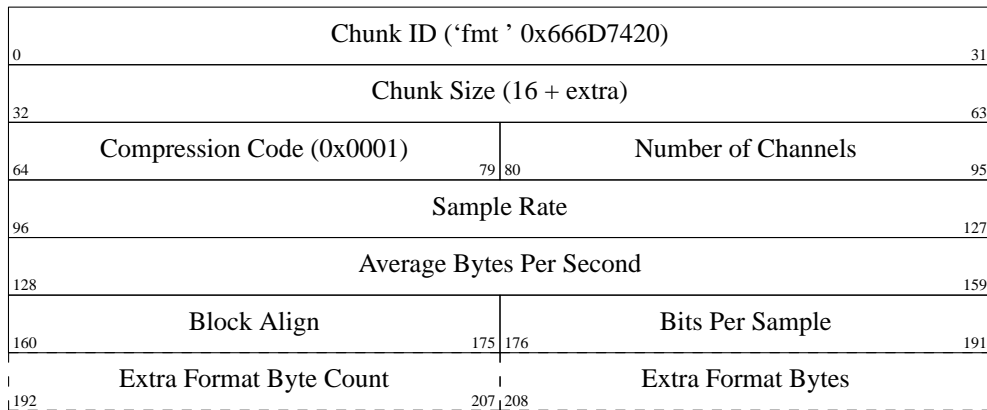
RIFF WAVE is the most common form of PCM container. What that means is that the file is mostly PCM data with a small amount of header data to tell applications what format the PCM data is in. Since RIFF WAVE originated on Intel processors, everything in it is little-endian.

1.2.1. the RIFF WAVE stream



'Chunk Size' is the total size of the chunk, minus 8 bytes for the chunk header.

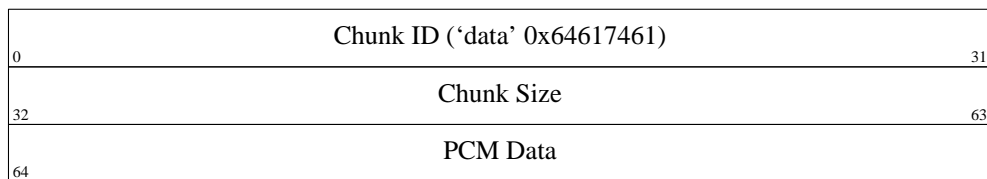
1.2.2. the fmt chunk



$$\text{Average Bytes Per Second} = \text{Sample Rate} \times \text{Number of Channels} \times \text{Bits Per Sample} \div 8$$

$$\text{Block Align} = \text{Number of Channels} \times \text{Bits Per Sample} \div 8$$

1.2.3. the data chunk

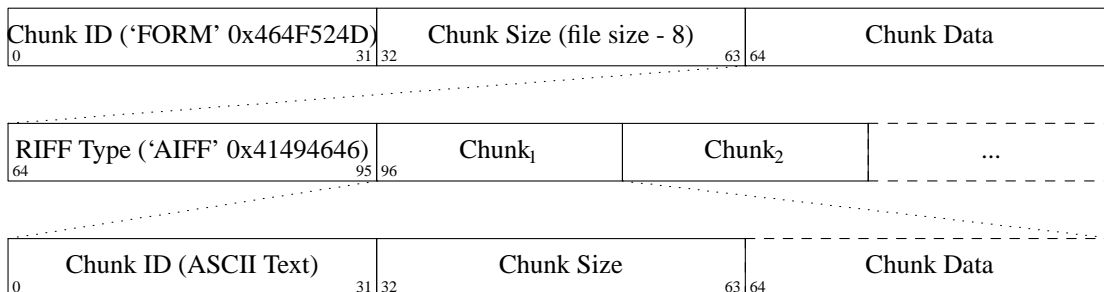


'PCM Data' is a stream of PCM samples stored in little-endian format.

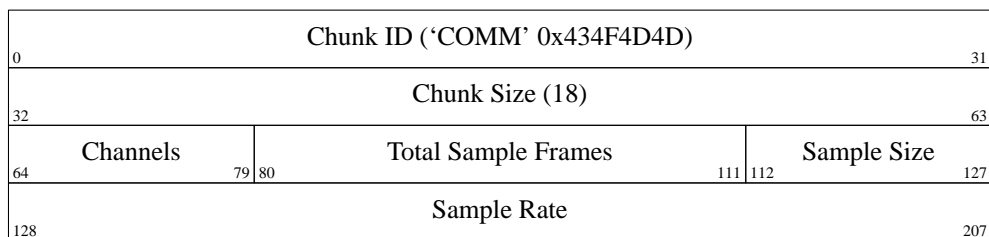
1.3. AIFF

AIFF is the Audio Interchange File Format. It is popular on Apple computers and is a precursor to the more widespread WAVE format. All values in AIFF are stored as big-endian.

1.3.1. the AIFF stream

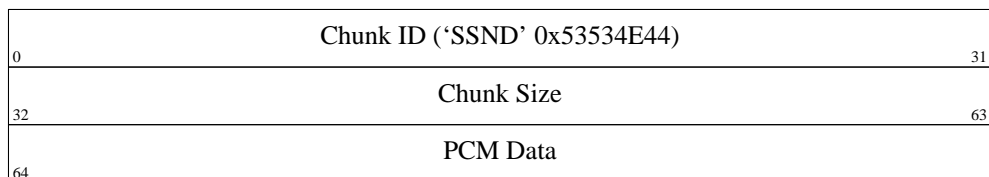


1.3.2. the COMM chunk



The Sample Rate field is an 80-bit IEEE Standard 754 floating point value instead of the big-endian integers common to all the other fields.

1.3.3. the SSND chunk



1.4. Sun AU

The AU file format was invented by Sun Microsystems and also used on NeXT systems. All values in AU are stored as big-endian. It supports a wide array of data formats, including μ -law logarithmic encoding, but can also be used as a PCM container.

1.4.1. the AU stream



1.4.2. the AU header

0	Magic Number ('.snd' 0x2e736e64)	31
32	Data Offset	63
64	Data Size	95
96	Encoding Format	127
128	Sample Rate	159
160	Channels	191

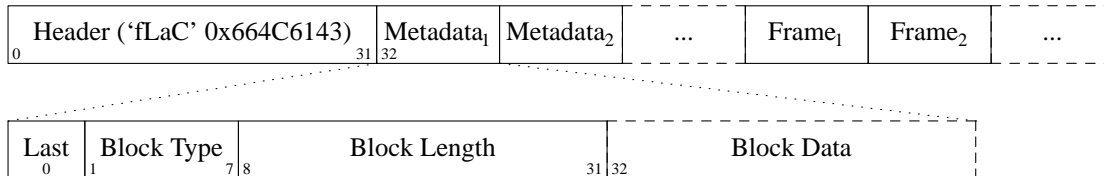
Encoding Formats	
value	format
1	8-bit G.711 μ -law
2	8-bit linear PCM
3	16-bit linear PCM
4	24-bit linear PCM
5	32-bit linear PCM
6	32-bit IEEE floating point
7	64-bit IEEE floating point
8	Fragmented sample data
9	DSP program
10	8-bit fixed point
11	16-bit fixed point
12	24-bit fixed point
13	32-bit fixed point
18	16-bit linear with emphasis
19	16-bit linear compressed
20	16-bit linear with emphasis and compression
21	Music kit DSP commands
23	4-bit ISDN μ -law compressed using the ITU-T G.721 ADPCM voice data encoding scheme
24	ITU-T G.722 ADPCM
25	ITU-T G.723 3-bit ADPCM
26	ITU-T G.723 5-bit ADPCM
27	8-bit G.711 A-law

1.5. FLAC

FLAC is the Free Lossless Audio Codec. It compresses PCM audio data losslessly using predictors and a residual. FLACs are smaller than WAVs, contain checksumming to verify their integrity, contain comment tags for metadata and are streamable.

Except for the contents of the VORBIS_COMMENT metadata block, everything in FLAC is big-endian.

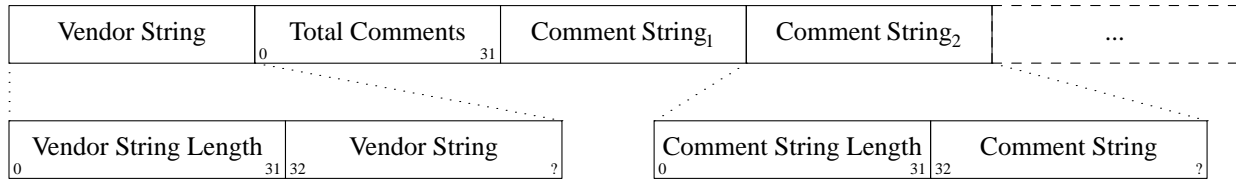
1.5.1. the FLAC file stream



"Last" is 0 when there are additional metadata blocks and 1 when it is the final block before the the audio frames. "Block Length" is the size of the metadata block data to follow, not including the header.

Block Types	
value	type
0	STREAMINFO
1	PADDING
2	APPLICATION
3	SEEKTABLE
4	VORBIS_COMMENT
5	CUESHEET
6	PICTURE
7-126	reserved
127	invalid

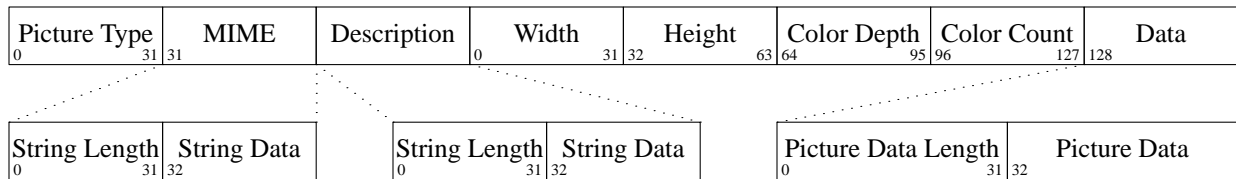
1.5.2.5. the VORBIS_COMMENT metadata block



The length fields are all little-endian. The Vendor String and Comment Strings are all UTF-8 encoded. Keys are not case-sensitive and may occur multiple times, indicating multiple values for the same field. For instance, a track with multiple artists may have more than one ARTIST.

Comment Strings			
key	value	key	value
ALBUM	album name	ARTIST	artist name, band name, composer, author, etc.
CONTACT	contact information	COPYRIGHT	copyright attribution
DATE	recording date	DESCRIPTION	a short description
GENRE	a short music genre label	ISRC	ISRC number for the track
LICENSE	license information	LOCATION	recording location
ORGANIZATION	record label	PERFORMER	performer name, orchestra, actor, etc.
TITLE	track name	TRACKNUMBER	the track number
VERSION	track version		

1.5.2.6. the PICTURE metadata block



Picture Types			
value	type	value	type
0	Other	1	32x32 pixels 'file icon' (PNG only)
2	Other file icon	3	Cover (front)
4	Cover (back)	5	Leaflet page
6	Media (e.g. label side of CD)	7	Lead artist / Lead performer / Soloist
8	Artist / Performer	9	Conductor
10	Band / Orchestra	11	Composer
12	Lyricist / Text writer	13	Recording location
14	During recording	15	During performance
16	Movie / Video screen capture	17	A bright coloured fish
18	Illustration	19	Band / Artist logotype
20	Publisher / Studio logotype		

1.5.2.7. the CUESHEET metadata block

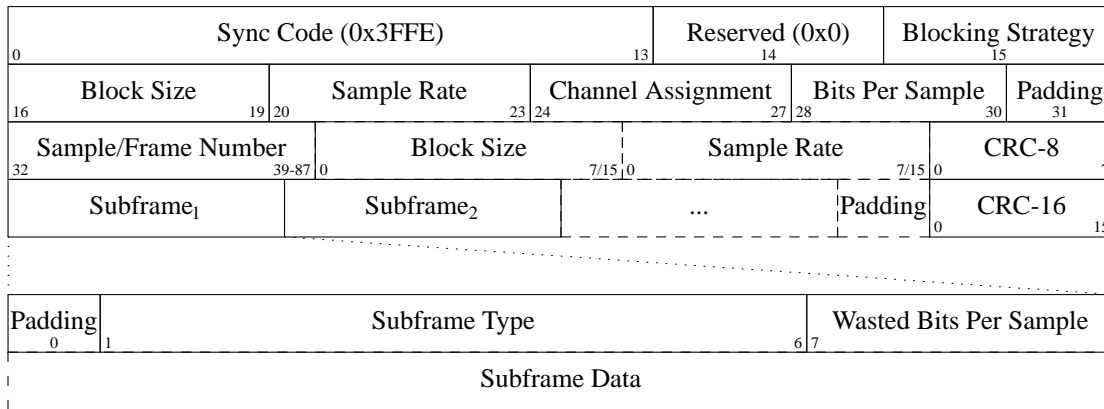
Catalog Number	Lead-in Samples	is CDDA	NULL (0x00)	Track Count	Track ₁	...
0 1023	1024 1087	1088	1089 3159	3160 3167	3168	

Track Offset	Track Number	ISRC	Type	Pre-Emph.	NULL (0x00)	Index Points	Index ₁	...
0 63	64 71	72 167	168	169	170 279	280 287	288	

Index Offset	Index Number	NULL (0x00)
0 63	64 71	72 95

1.5.3. FLAC decoding

A FLAC stream is made up of individual FLAC frames, as follows:



Bits	Block Size (in samples)	Sample Rate	Channel Assignment		Bits
			channels	assignment	
0000	get from STREAMINFO	get from STREAMINFO	1	mono	0000
0001	192	88200	2	left, right	0001
0010	576	176400	3	left, right, center	0010
0011	1152	192000	4	left, right, back left, back right	0011
0100	2304	8000	5	left, right, center, back left, back right	0100
0101	4608	16000	6	left, right, center, LFE, back left, back right	0101
0110	8 bit from end of header (+1)	22050	7	undefined	0110
0111	16 bit from end of header (+1)	24000	8	undefined	0111
1000	256	32000	2	0 left, 1 difference	1000
1001	512	44100	2	0 difference, 1 right	1001
1010	1024	48000	2	0 average, 1 difference	1010
1011	2048	96000		reserved	1011
1100	4096	get 8 bit from end of header (in kHz)		reserved	1100
1101	8192	get 16 bit from end of header (in Hz)		reserved	1101
1110	16384	get 16 bit from end of header (in 10s of Hz)		reserved	1110
1111	32768	invalid		reserved	1111

Bits Per Sample	
bits	per sample
000	get from STREAMINFO
001	8
010	12
011	reserved
100	16
101	20
110	24
111	reserved

Subframe Type	
bits	type
000000	SUBFRAME_CONSTANT
000001	SUBFRAME_VERBATIM
00001x	reserved
0001xx	reserved
001xxx	SUBFRAME_FIXED (xxx = Predictor Order)
01xxxx	reserved
1xxxxx	SUBFRAME_LPC (xxxxx = LPC Order - 1)

Sample/Frame Number is a UTF-8 coded value. If the blocking strategy is 0, it decodes to a 32-bit frame number. If the blocking strategy is 1, it decodes to a 36-bit sample number. There is one Subframe per channel.

‘Wasted Bits Per Sample’ is typically a single bit set to 0, indicating no wasted bits per sample. If set to 1, a unary-encoded value follows which indicates how many bits are wasted per sample. Padding is added as needed between the final subframe and CRC-16 in order to byte-align frames.

1.5.3.1. the CONSTANT subframe

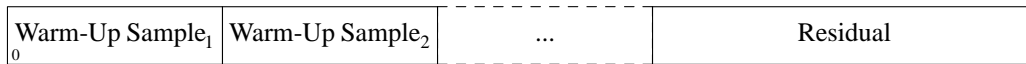
This is the simplest possible subframe. It consists of a single value whose size is equal to the frame’s ‘Bits Per Sample’. For instance, a 16-bit frame would have CONSTANT subframes 16 bits in length. The value of the subframe is the value of all samples the subframe contains. An obvious use of this subframe is to store an entire subframe’s worth of digital silence (samples with a value of 0) very efficiently.

1.5.3.2. the VERBATIM subframe



This subframe’s length is equal to the ‘Bits Per Sample’ multiplied by the frame’s ‘Block Size’. Since it does no compression whatsoever and simply stores audio samples as-is, this subframe is only suitable for especially noisy portions of a track where no predictor can be found.

1.5.3.3. the FIXED subframe



The number of warm-up samples equals the ‘Predictor Order’ (which is encoded in the ‘Subframe Type’). These samples are sent out as-is; they are the subframe’s ‘starting point’ upon which further samples build when decompressing the stream. Determining the value of the current sample is then a matter of looking backwards at previously decoded samples (or warm-up samples), applying a simple formula on their values (which depends on the Predictor Order) and adding the residual.

Predictor Order	Calculation
0	$Sample_i = Residual_i$
1	$Sample_i = Sample_{i-1} + Residual_i$
2	$Sample_i = (2 \times Sample_{i-1}) - Sample_{i-2} + Residual_i$
3	$Sample_i = (3 \times Sample_{i-1}) - (3 \times Sample_{i-2}) + Sample_{i-3} + Residual_i$
4	$Sample_i = (4 \times Sample_{i-1}) - (6 \times Sample_{i-2}) + (4 \times Sample_{i-3}) - Sample_{i-4} + Residual_i$

Let’s run through a simple example in which the Predictor Order is 1. Note that residual does not apply to warm-up samples. How to extract the encoded residual will be covered in a later section.

Index	Residual	Sample
0		(warm-up) 10
1	1	$10 + 1 = \mathbf{11}$
2	2	$11 + 2 = \mathbf{13}$
3	-2	$13 - 2 = \mathbf{11}$
4	1	$11 + 1 = \mathbf{12}$
5	-1	$12 - 1 = \mathbf{11}$

1.5.3.4. the LPC subframe

Warm-Up Sample ₁			Warm-Up Sample ₂			...			Warm-Up Sample _n					
QLP Precision			QLP Shift Needed			QLP Coefficient ₁			QLP Coefficient ₂			...		
Residual														

The number of warm-up samples equals the ‘LPC Order’ (which is encoded in the ‘Subframe Type’). The size of each QLP Coefficient is equal to ‘QLP Precision’ number of bits, plus 1. The value of each Coefficient is a signed two’s-complement integer. The number of Coefficients equals the ‘LPC Order’.

$$Sample_i = \left[\frac{\sum_{j=0}^{Order-1} QLP\ Coefficient_j \times Sample_{i-j-1}}{2^{QLP\ Shift\ Needed}} \right] + Residual_i$$

Don’t be intimidated by the Σ if you’re not math-inclined. It simply means we’re taking the sum of the calculated values from 0 to Order - 1, bit-shifting that sum down and added the residual when determining the current sample. Much like the FIXED subframe, LPC subframes also contain warm-up samples which serve as our calculation’s starting point.

In this example, the LPC Order is 5, the QLP Shift Needed is 9 and the encoded Coefficients are as follows:

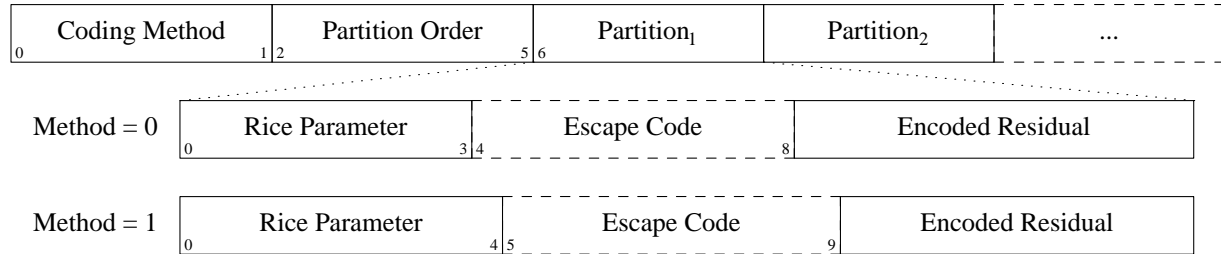
- QLP Coefficient₀ 1241
- QLP Coefficient₁ -944
- QLP Coefficient₂ 14
- QLP Coefficient₃ 342
- QLP Coefficient₄ -147

Index	Residual	Sample
0		(warm-up) 1053
1		(warm-up) 1116
2		(warm-up) 1257
3		(warm-up) 1423
4		(warm-up) 1529
5	11	(1241 × 1529) + (-944 × 1423) + (14 × 1257) + (342 × 1116) + (-147 × 1053) = 798656 (798656 / 2 ⁹) = 1559 + 11 = 1570
6	79	(1241 × 1570) + (-944 × 1529) + (14 × 1423) + (342 × 1257) + (-147 × 1116) = 790758 (790758 / 2 ⁹) = 1544 + 79 = 1623
7	24	(1241 × 1623) + (-944 × 1570) + (14 × 1529) + (342 × 1423) + (-147 × 1257) = 855356 (855356 / 2 ⁹) = 1670 + 24 = 1694
8	-81	(1241 × 1694) + (-944 × 1623) + (14 × 1570) + (342 × 1529) + (-147 × 1423) = 905859 (905859 / 2 ⁹) = 1769 - 81 = 1688
9	-72	(1241 × 1688) + (-944 × 1694) + (14 × 1623) + (342 × 1570) + (-147 × 1529) = 830571 (830571 / 2 ⁹) = 1622 - 72 = 1550

In this instance, division should always round down and *not* towards zero. For example: -5 / 2¹ = -3

1.5.3.5. the Residual

Though the FLAC format allows for different forms of residual coding, two forms of partitioned Rice are the only ones currently supported. The difference between the two is that when ‘Coding Method’ is 0, the Rice Parameter in each partition is 4 bits. When the ‘Coding Method’ is 1, that parameter is 5 bits.



There are $2^{\text{Partition Order}}$ number of Partitions. The number of decoded samples in a Partition depends on the its position in the subframe. The first partition in the subframe contains:

$$\text{Total Samples} = \left(\frac{\text{Frame's Block Size}}{2^{\text{Partition Order}}} \right) - \text{Predictor Order}$$

Subsequent partitions contain:

$$\text{Total Samples} = \frac{\text{Frame's Block Size}}{2^{\text{Partition Order}}}$$

Unless, of course, the Partition Order is 0. In that case:

$$\text{Total Samples} = \text{Frame's Block Size} - \text{Predictor Order}$$

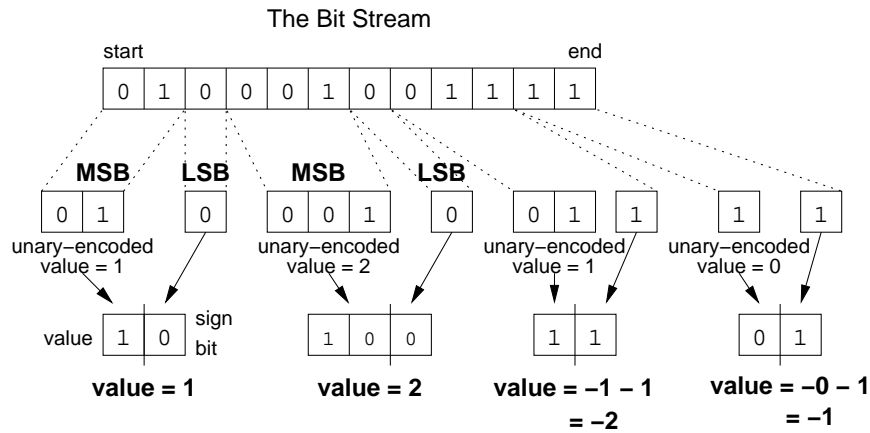
since there is only one partition which takes up the entire block.

If all of the bits in ‘Rice Parameter’ are set, the partition is unencoded binary using ‘Escape Code’ number of bits per sample.

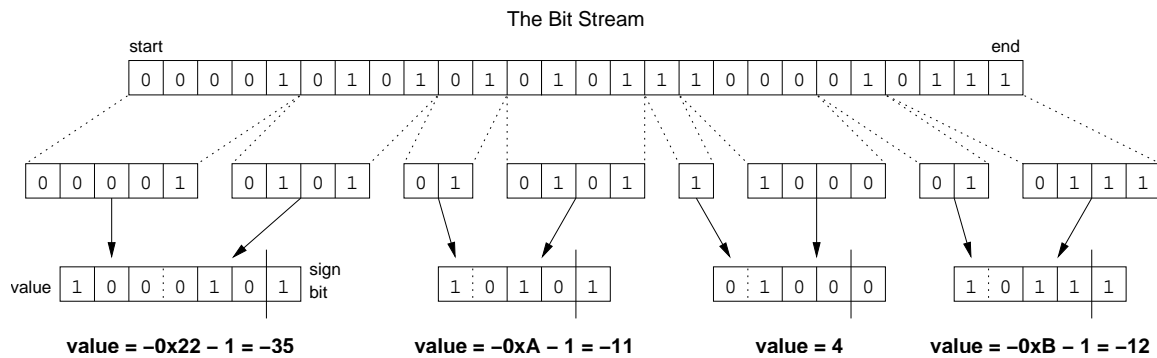
1.5.3.5.1. Rice Encoding

The residual uses Rice coding to compress lots of mostly small values in a very small amount of space. To decode it, one first needs the Rice parameter. Take a unary-encoded value[†] from the bit stream, which are our most significant bits (MSB). Then take 'parameter' number of additional bits, which are our least significant bits (LSB). Combine the two sets into our new value, making the MSB set as the high bits and the LSB set as the low bits. Bit 0 of this new value is the sign bit. If it is 0, the actual value is equal to the rest of the bits. If it is 1, the actual value is equal to the rest of the bits, multiplied by -1 and minus 1.

This is less complicated than it sounds, so let's run through an example in which the Rice parameter is 1:



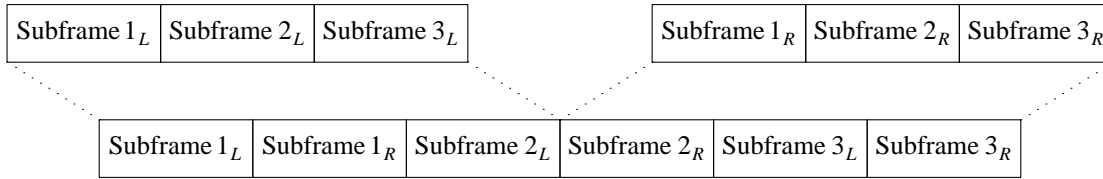
Now, let's run through another example in which the Rice parameter is 4:



[†] In this instance, unary-encoding is a simple matter of counting the number of 0 bits before the next 1 bit. The resulting sum is the value.

1.5.3.6. Channels

Since most audio has more than one channel, it is important to understand how FLAC handles putting it back together. When channels are stored independently, one simply interleaves them together in the proper order. Let's take an example of 2 channel, 16-bit audio stored this way:



This is the simplest case. However, in the case of difference channels, one subframe will contain actual channel data and the other channel will contain signed difference data which is applied to the first channel in order to reconstruct both channels. It's very important to remember that the difference channel has 1 additional bit per sample which will be consumed during reconstruction. Why 1 additional bit? Let's take an example where the left sample's value is -30000 and the right sample's value is +30000. Storing this pair as left + difference means the left sample remains -30000 and the difference is -60000 (-30000 - 30000 = -60000). -60000 won't fit into a 16-bit signed integer. Adding that 1 additional bit doubles our range of values and that's just enough to cover any possible difference between two samples.

Channel Calculation				
Assignment	Channel 0	Channel 1	Left Channel	Right Channel
1000	left	difference	left	left - difference
1001	difference	right	right + difference	right
1010	mid	side	$((\text{mid} \ll 1) (\text{side} \& 1)) + \text{side} \gg 1$	$((\text{mid} \ll 1) (\text{side} \& 1)) - \text{side} \gg 1$

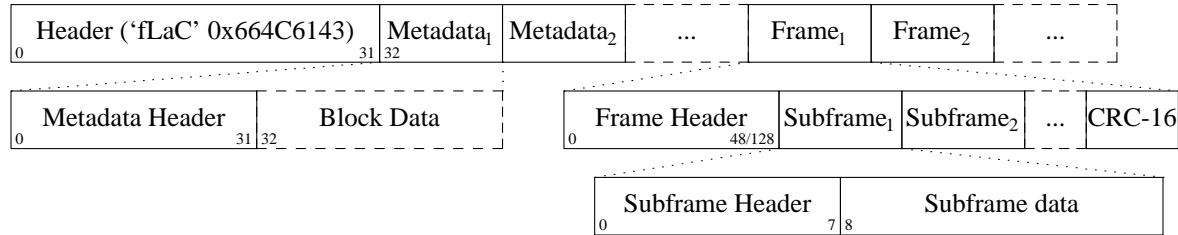
The mid channel case is another unusual exception. We're prepending the mid channel with bit 0 from the side channel, performing the addition/subtraction and then discarding that bit before assigning the results to the left and right channels.

1.5.3.7. Wasted bits per sample

Though rare in practice, FLAC subframes support 'wasted bits per sample'. Put simply, these wasted bits are removed during subframe calculation and restored to the subframe's least significant bits as zero value bits when it is returned. For instance, a subframe with 1 wasted bit per sample in a 16-bit FLAC stream is treated as having only 15 bits per sample when reading warm-up samples and then all through the rest of the subframe calculation. That wasted zero bit is then prepended to each sample prior to returning the subframe.

1.5.4. FLAC encoding

For the purposes of discussing FLAC encoding, we'll assume one has a stream of input PCM values along with the stream's sample rate, number of channels and bits per sample. Creating a valid FLAC file is then a matter of writing the proper file header, metadata blocks and FLAC frames.



1.5.4.1. Metadata header

bits	value
1	0 if addition metadata blocks follow, 1 if not
7	0 for STREAMINFO, 1 for PADDING, 4 for VORBIS_COMMENT, etc.
24	the length of the block data in bytes, not including the header

1.5.4.2. the STREAMINFO metadata block

bits	value
16	the minimum FLAC frame size, in PCM frames
16	the maximum FLAC frame size, in PCM frames
24	the minimum FLAC frame size, in bytes
24	the maximum FLAC frame size, in bytes
20	the stream's sample rate, in Hz
3	the stream's channel count, minus one
5	the stream's bit-per-sample, minus one
36	the stream's total number of PCM frames
128	an MD5 sum of the PCM stream's bytes

When encoding a FLAC file, many of these fields cannot be known in advance. Instead, one must keep track of those values during encoding and then rewrite the STREAMINFO block when finished.

1.5.4.3. the VORBIS_COMMENT metadata block

bits	value
32‡	vendor string length, in bytes
string length × 8	vendor string data, as UTF-8 encoded text
32‡	total number of comment strings
32‡	comment string ₁ length, in bytes
string length × 8	comment string ₁ , as UTF-8 encoded text
...	...

Fields marked with ‡ are little-endian integers.

1.5.4.4. the PADDING metadata block

This is simply an empty block full of 0x00 bytes.

1.5.4.5. Frame header

bits	value
14	0x3FFE sync code
1	0 reserved
1	0 if the header encodes the frame number, 1 if it encodes the sample number
4	this frame's block size, as encoded PCM frames†
4	this frame's encoded sample rate†
4	this frame's encoded channel assignment†
3	this frame's encoded bits per sample†
1	0 padding
8-56	the frame number, or sample number, UTF-8 encoded and starting from 0
0/8/16	the number of PCM frames (minus one) in this FLAC frame, if block size is 0x6 (8 bits) or 0x7 (16 bits)
0/8/16	the sample rate of this FLAC frame, if sample rate is 0xC (8 bits), 0xD (16 bits) or 0xE (16 bits)
8	the CRC-8 of all data from the beginning of the frame header

The FLAC frame's block size in PCM frames (called "channel independent samples" in FLAC's documentation) is typically encoded in the 4 bit 'block size' field. But for odd-sized frames - which often occur at the end of the stream - that value is stored as an 8 or 16 bit integer following the UTF-8 encoded frame number.

In addition, odd sample rate values are stored as 8 bit (in kHz), 16 bit (in Hz) or 16 bit (in 10s of Hz) prior to the CRC-8, should a predefined value not be available.

Up until this point, nearly all of these fields can be filled from the PCM stream data. Unless you're writing a variable block size encoder (which no one has), one should encode the frame number starting from 0 in the frame header and choose a predefined block size for as many FLAC frames as possible.

1.5.4.6. Channel assignment

If the input stream has a number of channels other than 2, one has no choice but to store them independently. If the number of channels equals 2, one can try all four possible assignments (left-difference, difference-right, mid-side and independent) and use the one which takes the least amount of space.

1.5.4.7. Subframe header

bits	value
1	0 padding
6	subframe type, with optional predictor order
1	0 if no wasted bits per sample, 1 if a unary-encoded number follows
0+	the number of wasted bits per sample (minus one) encoded as unary

Subframe Type	
bits	type
000000	SUBFRAME_CONSTANT
000001	SUBFRAME_VERBATIM
00001x	reserved
0001xx	reserved
001xxx	SUBFRAME_FIXED (xxx = Predictor Order)
01xxxx	reserved
1xxxxx	SUBFRAME_LPC (xxxxx = Predictor Order - 1)

† See table on page 12

1.5.4.8. the CONSTANT subframe

If all the samples in a subframe are identical, one can encode them using a CONSTANT subframe, which is essentially a single sample value that gets duplicated ‘block size’ number of times when decoded.

1.5.4.9. the VERBATIM subframe

This subframe simply stores all the samples as-is, with no compression whatsoever. It is a ‘fallback’ encoding method for when no other subframe makes one’s data any smaller.

1.5.4.10. the FIXED subframe

This subframe consists of ‘predictor order’ number of unencoded warm-up samples followed by a residual. Determining which predictor order to use on a given set of input samples depends on their minimum delta sum. This process is best explained by example:

index	sample	Δ^0	Δ^1	Δ^2	Δ^3	Δ^4
0	-40					
1	-41	<i>-41</i>				
2	-40	<i>-40</i>	<i>-1</i>			
3	-39	<i>-39</i>	<i>-1</i>	<i>0</i>		
4	-38	<i>-38</i>	<i>-1</i>	<i>0</i>	<i>0</i>	
5	-38	<i>-38</i>	<i>0</i>	<i>-1</i>	<i>1</i>	<i>-1</i>
6	-35	<i>-35</i>	<i>-3</i>	<i>3</i>	<i>-4</i>	<i>5</i>
7	-35	<i>-35</i>	<i>0</i>	<i>-3</i>	<i>6</i>	<i>-10</i>
8	-39	<i>-39</i>	<i>4</i>	<i>-4</i>	<i>1</i>	<i>5</i>
9	-40	<i>-40</i>	<i>1</i>	<i>3</i>	<i>-7</i>	<i>8</i>
10	-40	<i>-40</i>	<i>0</i>	<i>1</i>	<i>2</i>	<i>-9</i>
11	-39	<i>-39</i>	<i>-1</i>	<i>1</i>	<i>0</i>	<i>2</i>
12	-38	<i>-38</i>	<i>-1</i>	<i>0</i>	<i>1</i>	<i>-1</i>
13	-37	<i>-37</i>	<i>-1</i>	<i>0</i>	<i>0</i>	<i>1</i>
14	-33	<i>-33</i>	<i>-4</i>	<i>3</i>	<i>-3</i>	<i>3</i>
15	-36	<i>-36</i>	<i>3</i>	<i>-7</i>	<i>10</i>	<i>-13</i>
16	-35	<i>-35</i>	<i>-1</i>	<i>4</i>	<i>-11</i>	<i>21</i>
17	-31	<i>-31</i>	<i>-4</i>	<i>3</i>	<i>1</i>	<i>-12</i>
18	-32	<i>-32</i>	<i>1</i>	<i>-5</i>	<i>8</i>	<i>-7</i>
19	-33	<i>-33</i>	<i>1</i>	<i>0</i>	<i>-5</i>	<i>13</i>
sum		579	26	38	60	111

Note that the numbers in italics play a part in the delta calculation to their right, but do **not** figure into the delta’s absolute value sum, below.

In this example, Δ^1 ’s value of 26 is the smallest. Therefore, when compressing this set of samples in a FIXED subframe, it’s best to use a predictor order of 1.

The predictor order indicates how many warm-up samples to take from the PCM stream. Determining the residual values can then be done automatically based on the current $Sample_i$ and previously encoded samples, or warm-up samples.

Predictor Order	Calculation
0	$Residual_i = Sample_i$
1	$Residual_i = Sample_i - Sample_{i-1}$
2	$Residual_i = Sample_i - ((2 \times Sample_{i-1}) - Sample_{i-2})$
3	$Residual_i = Sample_i - ((3 \times Sample_{i-1}) - (3 \times Sample_{i-2}) + Sample_{i-3})$
4	$Residual_i = Sample_i - ((4 \times Sample_{i-1}) - (6 \times Sample_{i-2}) + (4 \times Sample_{i-3}) - Sample_{i-4})$

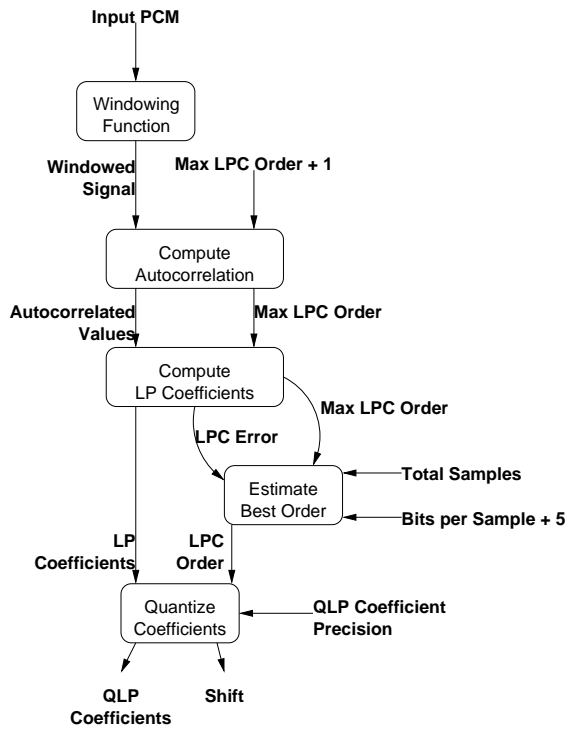
1.5.4.11. the LPC subframe

Unlike the FIXED subframe which required only input samples and a predictor order, LPC subframes also require a list of QLP coefficients, a QLP precision value of those coefficients, and a QLP shift needed value.

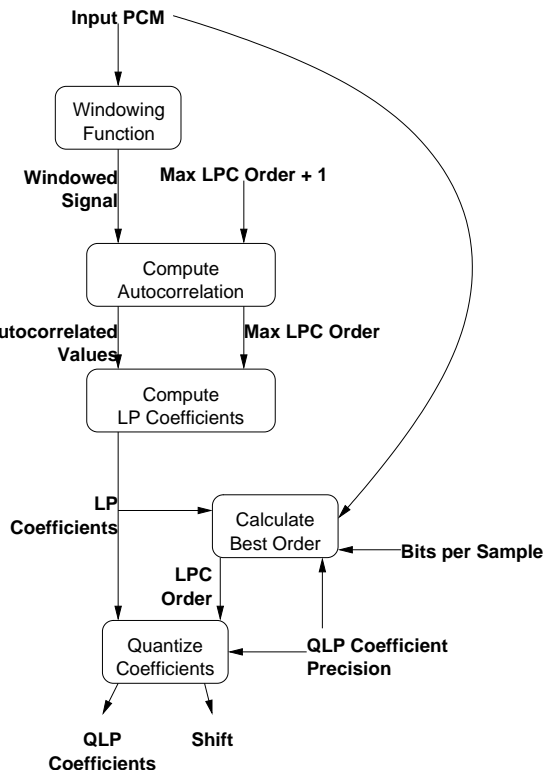
Warm-Up Sample ₁			Warm-Up Sample ₂			...			Warm-Up Sample _n					
QLP Precision			QLP Shift Needed			QLP Coefficient ₁			QLP Coefficient ₂			...		
Residual														

Determining these values for a given input PCM signal is a somewhat complicated process which depends on whether one is performing an exhaustive LP coefficient order search or not:

Non-exhaustive search



Exhaustive search



1.5.4.11.1. Windowing

The first step in LPC subframe encoding is ‘windowing’ the input signal. Put simply, this is a process of multiplying each input sample by an equivalent value from the window, which are floats from 0.0 to 1.0. In this case, the default is a Tukey window with a ratio of 0.5. A Tukey window is a combination of the Hann and Rectangular windows. The ratio of 0.5 means there’s 0.5 samples in the Hann window per sample in the Rectangular window.

The Hann window is defined by the function:

$$\text{hann}(n) = \frac{1}{2} \left(1 - \cos \left(\frac{2\pi n}{\text{sample count} - 1} \right) \right)$$

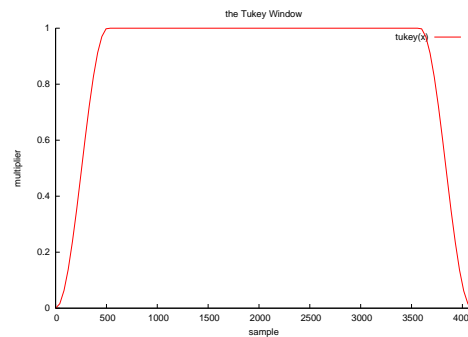
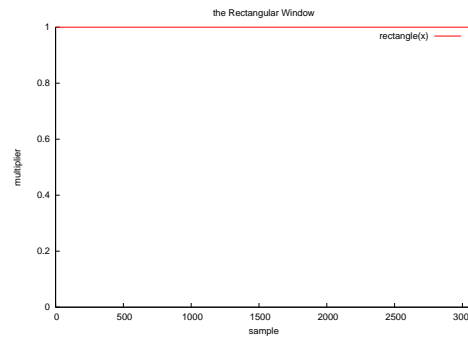
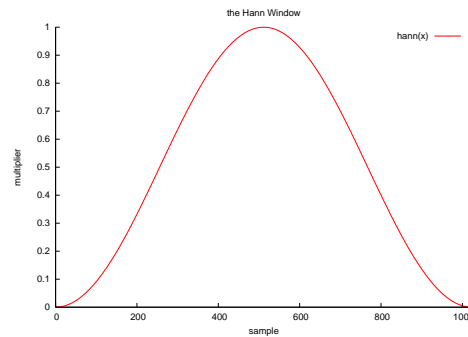
The Rectangular window is defined by the function:

$$\text{rectangle}(n) = 1.0$$

The Tukey window is defined by taking a Hann window, splitting it at the halfway point, and inserting a Rectangular window between the two.

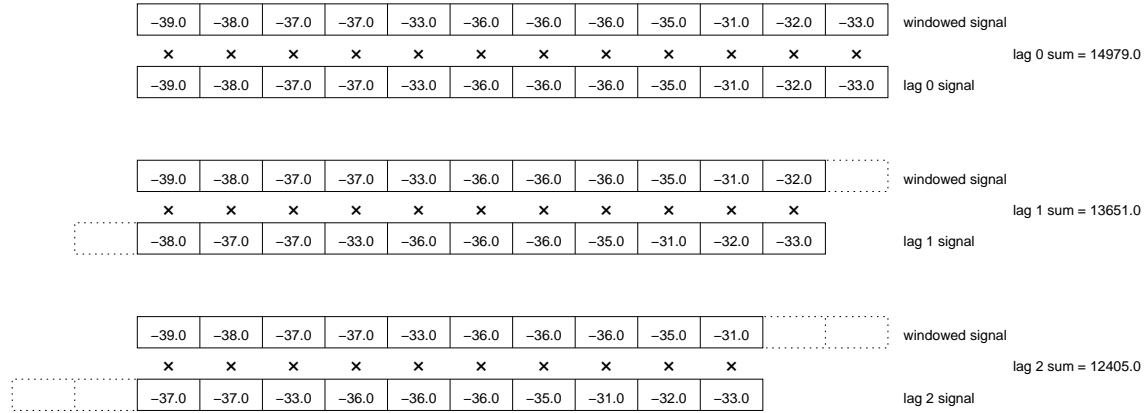
Let’s run through a short example with 20 samples:

index	input sample	Tukey window	windowed signal
0	-40	×	0.0000 = 0.00
1	-41	×	0.1464 = -6.00
2	-40	×	0.5000 = -20.00
3	-39	×	0.8536 = -33.29
4	-38	×	1.0000 = -38.00
5	-38	×	1.0000 = -38.00
6	-35	×	1.0000 = -35.00
7	-35	×	1.0000 = -35.00
8	-39	×	1.0000 = -39.00
9	-40	×	1.0000 = -40.00
10	-40	×	1.0000 = -40.00
11	-39	×	1.0000 = -39.00
12	-38	×	1.0000 = -38.00
13	-37	×	1.0000 = -37.00
14	-33	×	1.0000 = -33.00
15	-36	×	1.0000 = -36.00
16	-35	×	0.8536 = -29.88
17	-31	×	0.5000 = -15.50
18	-32	×	0.1464 = -4.68
19	-33	×	0.0000 = 0.00



1.5.4.11.2. Computing autocorrelation

Once our input samples have been converted to a windowed signal, we then compute the autocorrelation values from that signal. Each autocorrelation value is determined by multiplying the signal's samples by the samples of a lagged version of that same signal, and then taking the sum. The lagged signal is simply the original signal with 'lag' number of samples removed from the beginning.



The lagged sums from 0 to the maximum LPC order are our autocorrelation values. In this example, they are 14979.0, 13651.0 and 12405.0.

1.5.4.11.3. LP coefficient calculation

Calculating the LP coefficients uses the Levinson-Durbin recursive method.† Our inputs are M , the maximum LPC order minus 1, and r autocorrelation values, from $r(0)$ to $r(M-1)$. Our outputs are a , a list of LP coefficient lists from a_{11} to $a_{(M-1)(M-1)}$, and E , a list of error values from E_0 to $E_{(M-1)}$. q_m and κ_m are temporary values.

Initial values:

$$\begin{aligned} E_0 &= r(0) \\ a_{11} &= \kappa_1 = \frac{r(1)}{E_0} \\ E_1 &= E_0(1 - \kappa_1^2) \end{aligned}$$

With $m \geq 2$, the following recursive algorithm is performed:

- step 1. $q_m = r(m) - \sum_{i=1}^{m-1} a_{i(m-1)}r(m-i)$
- step 2. $\kappa_m = \frac{q_m}{E_{(m-1)}}$
- step 3. $a_{mm} = \kappa_m$
- step 4. $a_{im} = a_{i(m-1)} - \kappa_m a_{(m-i)(m-1)}$ for $i = 1, i = 2, \dots, i = m-1$
- step 5. $E_m = E_{m-1}(1 - \kappa_m^2)$
- step 6. If $m < M$ then increment m to $m+1$ and return to step 1. If $m = M$ then stop.

Let's run through an example in which $M = 4$, $r(0) = 11018$, $r(1) = 9690$, $r(2) = 8443$ and $r(3) = 7280$:

$$\begin{aligned} E_0 &= r(0) = 11018 \\ a_{11} &= \kappa_1 = \frac{r(1)}{E_0} = \frac{9690}{11018} = 0.8795 \\ E_1 &= E_0(1 - \kappa_1^2) = 11018(1 - 0.8795^2) = 2495 \\ q_2 &= r(2) - \sum_{i=1}^1 a_{i1}r(2-i) = 8443 - (0.8795)(9690) = -79.35 \\ \kappa_2 &= \frac{q_2}{E_1} = \frac{-79.35}{2495} = -0.0318 \\ a_{22} &= \kappa_2 = -0.0318 \\ a_{12} &= a_{11} - \kappa_2 a_{11} = 0.8795 - (-0.0318)(0.8795) = 0.9074 \\ E_2 &= E_1(1 - \kappa_2^2) = 2495(1 - (-0.0318)^2) = 2492 \\ q_3 &= r(3) - \sum_{i=1}^2 a_{i2}r(3-i) = 7280 - ((0.9074)(8443) + (-0.0318)(9690)) = -73.04 \\ \kappa_3 &= \frac{q_3}{E_2} = \frac{-73.04}{2492} = -0.0293 \\ a_{33} &= \kappa_3 = -0.0293 \\ a_{13} &= a_{12} - \kappa_3 a_{22} = 0.9074 - (-0.0293)(-0.0318) = 0.9065 \\ a_{23} &= a_{22} - \kappa_3 a_{12} = -0.0318 - (-0.0293)(0.9074) = -0.0052 \\ E_3 &= E_2(1 - \kappa_3^2) = 2492(1 - (-0.0293)^2) = 2490 \end{aligned}$$

Our final values are:

$$\begin{aligned} a_{11} &= 0.8795 \\ a_{12} &= 0.9074 \quad a_{22} = -0.0318 \\ a_{13} &= 0.9065 \quad a_{23} = -0.0052 \quad a_{33} = -0.0293 \\ E_1 &= 2495 \quad E_2 = 2492 \quad E_3 = 2490 \end{aligned}$$

These values have been rounded to the nearest significant digit and will not be an exact match to those generated by a computer.

† This algorithm is taken from <http://www.engineer.tamuk.edu/SPark/chap7.pdf>

1.5.4.11.4. Best order estimation

At this point, we have an array of prospective LP coefficient lists, a list of error values and must decide which LPC order to use. There are two ways to accomplish this: we can either estimate the total bits from the error values or perform an exhaustive search. Making the estimation requires the total number of samples in the subframe, the number of overhead bits per order (by default, this is the number of bits per sample in the subframe, plus 5), and an error scale constant in addition to the LPC error values:

$$Error\ Scale = \frac{\ln(2)^2}{2 \times Total\ Samples}$$

Once the error scale has been calculated, one can generate a 'Bits per Residual' estimation function which, given an LPC Error value, returns what its name implies:

$$Bits\ per\ Residual(LPC\ Error) = \frac{\ln(Error\ Scale \times LPC\ Error)}{2 \times \ln(2)}$$

With this function, we can estimate how many bits the entire LPC subframe will take for each LPC Error value and its associated Order:

$$Total\ Bits(LPC\ Error, Order) = \left(Bits\ per\ Residual(LPC\ Error) \times (Total\ Samples - Order) \right) + (Order \times Overhead\ Bits\ per\ Order)$$

Picking the best LPC Order is then done exhaustively by calculating the total estimated bits for each error value and using the order which uses the fewest.

1.5.4.11.5. Best order exhaustive search

In a curious bit of recursion, finding the best order for an LPC subframe via an exhaustive search requires taking each list of LP Coefficients calculated previously, quantizing them into a list of QLP Coefficients and a QLP Shift Needed value,[†] determining the total amount of bits each hypothetical LPC subframe uses and using the LPC order which uses the fewest.

Remember that building an LPC subframe requires the following values: LPC Order, QLP Precision, QLP Shift Needed and QLP Coefficients along with the subframe's samples and bits-per-sample. For each possible LPC Order, the QLP Shift Needed and the QLP Coefficient list values can be calculated by quantizing the LP Coefficients. QLP Precision is the size of each QLP Coefficient list value in the subframe header. Simply choose the field with the largest number of bits in the QLP Coefficient list for the QLP Precision value.

Finally, instead of writing these hypothetical LPC subframes directly to disk, one only has to capture how many bits they *would* use. The hypothetical LPC subframe that uses the fewest number of bits is the one we should actually write to disk.

[†] Quantizing coefficients will be covered in the next section.

1.5.4.11.6. Quantizing coefficients

Quantizing coefficients is a process of taking a list of LP Coefficients along with a QLP Coefficients Precision value and returning a list of QLP Coefficients and a QLP Shift Needed value. The first step is determining the upper and lower limits of the QLP Coefficients:

$$QLP\ coefficient\ maximum = 2^{precision} - 1$$

$$QLP\ coefficient\ minimum = -2^{precision}$$

The next step is determining the maximum shift limit and minimum shift limit constants, which are what their names imply:

$$max\ shift\ limit = 2^{QLP\ shift\ length-1} - 1 = 2^{5-1} - 1 = 2^4 - 1 = 16 - 1 = 15$$

$$min\ shift\ limit = -(max\ shift\ limit - 1) = -14$$

Now we determine the initial QLP Shift Needed value:

$$shift = precision - \left\lceil \frac{\log(\max(|LP\ Coefficients|))}{\log(2)} \right\rceil$$

where 'shift' is adjusted if necessary such that: $min\ shift\ limit \leq shift \leq max\ shift\ limit$

Finally, we determine the QLP Coefficient values themselves via a small recursive routine:

$$X(i) = E(i - 1) + (LP\ Coefficient_i \times 2^{shift})$$

$$QLP\ Coefficient_i = round(X(i))$$

$$E(i) = X(i) - QLP\ Coefficient_i$$

where $E(-1) = 0$ and each QLP Coefficient is adjusted prior to calculating the next $E(i)$ value such that: $QLP\ coefficient\ minimum \leq QLP\ Coefficient_i \leq QLP\ coefficient\ maximum$

The LPC Order, QLP Precision, QLP Shift Needed, and QLP Coefficients make up the LPC subframe.

1.5.4.11.7. Calculating Residual

A number of warm-up samples equal to LPC Order are taken from the input PCM and the subframe's residuals are calculated according to the following formula:

$$Residual_i = Sample_i - \left[\frac{\sum_{j=0}^{Order-1} QLP\ Coefficient_j \times Sample_{i-j-1}}{2^{QLP\ Shift\ Needed}} \right]$$

For example, given the samples 1053, 1116, 1257, 1423, 1529, 1570, 1623, 1694, 1688 and 1550, the coefficients:

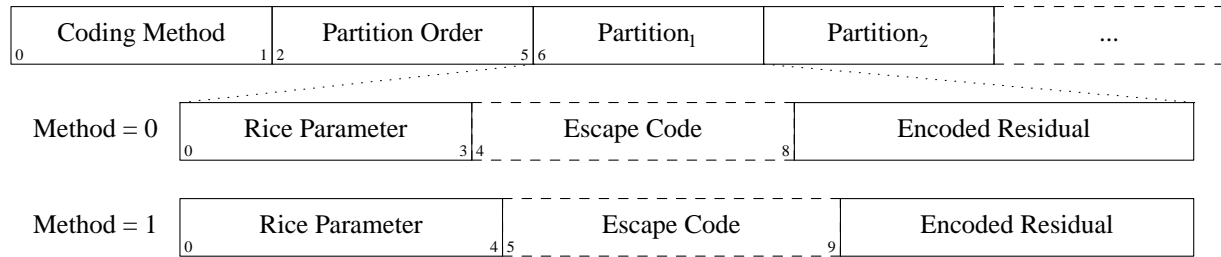
QLP Coefficient₀ 1241
 QLP Coefficient₁ -944
 QLP Coefficient₂ 14
 QLP Coefficient₃ 342
 QLP Coefficient₄ -147

and a QLP Shift Needed value of 9, our residuals are as follows:

Index	Sample	Residual
0	(warm-up) 1053	
1	(warm-up) 1116	
2	(warm-up) 1257	
3	(warm-up) 1423	
4	(warm-up) 1529	
5	1570	$1570 - \frac{(1241 \times 1529) + (-944 \times 1423) + (14 \times 1257) + (342 \times 1116) + (-147 \times 1053)}{2^9} = 1570 - \frac{798656}{512} = \mathbf{11}$
6	1623	$1623 - \frac{(1241 \times 1570) + (-944 \times 1529) + (14 \times 1423) + (342 \times 1257) + (-147 \times 1116)}{2^9} = 1623 - \frac{790758}{512} = \mathbf{79}$
7	1694	$1694 - \frac{(1241 \times 1623) + (-944 \times 1570) + (14 \times 1529) + (342 \times 1423) + (-147 \times 1257)}{2^9} = 1694 - \frac{855356}{512} = \mathbf{24}$
8	1688	$1688 - \frac{(1241 \times 1694) + (-944 \times 1623) + (14 \times 1570) + (342 \times 1529) + (-147 \times 1423)}{2^9} = 1688 - \frac{905859}{512} = \mathbf{-81}$
9	1550	$1550 - \frac{(1241 \times 1688) + (-944 \times 1694) + (14 \times 1623) + (342 \times 1570) + (-147 \times 1529)}{2^9} = 1550 - \frac{830571}{512} = \mathbf{-72}$

1.5.4.12. the Residual

Given a stream of residual values, one must place them in one or more partitions, each with its own Rice parameter, and prepended with a small header:



The residual’s coding method is typically 0, unless one is encoding audio with more than 16 bits-per-sample and one of the partitions requests a Rice parameter higher than 2^4 . The residual’s partition order is chosen exhaustively, which means trying all of them within a certain range (e.g. 0 to 5) such that the residuals can be divided evenly between them and then the partition order which uses the smallest estimated amount of space is chosen.

Choosing the best Rice parameter is a matter of selecting the smallest value of ‘x’ such that:

$$sample\ count \times 2^x > \sum_{i=0}^{residual\ count - 1} |residual_i|$$

Again, this is easier to understand with a block of example residuals, 19 in total:

index	$residual_i$	$ residual_i $
0	-1	1
1	1	1
2	1	1
3	1	1
4	0	0
5	3	3
6	0	0
7	-4	4
8	-1	1
9	0	0
10	1	1
11	1	1
12	1	1
13	4	4
14	-3	3
15	1	1
16	4	4
17	-1	1
18	-1	1
sum		29

19×2^0 is not larger than 29.

19×2^1 is larger than 29, so the best Rice parameter for this block of residuals is 1.

Remember that the Rice parameter’s maximum value is limited to 2^4 using coding method 0, or 2^5 using coding method 1.

1.5.5. the Checksums

Calculating the frame header's CRC-8 and frame footer's CRC-16 is necessary both for FLAC encoders and decoders, but the process is the same for each.

1.5.5.1. CRC-8

CRC-8 is used to checksum FLAC frame headers. Given a byte of input and the previous CRC-8 checksum, or 0 as an initial value, the current checksum can be calculated as follows:

$$checksum_i = CRC8(byte \text{ xor } checksum_{i-1})$$

CRC8																
	0x?0	0x?1	0x?2	0x?3	0x?4	0x?5	0x?6	0x?7	0x?8	0x?9	0x?A	0x?B	0x?C	0x?D	0x?E	0x?F
0x0?	0x00	0x07	0x0E	0x09	0x1C	0x1B	0x12	0x15	0x38	0x3F	0x36	0x31	0x24	0x23	0x2A	0x2D
0x1?	0x70	0x77	0x7E	0x79	0x6C	0x6B	0x62	0x65	0x48	0x4F	0x46	0x41	0x54	0x53	0x5A	0x5D
0x2?	0xE0	0xE7	0xEE	0xE9	0xFC	0xFB	0xF2	0xF5	0xD8	0xDF	0xD6	0xD1	0xC4	0xC3	0xCA	0xCD
0x3?	0x90	0x97	0x9E	0x99	0x8C	0x8B	0x82	0x85	0xA8	0xAF	0xA6	0xA1	0xB4	0xB3	0xBA	0xBD
0x4?	0xC7	0xC0	0xC9	0xCE	0xDB	0xDC	0xD5	0xD2	0xFF	0xF8	0xF1	0xF6	0xE3	0xE4	0xED	0xEA
0x5?	0xB7	0xB0	0xB9	0xBE	0xAB	0xAC	0xA5	0xA2	0x8F	0x88	0x81	0x86	0x93	0x94	0x9D	0x9A
0x6?	0x27	0x20	0x29	0x2E	0x3B	0x3C	0x35	0x32	0x1F	0x18	0x11	0x16	0x03	0x04	0x0D	0x0A
0x7?	0x57	0x50	0x59	0x5E	0x4B	0x4C	0x45	0x42	0x6F	0x68	0x61	0x66	0x73	0x74	0x7D	0x7A
0x8?	0x89	0x8E	0x87	0x80	0x95	0x92	0x9B	0x9C	0xB1	0xB6	0xBF	0xB8	0xAD	0xAA	0xA3	0xA4
0x9?	0xF9	0xFE	0xF7	0xF0	0xE5	0xE2	0xEB	0xEC	0xC1	0xC6	0xCF	0xC8	0xDD	0xDA	0xD3	0xD4
0xA?	0x69	0x6E	0x67	0x60	0x75	0x72	0x7B	0x7C	0x51	0x56	0x5F	0x58	0x4D	0x4A	0x43	0x44
0xB?	0x19	0x1E	0x17	0x10	0x05	0x02	0x0B	0x0C	0x21	0x26	0x2F	0x28	0x3D	0x3A	0x33	0x34
0xC?	0x4E	0x49	0x40	0x47	0x52	0x55	0x5C	0x5B	0x76	0x71	0x78	0x7F	0x6A	0x6D	0x64	0x63
0xD?	0x3E	0x39	0x30	0x37	0x22	0x25	0x2C	0x2B	0x06	0x01	0x08	0x0F	0x1A	0x1D	0x14	0x13
0xE?	0xAE	0xA9	0xA0	0xA7	0xB2	0xB5	0xBC	0xBB	0x96	0x91	0x98	0x9F	0x8A	0x8D	0x84	0x83
0xF?	0xDE	0xD9	0xD0	0xD7	0xC2	0xC5	0xCC	0xCB	0xE6	0xE1	0xE8	0xEF	0xFA	0xFD	0xF4	0xF3

For example, given the header bytes: 0xFF, 0xF8, 0xCC, 0x1C, 0x00 and 0xC0:

$$checksum_0 = CRC8(0xFF \text{ xor } 0) = CRC8(0xFF) = 0xF3$$

$$checksum_1 = CRC8(0xF8 \text{ xor } 0xF3) = CRC8(0x0B) = 0x31$$

$$checksum_2 = CRC8(0xCC \text{ xor } 0x31) = CRC8(0xFD) = 0xFD$$

$$checksum_3 = CRC8(0x1C \text{ xor } 0xFD) = CRC8(0xE1) = 0xA9$$

$$checksum_4 = CRC8(0x00 \text{ xor } 0xA9) = CRC8(0xA9) = 0x56$$

$$checksum_5 = CRC8(0xC0 \text{ xor } 0x56) = CRC8(0x96) = 0xEB$$

Thus, the next byte after the header should be 0xEB. Furthermore, when the checksum byte itself is run through the checksumming procedure:

$$checksum_6 = CRC8(0xEB \text{ xor } 0xEB) = CRC8(0x00) = 0x00$$

the result will always be 0. This is a handy way to verify a frame header's checksum since the checksum of the header's bytes along with the header's checksum itself will always result in 0.

1.5.5.2. CRC-16

CRC-16 is used to checksum the entire FLAC frame, including the header and any padding bits after the final subframe. Given a byte of input and the previous CRC-16 checksum, or 0 as an initial value, the current checksum can be calculated as follows:

$$checksum_i = CRC16(byte \text{ xor } (checksum_{i-1} \gg 8)) \text{ xor } (checksum_{i-1} \ll 8)$$

the checksum is always truncated to 16-bits.

CRC16																
	0x?0	0x?1	0x?2	0x?3	0x?4	0x?5	0x?6	0x?7	0x?8	0x?9	0x?A	0x?B	0x?C	0x?D	0x?E	0x?F
0x0?	0x0000	0x8005	0x800f	0x000a	0x801b	0x001e	0x0014	0x8011	0x8033	0x0036	0x003c	0x8039	0x0028	0x802d	0x8027	0x0022
0x1?	0x8063	0x0066	0x006c	0x8069	0x0078	0x807d	0x8077	0x0072	0x0050	0x8055	0x805f	0x005a	0x804b	0x004e	0x0044	0x8041
0x2?	0x80c3	0x00c6	0x00cc	0x80c9	0x00d8	0x80dd	0x80d7	0x00d2	0x00f0	0x80f5	0x80ff	0x00fa	0x80eb	0x00ee	0x00e4	0x80e1
0x3?	0x00a0	0x80a5	0x80af	0x00aa	0x80bb	0x00be	0x00b4	0x80b1	0x8093	0x0096	0x009c	0x8099	0x0088	0x808d	0x8087	0x0082
0x4?	0x8183	0x0186	0x018c	0x8189	0x0198	0x819d	0x8197	0x0192	0x01b0	0x81b5	0x81bf	0x01ba	0x81ab	0x01ae	0x01a4	0x81a1
0x5?	0x01e0	0x81e5	0x81ef	0x01ea	0x81fb	0x01fe	0x01f4	0x81f1	0x81d3	0x01d6	0x01dc	0x81d9	0x01c8	0x81cd	0x81c7	0x01c2
0x6?	0x0140	0x8145	0x814f	0x014a	0x815b	0x015e	0x0154	0x8151	0x8173	0x0176	0x017c	0x8179	0x0168	0x816d	0x8167	0x0162
0x7?	0x8123	0x0126	0x012c	0x8129	0x0138	0x813d	0x8137	0x0132	0x0110	0x8115	0x811f	0x011a	0x810b	0x010e	0x0104	0x8101
0x8?	0x8303	0x0306	0x030c	0x8309	0x0318	0x831d	0x8317	0x0312	0x0330	0x8335	0x833f	0x033a	0x832b	0x032e	0x0324	0x8321
0x9?	0x0360	0x8365	0x836f	0x036a	0x837b	0x037e	0x0374	0x8371	0x8353	0x0356	0x035c	0x8359	0x0348	0x834d	0x8347	0x0342
0xA?	0x03c0	0x83c5	0x83cf	0x03ca	0x83db	0x03de	0x03d4	0x83d1	0x83f3	0x03f6	0x03fc	0x83f9	0x03e8	0x83ed	0x83e7	0x03e2
0xB?	0x83a3	0x03a6	0x03ac	0x83a9	0x03bb	0x83bd	0x83b7	0x03b2	0x0390	0x8395	0x839f	0x039a	0x838b	0x038e	0x0384	0x8381
0xC?	0x0280	0x8285	0x828f	0x028a	0x829b	0x029e	0x0294	0x8291	0x82b3	0x02b6	0x02bc	0x82b9	0x02a8	0x82ad	0x82a7	0x02a2
0xD?	0x82e3	0x02e6	0x02ec	0x82e9	0x02f8	0x82fd	0x82f7	0x02f2	0x02d0	0x82d5	0x82df	0x02da	0x82cb	0x02ce	0x02c4	0x82c1
0xE?	0x8243	0x0246	0x024c	0x8249	0x0258	0x825d	0x8257	0x0252	0x0270	0x8275	0x827f	0x027a	0x826b	0x026e	0x0264	0x8261
0xF?	0x0220	0x8225	0x822f	0x022a	0x823b	0x023e	0x0234	0x8231	0x8213	0x0216	0x021c	0x8219	0x0208	0x820d	0x8207	0x0202

For example, given the frame bytes: 0xFF, 0xF8, 0xCC, 0x1C, 0x00, 0xC0, 0xEB, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 and 0x00, the frame's CRC-16 can be calculated as follows:

$$checksum_0 = CRC16(0xFF \text{ xor } (0 \gg 8)) \text{ xor } (0 \ll 8) = CRC16(0xFF) \text{ xor } 0 = 0x0202$$

$$checksum_1 = CRC16(0xF8 \text{ xor } (0x0202 \gg 8)) \text{ xor } (0x0202 \ll 8) = CRC16(0xFA) \text{ xor } 0x0200 = 0x001C$$

$$checksum_2 = CRC16(0xCC \text{ xor } (0x001C \gg 8)) \text{ xor } (0x001C \ll 8) = CRC16(0xCC) \text{ xor } 0x1C00 = 0x1EA8$$

$$checksum_3 = CRC16(0x1C \text{ xor } (0x1EA8 \gg 8)) \text{ xor } (0x1EA8 \ll 8) = CRC16(0x02) \text{ xor } 0xA800 = 0x280F$$

$$checksum_4 = CRC16(0x00 \text{ xor } (0x280F \gg 8)) \text{ xor } (0x280F \ll 8) = CRC16(0x28) \text{ xor } 0xF000 = 0x0FF0$$

$$checksum_5 = CRC16(0xC0 \text{ xor } (0x0FF0 \gg 8)) \text{ xor } (0x0FF0 \ll 8) = CRC16(0xCF) \text{ xor } 0xF000 = 0xF2A2$$

$$checksum_6 = CRC16(0xEB \text{ xor } (0xF2A2 \gg 8)) \text{ xor } (0xF2A2 \ll 8) = CRC16(0x19) \text{ xor } 0xA200 = 0x2255$$

$$checksum_7 = CRC16(0x00 \text{ xor } (0x2255 \gg 8)) \text{ xor } (0x2255 \ll 8) = CRC16(0x22) \text{ xor } 0x5500 = 0x55CC$$

$$checksum_8 = CRC16(0x00 \text{ xor } (0x55CC \gg 8)) \text{ xor } (0x55CC \ll 8) = CRC16(0x55) \text{ xor } 0xCC00 = 0xCDFE$$

$$checksum_9 = CRC16(0x00 \text{ xor } (0xCDFE \gg 8)) \text{ xor } (0xCDFE \ll 8) = CRC16(0xCD) \text{ xor } 0xFE00 = 0x7CAD$$

$$checksum_{10} = CRC16(0x00 \text{ xor } (0x7CAD \gg 8)) \text{ xor } (0x7CAD \ll 8) = CRC16(0x7C) \text{ xor } 0xAD00 = 0x2C0B$$

$$checksum_{11} = CRC16(0x00 \text{ xor } (0x2C0B \gg 8)) \text{ xor } (0x2C0B \ll 8) = CRC16(0x2C) \text{ xor } 0xB000 = 0x8BEB$$

$$checksum_{12} = CRC16(0x00 \text{ xor } (0x8BEB \gg 8)) \text{ xor } (0x8BEB \ll 8) = CRC16(0x8B) \text{ xor } 0xEB00 = 0xE83A$$

$$checksum_{13} = CRC16(0x00 \text{ xor } (0xE83A \gg 8)) \text{ xor } (0xE83A \ll 8) = CRC16(0xE8) \text{ xor } 0x3A00 = 0x3870$$

$$checksum_{14} = CRC16(0x00 \text{ xor } (0x3870 \gg 8)) \text{ xor } (0x3870 \ll 8) = CRC16(0x38) \text{ xor } 0x7000 = 0xF093$$

Thus, the next two bytes after the final subframe should be 0xF0 and 0x93. Again, when the checksum bytes are run through the checksumming procedure:

$$checksum_{15} = CRC16(0xF0 \text{ xor } (0xF093 \gg 8)) \text{ xor } (0xF093 \ll 8) = CRC16(0x00) \text{ xor } 0x9300 = 0x9300$$

$$checksum_{16} = CRC16(0x93 \text{ xor } (0x9300 \gg 8)) \text{ xor } (0x9300 \ll 8) = CRC16(0x00) \text{ xor } 0x0000 = 0x0000$$

the result will also always be 0, just as in the CRC-8.

1.6. Monkey's Audio

Monkey's Audio is a lossless RIFF WAVE compressor. Unlike FLAC, which is a PCM compressor, Monkey's Audio also stores IFF chunks and reproduces the original WAVE file in its entirety rather than storing only the data it contains. All of its fields are little-endian.

1.6.1. the Monkey's Audio stream

APE Descriptor 0 415	APE Header 416 607	Seektable	Header Data	Frame ₁	Frame ₂	...	APEv2 Tag
-------------------------	-----------------------	-----------	-------------	--------------------	--------------------	-----	-----------

1.6.2. the APE Descriptor

0	ID ('MAC' 0x4D414320)	31	32	Version	63
64	Descriptor Bytes	95	96	Header Bytes	127
128	Seektable Bytes	159	160	Header Data Bytes	191
192	Frame Data Bytes	223	224	Frame Data Bytes (High)	255
256	Terminating Data Bytes				287
288	MD5 Sum				415

Version is the encoding software's version times 1000. i.e. Monkey's Audio 3.99 = 3990

1.6.3. the APE Header

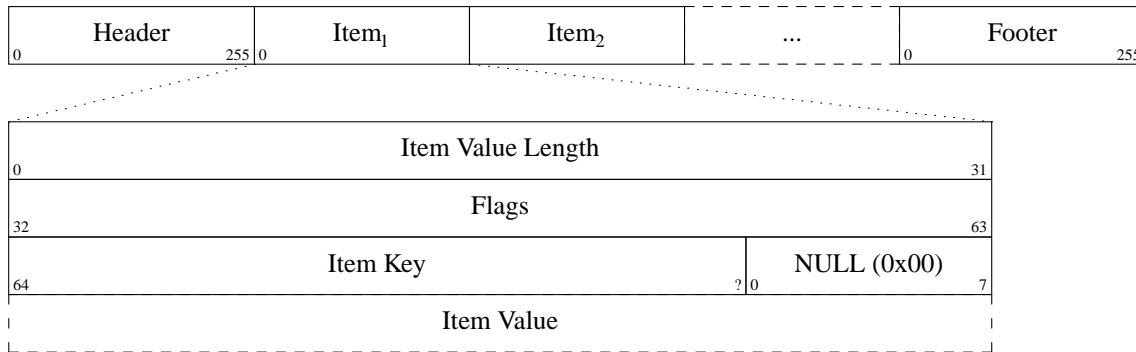
0	Compression Level	15	16	Format Flags	31
32	Blocks Per Frame				63
64	Final Frame Blocks				95
96	Total Frames				127
128	Bits Per Sample	143	144	Channels	159
160	Sample Rate				191

'Bits Per Sample', 'Channels' and 'Sample Rate' are self-explanatory. A 'block' refers to a group of samples to be sent across all channels at any one instant. For instance, a file with a sample rate of 44100 has exactly 44100 blocks per second, regardless of the number of channels or bits per sample. Calculating the total length of the file is a matter of multiplying the number of frames by the number of blocks per frame and adding any remaining blocks in the final frame.

$$\text{Length in Seconds} = \frac{((\text{Total Frames} - 1) \times \text{Blocks Per Frame}) + \text{Final Frame Blocks}}{\text{Sample Rate}}$$

1.6.4. the APEv2 tag

The APEv2 tag is a little-endian metadata tag appended to Monkey's Audio files.



Item Key is an ASCII string from the range 0x20 to 0x7E. Item Value is typically a UTF-8 encoded string, but may also be binary depending on the Flags.

Item Keys			
key	value	key	value
Abstract	Abstract	Album	album name
Artist	performing artist	Bibliography	Bibliography/Discography
Catalog	catalog number	Comment	user comment
Composer	original composer	Conductor	conductor
Copyright	copyright holder	Debut album	debut album name
Dummy	place holder	EAN/UPC	EAN-13/UPC-A bar code identifier
File	file location	Genre	genre
Index	indexes for quick access	Introplay	characteric part of piece for intro playing
ISBN	ISBN number with check digit	ISRC	International Standard Recording Number
Language	used Language(s) for music/spoken words	LC	Label Code
Media	source media	Publicationright	publication right holder
Publisher	record label or publisher	Record Date	record date
Record Location	record location	Related	location of related information
Subtitle	track subtitle	Title	track title
Track	track number	Year	release date

1.6.5. the APEv2 tag header/footer

0		Preamble ('APETAGEX' 0x4150455441474558)		63	
64		Version (0xD0070000)	95	96	Tag Size
128		Item Count	159	160	Flags
192		Reserved		255	

The format of the APEv2 header and footer are identical except for one of the flags. 'Version' is typically 2000 (stored little-endian). 'Tag Size' is the size of the entire APEv2 tag, including the footer but excluding the header. 'Item Count' is the number of individual tag items.

1.6.6. the APEv2 flags

The 'Flags' field is used by both the APEv2 header/footer and the individual tag items.

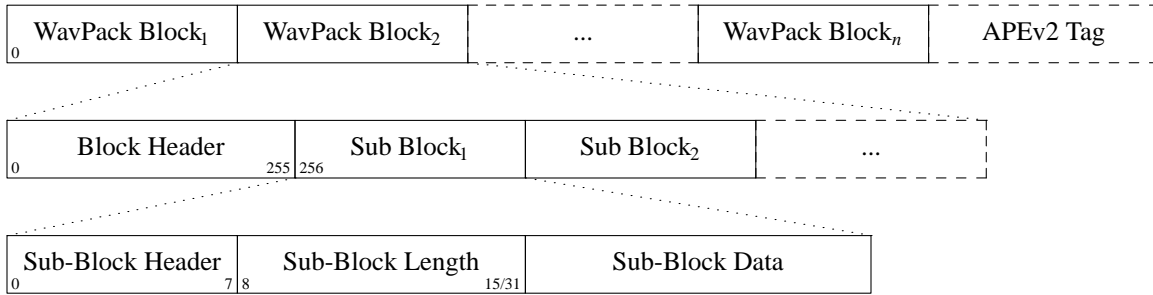
0		Undefined (0x00)		4		5		Encoding		6		7		Read-Only					
8		Undefined (0x00)												15					
16		Undefined (0x00)												23					
Container Header				Contains no Footer				Is Header				Undefined (0x00)							
24				25				26				27				31			

Encoding	
bits	value
00	UTF-8
01	Binary
10	External Link
11	Reserved

1.7. WavPack

WavPack is a lossless audio format. It is a sequence of WavPack blocks followed by an optional APEv2 tag. Each block consists of one or more sub-blocks which contain the compressed audio data or metadata. All of its fields are little-endian.

1.7.1. the WavPack file stream



1.7.2. a WavPack block header

Block ID 'wvpk' (0x7776706B)				0	31
Block Size				32	63
Version		Track Number		Index Number	
64	79	80	87	88	95
Total Samples				96	127
Block Index				128	159
Block Samples				160	191
Floating Point Data		Hybrid Noise Shaping		Channel Decorrelation	
192		193		194	
Hybrid Mode		Mono Output		Bits-per-Sample	
196		197		198	
Left Shift Data (low)				Final Block	
200				202	
Initial Block		Hbd. Noise Balanced		Hbd. Controls Bitrate	
204		205		206	
Sampling Rate (low)		Maximum Magnitude			
208		209			
Maximum Magnitude (cont.)			Left Shift Data (high)		
212			213		
Reserved		False Stereo		Use IIR	
216		217		218	
Reserved		Sampling Rate (high)			
220		221			
CRC				224	
				255	

'Block Size' is the length of everything past everything past the block header, minus 24 bytes.

Bits-per-Sample	
bits	per-sample
00	8
01	16
10	24
11	32

Sampling Rate			
bits	rate	bits	rate
0000	6000	1000	32000
0001	8000	1001	44100
0010	9600	1010	48000
0011	11025	1011	64000
0100	12000	1100	88200
0101	16000	1101	96000
0110	22050	1110	192000
0111	24000	1111	reserved

The 'flags' field is stored as a little-endian 32-bit integer. Since some fields cross byte boundaries, their high and low bits will be far apart when written in this format where the bits are ordered the way they appear in the file.

The 'Mono Output' bit indicates the channel count. If 1, this block has 1 channel. If 0, this block has 2 channels. For an audio stream with more than 2 channels, check the 'Initial Block' and 'Final Block' bits to indicate the start and end of the channels. As an example:

Initial Block	Final Block	Mono Output	Channels
1	0	0	2
0	0	1	1
0	0	1	1
0	1	0	2
Total			6

1.7.3. a WavPack sub-block header

Large Block 0	Actual Size 1 Less 1	Nondecoder Data 2	Metadata Function 3 7
Block Size 8 15/31			
Block Data			

If the 'Large Block' field is 0, the 'Block Size' field is 8 bits long. If it is 1, the 'Block Size' field is 24 bits long. The 'Block Size' field is the length of 'Block Data', in 16-bit words rather than bytes. If 'Actual Size 1 Less' is set, that means 'Block Data' doesn't contain an even number of bytes; it is padded with a single null byte at the end in order to fit. If 'Nondecoder Data' is set, that means the decoder does not have to understand the contents of this particular sub-block in order to decode the audio.

1.8. MP3

MP3 is the de-facto standard for lossy audio. It is little more than a series of MPEG frames with an optional ID3v2 metadata header and optional ID3v1 metadata footer.

MP3 decoders are assumed to be very tolerant of anything in the stream that doesn't look like an MPEG frame, ignoring such junk until the next frame is found. Since MP3 files have no standard container format in which non-MPEG data can be placed, metadata such as ID3 tags are often made 'sync-safe' by formatting them in a way that decoders won't confuse tags for MPEG frames.

1.8.1. the MP3 file stream



1.8.2. an MPEG frame header

Frame Sync (all set)										
0										7
Frame Sync				MPEG ID				Layer Description		Prot.
8	10	11	12	13	14	15				
Bitrate					Sampling		Pad	Private		
16	19	20	21	22	23					
Channel			Mode Extension		Copyright	Original	Emphasis			
24	25	26	27	28	29	30	31			

Layer I frames always contain 384 samples.

Layer II and Layer III frames always contain 1152 samples.

To find the total size of an MPEG frame, use one of the following formulas:

- Layer I : $Byte\ Length = \left(\frac{12 \times Bitrate}{Sampling} + Pad \right) \times 4$
- Layer II/III : $Byte\ Length = \frac{144 \times Bitrate}{Sampling} + Pad$

For example, an MPEG-1 Layer III frame with a sampling rate of 44100, a bitrate of 128kbps and a set pad bit is 418 bytes long, including the header.

$$418 = \frac{144 \times 128000}{44100} + 1$$

MPEG ID		Layer Description	
bits	version	bits	layer
00	MPEG 2.5	00	reserved
01	reserved	01	Layer III
10	MPEG 2	10	Layer II
11	MPEG 1	11	Layer I

If the 'Protection' bit is set, the frame header is followed by a 16 bit CRC.

Sampling			
bits	MPEG-1	MPEG-2	MPEG-2.5
00	44100	22050	11025
01	48000	24000	12000
10	32000	16000	8000
11	reserved	reserved	reserved

Channel	
bits	type
00	Stereo
01	Joint stereo
10	Dual channel stereo
11	Mono

bits	Bitrate (in 1000 bits per second)				
	MPEG-1 Layer-1	MPEG-1 Layer-2	MPEG-1 Layer-3	MPEG-2 Layer-1	MPEG-2 Layer-2/3
0000	free	free	free	free	free
0001	32	32	32	32	8
0010	64	48	40	48	16
0011	96	56	48	56	24
0100	128	64	56	64	32
0101	160	80	64	80	40
0110	192	96	80	96	48
0111	224	112	96	112	56
1000	256	128	112	128	64
1001	288	160	128	144	80
1010	320	192	160	160	96
1011	352	224	192	176	112
1100	384	256	224	192	128
1101	416	320	256	224	144
1110	448	384	320	256	160
1111	bad	bad	bad	bad	bad

1.8.2.1. the Xing header

An MP3 frame header contains the track's sampling rate, bits-per-sample and number of channels. However, because MP3 files are little more than concatenated MPEG frames, there is no obvious place to store the track's total length. Since the length of each frame is a constant number of samples, one can calculate the track length by counting the number of frames. This method is the most accurate but is also quite slow.

For MP3 files in which all frames have the same bitrate - also known as constant bitrate, or CBR files - one can divide the total size of file (minus any ID3 headers/footers), by the bitrate to determine its length. If an MP3 file has no Xing header in its first frame, one can assume it is CBR.

An MP3 file that does contain a Xing header in its first frame can be assumed to be variable bitrate, or VBR. In that case, the rate of the first frame cannot be used as a basis to calculate the length of the entire file. Instead, one must use the information from the Xing header which contains that length.

All of the fields within a Xing header are big-endian.

0		Header 'Xing' (0x58696E67)		31							
32		Flags		63							
64		Number of Frames		95							
96		Bytes		127							
128	TOC Entry ₁	135	136	TOC Entry ₂	143	144	...	919	920	TOC Entry ₁₀₀	927
928		Quality		959							

1.8.3. the ID3v1 tag

ID3v1 tags are very simple metadata tags appended to an MP3 file. All of the fields are fixed length and the text encoding is undefined. There are two versions of ID3v1 tags. ID3v1.1 has a track number field as a 1 byte value at the end of the comment field. If the byte just before the end is not null (0x00), assume we're dealing with a classic ID3v1 tag without a track number.

Obviously, ID3v1 tags are insufficient for any nontrivial metadata. They are often still added for compatibility reasons, however.

1.8.3.1. ID3v1

0	Header ('TAG' 0x544147)		23
24	Track Title		263
264	Artist Name		503
504	Album Name		743
744	Year	775	
776	Comment		1015
1016	Genre	1023	

1.8.3.2. ID3v1.1

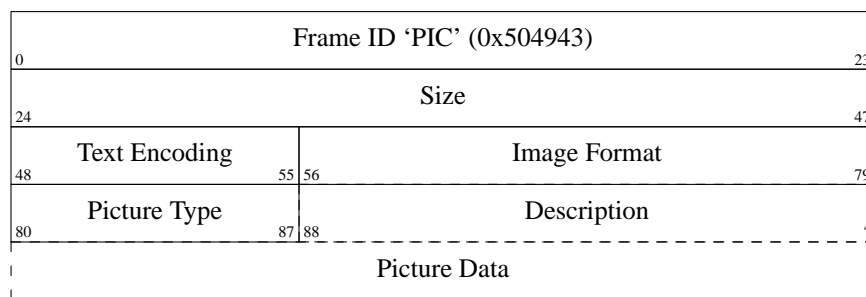
0	Header ('TAG' 0x544147)		23
24	Track Title		263
264	Artist Name		503
504	Album Name		743
744	Year	775	
776	Comment	999	
		1000	1007
		1008	1015
1016	Genre	1023	

1.8.4.2.3. ID3v2.2 Frame IDs

ID	Description	ID	Description
BUF	Recommended buffer size	CNT	Play counter
COM	Comments	CRA	Audio encryption
CRM	Encrypted meta frame	ETC	Event timing codes
EQU	Equalization	GEO	General encapsulated object
IPL	Involved people list	LNK	Linked information
MCI	Music CD Identifier	MLL	MPEG location lookup table
PIC	Attached picture	POP	Popularimeter
REV	Reverb	RVA	Relative volume adjustment
SLT	Synchronized lyric/text	STC	Synced tempo codes
TAL	Album/Movie/Show title	TBP	BPM (Beats Per Minute)
TCM	Composer	TCO	Content type
TCR	Copyright message	TDA	Date
TDY	Playlist delay	TEN	Encoded by
TFT	File type	TIM	Time
TKE	Initial key	TLA	Language(s)
TLE	Length	TMT	Media type
TOA	Original artist(s)/performer(s)	TOF	Original filename
TOL	Original Lyricist(s)/text writer(s)	TOR	Original release year
TOT	Original album/Movie/Show title	TP1	Lead artist(s)/Lead performer(s)/Soloist(s)/Performing group
TP2	Band/Orchestra/Accompaniment	TP3	Conductor/Performer refinement
TP4	Interpreted, remixed, or otherwise modified by	TPA	Part of a set
TPB	Publisher	TRC	ISRC (International Standard Recording Code)
TRD	Recording dates	TRK	Track number/Position in set
TSI	Size	TSS	Software/hardware and settings used for encoding
TT1	Content group description	TT2	Title/Songname/Content description
TT3	Subtitle/Description refinement	TXT	Lyricist/text writer
TXX	User defined text information frame	TYE	Year
UFI	Unique file identifier	ULT	Unsynchronized lyric/text transcription
WAF	Official audio file webpage	WAR	Official artist/performer webpage
WAS	Official audio source webpage	WCM	Commercial information
WCP	Copyright/Legal information	WPB	Publishers official webpage
WXX	User defined URL link frame		

1.8.4.2.4. the PIC Frame

'PIC' frames are attached pictures. This allows an ID3v2.2 tag to contain a JPEG or PNG image, typically of album artwork which can be displayed to the user when the track is played.



Text Encoding is the encoding of the Description field. Its value is either ISO-8859-1 or UCS-16 - the same as in text frames. Image Format is a 3 byte string indicating the format of the image, typically 'JPG' for JPEG images or 'PNG' for PNG images. Description is a NULL-terminated C-string which contains a text description of the image.

Picture Types			
value	type	value	type
0	Other	1	32x32 pixels 'file icon' (PNG only)
2	Other file icon	3	Cover (front)
4	Cover (back)	5	Leaflet page
6	Media (e.g. label side of CD)	7	Lead artist / Lead performer / Soloist
8	Artist / Performer	9	Conductor
10	Band / Orchestra	11	Composer
12	Lyricist / Text writer	13	Recording location
14	During recording	15	During performance
16	Movie / Video screen capture	17	A bright coloured fish
18	Illustration	19	Band / Artist logotype
20	Publisher / Studio logotype		

1.8.4.3. ID3v2.3

1.8.4.3.1. the ID3v2.3 Header

ID ('ID3' 0x494433)										Version (0x0300)						
Unsync				Extended		Experimental		Footer		NULL (0x00)						
0x00		Size			0x00		Size		0x00		Size		0x00		Size	

The single Size field is split by NULL (0x00) bytes in order to make it 'sync-safe'. That is, no possible size value will result in a false MP3 frame sync (11 bits set in a row).

1.8.4.3.2. an ID3v2.3 Frame

Frame ID									
Size									
Tag Alter		File Alter		Read Only		0x00			
Compression		Encryption		Grouping		0x00			
Frame Data									

Frame ID's that begin with the letter 'T' (0x54) are text frames. These have an additional text encoding byte before the actual text data. All text strings are terminated by a null character (0x00 or 0x0000, depending on the encoding).

Frame ID 'TXXX' (0x54XXXXXX)									
Size									
Tag Alter		File Alter		Read Only		0x00			
Compression		Encryption		Grouping		0x00			
Encoding				Text					

Encoding Byte	Text Encoding
0x00	ISO-8859-1
0x01	UCS-16

1.8.4.3.3. ID3v2.3 Frame IDs

ID	Description	ID	Description
AENC	Audio encryption	APIC	Attached picture
COMM	Comments	COMR	Commercial frame
ENCR	Encryption method registration	EQUA	Equalization
ETCO	Event timing codes	GEOB	General encapsulated object
GRID	Group identification registration	IPLS	Involved people list
LINK	Linked information	MCDI	Music CD identifier
MLLT	MPEG location lookup table	OWNE	Ownership frame
PRIV	Private frame	PCNT	Play counter
POPM	Popularimeter	POSS	Position synchronisation frame
RBUF	Recommended buffer size	RVAD	Relative volume adjustment
RVRB	Reverb	SYLT	Synchronized lyric/text
SYTC	Synchronized tempo codes	TALB	Album/Movie/Show title
TBPM	BPM (beats per minute)	TCOM	Composer
TCO	Content type	TCOP	Copyright message
TDAT	Date	TDLY	Playlist delay
TENC	Encoded by	TEXT	Lyricist/Text writer
TFLT	File type	TIME	Time
TIT1	Content group description	TIT2	Title/songname/content description
TIT3	Subtitle/Description refinement	TKEY	Initial key
TLAN	Language(s)	TLEN	Length
TMED	Media type	TOAL	Original album/movie/show title
TOFN	Original filename	TOLY	Original lyricist(s)/text writer(s)
TOPE	Original artist(s)/performer(s)	TORY	Original release year
TOWN	File owner/licensee	TPE1	Lead performer(s)/Soloist(s)
TPE2	Band/orchestra/accompaniment	TPE3	Conductor/performer refinement
TPE4	Interpreted, remixed, or otherwise modified by	TPOS	Part of a set
TPUB	Publisher	TRCK	Track number/Position in set
TRDA	Recording dates	TRSN	Internet radio station name
TRSO	Internet radio station owner	TSIZ	Size
TSRC	ISRC (international standard recording code)	TSSE	Software/Hardware and settings used for encoding
TYER	Year	TXXX	User defined text information frame
UFID	Unique file identifier	USER	Terms of use
USLT	Unsynchronized lyric/text transcription	WCOM	Commercial information
WCOP	Copyright/Legal information	WOAF	Official audio file webpage
WOAR	Official artist/performer webpage	WOAS	Official audio source webpage
WORS	Official internet radio station homepage	WPAY	Payment
WPUB	Publishers official webpage	WXXX	User defined URL link frame

1.8.4.3.4. the APIC Frame

'APIC' frames are attached pictures.

Frame ID 'APIC' (0x41504943)				0	31
Size				32	63
Tag Alter 64	File Alter 65	Read Only 66	0x00	67	71
Compression 72	Encryption 73	Grouping 74	0x00	75	79
Text Encoding 80		MIME Type 87 88			?
Picture Type 0		Description 7			?
Picture Data					

Text Encoding is the encoding of the Description field. Its value is either ISO-8859-1 or UCS-16 - the same as in text frames. MIME Type is a NULL-terminated, ASCII C-string which contains the image's MIME type, such as 'image/jpeg' or 'image/png'. Description is a NULL-terminated C-string which contains a text description of the image.

Picture Types			
value	type	value	type
0	Other	1	32x32 pixels 'file icon' (PNG only)
2	Other file icon	3	Cover (front)
4	Cover (back)	5	Leaflet page
6	Media (e.g. label side of CD)	7	Lead artist / Lead performer / Soloist
8	Artist / Performer	9	Conductor
10	Band / Orchestra	11	Composer
12	Lyricist / Text writer	13	Recording location
14	During recording	15	During performance
16	Movie / Video screen capture	17	A bright coloured fish
18	Illustration	19	Band / Artist logotype
20	Publisher / Studio logotype		

1.8.4.4. ID3v2.4

1.8.4.4.1. the ID3v2.4 Header

ID ('ID3' 0x494433)										Version (0x0400)			
Unsync		Extended		Experimental		Footer		NULL (0x00)					
0x00	Size	0x00	Size	0x00	Size	0x00	Size	0x00	Size	0x00	Size		

The single Size field is split by NULL (0x00) bytes in order to make it 'sync-safe'. That is, no possible size value will result in a false MP3 frame sync (11 bits set in a row).

1.8.4.4.2. an ID3v2.4 Frame

Frame ID									
0x00	Size	0x00	Size	0x00	Size	0x00	Size	0x00	Size
0x00	Tag Alter	File Alter	Read Only	0x00					
0x00	Grouping	0x00	Compression	Encryption	Unsync	Data Length			
Frame Data									

Frame ID's that begin with the letter 'T' (0x54) are text frames. These have an additional text encoding byte before the actual text data. All text strings are terminated by a null character (0x00 or 0x0000, depending on the encoding).

Frame ID 'TXXX' (0x54XXXXXX)									
0x00	Size	0x00	Size	0x00	Size	0x00	Size	0x00	Size
0x00	Tag Alter	File Alter	Read Only	0x00					
0x00	Grouping	0x00	Compression	Encryption	Unsync	Data Length			
Encoding			Text						

Encoding Byte	Text Encoding
0x00	ISO-8859-1
0x01	UTF-16
0x02	UTF-16BE
0x03	UTF-8

1.8.4.4.3. ID3v2.4 Frame IDs

ID	Description	ID	Description
AENC	Audio encryption	APIC	Attached picture
ASPI	Audio seek point index	COMM	Comments
COMR	Commercial frame	ENCR	Encryption method registration
EQU2	Equalisation (2)	ETCO	Event timing codes
GEOB	General encapsulated object	GRID	Group identification registration
LINK	Linked information	MCDI	Music CD identifier
MLLT	MPEG location lookup table	OWNE	Ownership frame
PRIV	Private frame	PCNT	Play counter
POPM	Popularimeter	POSS	Position synchronisation frame
RBUF	Recommended buffer size	RVA2	Relative volume adjustment (2)
RVRB	Reverb	SEEK	Seek frame
SIGN	Signature frame	SYLT	Synchronised lyric/text
SYTC	Synchronised tempo codes	TALB	Album/Movie/Show title
TBPM	BPM (beats per minute)	TCOM	Composer
TCON	Content type	TCOP	Copyright message
TDEN	Encoding time	TDLY	Playlist delay
TDOR	Original release time	TDRC	Recording time
TDRL	Release time	TDTG	Tagging time
TENC	Encoded by	TEXT	Lyricist/Text writer
TFLT	File type	TIPL	Involved people list
TIT1	Content group description	TIT2	Title/songname/content description
TIT3	Subtitle/Description refinement	TKEY	Initial key
TLAN	Language(s)	TLEN	Length
TMCL	Musician credits list	TMED	Media type
TMOO	Mood	TOAL	Original album/movie/show title
TOFN	Original filename	TOLY	Original lyricist(s)/text writer(s)
TOPE	Original artist(s)/performer(s)	TOWN	File owner/licensee
TPE1	Lead performer(s)/Soloist(s)	TPE2	Band/orchestra/accompaniment
TPE3	Conductor/performer refinement	TPE4	Interpreted, remixed, or otherwise modified by
TPOS	Part of a set	TPRO	Produced notice
TPUB	Publisher	TRCK	Track number/Position in set
TRSN	Internet radio station name	TRSO	Internet radio station owner
TSOA	Album sort order	TSOP	Performer sort order
TSOT	Title sort order	TSRC	ISRC (international standard recording code)
TSSE	Software/Hardware and settings used for encoding	TSST	Set subtitle
TXXX	User defined text information frame	UFID	Unique file identifier
USER	Terms of use	USLT	Unsynchronised lyric/text transcription
WCOM	Commercial information	WCOP	Copyright/Legal information
WOAF	Official audio file webpage	WOAR	Official artist/performer webpage
WOAS	Official audio source webpage	WORS	Official Internet radio station homepage
WPAY	Payment	WPUB	Publishers official webpage
WXXX	User defined URL link frame		

1.8.4.4.4. the APIC Frame

'APIC' frames are attached pictures.

Frame ID 'APIC' (0x41504943)														
0	0x00		Size		0x00		Size		0x00		Size		31	
	32	33	39	40	41	47	48	49	55	56	57	63		
	0x00		Tag Alter	File Alter		Read Only	0x00						71	
	64	65		66		67	68							
	0x00		Grouping		0x00		Compression	Encryption	Unsync	Data Length				
	72	73		74		75		76	77	78	79			
	Text Encoding			MIME Type							1			
80											2			
	Picture Type			Description							1			
0											2			
	Picture Data												1	

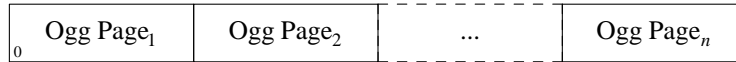
Text Encoding is the encoding of the Description field. Its value is either ISO-8859-1, UTF-16 or UTF-8 - the same as in text frames. MIME Type is a NULL-terminated, ASCII C-string which contains the image's MIME type, such as 'image/jpeg' or 'image/png'. Description is a NULL-terminated C-string which contains a text description of the image.

Picture Types			
value	type	value	type
0	Other	1	32x32 pixels 'file icon' (PNG only)
2	Other file icon	3	Cover (front)
4	Cover (back)	5	Leaflet page
6	Media (e.g. label side of CD)	7	Lead artist / Lead performer / Soloist
8	Artist / Performer	9	Conductor
10	Band / Orchestra	11	Composer
12	Lyricist / Text writer	13	Recording location
14	During recording	15	During performance
16	Movie / Video screen capture	17	A bright coloured fish
18	Illustration	19	Band / Artist logotype
20	Publisher / Studio logotype		

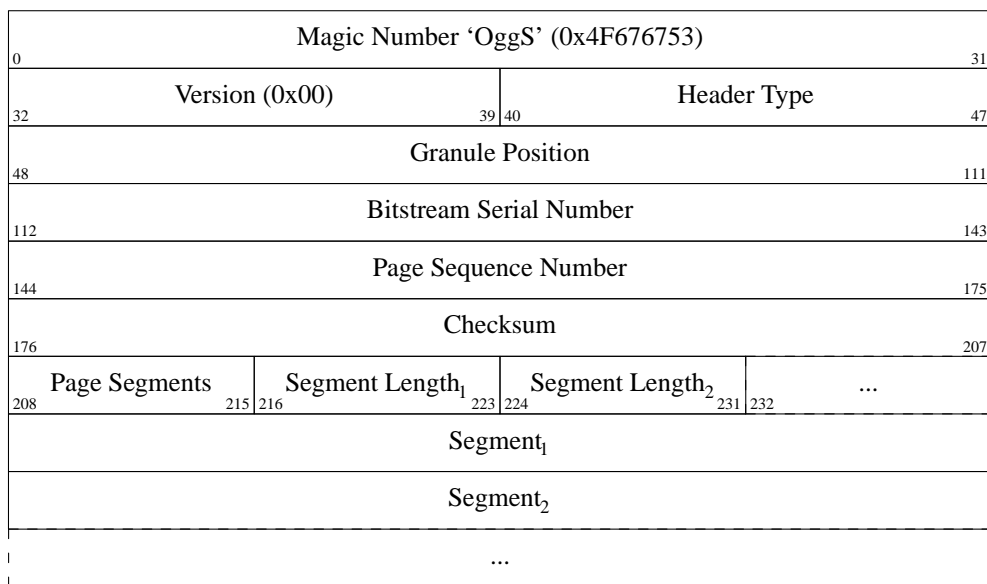
1.9. Ogg Vorbis

Ogg Vorbis is Vorbis audio in an Ogg container. Ogg containers are a series of Ogg pages, each containing one or more segments of data. All of the fields within Ogg Vorbis are little-endian.

1.9.1. the Ogg file stream



1.9.2. an Ogg page



Header Type	
bits	page type
001	Continuation
010	Beginning of Stream
100	End of Stream

Granule position is a time marker. In the case of Ogg Vorbis, it is the sample count.

Bitstream Serial Number is an identifier for the given bitstream which is unique within the Ogg file. For instance, an Ogg file might contain both video and audio pages, interleaved. The Ogg pages for the audio will have a different serial number from those of the video so that the decoder knows where to send the data of each.

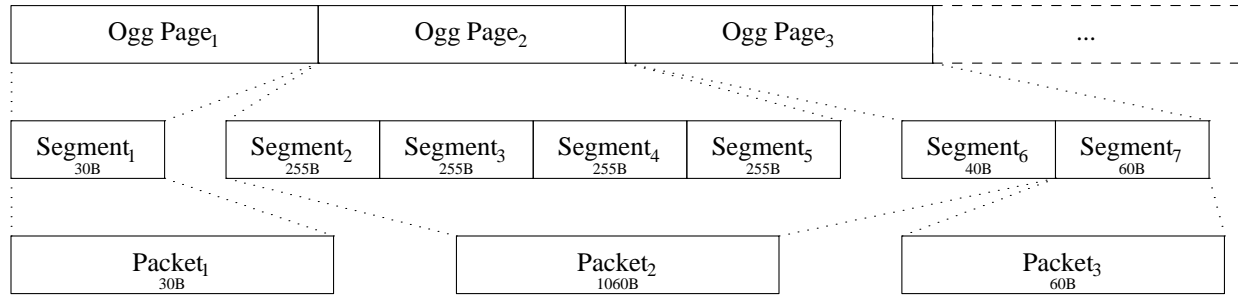
Page sequence number is an integer counter which starts from 0 and increments 1 for each Ogg page. Multiple bitstreams will have separate

sequence numbers.

Checksum is a 32-bit checksum of the entire Ogg page.

The Page Segments value indicates how many segments are in this Ogg page. Each segment will have an 8-bit length. If that length is 255, it indicates the next segment is part of the current one and should be concatenated with it when creating packets from the segments. In this way, packets larger than 255 bytes can be stored in an Ogg page. If the final segment in the Ogg page has a length of 255 bytes, the packet it is a part of continues into the next Ogg page.

1.9.3. Ogg packets



This is an example Ogg stream to illustrate a few key points about the format. Note that Ogg pages may have one or more segments, and packets are composed of one or more segments, yet the boundaries between packets are segments that are less than 255 bytes long. Which segment belongs to which Ogg page is not important for building packets.

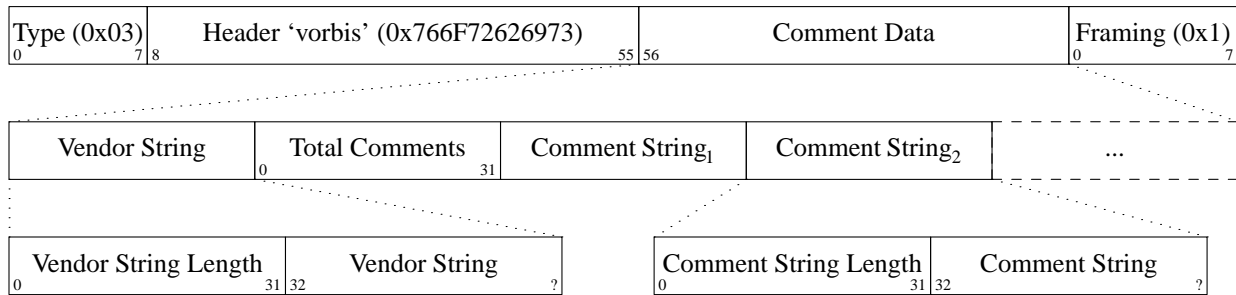
1.9.4. the Identification packet

The first packet within a Vorbis stream is the Identification packet. This contains the sample rate and number of channels. Vorbis does not have a bits-per-sample field, as samples are stored internally as floating point values and are converted into a certain number of bits in the decoding process. To find the total samples, use the Granule Position value in the stream's final Ogg page.

Type (0x01)		Header 'vorbis' (0x766F72626973)	
0	7	8	55
Vorbis version (0x00000000)		Channels	
56		87	88
Sample Rate			
96		127	
Maximum Bitrate			
128		159	
Nominal Bitrate			
160		191	
Minimum Bitrate			
192		223	
Blocksize ₀	Blocksize ₁	Framing flag (0x01)	
224	227	228	231
		232	239

1.9.5. the Comment packet

The second packet within a Vorbis stream is the Comment packet.



The length fields are all little-endian. The Vendor String and Comment Strings are all UTF-8 encoded. Keys are not case-sensitive and may occur multiple times, indicating multiple values for the same field. For instance, a track with multiple artists may have more than one ARTIST.

Comment Strings			
key	value	key	value
ALBUM	album name	ARTIST	artist name, band name, composer, author, etc.
CONTACT	contact information	COPYRIGHT	copyright attribution
DATE	recording date	DESCRIPTION	a short description
GENRE	a short music genre label	ISRC	ISRC number for the track
LICENSE	license information	LOCATION	recording location
ORGANIZATION	record label	PERFORMER	performer name, orchestra, actor, etc.
TITLE	track name	TRACKNUMBER	the track number
VERSION	track version		

1.10. Ogg Speex

Ogg Speex is Speex audio in an Ogg container. Speex is a lossy audio codec optimized for speech. All of the fields within Ogg Speex are little-endian.

How Ogg containers break up data packets into segments and pages has already been explained in the Ogg Vorbis section. Therefore, I shall move directly to the Ogg Speex packets themselves.

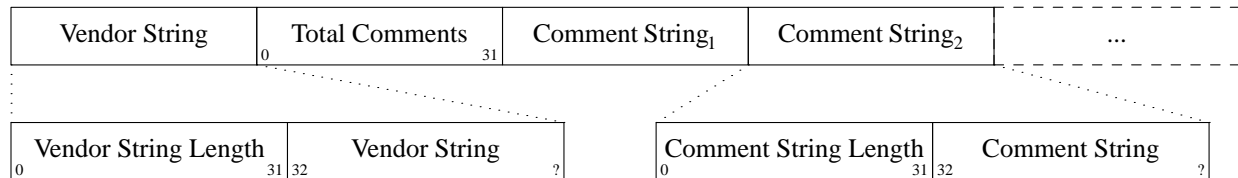
1.10.1. the Header packet

The first packet within a Speex stream is the Header packet. It contains the number of channels and sampling rate. Like Vorbis, the number of bits per sample is generated during decoding and the total number of samples is pulled from the 'Granule Position' field in the Ogg stream.

Speex String 'Speex' (0x5370656578202020)					
0					63
Speex Version			223	Speex Version ID	
64			224	255	
Header Size		287	Sampling Rate		Mode
256	288	319	320	351	
Mode Bitstream Version		Number of Channels		Bitrate	
352	383	384	415	447	
Frame Size		VBR		Frames Per Packet	
448	479	480	511	512	
Extra Headers		Reserved ₁		Reserved ₂	
544	575	576	607	608	
				639	

1.10.2. the Comment packet

The second packet within a Speex stream is the Comment packet.



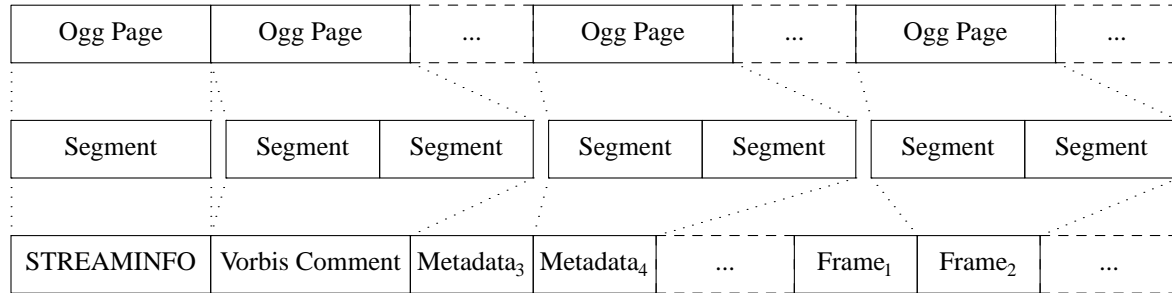
The length fields are all little-endian. The Vendor String and Comment Strings are all UTF-8 encoded. Keys are not case-sensitive and may occur multiple times, indicating multiple values for the same field. For instance, a track with multiple artists may have more than one ARTIST.

Comment Strings			
key	value	key	value
ALBUM	album name	ARTIST	artist name, band name, composer, author, etc.
CONTACT	contact information	COPYRIGHT	copyright attribution
DATE	recording date	DESCRIPTION	a short description
GENRE	a short music genre label	ISRC	ISRC number for the track
LICENSE	license information	LOCATION	recording location
ORGANIZATION	record label	PERFORMER	performer name, orchestra, actor, etc.
TITLE	track name	TRACKNUMBER	the track number
VERSION	track version		

1.11. Ogg FLAC

Ogg FLAC is a FLAC audio stream in an Ogg container. Although it contains all the features of native FLAC, Ogg FLAC is rarely used as a standalone audio format. It is more useful when interleaved with an Ogg video stream so that a movie can be presented with lossless audio. However, this section will only deal with Ogg FLAC.

1.11.1. the Ogg FLAC file stream



1.11.2. the STREAMINFO metadata packet

0	Packet Byte (0x7F)	7	8	Signature 'FLAC' (0x464C4143)			39			
40	Major Version (0x1)			47	48	Minor Version (0x0)		55		
56	Header Packets	71	72	FLAC Signature 'fLaC' (0x664C6143)			103			
	Last Block (0)	104	105	Block Type (0x0)	111	112	Block Length	135		
136	Minimum Block Size (in samples)			151	152	Maximum Block Size (in samples)		167		
168	Minimum Frame Size (in bytes)			191	192	Maximum Frame Size (in bytes)		215		
216	Sample Rate			235	236	Channels	238	239	Bits Per Sample	243
244	Total Samples						279			
280	MD5 Sum of PCM Data						407			

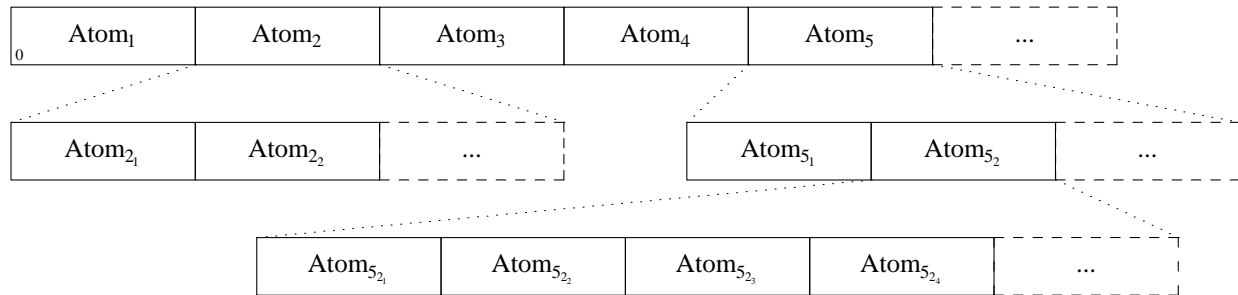
1.11.3. the Metadata packets

Subsequent FLAC metadata blocks are stored 1 per packet. Each contains the 32-bit FLAC metadata block header in addition to the metadata itself. The VORBIS_COMMENT metadata block is required to immediately follow the STREAMINFO block, but all others may appear in any order.

1.12. M4A

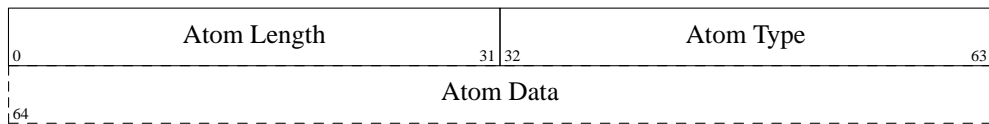
M4A is typically AAC audio in a QuickTime container stream, though it may also contain other formats such as MPEG-1 audio.

1.12.1. the QuickTime file stream

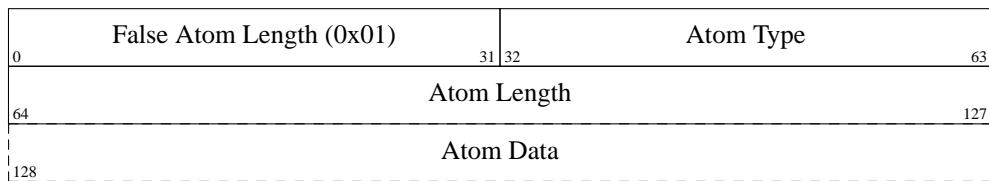


Unlike other chunked formats such as RIFF WAVE, QuickTime's atom chunks may be containers for other atoms. All of its fields are big-endian.

1.12.2. a QuickTime atom



Atom Type is an ASCII string. Atom Length is the length of the entire atom, including the header. If Atom Length is 0, the atom continues until the end of the file. If Atom Length is 1, the atom has an extended size. This means there is a 64-bit length field immediately after the header which is the atom's actual size.



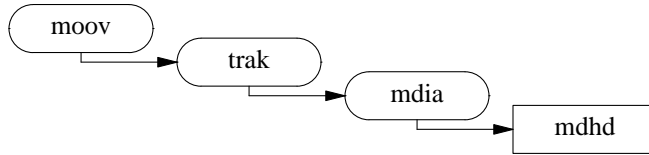
1.12.3. Container atoms

There appears to be no flag or field to tell a QuickTime parser which of its atoms are containers and which ones are not. Instead, one must check against a table of known container atom types. If an atom is known to be a container, one can treat its Atom Data as a QuickTime stream and parse it in a recursive fashion.

Known Container Atoms							
dinf	edts	imag	imap	mdia	mdra	minf	moov
rmra	stbl	trak	tref	udta	vnrp		

1.12.4. the mdhd atom

The mdhd atom contains track information such as samples-per-second, track length and creation/modification times.



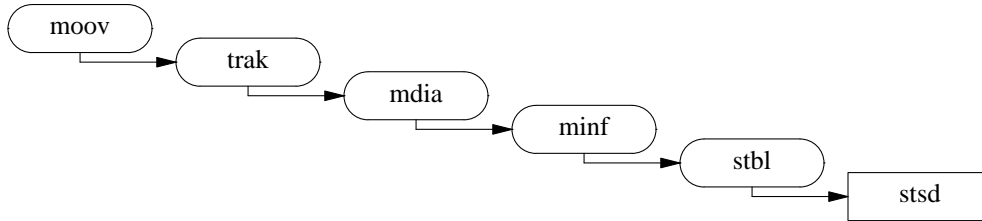
0		mdhd Length		31	32	'mdhd' (0x6D646864)		63	
64	Version (0x00)		71	72	Flags (0x000000)				95
96		Creation Date						127	
128		Modification Date						159	
160		Sample Rate						191	
192		Track Length						223	
224		225	Language		239	240	Quality		255

If version is 1, the date and length fields are 64-bit.

0		mdhd Length		31	32	'mdhd' (0x6D646864)		63	
64	Version (0x01)		71	72	Flags (0x000000)				95
96		Creation Date						159	
160		Modification Date						223	
224		Sample Rate						255	
256		Track Length						319	
320		321	Language		335	336	Quality		351

1.12.5. the mp4a atom

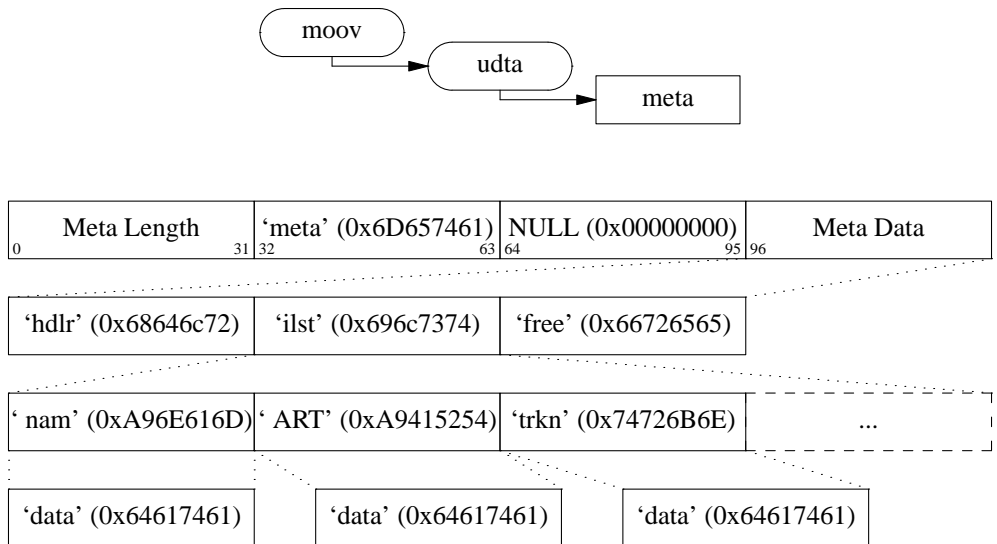
The mp4a atom contains information such as the number of channels and bits-per-sample. It can be found in the stsd atom, offset 8 bytes.



stsd Length		'stsd' (0x73747364)		Flags		mp4a Atom	
0	31	32	63	64	127	128	
mp4a Length		'mp4a' (0x6D703461)					
0	31	32	63				
Reserved (0x00000000)							
64							95
Reserved (0x0000)		Reference Index					
96	111	112					
Version (0x0000)		Revision Level					
128	143	144					
Audio Encoding Vendor							
160							191
Channels		Bits Per Sample					
192	207	208					
Additional Data							
224							

1.12.6. the meta atom

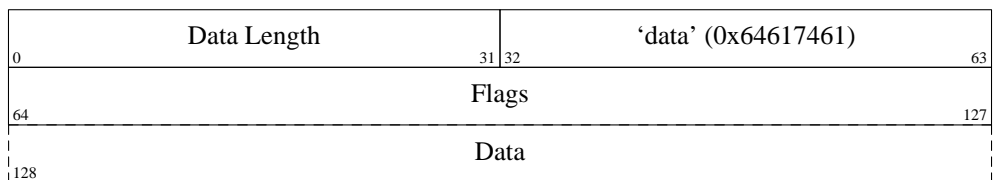
The meta atom contains track metadata such as the track name, artist, album name, etc. meta is like other QuickTime containers, except that it contains an additional 4 bytes prior to its QuickTime Atom Data payload.



The atoms within *ilst* are all containers, each with a *data* atom of its own. Notice that *ilst*'s sub-atoms often have non-ASCII Atom Type values - many are preceded by the 0xA9 byte. Such atoms are human-readable.

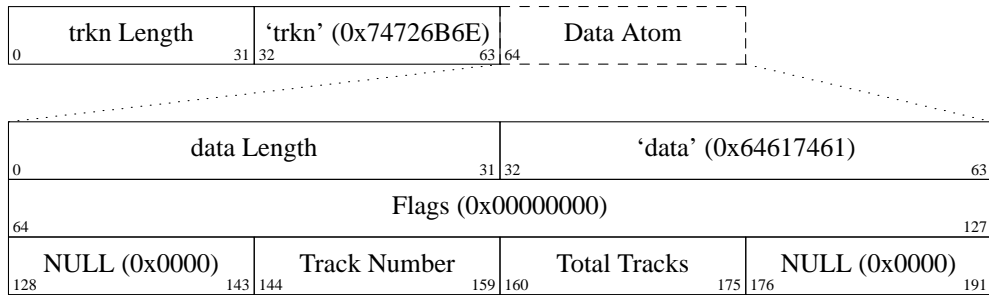
Known meta Sub-Atoms					
Atom Type	Hex	Description	Atom Type	Hex	Description
aaid	0x61616964	Artist Name	alb	0xA9616C62	Album Name
akid	0x616B6964	Alternative ID ?	apid	0x61706964	Apple ID
ART	0xA9415254	Performer Name	cmt	0xA9636D74	Comment
com	0xA9636F6D	Composer	covr	0x636F7672	Cover Image
cpil	0x6370696C	Compilation	cprt	0x63707274	Copyright ?
day	0xA9646179	Date	disk	0x6469736B	Disc X of Y
geid	0x67656964	iTMS ID ?	gnre	0x676E7265	Genre
grp	0xA9677270	Group ?	nam	0xA96E616D	Track Name
plid	0x706C6964	Purchase ID ?	rtng	0x72746E67	Rating
stik	0x7374696B	Movie Type	tmpo	0x746D706F	Tempo
too	0xA9746F6F	Encoder	trkn	0x74726B6E	Track Number
wrt	0xA9777274	Composer	----	0x2D2D2D2D	iTunes-specific

The *data* sub-atoms contain the actual metadata, naturally, but that data is preceded by 8 bytes of additional information.



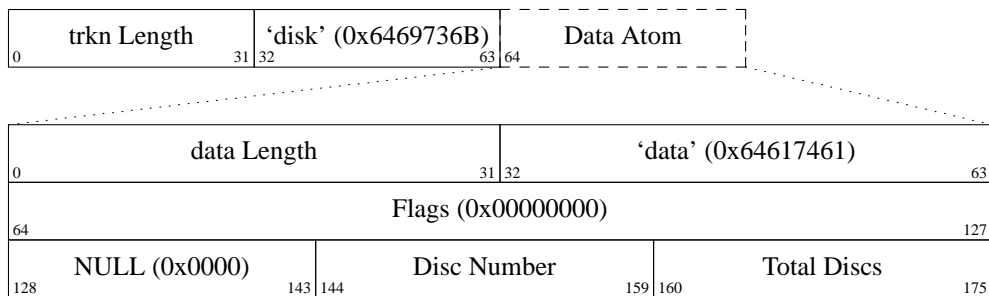
1.12.6.1. the trkn sub-atom

trkn is a binary sub-atom of meta which contains the track number.



1.12.6.2. the disk sub-atom

disk is a binary sub-atom of meta which contains the disc number. For example, if the track belongs to the first disc in a set of two discs, the sub-atom will contain that information.



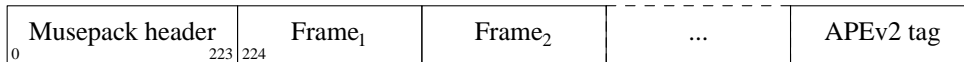
1.13. Musepack

Musepack is a lossy audio format based on MP2 and designed for transparency. It comes in two varieties: SV7 and SV8 where ‘SV’ stands for Stream Version. These container versions differ so heavily that they must be considered separately from one another.

1.13.1. Musepack SV7

This is the earliest version of Musepack with wide support. All of its fields are little-endian.

1.13.1.1. the Musepack SV7 file stream



1.13.1.2. the Musepack SV7 header

Signature ('MP+' 0x4D502B)		Version (0x07)	
0	23	24	31
Frame Count			
32			63
Max Level			
64			79
Profile		Link	Sample Rate
80	83	84	85 86 87
Intensity Stereo	Midside Stereo	Max Band	
88	89	90	95
Title Gain		Title Peak	
96	111	112	127
Album Gain		Album Peak	
128	143	144	159
Unused (0x00)			
160			175
Last Frame Samples (low)		True Gapless	Unused (0x00)
176	179	180	181 183
Fast Seeking	Last Frame Samples (high)		
184	185	191	
Unknown		Encoder Version	
192	215	216	223

Each frame contains 1152 samples per channel. Therefore:

$$Total\ Samples = ((Frame\ Count - 1) \times 1152) + Last\ Frame\ Samples$$

Musepack files always have exactly 2 channels and always have 16 bits per sample.

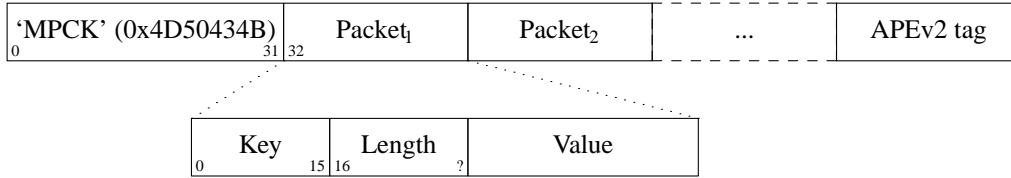
Sampling Rate	
bits	rate
00	44100
01	48000
10	37800
11	32000

Profile			
bits	used profile	bits	used profile
0000	no profile	0001	Unstable/Experimental
0010	unused	0011	unused
0100	unused	0101	below Telephone
0110	below Telephone	0111	Telephone
1000	Thumb	1001	Radio
1010	Standard	1011	Xtreme
1100	Insane	1101	Braindead
1110	above Braindead	1111	above Braindead

1.13.2. Musepack SV8

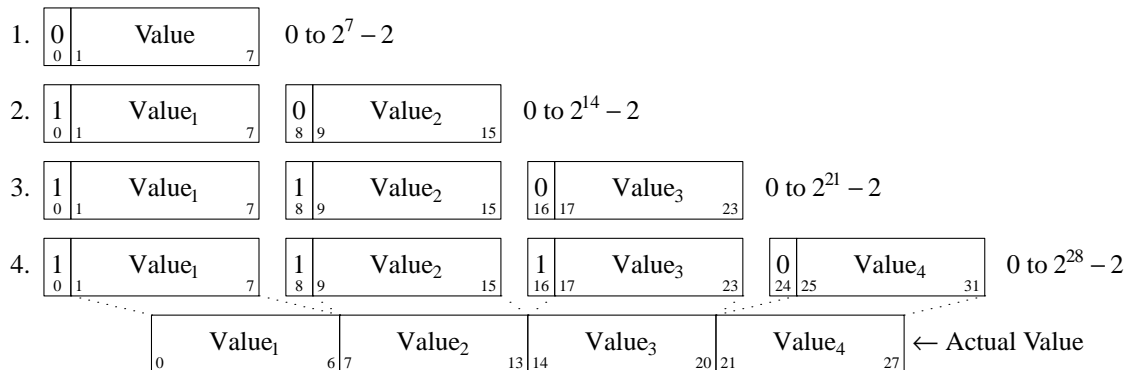
This is the latest version of the Musepack stream. All of its fields are big-endian.

1.13.2.1. the Musepack SV8 file stream



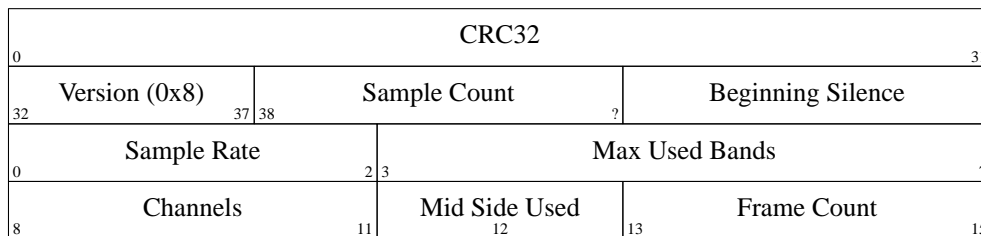
‘Key’ is a two character uppercase ASCII string (i.e. each digit must be between the characters 0x41 and 0x5A, inclusive). ‘Length’ is a variable length field indicating the size of the entire packet, including the header. This is a Nut-encoded field whose total size depends on whether the eighth bit of each byte is 0 or 1. The remaining seven bits of each byte combine to form the field’s value, which is big-endian.

1.13.2.2. Nut-encoded values



1.13.2.3. the SH packet

This is the Stream Header, which must be found before the first audio packet in the Musepack file.



‘CRC32’ is a checksum of everything in the header, not including the checksum itself. ‘Sample Count’ is the total number of samples, as a Nut-encoded value. ‘Beginning Silence’ is the number of silence samples at the start of the stream, also as a Nut-encoded value. ‘Channels’ is the total number of channels in the stream, minus 1. ‘Mid Side Used’ indicates the channels are stored using mid-side stereo. ‘Frame Count’

is used to calculate the total number of frames per audio packet ($Number\ of\ Frames = 4^{Frame\ Count}$).

Sampling Rate			
bits	rate	bits	rate
000	44100	001	48000
010	37800	011	32000

1.13.2.4. the SE packet

This is an empty packet that denotes the end of the Musepack stream. A decoder should ignore everything after this packet, which allows for metadata tags such as APEv2 to be placed at the end of the file.

1.13.2.5. the RG packet

This is ReplayGain information about the audio file.

Version (0x1)		0	7
Title Gain		8	23
Title Peak		24	39
Album Gain		40	55
Album Peak		56	71

1.13.2.6. the EI packet

This is information about the Musepack encoder.

Profile		0	6
PNS		7	7
Major Version		8	15
Minor Version		16	23
Build		24	31

2. FreeDB

Because compact discs do not usually contain metadata about track names, album names and so forth, that information must be retrieved from an external source. FreeDB is a service which allows users to submit CD metadata and to retrieve the meta-data submitted by others. Both actions require a category and a 32-bit disc ID number, which combine to form a unique identifier for a particular CD.

The 32-bit disc ID is calculated from the total number of tracks, the track offsets (in CD sectors) and the total length of the CD (in seconds).

In the case of CD submission, the genre is decided by the submitter. In the case of CD retrieval, the user must choose from a list of possible choices when there is a collision between 32-bit disc ID numbers.

The actual metadata is stored as XMCD files.

2.1. Native Protocol

FreeDB's native protocol runs as a service on TCP port 8880. After connecting, the client and server exchange a handshake using the 'hello' command. The server will not do anything without this handshake.

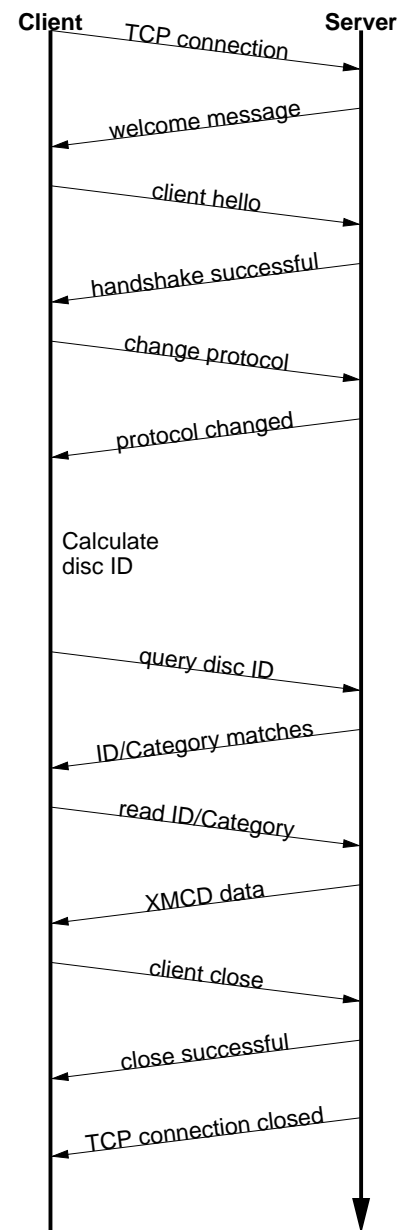
Next the client changes to protocol level 6 with the 'proto' command. This is necessary because only the highest protocol supports UTF-8 text encoding. Without this, any characters not in the latin-1 set will not be sent properly.

Once that is accomplished, the client should calculate the 32-bit disc ID from the track information.

One then sends the 32-bit disc ID and additional disc information to the server with the 'query' command to retrieve a list of matching disc IDs, genres and titles. If there are multiple matches, the user must be prompted to choose one of the matches.

When our match is known, the client uses the 'read' command to retrieve the actual XMCD data.

Finally, the 'close' command is used to sever the connection and complete the transaction.



2.1.1. The Disc ID

FreeDB uses a big-endian 32-bit disc ID to differentiate one disc from another.

Offset Seconds Digit Sum <small>0 7 8</small>	Total Length in Seconds <small>23 24</small>	Track Count <small>31</small>
--	---	----------------------------------

‘Track Count’ is self-explanatory. ‘Total Length’ is the total length of all the tracks, not counting the initial 2 second lead-in. ‘Offset Seconds Digit Sum’ is the sum of the digits of all the disc’s track offsets, in seconds, and truncated to 8 bits. Remember to count the initial 2 second/150 frame lead-in when calculating offsets.

Example Audio Disc						
Track Number	Length			Offset		
	in Minutes:Seconds	in Seconds	in Frames	in Minutes:Seconds	in Seconds	in Frames
1	3:37	217	16340	0:02	2	150
2	3:23	203	15294	3:39	219	16490
3	3:37	217	16340	7:03	423	31784
4	3:20	200	15045	10:41	641	48124

In this example, ‘Track Count’ is 4. ‘Total Length’ is $\frac{16340 + 15294 + 16340 + 15045}{75} = 840$

There are 75 frames per second, but one must remember to count fractions of seconds when calculating the total disc length. The ‘Offset Seconds Digit Sum’ is calculated by looking at the ‘Offset in Seconds’ column. Those values are 2, 219, 423 and 641. One must take all of those digits and add them, which works out to $2 + 2 + 1 + 9 + 4 + 2 + 3 + 6 + 4 + 1 = 34$

This means our three values are 34, 840 and 4. In hexadecimal, they are 0x22, 0x0348 and 0x04. Combining them into a single value yields 0x22034804. Thus, our FreeDB disc ID is ‘22034804’.

2.1.2. Initial Greeting

From server:

```
<code> <host> CDDBP server <version> ready at <datetime>
```

	200	OK, reading/writing allowed
	201	OK, read-only
<code>	432	No connections allowed: permission denied
	433	No connections allowed: X users allowed, Y currently active
	434	No connections allowed: system load too high

<hostname> the server's host name
<version> the server's version
<datetime> the current date and time

2.1.3. Client-Server Handshake

To server:

```
cddb hello <username> <hostname> <clientname> <version>
```

<username>	login name of user
<hostname>	host name of client
<clientname>	name of client program
<version>	version of client program

From server:

```
<code> hello and welcome <username>@<hostname> running <clientname> <version>
```

	200	handshake successful
<code>	402	already shook hands
	431	handshake unsuccessful, closing connection

<username> login name of user
<hostname> host name of client
<clientname> name of client program
<version> version of client program

2.1.4. Set Protocol Level

To server:

proto [level]

<level> | protocol level as integer (optional)

From server:

<code> CDDb protocol level: <current>, supported <supported>

OR

<code> OK, protocol version now: <current>

	200	displaying current protocol level
<code>	201	protocol level set
	501	illegal protocol level
	502	protocol level already at <current>
<current>		the current protocol level of this connection
<supported>		the maximum supported protocol level

2.1.5. Calculate Disc ID

To server:

discid <track count> <offset 1> ... <offset n> <total seconds>

<track count>	number of tracks in CD
<offset>	frame offset of each track
<total seconds>	total length of CD in seconds

From server:

<code> Disc ID is <disc id>

<code>	200	disc ID calculated
	500	syntax error
<disc id>		32-bit disc ID

2.1.6. Query Database

To server:

```
cddb query <disc id> <track count> <offset 1> ... <offset n> <total seconds>
```

<disc id>	32-bit disc ID
<track count>	number of tracks in CD
<offset>	frame offset of each track
<total seconds>	total length of CD in seconds

From server:

```
<code> <category> <disc id> <disc title>
```

OR

```
<code> close matches found  
<category> <disc id> <disc title>  
<category> <disc id> <disc title>  
<...>  
.
```

OR

```
<code> exact matches found  
<category> <disc id> <disc title>  
<category> <disc id> <disc title>  
<...>  
.
```

<code>	200	Found exact match
	211	Found inexact matches, list follows
	202	No match found
	210	Found exact matches, list follows
	403	Database entry corrupt
	409	No handshake
<category>		category string
<disc id>		32-bit disc ID
<disc title>		disc title string

2.1.7. Read XMCD Data

To server:

```
cddb read <category> <disc id>
```

<category>		category string
<disc id>		32-bit disc ID

From server:

```
<code> <category> <disc id>  
# XMCD file data  
# ...  
.
```

		210	XMCD data follows
		401	XMCD data not found
<code>		402	server error
		403	database entry corrupt
		409	no handshake
<category>			category string
<disc id>			32-bit disc ID

2.1.8. Close Connection

To server:

```
quit
```

From server:

```
<code> <hostname> <message>  
  
<code> <message> | 230 Closing connection. Goodbye.  
                 | 530 error, closing connection.  
<hostname> | server's host name
```

2.1.9. List Genres

To server:

cddb lscat

From server:

```
<code> Okay category list follows (until terminating marker)
<category>
<category>
<...>
.
<code>          210   okay
<category> | category string
```

2.1.10. List Mirrors

To server:

sites

From server:

```
<code> OK, site information follows (until terminating '.')
<site> <protocol> <port> <address> <latitude> <longitude> <description>
<...>
.
<code>          210   site information follows
          401   no site information available
<site>          internet address of site
<protocol>     the protocol level supported by the site
<port>        internet port of site
<address>     additional addressing required, or '-' if none necessary
<latitude>    latitude of site
<longitude>   longitude of site
<description> short text description of site
```

2.2. Web Protocol

FreeDB's web protocol runs as a service on HTTP port 80. A web client POSTs data to a location, typically: `~cddb/cddb.cgi` and retrieves results. This method is similar to the native protocol and the returned data is identical. However, since HTTP POST requests are stateless, there is no separate 'hello', 'proto' and 'quit' commands; these are issued along with the primary server command or are implied.

POST Arguments	
key	value
hello	<username> <hostname> <clientname> <version>
proto	<protocol level>
cmd	<server command>

For example, to execute the 'read' command on disc ID 'AABBCCDD' in the 'soundtrack' category, one can post the following string:

```
cmd=read+soundtrack+aabbccdd&hello=username+hostname+audiotools+1.0&proto=6
```

2.3. XMCD

XMCD files are text files encoded either in UTF-8, ISO-8859-1 or US-ASCII that begin with the string '# XMCD'. Lines are delimited by either the 0x0A character or the 0x0D 0x0A character pair. All lines must be less than 256 characters long, including delimiters. Blank lines are prohibited. Lines that begin with the '#' character are comments. Curiously, the comments themselves are expected by FreeDB to contain important information such as track offsets and disc length. Fortunately, FreeDB clients can safely ignore such information unless submitting a new disc entry.

What we are interested in are the KEY=value pairs in the rest of the file.

key	value
DISCID	a comma-separated list of 32-bit disc IDs
DTITLE	an artist name and album name, separated by '/'
DYEAR	a 4 digit disc release year
DGENRE	the disc's FreeDB category string
TITLEX	the track title, or the track artist name and track title, separated by '/' X is an integer starting from 0
EXTD	extended data about the disc
EXTTX	extended data about the track X is an integer starting from 0
PLAYORDER	a comma-separated list of track numbers

Multiple identical keys should have their values concatenated (minus the newline delimiter), which allows a single key to have a value longer than the 256 characters line length.

2.3.1. BNF

```

<XMCD file> ::= <header> <disc information> <file information> <blank> <data>
<header> ::= "# xmcd" <EOL> "#" <EOL>
<disc information> ::= "# Track frame offsets:" <EOL> <track offset>
<track offset> ::= "# " <offset> <EOL> | "# " <offset> <EOL> <track offset>
<offset> ::= [0-9]+
<file information> ::= <disc length> <blank> <revision> <processed by> <submitted via>
<disc length> ::= "# Disc length: " <seconds> " seconds" <EOL>
<seconds> ::= [0-9]+
<revision> ::= "# Revision: " <revision number> <EOL>
<revision number> ::= [0-9]+
<processed by> ::= "# Processed by: " <server name> <server version> <EOL>
<submitted via> ::= "# Submitted via: " <client name> <client version> <EOL>
<blank> ::= "#" <EOL> | "#" <EOL> <blank>
<data> ::= <key> "=" <value> <EOL> | <key> "=" <value> <EOL> <data>
<key> ::= [A-Z0-9]+
<value> ::= [UTF-8 value]+

```

3. ReplayGain

The ReplayGain standard is designed to address the problem of highly variable music loudness. For example, let's assume we have two audio tracks, A and B, and that track B is much louder than A. If played in sequence, the listener will have to scramble for the volume control once B starts in order to have a comfortable experience. ReplayGain solves this problem by calculating the overall loudness of a track as a delta (some positive or negative number of decibels, in relation to a reference loudness value). This delta is then applied during playback, which has the same effect as turning the volume up or down so that the user doesn't have to.

ReplayGain requires four floating-point values which are typically stored as metadata in each audio track: 'track gain', a positive or negative number of decibels representing the loudness delta of this particular track, 'track peak', the highest sample value of this particular track from a range of 0.0 to 1.0, 'album gain', a positive or negative number of decibels representing the loudness delta of the track's entire album and 'album peak', the highest sample value of the track's entire album from a range of 0.0 to 1.0.

3.1. Applying ReplayGain

The user will be expected to choose whether to apply 'album gain' or 'track gain' during playback. When listening to audio on an album-by-album basis, album gain keeps quiet tracks quiet and loud tracks loud within the context of that album. When listening to audio on a track-by-track basis, perhaps as a randomly shuffled set, track gain keeps them all to roughly the same loudness. So from an implementation perspective, a program only needs to apply the given gain and peak value to the stream being played back. Applying the gain value to each input PCM sample is quite simple:

$$Output_i = Input_i \times 10^{gain/20}$$

For example, if the gain is -2.19, each sample should be multiplied by $10^{-2.19/20}$ or about 0.777.

If the gain is negative, the PCM stream gets quieter than it was originally. If the gain is positive, the PCM stream gets louder. However, increasing the value of each sample may cause a problem if doing so sends any samples beyond the maximum value the stream can hold. For example, if the gain indicates we should be multiplying each sample by 1.28 and we encounter a 16-bit input sample with a value of 32000, the resulting output sample of 34560 is outside of the stream's 16-bit signed range (-32678 to 32767). That will result in 'clipping' the audio peaks, which doesn't sound good.

Preventing this is what ReplayGain's peak value is for; it's the highest PCM value in the stream and no multiplier should push that value beyond 1.0. Thus, if the peak value of a stream is 0.9765625, no ReplayGain value should generate a multiplier higher than 1.024 ($0.9765625 \times 1.024 = 1.0$).

3.2. Calculating ReplayGain

As explained earlier, ReplayGain requires a peak and gain value which are split into ‘track’ and ‘album’ varieties for a total of four. The ‘track’ values require the PCM data for the particular track we’re generating data for. The ‘album’ values require the PCM data for the entire album, concatenated together into a single stream.

Determining the peak value is very straightforward. We simply convert each sample’s value to the range of 0.0 to 1.0 and find the highest value which occurs in the stream. For signed samples, the conversion process is also simple:

$$Output_i = \frac{|Input_i|}{2^{bits\ per\ sample - 1}}$$

Determining the gain value is a more complicated process. It involves running the input stream through an equal loudness filter, breaking that stream into 50 millisecond long blocks, and then determining a final value based on the value of those blocks.

3.2.1. the Equal Loudness Filter

Because people don’t perceive all frequencies of sounds as having equal loudness, ReplayGain runs audio through a filter which emphasizes ones we hear as loud and deemphasizes ones we hear as quiet. This equal loudness filtering is actually comprised of two separate filters: Yule and Butter (these are Infinite Impulse Response filters named after their creators). Each works on a similar principle.

The basic premise is that each output sample is derived from multiplying ‘order’ number of previous input samples by certain values (which depend on the filter) *and* ‘order’ number of previous output samples by a different set of values (also depending on the filter) and then combining the results. This filter is applied independently to each channel. In purely mathematical terms, it looks like this:

$$Output_i = \left(\sum_{j=i-order}^i Input_j \times Input\ Filter_j \right) - \left(\sum_{k=i-order}^{i-1} Output_k \times Output\ Filter_k \right)$$

‘Input Filter’ and ‘Output Filter’ are lists of predefined values. ‘Order’ refers to the size of those lists. When filtering at the start of the stream, treat any samples before the beginning as 0.

Since this explanation is getting vague and theoretical, let’s move on to the filter values themselves and an example of how to apply them. There’s a lot of numbers here, each with a lot of digits. Don’t be too concerned about the results of the math; focus instead on which sample value gets multiplied by which filter value in order to yield the sums we need.

3.2.1.1. the Yule Filter

Yule Input Filter				
Sample to Multiply		Sample Rate		
		48000Hz	44100Hz	32000Hz
$Input_i$	×	0.038575994352000001	0.054186564064300002	0.15457299681924
$Input_{i-1}$	×	-0.021603671841850001	-0.029110078089480001	-0.093310490563149995
$Input_{i-2}$	×	-0.0012339531685100001	-0.0084870937985100006	-0.062478801536530001
$Input_{i-3}$	×	-9.2916779589999993e-05	-0.0085116564546900003	0.021635418887979999
$Input_{i-4}$	×	-0.016552603416190002	-0.0083499090493599996	-0.05588393329856
$Input_{i-5}$	×	0.02161526843274	0.022452932533390001	0.047814766749210001
$Input_{i-6}$	×	-0.02074045215285	-0.025963385129149998	0.0022231259774300001
$Input_{i-7}$	×	0.0059429806512499997	0.016248649629749999	0.031740925400489998
$Input_{i-8}$	×	0.0030642802319099998	-0.0024087905158400001	-0.013905894218979999
$Input_{i-9}$	×	0.00012025322027	0.0067461368224699999	0.00651420667831
$Input_{i-10}$	×	.0028846368391600001	-0.00187763777362	-0.0088136273383899993

Yule Output Filter				
Sample to Multiply		Sample Rate		
		48000Hz	44100Hz	32000Hz
$Output_{i-1}$	×	-3.8466461711806699	-3.4784594855007098	-2.3789883497308399
$Output_{i-2}$	×	7.81501653005538	6.3631777756614802	2.84868151156327
$Output_{i-3}$	×	-11.341703551320419	-8.5475152747187408	-2.6457717022982501
$Output_{i-4}$	×	13.055042193275449	9.4769360780128	2.2369765745171302
$Output_{i-5}$	×	-12.28759895145294	-8.8149868137015499	-1.67148153367602
$Output_{i-6}$	×	9.4829380631978992	6.8540154093699801	1.0059595480854699
$Output_{i-7}$	×	-5.8725786177599897	-4.3947099607955904	-0.45953458054982999
$Output_{i-8}$	×	2.7546586187461299	2.1961168489077401	0.16378164858596
$Output_{i-9}$	×	-0.86984376593551005	-0.75104302451432003	-0.050320777171309998
$Output_{i-10}$	×	0.13919314567432001	0.13149317958807999	0.023478974070199998

3.2.1.2. the Butter Filter

Butter Input Filter				
Sample to Multiply		Sample Rate		
		48000Hz	44100Hz	32000Hz
$Input_i$	×	0.98621192462707996	0.98500175787241995	0.97938932735214002
$Input_{i-1}$	×	-1.9724238492541599	-1.9700035157448399	-1.95877865470428
$Input_{i-2}$	×	0.98621192462707996	0.98500175787241995	0.97938932735214002

Butter Output Filter				
Sample to Multiply		Sample Rate		
		48000Hz	44100Hz	32000Hz
$Output_{i-1}$	×	-1.9722337291952701	-1.96977855582618	-1.9583538097539801
$Output_{i-2}$	×	0.97261396931305999	0.97022847566350001	0.95920349965458995

3.2.1.3. a Filtering Example

When performing ReplayGain calculations, we'll start by converting all our samples to floating-point values between -1.0 and 1.0. This is a simple matter of dividing each sample by $2^{\text{bits-per-sample} - 1}$. So for 16-bit samples, divide each one by 32768.

Next, let's assume we have a 44100Hz stream and our previous input and output samples are as follows:

Sample	Input	Output
89	-0.001007080078125	-0.00045495715387008651
90	-0.0009765625	-0.00045569008938487577
91	-0.001068115234375	-0.00044710087844377787
92	-0.0009765625	-0.00044127330865733358
93	-0.00091552734375	-0.00043189463254365861
94	-0.0009765625	-0.00041441662610518335
95	-0.001007080078125	-0.00040230590245440639
96	-0.00091552734375	-0.0004015602553121536
97	-0.00091552734375	-0.00040046613041640292
98	-0.00091552734375	-0.00039336026519054979
99	-0.0009765625	-0.00039087401794557448

If the value of sample 100 from the input stream is -0.00091552734375 ($-30/2^{15}$), here's how we calculate output sample 100:

Sample	Input Value		Yule Input Filter		Result	
90	-0.0009765625	×	-0.00187763777362	=	1.8336306383007813e-06	
91	-0.001068115234375	×	0.0067461368224699999	=	-7.2056515132583621e-06	
92	-0.0009765625	×	-0.0024087905158400001	=	2.3523344881250001e-06	
93	-0.00091552734375	×	0.016248649629749999	=	-1.4876083035049437e-05	
94	-0.0009765625	×	-0.025963385129149998	=	2.5354868290185545e-05	
95	-0.001007080078125	×	0.022452932533390001	=	-2.2611901049861755e-05	
96	-0.00091552734375	×	-0.0083499090493599996	=	7.6445700525146477e-06	
97	-0.00091552734375	×	-0.0085116564546900003	=	7.7926542248748791e-06	
98	-0.00091552734375	×	-0.0084870937985100006	=	7.770166441506958e-06	
99	-0.0009765625	×	-0.029110078089480001	=	2.8427810634257813e-05	
100	-0.00091552734375	×	0.054186564064300002	=	-4.9609281064727785e-05	
Input Values Sum					=	-1.3126881893131719e-05

Sample	Output Value		Yule Output Filter		Result	
90	-0.00045569008938487577	×	0.13149317958807999	=	-5.9920138759993691e-05	
91	-0.00044710087844377787	×	-0.75104302451432003	=	0.00033579199600942429	
92	-0.00044127330865733358	×	2.1961168489077401	=	-0.00096908774811563594	
93	-0.00043189463254365861	×	-4.3947099607955904	=	0.0018980516436537679	
94	-0.00041441662610518335	×	6.8540154093699801	=	-0.002840417941224044	
95	-0.00040230590245440639	×	-8.8149868137015499	=	0.0035463212252098944	
96	-0.0004015602553121536	×	9.4769360780128	=	-0.0038055608710637796	
97	-0.00040046613041640292	×	-8.5475152747187408	=	0.0034229903667417111	
98	-0.00039336026519054979	×	6.3631777756614802	=	-0.0025030212972888127	
99	-0.00039087401794557448	×	-3.4784594855007098	=	0.0013596394353585582	
Output Values Sum					=	0.00038478667052108985

-1.3126881893131719e-05 (input values sum) - 0.00038478667052108985 (output values sum) = **-0.00039791355241422158** (output sample 100).

We're not quite done yet. Remember, ReplayGain's equal loudness filter requires both a Yule *and* Butter filter, in that order. Here's our set of previous Butter filter input and output values:

Sample	Input	Output
97	-0.00040046613041640292	1.2422165031560971e-05
98	-0.00039336026519054979	1.8657680223143899e-05
99	-0.00039087401794557448	2.0148828330135515e-05

Notice how Butter's input samples are Yule's output samples. Thus, our next input sample to the Butter filter is -0.00039791355241422158. Calculating sample 100 is now a similar process:

Sample	Input Value	Butter Input Filter	Result
98	-0.00039336026519054979	× 0.98500175787241995	= -0.00038746055268985282
99	-0.00039087401794557448	× -1.9700035157448399	= 0.0007700231895660934
100	-0.00039791355241422158	× 0.98500175787241995	= -0.0003919455486092676
Input Values Sum			= -9.38291173302702e-06

Sample	Output Value	Butter Output Filter	Result
98	1.8657680223143899e-05	× 0.97022847566350001	= 1.8102212642317936e-05
99	2.0148828330135515e-05	× -1.96977855582618	= -3.968872996972396e-05
Output Values Sum			= -2.1586517327406024e-05

-9.38291173302702e-06 (input values sum) - -2.1586517327406024e-05 (output values sum) = **1.2203605594379004e-05** (output sample 100).

The output from the Butter filter is the final result of ReplayGain's equal loudness filter.

3.2.2. RMS Energy Blocks

The next step is to take our stream of filtered samples and convert them to a list of blocks, each 1/20th of a second long. For example, a 44100Hz stream is sliced into blocks containing 2205 PCM frames each.

We then figure out the total energy value of each block by taking the Root Mean Square of the block's samples and converting to decibels, hence the name RMS. First, convert our floating-point samples back into integer samples by multiplying each one by $2^{\text{bits-per-sample} - 1}$. Next, run those samples through the following formulas:

$$Block_i = \frac{\left(\frac{\sum_{x=0}^{Block\ Length-1} Left\ Sample_x^2}{Block\ Length} \right) + \left(\frac{\sum_{y=0}^{Block\ Length-1} Right\ Sample_y^2}{Block\ Length} \right)}{2}$$

$$Block\ DB_i = 10 \times \log_{10}(Block_i + 10^{-10})$$

For mono streams, use the same value for both the left and right samples (this will cause the addition and dividing by 2 to cancel each other out). As a partial example involving 2205 PCM frames:

Sample	Left Value	Left Value ²	Right Value	Right Value ²
998	115	13225	-43	1849
999	111	12321	-38	1444
1000	107	11449	-36	1296
...
Left Value ² Sum = 7106715		Right Value ² Sum = 11642400		

$$\frac{(7106715 / 2205) + (11642400 / 2205)}{2} = 4251$$

$$10 \times \log_{10}(4251 + 10^{-10}) = 36.28$$

Thus, the decibel value of this block is 36.28.

3.2.3. Statistical Processing and Calibration

At this point, we've converted our stream of input samples into a list of RMS energy blocks. We now pick the 95th percentile value as the audio stream's representative value. That means we first sort them from lowest to highest, then pick the one at the 95% position. For example, if we have a total of 2400 decibel blocks (from a 2 minute song), the value of block 2280 is our representative.

Finally, we take the difference between a reference value of pink noise and our representative value for the final gain value. The reference pink noise value is typically 64.82 dB. Therefore, if our representative value is 67.01 dB, the resulting gain value is -2.19 dB (64.82 - 67.01 = -2.19).

4. References

- **The Wave File Format**
<http://www.borg.com/~jglatt/tech/wave.htm>
- **Audio Interchange File Format**
<http://www.borg.com/~jglatt/tech/aiff.htm>
- **AU Audio File Format**
<http://www.opengroup.org/public/pubs/external/auformat.html>
- **FLAC Format Specification**
<http://flac.sourceforge.net/format.html>
- **APEv2 Specification**
http://wiki.hydrogenaudio.org/index.php?title=APEv2_specification
- **WavPack 4.0 File / Block Format**
http://www.wavpack.com/file_format.txt
- **MPEG Audio Compression Basics**
<http://www.dv.co.yu/mpgscript/mpeghdr.htm>
- **What is ID3v1**
<http://www.id3.org/ID3v1>
- **The ID3v2 Documents**
http://www.id3.org/Developer_Information
- **The Ogg File Format**
http://en.wikipedia.org/wiki/Ogg#File_format
- **Vorbis I Specification**
http://xiph.org/vorbis/doc/Vorbis_I_spec.html
- **Speex Documentation**
<http://www.speex.org/docs/>
- **Musepack Stream Version 7 Format Specification**
<http://trac.musepack.net/trac/wiki/SV7Specification>
- **Parsing and Writing QuickTime Files in Java**
http://www.onjava.com/pub/a/onjava/2003/02/19/qt_file_format.html
- **ISO 14496-1 Media Format**
<http://www.geocities.com/xhelmboyx/quicktime/formats/mp4-layout.txt>
- **FreeDB Information**
http://www.freedb.org/en/download__miscellaneous.11.html
- **ReplayGain**
<http://replaygain.hydrogenaudio.org>