

Audio Formats Reference

Brian Langenberger

October 23, 2010

Contents

1	Introduction	9
2	the Basics	11
2.1	Hexadecimal	11
2.2	Signed Integers	12
2.3	Endianness	13
2.4	Character Encodings	13
2.5	PCM	14
2.6	Bitstreams	15
3	Waveform Audio File Format	17
3.1	the RIFF WAVE Stream	17
3.2	the Classic ‘fmt’ Chunk	17
3.3	the WAVEFORMATEXTENSIBLE ‘fmt’ Chunk	18
3.4	the ‘data’ Chunk	18
3.5	Channel Assignment	19
4	Audio Interchange File Format	21
4.1	the AIFF File Stream	21
4.2	the COMM Chunk	21
4.3	the SSND Chunk	22
5	Sun AU	23
5.1	the Sun AU File Stream	23
6	Shorten	25
6.1	Shorten Data Types	25
6.2	the Shorten File Stream	26
6.3	Shorten Decoding	27
6.3.1	the DIFF Command	27
6.3.2	Channels and Wrapping	28
6.3.3	the QUIT Command	28
6.3.4	the BLOCKSIZE Command	28
6.3.5	the ZERO Command	28
6.3.6	the BITSHIFT Command	28

6.3.7	the QLPC Command	29
6.3.8	the Coffset	30
6.3.9	the VERBATIM Command	30
6.4	Shorten Encoding	31
6.4.1	the VERBATIM Command	32
6.4.2	the BLOCKSIZE Command	32
6.4.3	the QUIT Command	32
6.4.4	the ZERO Command	32
6.4.5	the DIFF Commands	33
7	Free Lossless Audio Codec	35
7.1	the FLAC File Stream	35
7.2	FLAC Metadata Blocks	36
7.2.1	STREAMINFO	36
7.2.2	PADDING	36
7.2.3	APPLICATION	36
7.2.4	SEEKTABLE	36
7.2.5	VORBIS_COMMENT	37
7.2.6	CUESHEET	38
7.2.7	PICTURE	38
7.3	FLAC Decoding	39
7.3.1	CONSTANT Subframe	40
7.3.2	VERBATIM Subframe	40
7.3.3	FIXED Subframe	40
7.3.4	LPC Subframe	42
7.3.5	the Residual	44
7.3.6	Channel Assignment	46
7.3.7	Wasted Bits per Sample	47
7.4	FLAC Encoding	48
7.4.1	the STREAMINFO Metadata Block	48
7.4.2	the Frame Header	49
7.4.3	Channel Assignment	50
7.4.4	the Subframe Header	50
7.4.5	the CONSTANT Subframe	50
7.4.6	the VERBATIM Subframe	51
7.4.7	the FIXED Subframe	51
7.4.8	the LPC Subframe	54
7.4.9	the Residual	64
7.4.10	Checksums	66
8	WavPack	69
8.1	the WavPack File Stream	69

8.2	the WavPack Block Header	70
8.2.1	WavPack Sub-Block	71
8.3	WavPack Decoding	71
8.3.1	False Stereo	72
8.3.2	the Decorrelation Terms Sub-Block	72
8.3.3	the Decorrelation Weights Sub-Block	73
8.3.4	the Decorrelation Samples Sub-Block	74
8.3.5	the Entropy Variables Sub-Block	76
8.3.6	the Bitstream Sub-Block	77
8.3.7	Sample Decorrelation	83
8.3.8	Joint Stereo	88
8.3.9	the CRC	88
8.3.10	Extended/Shifted Integers	89
8.3.11	False Stereo	89
8.3.12	RIFF WAVE Header/Footer	90
8.3.13	MD5	90
8.4	WavPack Encoding	91
8.4.1	False Stereo	92
8.4.2	Extended/Shifted Integers	92
8.4.3	the CRC	92
8.4.4	Joint Stereo	92
8.4.5	Block Header	93
8.4.6	Decorrelation Terms/Deltas	94
8.4.7	Decorrelation Passes	95
8.4.8	Decorrelation Weights	98
8.4.9	Decorrelation Samples	99
8.4.10	the Entropy Variables Sub-Block	101
8.4.11	the Bitstream Sub-Block	102
8.4.12	Extended Integers	106
8.4.13	RIFF WAVE Header	107
8.4.14	the Footer Block	107
9	Monkey's Audio	109
9.1	the Monkey's Audio File Stream	109
9.2	the Monkey's Audio Descriptor	109
9.3	the Monkey's Audio header	109
9.4	the APEv2 Tag	110
9.4.1	the APEv2 Tag Header/Footer	111
9.4.2	the APEv2 Flags	111
10	MP3	113
10.1	the MP3 File Stream	113

Contents

10.1.1	the Xing Header	114
10.2	ID3v1 Tags	115
10.2.1	ID3v1	115
10.2.2	ID3v1.1	115
10.3	ID3v2 Tags	116
10.3.1	ID3v2.2	116
10.3.2	ID3v2.3	119
10.3.3	ID3v2.4	122
11	M4A	125
11.1	the QuickTime File Stream	125
11.1.1	a QuickTime Atom	125
11.1.2	Container Atoms	125
11.2	M4A Atoms	126
11.2.1	the ftyp Atom	126
11.2.2	the mvhd Atom	126
11.2.3	the tkhd Atom	127
11.2.4	the mdhd Atom	127
11.2.5	the hdlr Atom	128
11.2.6	the smhd Atom	128
11.2.7	the dref Atom	128
11.2.8	the stsd Atom	129
11.2.9	the mp4a Atom	129
11.2.10	the stts Atom	130
11.2.11	the stsc Atom	130
11.2.12	the stsz Atom	130
11.2.13	the stco Atom	131
11.2.14	the meta Atom	131
12	Apple Lossless	133
12.1	the ALAC File Stream	133
12.2	ALAC Decoding	134
12.2.1	Residual Decoding	136
12.2.2	Subframe Calculation	139
12.2.3	Channel Decorrelation	142
12.2.4	Wasted Bits	142
12.3	ALAC Encoding	143
12.3.1	ALAC Atoms	143
12.3.2	Compressed and Uncompressed Frames	153
12.3.3	Channel Correlation	154
12.3.4	Coefficient Calculation	154
12.3.5	Subframe Calculation	161

12.3.6 Residual Encoding	164
13 Ogg Vorbis	167
13.1 Ogg File Stream	167
13.1.1 Ogg Packets	168
13.2 the Identification Packet	168
13.3 the Comment Packet	169
13.4 Channel Assignment	170
14 Ogg FLAC	171
14.1 the Ogg FLAC File Stream	171
15 Ogg Speex	173
15.1 the Header Packet	173
15.2 the Comment Packet	173
16 Musepack	175
16.1 the SV7 File Stream	175
16.2 the SV8 File Stream	176
16.2.1 the SH Packet	176
16.2.2 the SE Packet	177
16.2.3 the RG Packet	177
16.2.4 the EI Packet	177
17 FreeDB	179
17.1 Native Protocol	179
17.1.1 the Disc ID	180
17.1.2 Initial Greeting	180
17.1.3 Client-Server Handshake	181
17.1.4 Set Protocol Level	181
17.1.5 Query Database	182
17.1.6 Read XMCD Data	183
17.1.7 Close Connection	183
17.2 Web Protocol	184
17.3 XMCD	184
18 MusicBrainz	185
18.1 Searching Releases	185
18.1.1 the Disc ID	186
18.1.2 Server Query	187
18.1.3 Release XML	187
18.2 MusicBrainz XML	188

Contents

19 ReplayGain	195
19.1 Applying ReplayGain	195
19.2 Calculating ReplayGain	196
19.2.1 the Equal Loudness Filter	196
19.2.2 RMS Energy Blocks	198
19.2.3 Statistical Processing and Calibration	198
Appendices	199
A References	201
B License	203
B.1 Definitions	203
B.2 Fair Dealing Rights.	205
B.3 License Grant.	205
B.4 Restrictions.	206
B.5 Representations, Warranties and Disclaimer	208
B.6 Limitation on Liability.	209
B.7 Termination	209
B.8 Miscellaneous	209

1 Introduction

This book is intended as a reference for anyone who's ever looked at their collection of audio files and wondered how they worked. Though still a work-in-progress, my goal is to create documentation on the full decoding/encoding process of as many audio formats as possible.

Though to be honest, the audience for this is myself. I enjoy figuring out the little details of how these formats operate. And as I figure them out and implement them in Python Audio Tools, I then add some documentation here on what I've just discovered. That way, when I have to come back to something six months from now, I can return to some written documentation instead of having to go directly to my source code.

Therefore, I try to make my documentation as clear and concise as possible. Otherwise, what's the advantage over simply diving back into the source? Yet this process often turns into a challenge of its own; I'll discover that a topic I thought I'd understood wasn't so easy to grasp once I had to simplify and explain it to some hypothetical future self. Thus, I'll have to learn it better in order to explain it better.

That said, there's still much work left to do. Because it's a repository of my knowledge, it also illustrates the limits of my knowledge. Many formats are little more than "stubs", containing just enough information to extract such metadata as sample rate or bits-per-sample. These are formats in which my Python Audio Tools passes the encoding/decoding task to a binary "black-box" executable since I haven't yet taken the time to learn how to perform that work myself. But my hope is that as I learn more, this work will become more fleshed-out and widely useful.

In the meantime, by including it with Python Audio Tools, my hope is that someone else with some passing interest might also get some use out of what I've learned. And though I strive for accuracy (for my own sake, at least) I cannot guarantee it. When in doubt, consult the references on page 201 for links to external sources which may have additional information.

2 the Basics

2.1 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:

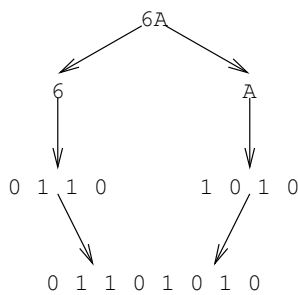
$$(6 \times 10^2) + (7 \times 10^1) + (5 \times 10^0) = 675$$

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 ($16 = 2^4$). This makes it very easy to go back and forth between hexadecimal and binary. For instance, let's take the byte 6A:



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

Going from binary
to hexadecimal is a simple matter of reversing the process.

2.2 Signed Integers

Signed integers are typically stored as “2’s-complement” values. To decode them, one needs to know the integer’s size in bits, its topmost (most-significant) bit value and the value of its remaining bits.

$$\text{signed value} = \begin{cases} \text{remaining bits} & \text{if topmost bit} = 0 \\ \text{remaining bits} - (2^{\text{integer size}-1}) & \text{if topmost bit} = 1 \end{cases} \quad (2.1)$$

For example, take an 8-bit integer whose bit values are 00000101. Since the topmost bit is 0, its value is simply 0000101, which is 5 in base-10 ($2^2 + 2^0 = 5$).

Next, let’s take an integer whose bit values are 11111011. Its topmost bit is 1 and its remaining bits are 1111011, which is 123 in base-10 ($2^6 + 2^5 + 2^4 + 2^3 + 2^1 + 2^0 = 123$). Therefore:

$$\begin{aligned} \text{signed value} &= 123 - 2^{8-1} \\ &= 123 - 128 \\ &= -5 \end{aligned}$$

Transforming a signed integer into its unsigned 2’s-complement value is a simple matter of reversing the process.

$$\text{unsigned value} = \begin{cases} \text{signed value} & \text{if signed value} \geq 0 \\ 2^{\text{integer size}} - (-\text{signed value}) & \text{if signed value} < 0 \end{cases} \quad (2.2)$$

For example, let’s convert the value -20 to a signed, 8-bit integer:

$$\begin{aligned} \text{unsigned value} &= 2^8 - (- - 20) \\ &= 256 - 20 \\ &= 236 \end{aligned}$$

which is 11101100 in binary ($2^7 + 2^6 + 2^5 + 2^3 + 2^2 = 236$).

2.3 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.

When converting a signed, little-endian value to an integer, the 2's-complement decoding comes *after* one performs the endianness reversing. For example, given a signed 16-bit little-endian value of 0xFBFF, one firsts reorders the bytes to 0xFFFF before decoding it to a signed value ($32763 - 2^{15} = -5$).

Conversely, when converting a signed integer to a little-endian value, the endian reversing comes *after* one performs the 2's-complement encoding.

This is covered in more depth in the bitstreams section on page 15.

2.4 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.

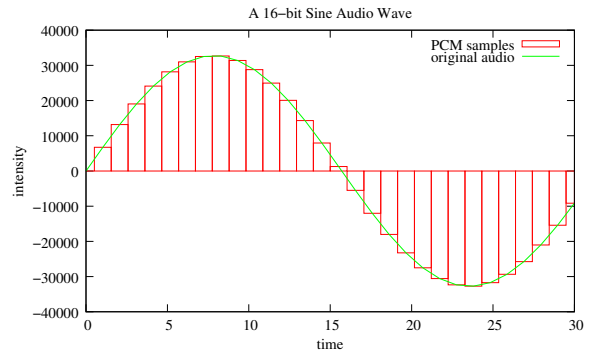
As an example, take the simple character é. In Latin-1 encoding, it is stored as a single byte 0xE9. In UTF-8 encoding, it is stored as the bytes 0xC3A9. In UTF-16BE encoding, it is stored as the bytes 0x00E9.

Although decoding and encoding text is a complex subject beyond the scope of this document, you must always be aware that 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.

2.5 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 4 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.

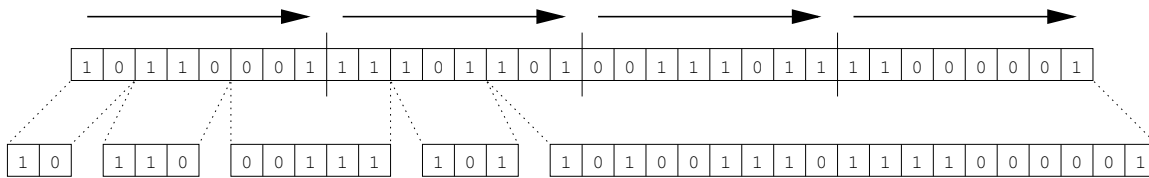
2.6 Bitstreams

Many formats are broken up into pieces smaller than an individual byte. As an example, let's take a 32-bit structure of data broken up into 5 fields as follows:



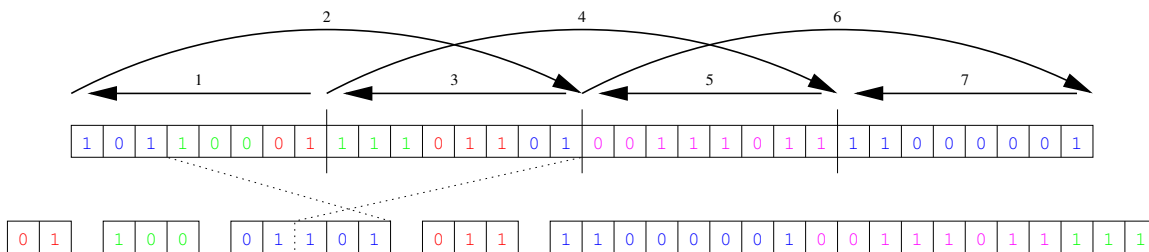
Note that field A is 2 bits, B is 3 bits, C is 5 bits, D is 3 bits and E is 19 bits.

Given the bytes B1 ED 3B C1 in hexadecimal, how we place those bytes within our structure depends whether or not our stream is big-endian or little-endian. For big-endian, the breakdown is largely intuitive:



field	base-2	base-16	base-10
A	10	2	2
B	110	6	6
C	0 0111	07	7
D	101	5	5
E	101 0011 1011 1100 0001	53BC1	342977

However, for little-endian streams, the same bits are read quite differently:



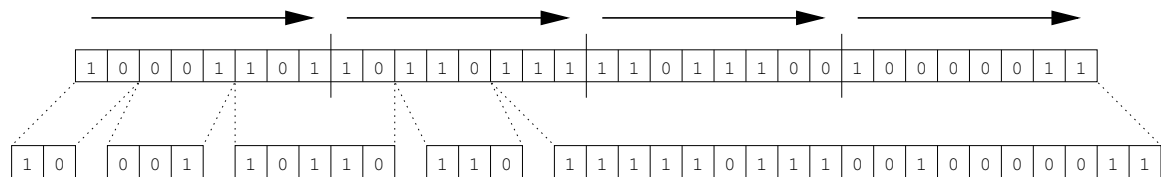
field	base-2	base-16	base-10
A	01	1	1
B	100	4	4
C	0 1101	0D	13
D	011	3	3
E	110 0000 1001 1101 1111	609DF	395743

2 the Basics

This arrangement seems bizarre and counterintuitive when using the same hexadecimal-to-binary conversion we've been using thus far. However, if we *reverse* our conversion process and place the least-significant bits on the left, as follows:

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

The bytes B1 ED 3B C1 are then laid out differently also and can now be read in a linear fashion:



As a further example of how this works, we'll convert the first four fields' binary digits to base-10:

$$A = (1 \times 2^0) + (0 \times 2^1) = 1$$

$$B = (0 \times 2^0) + (0 \times 2^1) + (1 \times 2^2) = 4$$

$$C = (1 \times 2^0) + (0 \times 2^1) + (1 \times 2^2) + (1 \times 2^3) + (0 \times 2^4) = 13$$

$$D = (1 \times 2^0) + (1 \times 2^1) + (0 \times 2^2) = 3$$

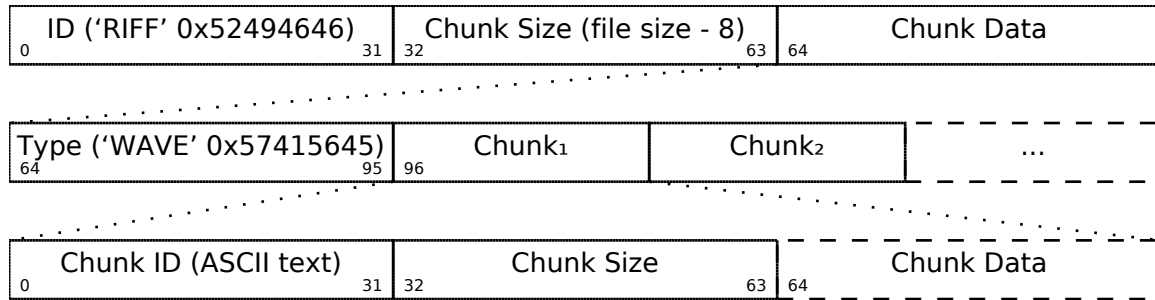
Computing field E is left as an exercise for the reader.

Naturally, these fields from the same bytes have the same values as before; only our way of visualizing them has changed.

3 Waveform Audio File Format

The Waveform Audio File Format 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.

3.1 the RIFF WAVE Stream



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

3.2 the Classic ‘fmt’ Chunk

Wave files with 2 channels or less, and 16 bits-per-sample or less, use a classic ‘fmt’ chunk to indicate its PCM data format. This chunk is required to appear before the ‘data’ chunk.

0	Chunk ID ('fmt' 0x666D7420)	31	32	Chunk Size (16)	63
64	Compression Code (0x0001)	79	80	Channel Count	95
96	Sample Rate				127
128	Average Bytes per Second				159
160	Block Align	175	176	Bits per Sample	191

$$\text{Average Bytes per Second} = \frac{\text{Sample Rate} \times \text{Channel Count} \times \text{Bits per Sample}}{8} \quad (3.1)$$

$$\text{Block Align} = \frac{\text{Channel Count} \times \text{Bits per Sample}}{8} \quad (3.2)$$

3.3 the WAVEFORMATEXTENSIBLE ‘fmt’ Chunk

Wave files with more than 2 channels or more than 16 bits-per-sample should use a WAVEFORMATEXTENSIBLE ‘fmt’ chunk which contains additional fields for channel assignment.

0		Chunk ID ('fmt ' 0x666D7420)		31	32		Chunk Size (40)		63	
64		Compression Code (0xFFFE)		79	80		Channel Count		95	
96		Sample Rate								127
128		Average Bytes per Second								159
160		Block Align		175	176		Bits per Sample		191	
192		CB Size (22)		207	208		Valid Bits per Sample		223	
Front Right of Center 224		Front Left of Center 225		Rear Right 226		Rear Left 227				
LFE 228		Front Center 229		Front Right 230		Front Left 231				
Top Back Left 232		Top Front Right 233		Top Front Center 234		Top Front Left 235				
Top Center 236		Side Right 237		Side Left 238		Back Center 239				
240		Undefined 245		Top Back Right 246		Top Back Center 247		248 255		
256		Sub Format (0x0100000000001000800000aa00389b71)								383

Note that the ‘Average Bytes per Second’ and ‘Block Align’ fields are calculated the same as a classic fmt chunk.

3.4 the ‘data’ Chunk

0		Chunk ID ('data' 0x64617461)		31	32	Chunk Size		63
64		PCM Data						

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

3.5 Channel Assignment

Channels whose bits are set in the WAVEFORMATEXTENSIBLE ‘fmt’ chunk appear in the following order:

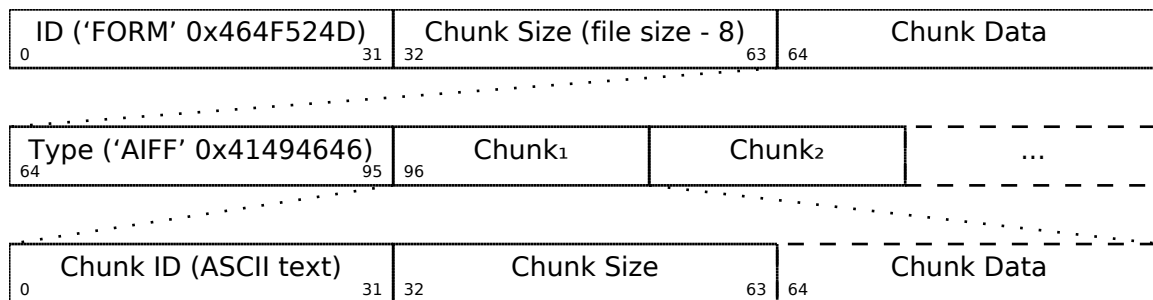
Index	Channel	Mask Bit
1	Front Left	0x1
2	Front Right	0x2
3	Front Center	0x4
4	LFE	0x8
5	Back Left	0x10
6	Back Right	0x20
7	Front Left of Center	0x40
8	Front Right of Center	0x80
9	Back Center	0x100
10	Side Left	0x200
11	Side Right	0x400
12	Top Center	0x800
13	Top Front Left	0x1000
14	Top Front Center	0x2000
15	Top Front Right	0x4000
16	Top Back Left	0x8000
17	Top Back Center	0x10000
18	Top Back Right	0x20000

For example, if the file’s channel mask is set to 0x33, it contains the channels ‘Front Left’, ‘Front Right’, ‘Back Left’ and ‘Back Right’, in that order.

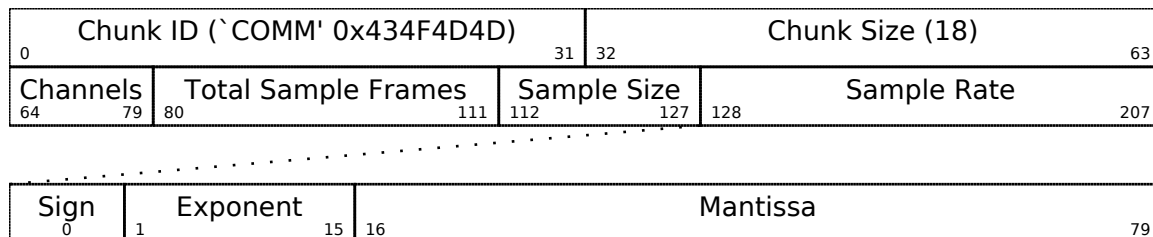
4 Audio Interchange File Format

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.

4.1 the AIFF File Stream



4.2 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.

$$\text{Value} = (-)^{\text{Sign}} \frac{\text{Mantissa}}{2^{63}} \times 2^{\text{Exponent} - 16383}$$

4 Audio Interchange File Format

For example, given a sign bit of 0, an exponent value of 0x400E and a mantissa value of 0xAC44000000000000:

$$\begin{aligned}\text{Value} &= \frac{12413046472939929600}{2^{63}} \times 2^{16398-16383} \\ &= 1.3458251953125 \times 2^{15} \\ &= 44100.0\end{aligned}$$

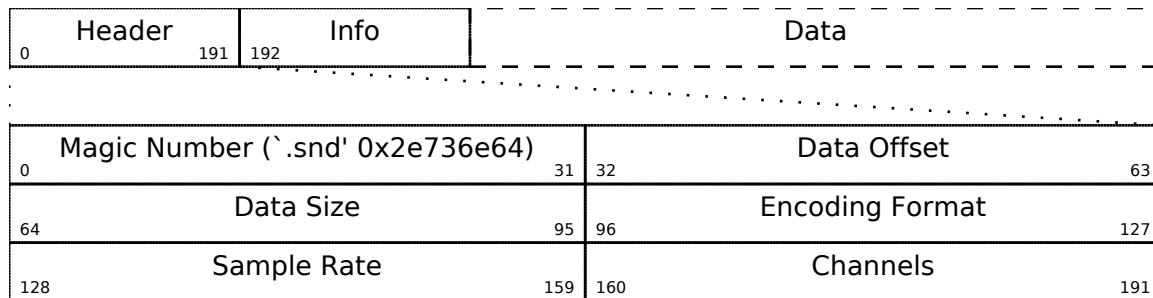
4.3 the SSND Chunk

0	Chunk ID (`SSND' 0x53534E44)	31	32	Chunk Size	63
64	Offset	95	96	Block Size	127
128	PCM Data				

5 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.

5.1 the Sun AU File Stream



value	encoding 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

5 Sun AU

6 Shorten

Shorten is one of the earliest lossless audio compression formats. Though superseded by FLAC and other formats, it remains interesting from a historical perspective.

6.1 Shorten Data Types

Notably, almost nothing in the Shorten file format is byte-aligned. Instead, it uses its own set of variable-length types which I'll refer to as **unsigned**, **signed** and **long**.

An **unsigned** field of a certain 'size' means we first take a unary-encoded¹, number of high bits and combine the resulting value with 'size' number of low bits. For example, given a 'size' of 2 and the bits '0 0 1 1 1', the high unary value of '0 0 1' combines with the low raw value of '1 1' resulting in a decimal value of 11.

A **signed** field is similar, but its low value contains one additional trailing bit for the sign value.

$$\text{signed value} = \begin{cases} \text{unsigned value} & \text{if sign bit} = 0 \\ -\text{unsigned value} - 1 & \text{if sign bit} = 1 \end{cases}$$

For example, given a 'size' of 3 and the bits '0 1 1 0 1 1', the high unary value of '0 1' combines with the low raw value of '1 0 1' and the sign bit '1' resulting in a decimal value of -14. Note that the sign bit is counted separately, so we're actually reading 4 additional bits after the unary value in this case.

Lastly, and most confusingly, a **long** field is the combination of two separate **unsigned** fields. The first, of size 2, determines the size value of the second. For example, given the bits '1 1 1 1 1 0 1', the first **unsigned** field of '1 1 1' has the value of 3 (unary 0 combined with a raw value of 3) - which is the size of the next **unsigned** field. That field, in turn, consists of the bits '1 1 0 1' which is 5 (unary 0 combined with a raw value of 5). So, the value of the entire **long** field is 5.

A Shorten file consists almost entirely of these three types in various sizes. Therefore, when one reads "**unsigned**(3)" in a Shorten field description, it means an **unsigned** field of size 3.

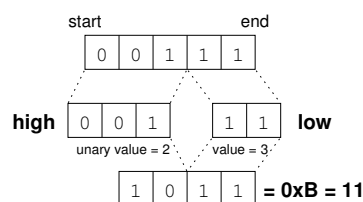


Figure 6.1: Unsigned

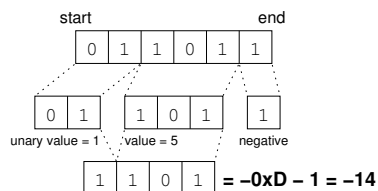
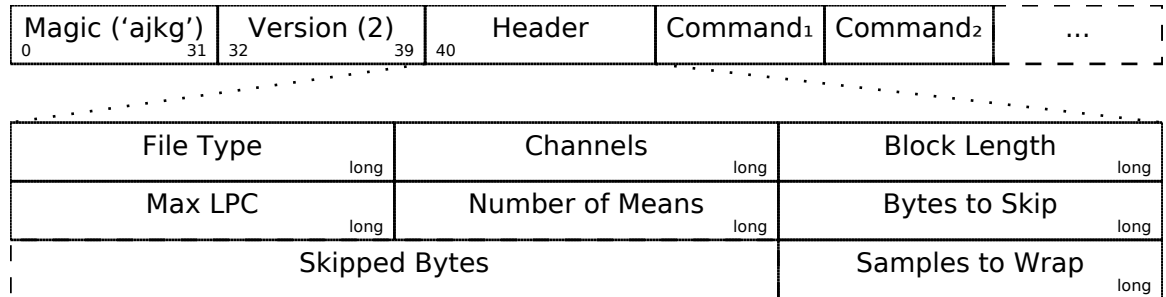


Figure 6.2: Signed

¹In 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.

6.2 the Shorten File Stream



file type	format
0	lossless μ -Law
1	signed 8 bit
2	unsigned 8 bit
3	signed 16 bit, big-endian
4	unsigned 16 bit, big-endian
5	signed 16 bit, little-endian
6	unsigned 16 bit, little-endian
7	lossy μ -Law
8	new μ -Law with zero mapping
9	lossless a-Law
10	lossy a-Law
11	Microsoft .wav
12	Apple .aiff

‘Channels’ is the number of channels in the audio stream. ‘Block Length’ is the length of each command block, in samples. ‘Max LPC’ is the maximum LPC value a block may have. ‘Samples to Wrap’ is the number of samples to be wrapped around from the top of an output block to the bottom. This will be explained in more detail in the decoding section.

6.3 Shorten Decoding

Internally, a Shorten file acts as a list of commands to be executed by a tiny virtual machine.² Each command is a `unsigned(2)` field followed by zero or more arguments.

value	command	value	command
0	DIFF0	5	BLOCKSIZE
1	DIFF1	6	BITSHIFT
2	DIFF2	7	QLPC
3	DIFF3	8	ZERO
4	QUIT	9	VERBATIM

6.3.1 the DIFF Command

All four DIFF commands are structured the same:

Energy Size <small>unsigned(3)</small>	Residual ₁ <small>signed(energy)</small>	Residual ₂ <small>signed(energy)</small>	...	Residual _x <small>signed(energy)</small>
---	--	--	-----	--

There are ‘Block Size’ number of residuals per DIFF (whose initial value is determined by the Shorten header) and each one’s size is determined by ‘Energy Size’. The process of transforming these residuals into samples depends on the DIFF command and the values of previously decoded samples.

Command	Calculation
DIFF0	$Sample_i = Residual_i + Offset^a$
DIFF1	$Sample_i = Sample_{i-1} + Residual_i$
DIFF2	$Sample_i = (2 \times Sample_{i-1}) - Sample_{i-2} + Residual_i$
DIFF3	$Sample_i = (3 \times (Sample_{i-1} - Sample_{i-2})) + Sample_{i-3} + Residual_i$

^aSee page 30

For example, given a DIFF1 command at the stream’s beginning and the residual values 10, 1, 2, -2, 1 and -1, samples are calculated as follows:

Index	Residual	Sample
-1	(before stream)	(not output) 0
0	10	$0 + 10 = \mathbf{10}$
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	$12 - 1 = \mathbf{11}$

²Interestingly, although Shorten’s successor, FLAC, presents its input as frames and subframes, references to a FLAC virtual machine are still present in its source code.

6.3.2 Channels and Wrapping

The audio commands `DIFF`, `QLPC` and `ZERO` send their samples to channels in order. For example, a stream of `DIFF` commands in a 2 channel stereo stream (a very typical configuration) sends `DIFF1` to the left channel, `DIFF2` to the right channel, `DIFF3` to left channel, `DIFF4` to the right channel and so on.

However, recall that most of the `DIFF` commands require previously decoded samples as part of their calculation. What this means is that `DIFF3` takes the last few samples from `DIFF1` in order to apply its residuals (since both are on the left channel) and `DIFF4` takes the last few samples from `DIFF2`.

This is where the header's 'Samples to Wrap' field comes into play. Its value is the number of samples to be wrapped from the top of the buffer to its pre-zero values. For example, if 'Sample Count' is 256 and 'Samples to Wrap' is 3 (another typical configuration), `Buffer-1` takes the value of `Buffer255`, `Buffer-2` takes the value of `Buffer254`, and `Buffer-3` takes the value of `Buffer253`. However, these pre-zero starting-point values are obviously not re-output when the buffer is finally completed and returned.

6.3.3 the QUIT Command

This command takes no arguments. It indicates the Shorten stream is finished and decoding is completed.

6.3.4 the BLOCKSIZE Command

This command takes a single `long` argument whose value is the new 'Block Size'. In effect, it modifies that variable in the Shorten virtual machine.

6.3.5 the ZERO Command

This command takes no arguments. It simply generates 'Block Size' number of zero samples into the current channel's output buffer.

6.3.6 the BITSHIFT Command

This commands takes a single `unsigned(2)` value and modifies the 'bitshift' variable in the Shorten virtual machine. This value is how many bits to left-shift all output samples prior to returning them.

For example, imagine a scenario in which all the samples in a set of blocks have 0 for their rightmost (least significant) bit. Setting a bitshift of 1 allows us to ignore that bit during calculation which, in turn, allows us to store those samples more efficiently.

Note that bit shifting is applied *after* channel wrapping and 'offset' calculation³.

³See page 30

6.3.7 the QLPC Command

The QLPC command is structured as follows:

Energy Size unsigned(2)	LPC Count unsigned(2)	LPC Coeff ₁ signed(5)	...	LPC Coeff _x signed(5)
Residual ₁ signed(energy)	Residual ₂ signed(energy)	Residual ₃ signed(energy)	...	Residual _x signed(energy)

So, given a set of LPC coefficients and a set of residuals, samples are calculated using the following formula:

$$\text{Sample}_i = \left\lfloor \frac{2^5 + \sum_{j=0}^{\text{Count}-1} \text{LPC Coefficient}_j \times \text{Sample}_{i-j-1}}{2^5} \right\rfloor + \text{Residual}_i \quad (6.1)$$

This simply means we're taking the sum of the calculated values from 0 to LPC Count - 1, bit-shifting that sum down and added the residual when determining the current sample. As with the **DIFF** commands, previously encoded samples (possibly from previous commands) are used to calculate the current sample.

For example, given the LPC Coefficients and previously encoded samples:

LPC Coefficient ₀	21		Sample ₁	-2
LPC Coefficient ₁	2		Sample ₂	-3
LPC Coefficient ₂	7		Sample ₃	-2

Index	Residual	Sample
1		-2
2		-3
3		-2
4	1	$\left\lfloor \frac{2^5 + (21 \times -2) + (2 \times -3) + (7 \times -2)}{2^5} \right\rfloor + 1 = \left\lfloor \frac{32-62}{32} \right\rfloor + 1 = -1 + 1 = 0$
5	-2	$\left\lfloor \frac{2^5 + (21 \times 0) + (2 \times -2) + (7 \times -3)}{2^5} \right\rfloor - 2 = \left\lfloor \frac{32-25}{32} \right\rfloor - 2 = 0 - 2 = -2$
6	-1	$\left\lfloor \frac{2^5 + (21 \times -2) + (2 \times 0) + (7 \times -2)}{2^5} \right\rfloor - 1 = \left\lfloor \frac{32-56}{32} \right\rfloor - 1 = -1 - 1 = -2$

Unfortunately, there's one more wrinkle to consider for proper QLPC command decoding: the 'offset'. How to calculate this value will be covered in the next section. But when a QLPC command is encountered, the offset value is subtracted from the QLPC's warm-up samples (taken from the top of the previous command, for the current channel). Then that offset value is re-added to our output samples after calculation.

For example, given a 'offset' value of 5, one would subtract 5 from Sample₃, Sample₂ and Sample₁, perform the QLPC calculation and then add 5 to our Sample₀, Sample₁, Sample₂, ... , Sample₂₅₅ before returning those values.

6.3.8 the Coffset

Calculating the ‘coffset’ value for a given command on a given channel requires a set of ‘offset’ values (whose count equals the ‘Number of Means’, from the Shorten header) and the ‘Number of Means’ value itself.

$$\text{coffset} = \frac{\frac{\text{nmeans}}{2} + \sum_{i=0}^{\text{nmeans}-1} \text{offset}_i}{\text{nmeans}} \quad (6.2)$$

For example, given a ‘Number of Means’ value of 4 and offsets of:

```
offset0 32
offset1 28
offset2 17
offset3 14
```

$$\text{coffset} = \frac{\frac{4}{2} + (32 + 28 + 17 + 14)}{4} = \frac{93}{4} = \mathbf{23} \quad (6.3)$$

The next obvious question is where do those ‘offset’ values come from? They’re actually a queue of (mostly) sample value averages on the given channel. So once we’ve decoded offset for command₅ on channel 0, offset₀ takes the value of offset₁, offset₁ takes the value of offset₂, offset₂ takes the value of offset₃, and command₅’s offset becomes the new offset₃.

However, the offset is not entirely a sample average. Its actual formula is as follows:

$$\text{offset} = \frac{\frac{\text{block size}}{2} + \sum_{i=0}^{\text{block size}-1} \text{sample}_i}{\text{block size}} \quad (6.4)$$

For example, if a command with a ‘block size’ of 256 has samples that total 1056, its offset value is:

$$\text{offset} = \frac{\frac{256}{2} + 1056}{256} = \frac{1184}{256} = \mathbf{4} \quad (6.5)$$

6.3.9 the VERBATIM Command

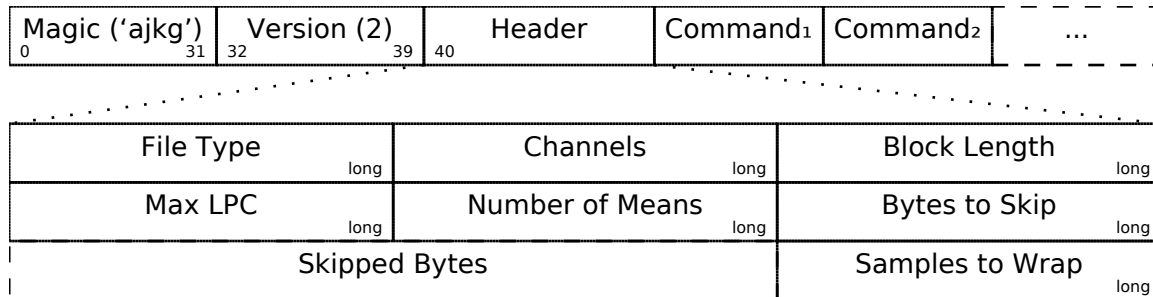
This command is for generating raw, non-audio data such as .wav or .aiff chunks and is structured as follows:

Total Bytes unsigned(5)	Byte ₁ unsigned(8)	Byte ₂ unsigned(8)	...	Byte _x unsigned(8)
----------------------------	----------------------------------	----------------------------------	-----	----------------------------------

These chunks of raw data are expected to be written in the order they appear in the Shorten file.

6.4 Shorten Encoding

For the purposes of Shorten encoding, one needs an entire PCM container file and its PCM values, number of channels and bits per sample. Recall that nearly the entire Shorten file format is made up of three variable-length data types with no byte-alignment of any sort. Because of that, there's no way to “rewind” the Shorten stream and replace values. Therefore, everything encoded to Shorten must be written in order.



Remember that `long` variables contain a size `unsigned(2)` field followed by its value. So what's the best size to use for a given value? Quite frankly, in my opinion, any small size will do. Most of a Shorten file is encoded residuals, so wasting a handful of bytes in the header won't make any difference.

A good set of header values to use are as follows:

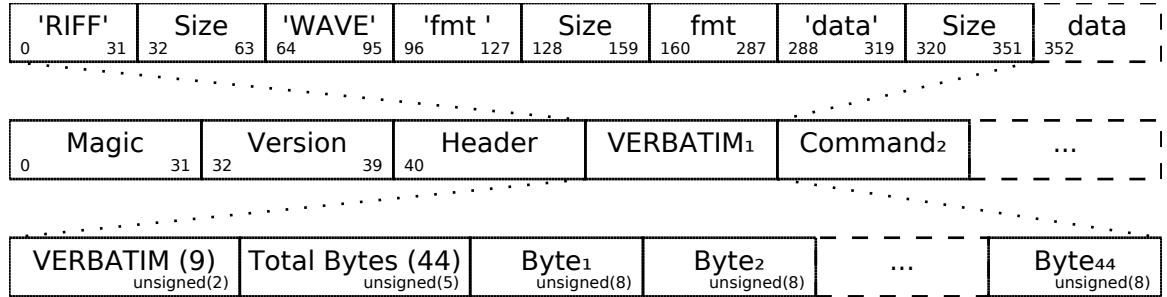
field	value
Field Type	2 for unsigned 8 bit input 5 for signed, little-endian, 16 bit input 3 for signed, big-endian, 16 bit input
Channels	from input
Block Length	256
Max LPC	0
Number of Means	0
Bytes to Skip	0
Samples to Wrap	3

The remainder of the stream is various Shorten commands. I'll be limiting this encoding documentation to the `DIFF1`, `DIFF2`, `DIFF3`, `QUIT`, `BLOCKSIZE`, `ZERO` and `VERBATIM` commands. `QLPC`'s implementation is actually broken in the reference decoder⁴ and tends not to produce smaller files. And, by omitting `DIFF0` which is rarely used, we can avoid 'offset' calculation during encoding.

⁴By limiting its accumulator to a 32-bit integer, it's prone to overflow at high LPC counts.

6.4.1 the VERBATIM Command

PCM containers such as Wave and AIFF consist of several blocks of data, one of which contains a large chunk of PCM data. Shorten encodes these files by turning non-audio data at the beginning and end of the container into VERBATIM commands, often in the following format:



These VERBATIM commands *must* appear in the Shorten stream in the same order as they appear in the PCM container itself.

6.4.2 the BLOCKSIZE Command

Most DIFF and ZERO commands should contain 'Block Length' number of samples, as indicated in the header. But when the end of the input stream is reached and a different number of samples are all that remain, a BLOCKSIZE is required. This is simply an **unsigned(2)** variable with a value of 5 (indicating the BLOCKSIZE command) followed by a **long** variable containing the new 'Block Length'.

6.4.3 the QUIT Command

When no more samples remain and all VERBATIM commands have been delivered (a Wave file may contain additional chunks after the 'data' chunk, for example), the QUIT command indicates the end of the Shorten stream. This is an **unsigned(2)** variable with the value 4.

However, the stream *must* be padded such that the total stream length, minus 5 bytes for the magic number and version, is a multiple of 4 bytes. If this padding is not performed, the reference decoder's bit stream implementation will exit with an error.

6.4.4 the ZERO Command

When a block's entire set of samples are 0, we can generate a ZERO command to represent them, which is simply the **unsigned(2)** command variable of 8.

6.4.5 the DIFF Commands

The bulk of a Shorten file is DIFF commands, which are **unsigned(2)** command variables 1, 2, and 3 for DIFF1, DIFF2 and DIFF3, respectively, followed by the same argument syntax:

Energy Size unsigned(3)	Residual ₁ signed(energy)	Residual ₂ signed(energy)	...	Residual _x signed(energy)
----------------------------	---	---	-----	---

Given a set of input samples, we decide which DIFF command to use by calculating their minimum delta sum, which is best explained by example:

index	sample	Δ^1	Δ^2	Δ^3
-1	0			
0	18	-18		
1	20	-2	-16	
2	26	-6	4	-20
3	24	2	-8	12
4	24	0	2	-10
5	23	1	-1	3
6	21	2	-1	0
7	24	-3	5	-6
8	23	1	-4	9
9	20	3	-2	-2
10	18	2	1	-3
11	18	0	2	-1
12	17	1	-1	3
13	17	0	1	-2
14	20	-3	3	-2
15	23	-3	0	3
16	21	2	-5	5
17	23	-2	4	-9
18	22	1	-3	7
19	18	4	-3	0
	sum	56	66	97

In this example, the |sum| of Δ^1 is the smallest value. Therefore, the best command to use for this set of samples is DIFF1. Once we know which DIFF command to use for a given set of input samples, calculating the command's set of residual values can be done automatically:

command	calculation
DIFF1	$\text{Residual}_i = \text{Sample}_i - \text{Sample}_{i-1}$
DIFF2	$\text{Residual}_i = \text{Sample}_i - ((2 * \text{Sample}_{i-1}) - \text{Sample}_{i-2})$
DIFF3	$\text{Residual}_i = \text{Sample}_i - ((3 \times (\text{Sample}_{i-1} - \text{Sample}_{i-2})) + \text{Sample}_{i-3})$

In this example, our residuals for DIFF1 are: 18, 2, 6, -2, 0, -1, -2, 3, -1, -3, -2, 0, -1, 0, 3, 3, -2, 2, -1, -4.

6 Shorten

Finally, given a set of residual values, we need to determine their ‘Energy Size’. This is done by selecting the smallest value of ‘x’ such that:

$$\text{sample count} \times 2^x > \sum_{i=0}^{\text{residual count}-1} |\text{residual}_i| \quad (6.6)$$

To finish our example, given we have 20 residuals:

index	residual _i	residual _i
0	18	18
1	2	2
2	6	6
3	-2	2
4	0	0
5	-1	1
6	-2	2
7	3	3
8	-1	1
9	-3	3
10	-2	2
11	0	0
12	-1	1
13	0	0
14	3	3
15	3	3
16	-2	2
17	2	2
18	-1	1
19	-4	4
sum		56

$$20 \times 2^0 \leq 56$$

$$20 \times 2^1 \leq 56$$

$$20 \times 2^2 > 56$$

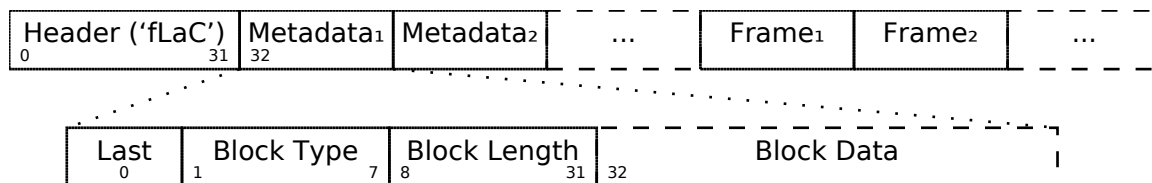
Therefore, the best ‘Energy Size’ for this set of residuals is 2.

7 Free Lossless Audio Codec

FLAC compresses PCM audio data losslessly using predictors and a residual. FLACs 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.

7.1 the FLAC File Stream



'Last' is 0 when there are no 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 Type	Block
0	STREAMINFO
1	PADDING
2	APPLICATION
3	SEEKTABLE
4	VORBIS_COMMENT
5	CUESHEET
6	PICTURE
7-126	reserved
127	invalid

7.2 FLAC Metadata Blocks

7.2.1 STREAMINFO

0	Minimum Block Size (in samples)	15	16	Maximum Block Size (in samples)	31
32	Minimum Frame Size (in bytes)	55	56	Maximum Frame Size (in bytes)	79
80	Sample Rate			99	100
				102	103
				107	107
108	Total Samples				143
144	MD5SUM of PCM Data				271

7.2.2 PADDING

PADDING is simply a block full of NULL (0x00) bytes. Its purpose is to provide extra metadata space within the FLAC file. By having a padding block, other metadata blocks can be grown or shrunk without having to rewrite the entire FLAC file by removing or adding space to the padding.

7.2.3 APPLICATION

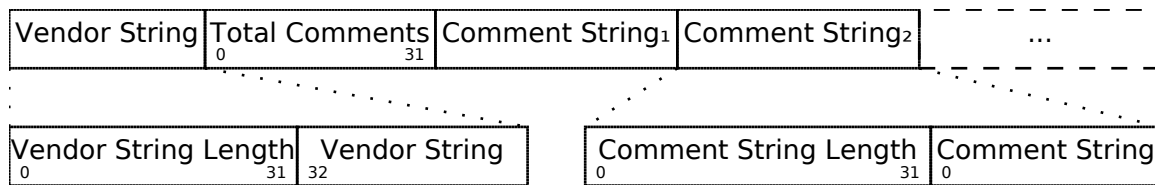
0	Application ID	31	32	Application Data
---	----------------	----	----	------------------

APPLICATION is a general-purpose metadata block used by a variety of different programs. Its contents are defined by the ASCII Application ID value.

7.2.4 SEEKTABLE

0	Seekpoint ₁	143	144	Seekpoint ₂	287	...
0	Sample Number in Target Frame	63	64	Byte Offset to Frame Header	127	128
						143

7.2.5 VORBIS_COMMENT

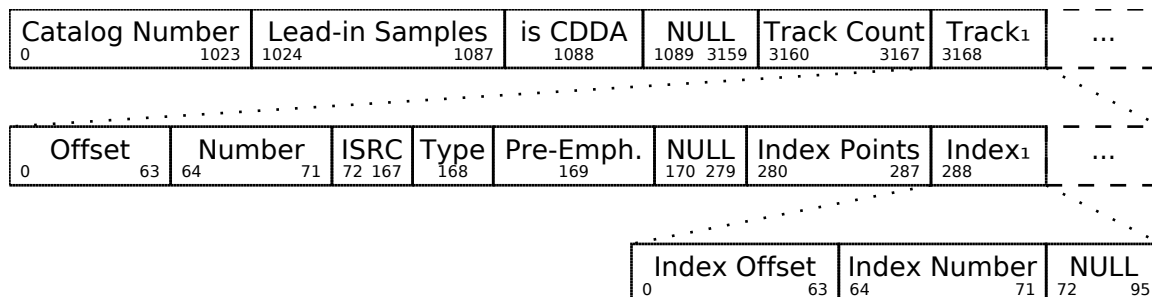


The length fields are all little-endian. The Vendor String and Comment Strings are all UTF-8 encoded. Keys are not case-sensitive and may occur multiple times, indicating multiple values for the same field. For instance, a track with multiple artists may have more than one **ARTIST**.

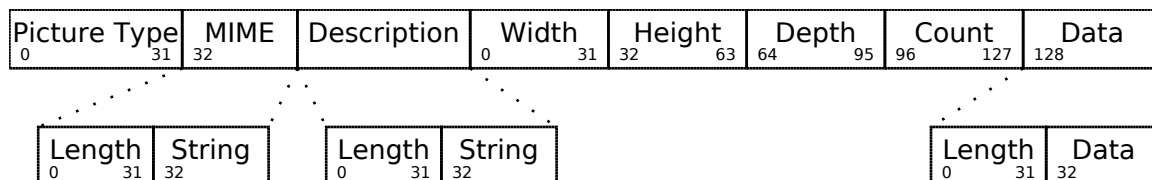
ALBUM	album name	ORGANIZATION	record label
ARTIST	artist name, band name, composer, author, etc.	PART*	track's movement title
CATALOGNUMBER*	CD spine number	PERFORMER	performer name, orchestra, actor, etc.
COMPOSER*	the work's author	PRODUCER*	person responsible for the project
CONDUCTOR*	performing ensemble's leader	PRODUCTNUMBER*	UPC, EAN, or JAN code
COPYRIGHT	copyright attribution	PUBLISHER*	album's publisher
DATE	recording date	RELEASE DATE*	date the album was published
DESCRIPTION	a short description	REMIKXER*	person who created the remix
DISCNUMBER*	disc number for multi-volume work	SOURCE ARTIST*	artist of the work being performed
ENGINEER*	the recording masterer	SOURCE MEDIUM*	CD, radio, cassette, vinyl LP, etc.
ENSEMBLE*	performing group	SOURCE WORK*	a soundtrack's original work
GENRE	a short music genre label	SPARS*	DDD, ADD, AAD, etc.
GUEST ARTIST*	collaborating artist	SUBTITLE*	for multiple track names in a single file
ISRC	ISRC number for the track	TITLE	track name
LICENSE	license information	TRACKNUMBER	track number
LOCATION	recording location	VERSION	track version
OPUS*	number of the work		

Fields marked with * are proposed extension fields and not part of the official Vorbis comment specification.

7.2.6 CUESHEET



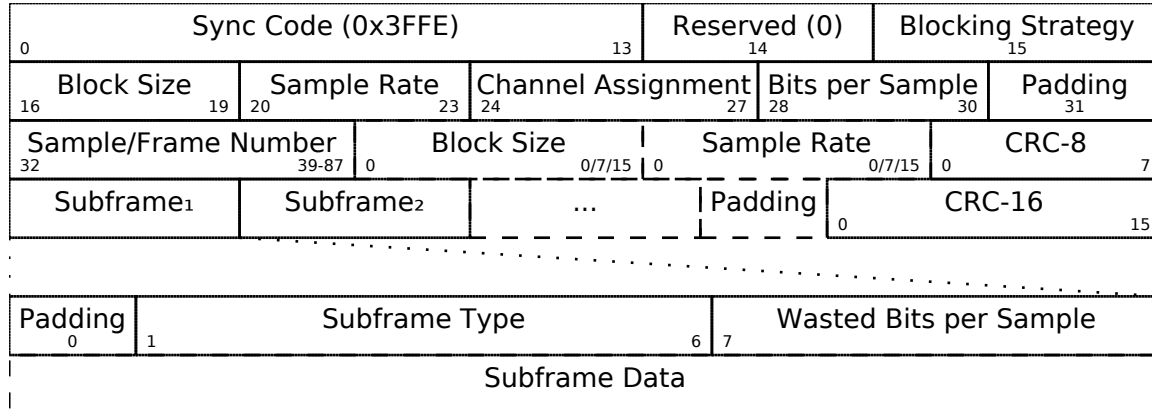
7.2.7 PICTURE



Picture Type	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 colored fish
18	Illustration
19	Band / Artist logotype
20	Publisher / Studio logotype

7.3 FLAC Decoding

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



Value	Block Size	Sample Rate	Channels	Assignment	Bits per Sample	Value
0000	STREAMINFO	STREAMINFO	1	front center	STREAMINFO	0000
0001	192	88200	2	front left, front right	8	0001
0010	576	176400	3	f. left, f. right, f. center	12	0010
0011	1152	192000	4	f. left, f. right, back left, back right	reserved	0011
0100	2304	8000	5	f. L, f. R, f. C, b. L, b. R	16	0100
0101	4608	16000	6	f. L, f. R, f. C, LFE, b. L, b. R	20	0101
0110	8 bits (+1)	22050	7	undefined	24	0110
0111	16 bits (+1)	24000	8	undefined	reserved	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	8 bits (in kHz)		reserved		1100
1101	8192	16 bits (in Hz)		reserved		1101
1110	16384	16 bits (in 10s of Hz)		reserved		1110
1111	32768	invalid		reserved		1111

‘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.

Value	Subframe Type
000000	SUBFRAME_CONSTANT
000001	SUBFRAME_VERBATIM
00001x	reserved
0001xx	reserved
001xxx	SUBFRAME_FIXED
	xxx = predictor order
01xxxx	reserved
1xxxxx	SUBFRAME_LPC
	xxxxxx = predictor order - 1

7.3.1 CONSTANT Subframe

This is the simplest possible subframe. It consists of a single value whose size is equal to the subframe's 'Bits per Sample'¹. For instance, a 16-bit subframe 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.

7.3.2 VERBATIM Subframe

0	Sample ₁	Sample ₂	Sample ₃	...	Sample _x
---	---------------------	---------------------	---------------------	-----	---------------------

Each sample is the size of the subframe's 'Bits per Sample'. The total number of samples equals the subframe'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 suitable predictor can be found.

7.3.3 FIXED Subframe

0	Warm-Up Sample ₁	Warm-Up Sample ₂	...	Residual
---	-----------------------------	-----------------------------	-----	----------

The number of warm-up samples equals the 'Predictor Order' (which is encoded in the 'Subframe Type'). Each warm-up sample is the same size as the subframe's 'Bits per Sample'. 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.

¹ The subframe's 'Bits per Sample' may not be the same size as the frame's 'Bits per Sample'; if the subframe is a difference or side channel, the subframe's 'Bits per Sample' is 1 bit larger than the frame's. Or, if the subframe has wasted-bits, it is treated as having that many bits fewer. This is explained in more detail on page 46.

For Order 0:

$\text{Sample}_i = \text{Residual}_i$ for $i = 0$ to Block Size $- 1$

For Order 1:

$$\text{Sample}_i = \begin{cases} \text{Warm Up}_i & \text{for } i = 0 \\ \text{Sample}_{i-1} + \text{Residual}_{i-1} & \text{for } i = 1 \text{ to Block Size} - 1 \end{cases}$$

For Order 2:

$$\text{Sample}_i = \begin{cases} \text{Warm Up}_i & \text{for } i = 0 \text{ to } 1 \\ (2 \times \text{Sample}_{i-1}) - \text{Sample}_{i-2} + \text{Residual}_{i-2} & \text{for } i = 2 \text{ to Block Size} - 1 \end{cases}$$

For Order 3:

$$\text{Sample}_i = \begin{cases} \text{Warm Up}_i & \text{for } i = 0 \text{ to } 2 \\ (3 \times \text{Sample}_{i-1}) - (3 \times \text{Sample}_{i-2}) + \text{Sample}_{i-3} + \text{Residual}_{i-3} & \text{for } i = 3 \text{ to Block Size} - 1 \end{cases}$$

For Order 4:

$$\text{Sample}_i = \begin{cases} \text{Warm Up}_i & \text{for } i = 0 \text{ to } 3 \\ (4 \times \text{Sample}_{i-1}) - (6 \times \text{Sample}_{i-2}) + (4 \times \text{Sample}_{i-3}) - \text{Sample}_{i-4} + \text{Residual}_{i-4} & \text{for } i = 4 \text{ to Block Size} - 1 \end{cases}$$

For example, given a FIXED order of 1, a Warm Up₀ of 10, the residuals 1, 2, -2, 1 -1, and a ‘Block Size’ of 6, we calculate samples as follows:

$$\begin{aligned} \text{Sample}_0 &= \text{Warm Up}_0 = \mathbf{10} \\ \text{Sample}_1 &= \text{Sample}_0 + \text{Residual}_0 = 10 + 1 = \mathbf{11} \\ \text{Sample}_2 &= \text{Sample}_1 + \text{Residual}_1 = 11 + 2 = \mathbf{13} \\ \text{Sample}_3 &= \text{Sample}_2 + \text{Residual}_2 = 13 - 2 = \mathbf{11} \\ \text{Sample}_4 &= \text{Sample}_3 + \text{Residual}_3 = 11 + 1 = \mathbf{12} \\ \text{Sample}_5 &= \text{Sample}_4 + \text{Residual}_4 = 12 - 1 = \mathbf{11} \end{aligned}$$

How to extract the encoded residual is explained on page 44.

7.3.4 LPC Subframe

Warm-Up Sample ₀ ₁	Warm-Up Sample ₂	...	Warm-Up Sample _x
QLP Precision ₀ ₃	QLP Shift Needed ₄ ₈	QLP Coefficient ₁ ₉	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 warm-up sample equals the subframe’s ‘Bits per Sample’. The size of each QLP Coefficient is equal to ‘QLP Precision’ number of bits, plus 1. ‘QLP Shift Needed’ and the value of each Coefficient are signed two’s-complement integers. The number of Coefficients equals the ‘LPC Order’.

For $i = 0$ to $Order - 1$:

$$Sample_i = Warm\ Up_i$$

For $i = Order$ to Block Size - 1:

$$Sample_i = \left\lfloor \frac{\sum_{j=0}^{Order-1} LPC\ Coefficient_j \times Sample_{i-j-1}}{2^{QLP\ Shift\ Needed}} \right\rfloor + Residual_{i-Order}$$

This 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, the ‘Block Size’ is 10 and the remaining subframe values are as follows:

Warm Up ₀	1053	LPC Coefficient ₀	1241	Residual ₀	11
Warm Up ₁	1116	LPC Coefficient ₁	-944	Residual ₁	79
Warm Up ₂	1257	LPC Coefficient ₂	14	Residual ₂	24
Warm Up ₃	1423	LPC Coefficient ₃	342	Residual ₃	-81
Warm Up ₄	1529	LPC Coefficient ₄	-147	Residual ₄	-72

Note that the number of residuals equals Block Size minus Order which happens to be 5 in this simple example ($10 - 5 = 5$). Most of the time it will be much larger.

For the sake of brevity, in this example we’ll abbreviate ‘QLP Coefficient’ as ‘C’ and ‘Sample’ as ‘S’.

$$\text{Sample}_0 = \text{Warm Up}_0 = \mathbf{1053}$$

$$\text{Sample}_1 = \text{Warm Up}_1 = \mathbf{1116}$$

$$\text{Sample}_2 = \text{Warm Up}_2 = \mathbf{1257}$$

$$\text{Sample}_3 = \text{Warm Up}_3 = \mathbf{1423}$$

$$\text{Sample}_4 = \text{Warm Up}_4 = \mathbf{1529}$$

$$\begin{aligned} \text{Sample}_5 &= \left\lfloor \frac{(C_0 \times S_4) + (C_1 \times S_3) + (C_2 \times S_2) + (C_3 \times S_1) + (C_4 \times S_0)}{2^9} \right\rfloor + \text{Residual}_0 \\ &= \left\lfloor \frac{(1241 \times 1529) + (-944 \times 1423) + (14 \times 1257) + (342 \times 1116) + (-147 \times 1053)}{512} \right\rfloor + 11 \\ &= \left\lfloor \frac{798656}{512} \right\rfloor + 11 = 1559 + 11 = \mathbf{1570} \end{aligned}$$

$$\begin{aligned} \text{Sample}_6 &= \left\lfloor \frac{(C_0 \times S_5) + (C_1 \times S_4) + (C_2 \times S_3) + (C_3 \times S_2) + (C_4 \times S_1)}{2^9} \right\rfloor + \text{Residual}_1 \\ &= \left\lfloor \frac{(1241 \times 1570) + (-944 \times 1529) + (14 \times 1423) + (342 \times 1257) + (-147 \times 1116)}{512} \right\rfloor + 79 \\ &= \left\lfloor \frac{790758}{512} \right\rfloor + 79 = 1544 + 79 = \mathbf{1623} \end{aligned}$$

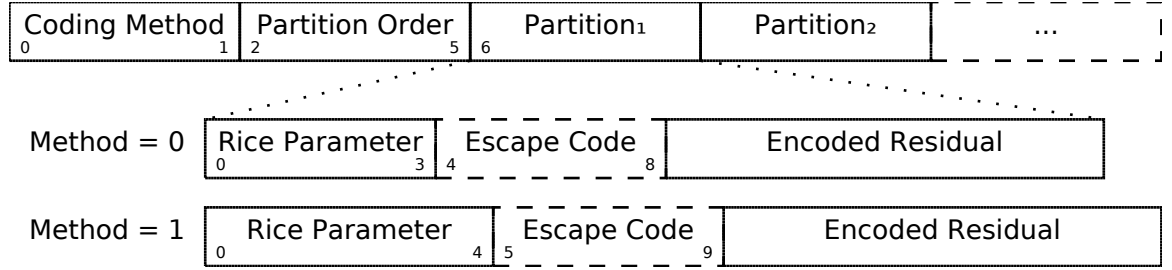
$$\begin{aligned} \text{Sample}_7 &= \left\lfloor \frac{(C_0 \times S_6) + (C_1 \times S_5) + (C_2 \times S_4) + (C_3 \times S_3) + (C_4 \times S_2)}{2^9} \right\rfloor + \text{Residual}_2 \\ &= \left\lfloor \frac{(1241 \times 1623) + (-944 \times 1570) + (14 \times 1529) + (342 \times 1423) + (-147 \times 1257)}{512} \right\rfloor + 24 \\ &= \left\lfloor \frac{855356}{512} \right\rfloor + 24 = 1670 + 24 = \mathbf{1694} \end{aligned}$$

$$\begin{aligned} \text{Sample}_8 &= \left\lfloor \frac{(C_0 \times S_7) + (C_1 \times S_6) + (C_2 \times S_5) + (C_3 \times S_4) + (C_4 \times S_3)}{2^9} \right\rfloor + \text{Residual}_3 \\ &= \left\lfloor \frac{(1241 \times 1694) + (-944 \times 1623) + (14 \times 1570) + (342 \times 1529) + (-147 \times 1423)}{512} \right\rfloor - 81 \\ &= \left\lfloor \frac{1769}{512} \right\rfloor - 81 = 1769 - 81 = \mathbf{1688} \end{aligned}$$

$$\begin{aligned} \text{Sample}_9 &= \left\lfloor \frac{(C_0 \times S_8) + (C_1 \times S_7) + (C_2 \times S_6) + (C_3 \times S_5) + (C_4 \times S_4)}{2^9} \right\rfloor + \text{Residual}_4 \\ &= \left\lfloor \frac{(1241 \times 1688) + (-944 \times 1694) + (14 \times 1623) + (342 \times 1570) + (-147 \times 1529)}{512} \right\rfloor - 72 \\ &= \left\lfloor \frac{830571}{512} \right\rfloor - 72 = 1622 - 72 = \mathbf{1550} \end{aligned}$$

7.3.5 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:

$$\text{Total Samples} = \frac{\text{Frame's Block Size}}{2^{\text{Partition Order}}} - \text{Predictor Order} \quad (7.1)$$

Subsequent partitions contain:

$$\text{Total Samples} = \frac{\text{Frame's Block Size}}{2^{\text{Partition Order}}} \quad (7.2)$$

Unless the Partition Order is 0. In that case:

$$\text{Total Samples} = \text{Frame's Block Size} - \text{Predictor Order} \quad (7.3)$$

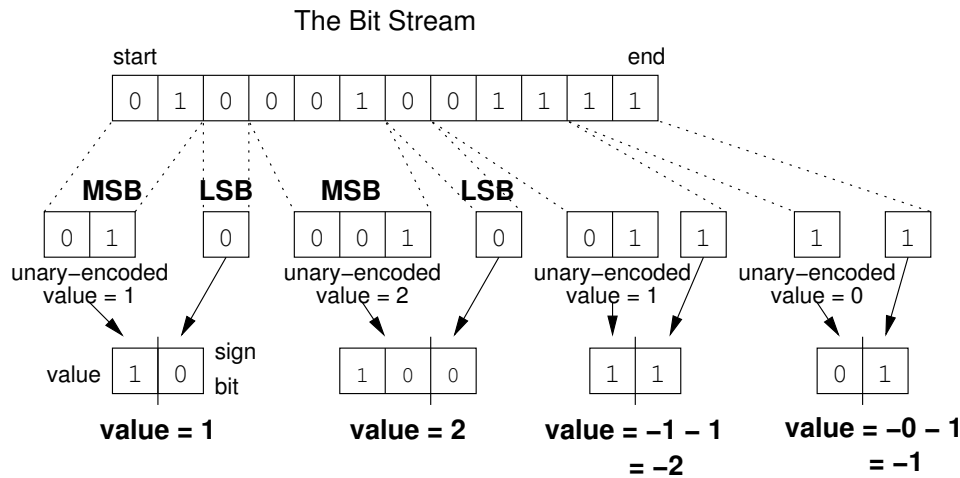
since there is only one partition which takes up the entire block.

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

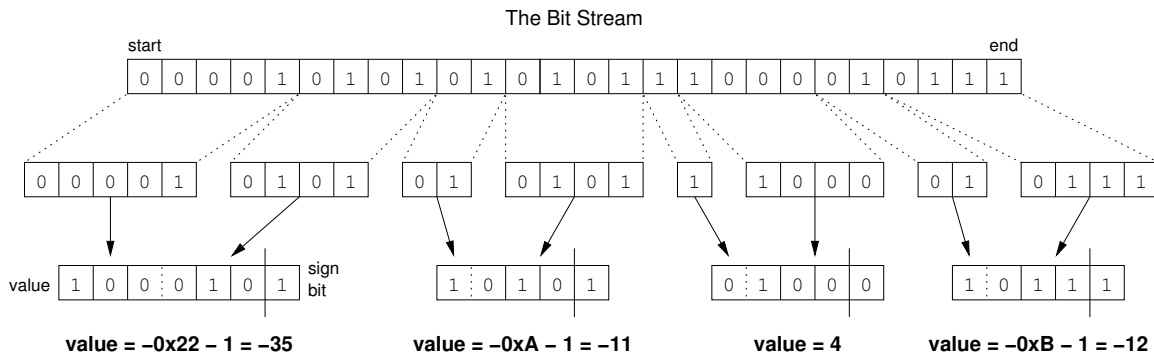
Rice Encoding

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 value² 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:



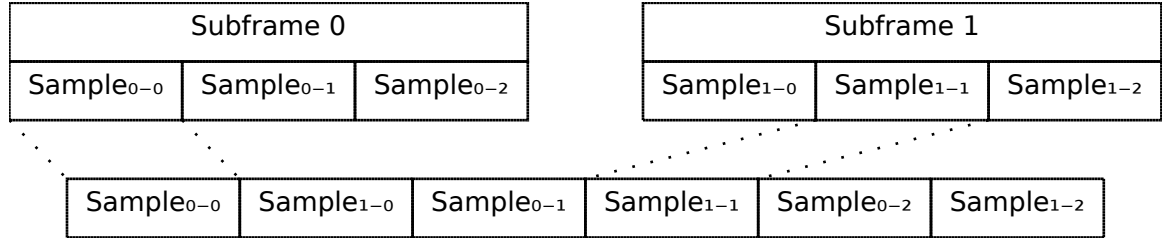
Now, let’s run through another example in which the Rice parameter is 4:



²In 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.

7.3.6 Channel Assignment

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 or mid-side channels, one subframe will contain actual channel data and the other channel will contain signed difference data which is applied to that actual data in order to reconstruct both channels. It's very important to remember that the difference (and side) 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.

Performing the actual channel decorrelation is as follows:

Left-Difference (assignment 0x8):

$$\begin{aligned}\text{Left}_i &= \text{Subframe}_{0i} \\ \text{Right}_i &= \text{Subframe}_{0i} - \text{Subframe}_{1i}\end{aligned}$$

Difference-Right (assignment 0x9):

$$\begin{aligned}\text{Left}_i &= \text{Subframe}_{0i} + \text{Subframe}_{1i} \\ \text{Right}_i &= \text{Subframe}_{1i}\end{aligned}$$

Mid-Side (assignment 0xA):

$$\begin{aligned}\text{Left}_i &= \lfloor (((\text{Subframe}_{0i} \times 2) + (\text{Subframe}_{1i} \bmod 2)) + \text{Subframe}_{1i}) \div 2 \rfloor \\ \text{Right}_i &= \lfloor (((\text{Subframe}_{0i} \times 2) + (\text{Subframe}_{1i} \bmod 2)) - \text{Subframe}_{1i}) \div 2 \rfloor\end{aligned}$$

Let's take an example of two Mid-Side encoded samples: 1533 and 3039 for Subframe₀ and Subframe₁, respectively:

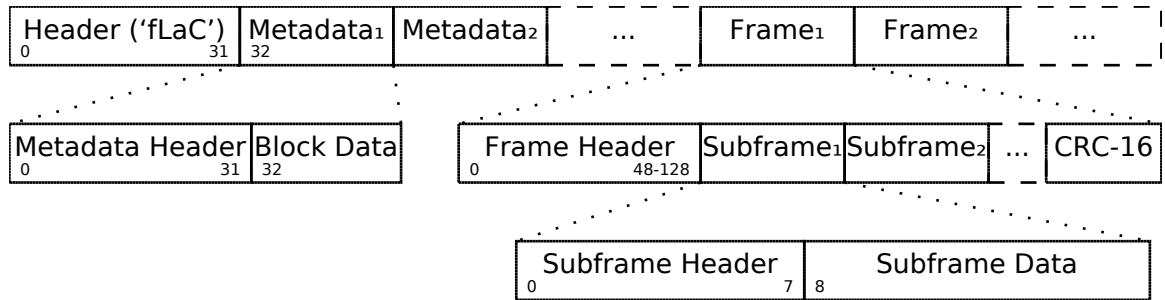
$$\begin{aligned}
 \text{Left} &= \lfloor (((1533 \times 2) + (3039 \bmod 2)) + 3039) \div 2 \rfloor \\
 &= \lfloor ((3066 + 1) + 3039) \div 2 \rfloor \\
 &= \lfloor 6106 \div 2 \rfloor = \mathbf{3053} \\
 \text{Right} &= \lfloor (((1533 \times 2) + (3039 \bmod 2)) - 3039) \div 2 \rfloor \\
 &= \lfloor ((3066 + 1) - 3039) \div 2 \rfloor \\
 &= \lfloor 28 \div 2 \rfloor = \mathbf{14}
 \end{aligned}$$

7.3.7 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.

7.4 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.



bits	value
1	0 if additional 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

Figure 7.1: Metadata Header

7.4.1 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

This metadata block must come first and is the only required block in a FLAC file.

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.

7.4.2 the 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, 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.

7.4.3 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 and use the one which takes the least amount of space.

Left-Difference (assignment 0x8):

$$\begin{aligned}\text{Subframe}_{0i} &= \text{Left}_i \\ \text{Subframe}_{1i} &= \text{Left}_i - \text{Right}_i\end{aligned}$$

Difference-Right (assignment 0x9):

$$\begin{aligned}\text{Subframe}_{0i} &= \text{Left}_i - \text{Right}_i \\ \text{Subframe}_{1i} &= \text{Right}_i\end{aligned}$$

Mid-Side (assignment 0xA):

$$\begin{aligned}\text{Subframe}_{0i} &= \lfloor (\text{Left}_i + \text{Right}_i) \div 2 \rfloor \\ \text{Subframe}_{1i} &= \text{Left}_i - \text{Right}_i\end{aligned}$$

Remember that the difference and side channels are treated as having 1 additional bit during encoding. Thus, a 16 bit frame would have a 17 bit difference subframe when calculating warm-up samples.

7.4.4 the Subframe Header

bits	value
1	0 padding
000000	SUBFRAME_CONSTANT
000001	SUBFRAME_VERBATIM
001xxx	SUBFRAME_FIXED (xxx = Predictor Order)
1xxxxx	SUBFRAME_LPC (xxxxx = Predictor Order - 1)
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

7.4.5 the CONSTANT Subframe

If all the samples in a subframe are identical, one can encode them using a **CONSTANT** subframe.

7.4.6 the VERBATIM Subframe

This subframe simply stores all the samples as-is, with no compression whatsoever.

7.4.7 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>
<i> sum </i>		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.

7 Free Lossless Audio Codec

The residual encoding process is then the simple inverse of the decoding process, as follows:

For Order 0:

$$\begin{aligned} \text{Residual}_i &= \text{Sample}_i \\ &\text{for } i = 0 \text{ to Block Size} - 1 \end{aligned}$$

For Order 1:

$$\begin{aligned} \text{Warm Up}_0 &= \text{Sample}_0 \\ \text{Residual}_i &= \text{Sample}_{i+1} - \text{Sample}_i \\ &\text{for } i = 0 \text{ to Block Size} - 2 \end{aligned}$$

For Order 2:

$$\begin{aligned} \text{Warm Up}_j &= \text{Sample}_j \\ &\text{for } j = 0 \text{ to } 1 \\ \text{Residual}_i &= \text{Sample}_{i+2} - ((2 \times \text{Sample}_{i+1}) - \text{Sample}_i) \\ &\text{for } i = 0 \text{ to Block Size} - 3 \end{aligned}$$

For Order 3:

$$\begin{aligned} \text{Warm Up}_j &= \text{Sample}_j \\ &\text{for } j = 0 \text{ to } 2 \\ \text{Residual}_i &= \text{Sample}_{i+3} - ((3 \times \text{Sample}_{i+2}) - (3 \times \text{Sample}_{i+1}) + \text{Sample}_i) \\ &\text{for } i = 0 \text{ to Block Size} - 4 \end{aligned}$$

For Order 4:

$$\begin{aligned} \text{Warm Up}_j &= \text{Sample}_j \\ &\text{for } j = 0 \text{ to } 3 \\ \text{Residual}_i &= \text{Sample}_{i+4} - ((4 \times \text{Sample}_{i+3}) - (6 \times \text{Sample}_{i+2}) + (4 \times \text{Sample}_{i+1}) - \text{Sample}_i) \\ &\text{for } i = 0 \text{ to Block Size} - 5 \end{aligned}$$

So to complete our FIXED subframe encoding example in which the predictor order is 1:

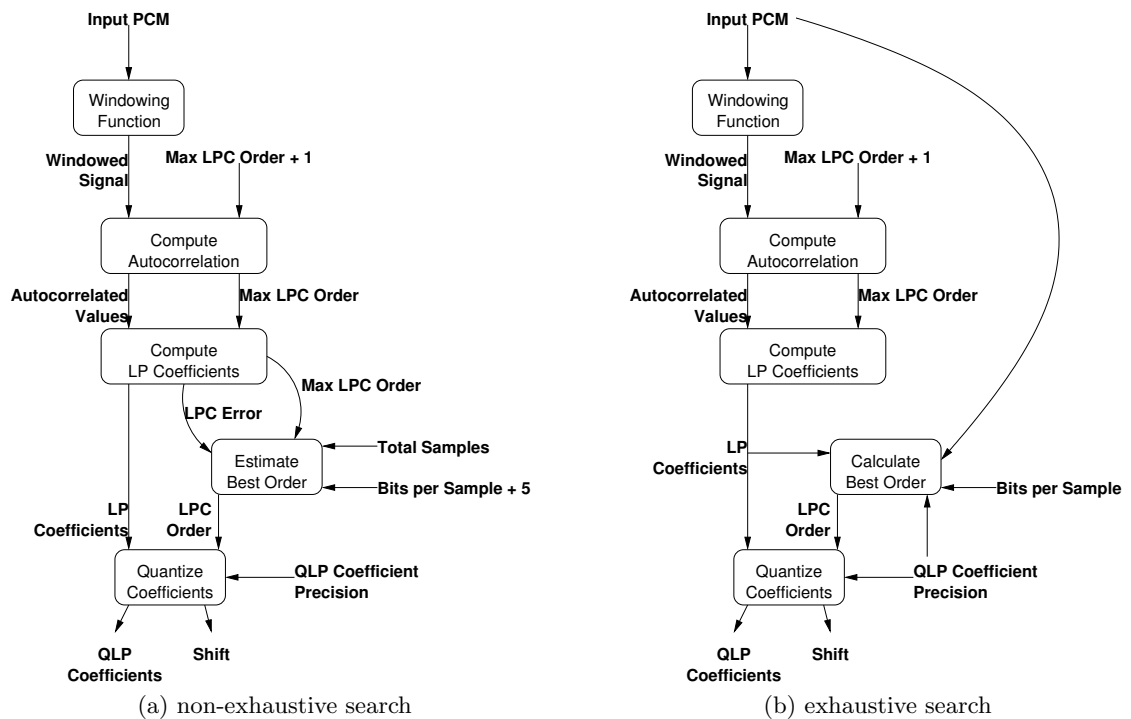
$$\begin{aligned}
\text{Warm Up}_0 &= \text{Sample}_0 = \mathbf{-40} \\
\text{Residual}_0 &= \text{Sample}_1 - \text{Sample}_0 = -41 - -40 = \mathbf{-1} \\
\text{Residual}_1 &= \text{Sample}_2 - \text{Sample}_1 = -40 - -41 = \mathbf{1} \\
\text{Residual}_2 &= \text{Sample}_3 - \text{Sample}_2 = -39 - -40 = \mathbf{1} \\
\text{Residual}_3 &= \text{Sample}_4 - \text{Sample}_3 = -38 - -39 = \mathbf{1} \\
\text{Residual}_4 &= \text{Sample}_5 - \text{Sample}_4 = -38 - -38 = \mathbf{0} \\
\text{Residual}_5 &= \text{Sample}_6 - \text{Sample}_5 = -35 - -38 = \mathbf{3} \\
\text{Residual}_6 &= \text{Sample}_7 - \text{Sample}_6 = -35 - -35 = \mathbf{0} \\
\text{Residual}_7 &= \text{Sample}_8 - \text{Sample}_7 = -39 - -35 = \mathbf{-4} \\
\text{Residual}_8 &= \text{Sample}_9 - \text{Sample}_8 = -40 - -39 = \mathbf{-1} \\
\text{Residual}_9 &= \text{Sample}_{10} - \text{Sample}_9 = -40 - -40 = \mathbf{0} \\
\text{Residual}_{10} &= \text{Sample}_{11} - \text{Sample}_{10} = -39 - -40 = \mathbf{1} \\
\text{Residual}_{11} &= \text{Sample}_{12} - \text{Sample}_{11} = -38 - -39 = \mathbf{1} \\
\text{Residual}_{12} &= \text{Sample}_{13} - \text{Sample}_{12} = -37 - -38 = \mathbf{1} \\
\text{Residual}_{13} &= \text{Sample}_{14} - \text{Sample}_{13} = -33 - -37 = \mathbf{4} \\
\text{Residual}_{14} &= \text{Sample}_{15} - \text{Sample}_{14} = -36 - -33 = \mathbf{-3} \\
\text{Residual}_{15} &= \text{Sample}_{16} - \text{Sample}_{15} = -35 - -36 = \mathbf{1} \\
\text{Residual}_{16} &= \text{Sample}_{17} - \text{Sample}_{16} = -31 - -35 = \mathbf{4} \\
\text{Residual}_{17} &= \text{Sample}_{18} - \text{Sample}_{17} = -32 - -31 = \mathbf{-1} \\
\text{Residual}_{18} &= \text{Sample}_{19} - \text{Sample}_{18} = -33 - -32 = \mathbf{-1}
\end{aligned}$$

7.4.8 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 _x
QLP Precision ₀ ₃	QLP Shift Needed ₄ ₈	QLP Coefficient ₁ ₉	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:



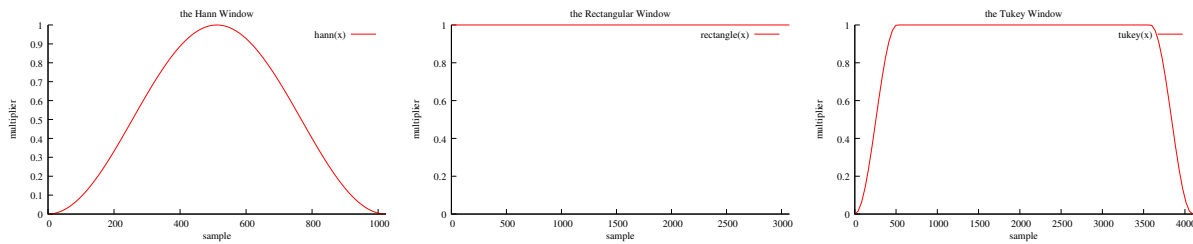
Windowing

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.

$$\text{hann}(n) = \frac{1}{2} \left(1 - \cos \left(\frac{2\pi n}{\text{sample count} - 1} \right) \right) \quad (7.4)$$

$$\text{rectangle}(n) = 1.0 \quad (7.5)$$

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

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.

-39.0	-38.0	-37.0	-37.0	-33.0	-36.0	-36.0	-36.0	-35.0	-31.0	-32.0	-33.0	windowed signal	
x	x	x	x	x	x	x	x	x	x	x	x		lag 0 sum = 14979.0
-39.0	-38.0	-37.0	-37.0	-33.0	-36.0	-36.0	-36.0	-35.0	-31.0	-32.0	-33.0	lag 0 signal	
-39.0	-38.0	-37.0	-37.0	-33.0	-36.0	-36.0	-36.0	-35.0	-31.0	-32.0	-33.0	windowed signal	
x	x	x	x	x	x	x	x	x	x	x	x		lag 1 sum = 13651.0
-38.0	-37.0	-37.0	-33.0	-36.0	-36.0	-36.0	-35.0	-31.0	-32.0	-33.0	-33.0	lag 1 signal	
-39.0	-38.0	-37.0	-37.0	-33.0	-36.0	-36.0	-36.0	-35.0	-31.0	-32.0	-33.0	windowed signal	
x	x	x	x	x	x	x	x	x	x	x	x		lag 2 sum = 12405.0
-37.0	-37.0	-33.0	-36.0	-36.0	-36.0	-35.0	-31.0	-32.0	-33.0	-33.0	-33.0	lag 2 signal	

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.

LP Coefficient Calculation

Calculating the LP coefficients uses the Levinson-Durbin recursive method.³ 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:

$$E_0 = r(0) \quad (7.6)$$

$$a_{11} = \kappa_1 = \frac{r(1)}{E_0} \quad (7.7)$$

$$E_1 = E_0(1 - \kappa_1^2) \quad (7.8)$$

³This algorithm is taken from <http://www.engineer.tamuk.edu/SPark/chap7.pdf>

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

$$\text{Step 1.} \quad q_m = r(m) - \sum_{i=1}^{m-1} a_{i(m-1)} r(m-i) \quad (7.9)$$

$$\text{Step 2.} \quad \kappa_m = \frac{q_m}{E_{(m-1)}} \quad (7.10)$$

$$\text{Step 3.} \quad a_{mm} = \kappa_m \quad (7.11)$$

$$\text{Step 4.} \quad a_{im} = a_{i(m-1)} - \kappa_m a_{(m-i)(m-1)} \text{ for } i = 1, i = 2, \dots, i = m-1 \quad (7.12)$$

$$\text{Step 5.} \quad E_m = E_{m-1}(1 - \kappa_m^2) \quad (7.13)$$

$$\text{Step 6.} \quad \text{If } m < M \text{ then } m \leftarrow m + 1 \text{ and goto step 1. If } m = M \text{ then stop.} \quad (7.14)$$

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:

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

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

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:

$$\text{Error Scale} = \frac{\ln(2)^2}{2 \times \text{Total Samples}} \quad (7.15)$$

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:

$$\text{Bits per Residual(LPC Error)} = \frac{\ln(\text{Error Scale} \times \text{LPC Error})}{2 \times \ln(2)} \quad (7.16)$$

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

$$\text{Total Bits(LPC Error, Order)} = \text{Bits per Residual(LPC Error)} \times (\text{Total Samples} - \text{Order}) + (\text{Order} \times \text{Overhead bits})$$

Continuing with our example, we have 20 samples and now have the error values of 2495, 2492 and 2490. This gives us an error scale of: $\frac{\ln(2)^2}{2 \times 20} = \frac{.6931^2}{40} = .01201$

At LPC order 1, our bits per residual are:

$$\frac{\ln(.01201 \times 2495)}{2 \times \ln(2)} = \frac{\ln(29.96)}{1.386} = 2.453$$

And our total bits are:

$$(2.453 \times (20 - 1)) + (1 \times (16 + 5)) = 46.61 + 21 = 67.61$$

At LPC order 2, our bits per residual are:

$$\frac{\ln(.01201 \times 2492)}{2 \times \ln(2)} = \frac{\ln(29.92)}{1.386} = 2.452$$

And our total bits are:

$$(2.452 \times (20 - 2)) + (2 \times (16 + 5)) = 44.14 + 42 = 86.14$$

At LPC order 3, our bits per residual are:

$$\frac{\ln(.01201 \times 2490)}{2 \times \ln(2)} = \frac{\ln(29.90)}{1.386} = 2.451$$

And our total bits are:

$$(2.451 \times (20 - 3)) + (3 \times (16 + 5)) = 41.67 + 63 = 104.7$$

Therefore, since the total bits for order 1 are the smallest, the best order for this group of samples is 1.

Though as you'll notice, the bits per residual for order 3 were the smallest. So if this group of samples was very large, it's likely that order 3 would prevail since the residuals multiplied by a smaller bits per residual would counteract the relatively fixed overhead bits per order value.

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,⁴ 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.

⁴Quantizing coefficients will be covered in the next section.

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:

$$\text{QLP coefficient maximum} = 2^{\text{precision}-1} - 1 \quad (7.17)$$

$$\text{QLP coefficient minimum} = -2^{\text{precision}-1} \quad (7.18)$$

The QLP Coefficients Precision value is typically based on the encoder's block size:

Block Size	Precision	Block Size	Precision
Size ≤ 192	7	Size ≤ 384	8
Size ≤ 576	9	Size ≤ 1152	10
Size ≤ 2304	11	Size ≤ 4608	12
Size > 4608	13		

So in our example of a block of 20 samples,

$$\text{QLP Coefficient maximum} = 2^{7-1} - 1 = 64 - 1 = 63$$

$$\text{QLP Coefficient minimum} = -2^{7-1} = -64$$

Now we determine the initial QLP Shift Needed value:

$$\text{shift} = \text{precision} - \left\lceil \frac{\log(\max(|\text{LP Coefficients}|))}{\log(2)} \right\rceil - 1 \quad (7.19)$$

where 'shift' is adjusted if necessary such that: $0 \leq \text{shift} \leq 0\text{x}\mathbf{F}$, since it must fit into a 5-bit signed field and negative shifts are no-ops in the FLAC decoder.

Continuing our ongoing example, let's assume we're quantizing the LP coefficients 0.9065, -0.0052 and -0.0293. So our shift should be:

$$\text{shift} = 7 - \left\lceil \frac{\log(0.9065)}{\log(2)} \right\rceil - 1 = 7 - \left\lceil \frac{-0.0981}{0.6931} \right\rceil - 1 = 7 - 0 - 1 = 6$$

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

$$X(i) = E(i-1) + (\text{LP Coefficient}_i \times 2^{\text{shift}}) \quad (7.20)$$

$$\text{QLP Coefficient}_i = \text{round}(X(i)) \quad (7.21)$$

$$E(i) = X(i) - \text{QLP Coefficient}_i \quad (7.22)$$

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

So to finish our LPC example:

$$X(1) = E(0) + (0.9065 \times 2^6) = 0 + 58.016 = \mathbf{58.016}$$

$$\text{QLP Coefficient}_1 = \text{round}(58.016) = \mathbf{58}$$

$$E(1) = X(1) - \text{QLP Coefficient}_1 = 58.016 - 58 = \mathbf{0.016}$$

$$X(2) = E(1) + (-0.0052 \times 2^6) = 0.016 + -0.3328 = \mathbf{0.3168}$$

$$\text{QLP Coefficient}_2 = \text{round}(0.3168) = \mathbf{0}$$

$$E(2) = X(2) - \text{QLP Coefficient}_2 = 0.3168 - 0 = \mathbf{0.3168}$$

$$X(3) = E(2) + (-0.0293 \times 2^6) = 0.3168 + -1.875 = \mathbf{-1.558}$$

$$\text{QLP Coefficient}_3 = \text{round}(-1.558) = \mathbf{-2}$$

$$E(3) = X(3) - \text{QLP Coefficient}_3 = -1.558 - -2 = \mathbf{0.4420}$$

Therefore, the LPC order is 3. The QLP Coefficients are 58, 0 and -2. The QLP Shift Needed value is 6. And, the QLP precision value can be calculated from the bits required for the largest absolute QLP Coefficient value. In this case, 6 bits are required to hold the value 58 so QLP precision can be 6.

Calculating LPC Residual

Warm-up samples and residuals are calculated as follows:

For $i = 0$ to $Order - 1$:

$$\text{Warm Up}_i = \text{Sample}_i$$

For $i = Order$ to Block Size - 1:

$$\text{Residual}_{i-Order} = \text{Sample}_i - \left\lfloor \frac{\sum_{j=0}^{Order-1} \text{QLP Coefficient}_j \times \text{Sample}_{i-j-1}}{2^{\text{QLP Shift Needed}}} \right\rfloor$$

Therefore, if ‘LPC Order’ is 3, ‘QLP Shift Needed’ is 6, and we have the following values:

Sample ₀	-37	LPC Coefficient ₀	58
Sample ₁	-33	LPC Coefficient ₁	0
Sample ₂	-36	LPC Coefficient ₂	-2
Sample ₃	-35		
Sample ₄	-31		
Sample ₅	-32		
Sample ₆	-33		

$$\text{Warm Up}_0 = \text{Sample}_0 = \mathbf{-37}$$

$$\text{Warm Up}_1 = \text{Sample}_1 = \mathbf{-33}$$

$$\text{Warm Up}_2 = \text{Sample}_2 = \mathbf{-36}$$

$$\begin{aligned} \text{Residual}_0 &= \text{Sample}_3 - \left\lfloor \frac{(C_0 \times S_2) + (C_1 \times S_1) + (C_2 \times S_2)}{2^6} \right\rfloor \\ &= -35 - \left\lfloor \frac{(58 \times -36) + (0 \times -33) + (-2 \times -37)}{64} \right\rfloor \\ &= -35 - \left\lfloor \frac{-2014}{64} \right\rfloor = -35 - -32 = \mathbf{-3} \end{aligned}$$

$$\begin{aligned} \text{Residual}_1 &= \text{Sample}_4 - \left\lfloor \frac{(C_0 \times S_3) + (C_1 \times S_2) + (C_2 \times S_1)}{2^6} \right\rfloor \\ &= -31 - \left\lfloor \frac{(58 \times -35) + (0 \times -36) + (-2 \times -33)}{64} \right\rfloor \\ &= -31 - \left\lfloor \frac{-1964}{64} \right\rfloor = -31 - -31 = \mathbf{0} \end{aligned}$$

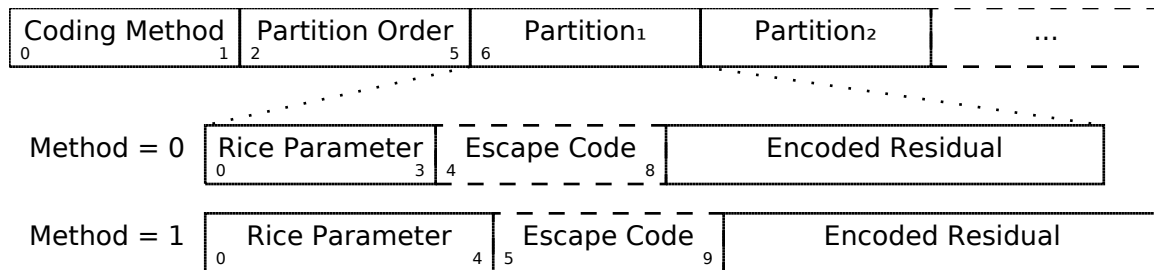
$$\begin{aligned} \text{Residual}_2 &= \text{Sample}_5 - \left\lfloor \frac{(C_0 \times S_4) + (C_1 \times S_3) + (C_2 \times S_2)}{2^6} \right\rfloor \\ &= -32 - \left\lfloor \frac{(58 \times -31) + (0 \times -35) + (-2 \times -36)}{64} \right\rfloor \\ &= -32 - \left\lfloor \frac{-1726}{64} \right\rfloor = -32 - -27 = \mathbf{-5} \end{aligned}$$

$$\begin{aligned} \text{Residual}_3 &= \text{Sample}_6 - \left\lfloor \frac{(C_0 \times S_5) + (C_1 \times S_4) + (C_2 \times S_3)}{2^6} \right\rfloor \\ &= -33 - \left\lfloor \frac{(58 \times -32) + (0 \times -31) + (-2 \times -35)}{64} \right\rfloor \\ &= -33 - \left\lfloor \frac{-1786}{64} \right\rfloor = -33 - -28 = \mathbf{-5} \end{aligned}$$

Therefore, our warm-up samples are -37, -33 and -36; and our residual values are -3, 0, -5 and -5.

7.4.9 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:

$$\text{sample count} \times 2^x > \sum_{i=0}^{\text{residual count}-1} |\text{residual}_i| \quad (7.23)$$

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

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.

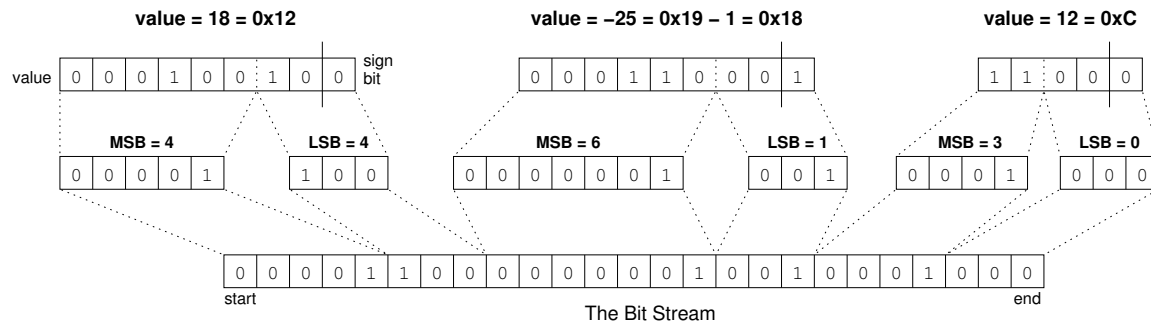
Remember that the Rice parameter's maximum value is limited to 2^4 using coding method 0, or 2^5 using coding method 1.

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

Residual Values

Encoding individual residual values to Rice coding requires only the Rice parameter and the values themselves. First, one must convert any negative values to positive by multiplying it by -1, subtracting 1 and prepending a 1 bit. If the value is already positive, prepend a 0 bit instead. Next, we split out new value into most significant bits (MSB) and least significant bits (LSB) where the length of the LSB is equal to the Rice parameter and MSB contains the remaining bits. The MSB value is written unary encoded, whereas the LSB is written as-is.

As with residual decoding, this process is not as difficult as it sounds and is best explained by an example, in this case using a parameter of 3 and encoding the residual values 18, -25 and 12:



7.4.10 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.

CRC-8

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:

$$\text{checksum}_i = \text{CRC8}(\text{byte} \mathbf{xor} \text{checksum}_{i-1}) \quad (7.24)$$

	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:

$$\begin{aligned} \text{checksum}_0 &= \text{CRC8}(0xFF \mathbf{xor} 0x00) = \text{CRC8}(0xFF) = 0xF3 \\ \text{checksum}_1 &= \text{CRC8}(0xF8 \mathbf{xor} 0xF3) = \text{CRC8}(0x0B) = 0x31 \\ \text{checksum}_2 &= \text{CRC8}(0xCC \mathbf{xor} 0x31) = \text{CRC8}(0xFD) = 0xFD \\ \text{checksum}_3 &= \text{CRC8}(0x1C \mathbf{xor} 0xFD) = \text{CRC8}(0xE1) = 0xA9 \\ \text{checksum}_4 &= \text{CRC8}(0x00 \mathbf{xor} 0xA9) = \text{CRC8}(0xA9) = 0x56 \\ \text{checksum}_5 &= \text{CRC8}(0xC0 \mathbf{xor} 0x56) = \text{CRC8}(0x96) = 0xEB \end{aligned}$$

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

$$\text{checksum}_6 = \text{CRC8}(0xEB \mathbf{xor} 0xEB) = \text{CRC8}(0x00) = 0x00$$

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.

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:

$$\text{checksum}_i = \text{CRC16}(\text{byte} \mathbf{xor} (\text{checksum}_{i-1} \gg 8)) \mathbf{xor} (\text{checksum}_{i-1} \ll 8) \quad (7.25)$$

and the checksum is always truncated to 16-bits.

	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?	0000	8005	800f	000a	801b	001e	0014	8011	8033	0036	003c	8039	0028	802d	8027	0022
0x1?	8063	0066	006c	8069	0078	807d	8077	0072	0050	8055	805f	005a	804b	004e	0044	8041
0x2?	80c3	00c6	00cc	80c9	00d8	80dd	80d7	00d2	00f0	80f5	80ff	00fa	80eb	00ee	00e4	80e1
0x3?	00a0	80a5	80af	00aa	80bb	00be	00b4	80b1	8093	0096	009c	8099	0088	808d	8087	0082
0x4?	8183	0186	018c	8189	0198	819d	8197	0192	01b0	81b5	81bf	01ba	81ab	01ae	01a4	81a1
0x5?	01e0	81e5	81ef	01ea	81fb	01fe	01f4	81f1	81d3	01d6	01dc	81d9	01c8	81cd	81c7	01c2
0x6?	0140	8145	814f	014a	815b	015e	0154	8151	8173	0176	017c	8179	0168	816d	8167	0162
0x7?	8123	0126	012c	8129	0138	813d	8137	0132	0110	8115	811f	011a	810b	010e	0104	8101
0x8?	8303	0306	030c	8309	0318	831d	8317	0312	0330	8335	833f	033a	832b	032e	0324	8321
0x9?	0360	8365	836f	036a	837b	037e	0374	8371	8353	0356	035c	8359	0348	834d	8347	0342
0xA?	03c0	83c5	83cf	03ca	83db	03de	03d4	83d1	83f3	03f6	03fc	83f9	03e8	83ed	83e7	03e2
0xB?	83a3	03a6	03ac	83a9	03b8	83bd	83b7	03b2	0390	8395	839f	039a	838b	038e	0384	8381
0xC?	0280	8285	828f	028a	829b	029e	0294	8291	82b3	02b6	02bc	82b9	02a8	82ad	82a7	02a2
0xD?	82e3	02e6	02ec	82e9	02f8	82fd	82f7	02f2	02d0	82d5	82df	02da	82cb	02ce	02c4	82c1
0xE?	8243	0246	024c	8249	0258	825d	8257	0252	0270	8275	827f	027a	826b	026e	0264	8261
0xF?	0220	8225	822f	022a	823b	023e	0234	8231	8213	0216	021c	8219	0208	820d	8207	0202

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:

```

checksum0 = CRC16(0xFF xor (0x0000 >> 8)) xor (0x0000 << 8) = CRC16(0xFF) xor 0x0000 = 0x0202
checksum1 = CRC16(0xF8 xor (0x0202 >> 8)) xor (0x0202 << 8) = CRC16(0xFA) xor 0x0200 = 0x001C
checksum2 = CRC16(0xCC xor (0x001C >> 8)) xor (0x001C << 8) = CRC16(0xCC) xor 0x1C00 = 0x1EA8
checksum3 = CRC16(0x1C xor (0x1EA8 >> 8)) xor (0x1EA8 << 8) = CRC16(0x02) xor 0xA800 = 0x280F
checksum4 = CRC16(0x00 xor (0x280F >> 8)) xor (0x280F << 8) = CRC16(0x28) xor 0x0F00 = 0x0FF0
checksum5 = CRC16(0xC0 xor (0x0FF0 >> 8)) xor (0x0FF0 << 8) = CRC16(0xCF) xor 0xF000 = 0xF2A2
checksum6 = CRC16(0xEB xor (0xF2A2 >> 8)) xor (0xF2A2 << 8) = CRC16(0x19) xor 0xA200 = 0x2255
checksum7 = CRC16(0x00 xor (0x2255 >> 8)) xor (0x2255 << 8) = CRC16(0x22) xor 0x5500 = 0x55CC
checksum8 = CRC16(0x00 xor (0x55CC >> 8)) xor (0x55CC << 8) = CRC16(0x55) xor 0xCC00 = 0xCDFE
checksum9 = CRC16(0x00 xor (0xCDFE >> 8)) xor (0xCDFE << 8) = CRC16(0xCD) xor 0xFE00 = 0x7CAD
checksum10 = CRC16(0x00 xor (0x7CAD >> 8)) xor (0x7CAD << 8) = CRC16(0x7C) xor 0xAD00 = 0x2C0B
checksum11 = CRC16(0x00 xor (0x2C0B >> 8)) xor (0x2C0B << 8) = CRC16(0x2C) xor 0x0B00 = 0x8BEB
checksum12 = CRC16(0x00 xor (0x8BEB >> 8)) xor (0x8BEB << 8) = CRC16(0x8B) xor 0xEB00 = 0xE83A
checksum13 = CRC16(0x00 xor (0xE83A >> 8)) xor (0xE83A << 8) = CRC16(0xE8) xor 0x3A00 = 0x3870
checksum14 = CRC16(0x00 xor (0x3870 >> 8)) xor (0x3870 << 8) = CRC16(0x38) 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:

```

checksum15 = CRC16(0xF0 xor (0xF093 >> 8)) xor (0xF093 << 8) = CRC16(0x00) xor 0x9300 = 0x9300
checksum16 = CRC16(0x93 xor (0x9300 >> 8)) xor (0x9300 << 8) = CRC16(0x00) xor 0x0000 = 0x0000

```

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

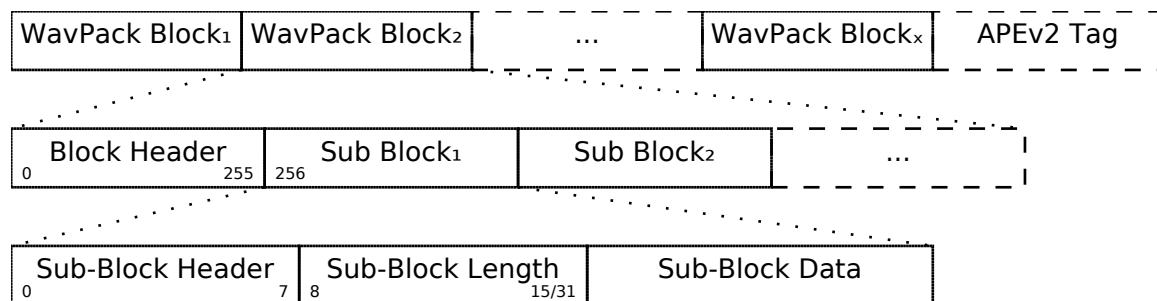
8 WavPack

WavPack is a format for compressing Wave files, typically in lossless mode. Notably, it also has a lossy mode and even a hybrid mode which allows the ‘correction’ file to be separated from a lossy core.

Metadata is stored as an APEv2 tag, which is described on page 110.

Its stream of data is stored little-endian, as described on page 15.

8.1 the WavPack File Stream



8.2 the WavPack Block Header

Block ID `wvpk' (0x6B707677)				Block Size			
0		31		32		63	
Version		Track Number		Index Number		Total Samples	
64	79	80	87	88	95	96	127
Block Index				Block Samples			
128		159		160		191	
Bits per Sample		Mono Output		Hybrid Mode		Joint Stereo	
192	193	194		195		196	
Channel Decorrelation		Hbd. Noise Shaping		Floating Point Data		Extended Size Integers	
197		198		199		200	
Hbd. Controls Bitrate		Hbd. Noise Balanced		Initial Block		Final Block	
201		202		203		204	
Left Shift Data		Maximum Magnitude		Sample Rate			
205	209	210	214	215			218
Reserved		Use IIR		False Stereo		Reserved	
219	220	221		222		223	
CRC							
224							255

‘Block Size’ is the length of everything in the block past the ‘Block Size’ field itself - or everything in the block past the CRC, minus 24 bytes.

‘Bits per Sample’ is one of 4 values:

00 = 8 bps, 01 = 16 bps, 10 = 24 bps, 11 = 32 bps .

‘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

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

8.2.1 WavPack Sub-Block

0	Metadata Function	4	Nondecoder Data	5	Actual Size 1 Less	6	Large Block	7
8	Block Size			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.

8.3 WavPack Decoding

Decoding each WavPack block requires reading its sub-blocks as ‘arguments’ to the decoder. One can envision them like named arguments to a function call since many sub-blocks may be optional or appear in an arbitrary order. As a sort of hypothetical high-level example:

```
decode_block(decorrelation_terms=sub_block[0],
             decorrelation_weights=sub_block[1],
             decorrelation_samples=sub_block[2],
             entropy_variables=sub_block[3],
             bitstream=sub_block[4])
```

Every block containing audio data requires ‘Entropy Variables’ and ‘Bitstream’ sub-blocks. The ‘Decorrelation Terms’, ‘Decorrelation Weights’ and ‘Decorrelation Samples’ sub-blocks are for performing one or more decorrelation passes over the bitstream’s samples.

Each block will decode to 1 or 2 channels of raw PCM output. Since files may have more than 2 channels, we may need to decode several blocks in order to retrieve all the channels of data so they can be properly combined.

8.3.1 False Stereo

If the ‘False Stereo’ bit is set in the block header, treat the block as mono for decoding purposes until just before the channel’s data is output.

8.3.2 the Decorrelation Terms Sub-Block

This block contains the decorrelation terms and deltas values. The quantity of those values indicates how many decorrelation passes we’ll be performing over the bitstream sub-block’s samples. One can presume this sub-block will occur prior to the ‘Decorrelation Weights’ and ‘Decorrelation Samples’ sub-blocks.

Metadata Function (2) 0 4	0 5	Actual Size 1 Less 6	Large Block 7	Block Size 8 15/31
...	Decorr. Term ₂ + 5 0 4	Decorr. Delta ₂ 5 7	Decorr. Term ₁ + 5 8 12	Decorr. Delta ₁ 13 15

The number of decorrelation terms and deltas equals the ‘Block Size’ times 2, and minus 1 if ‘Actual Size 1 Less’ is set. Each term and delta pair is 8 bits and stored in *reverse* order. In addition, one must subtract 5 from the stored value of each unsigned term to get its actual value.

For example, given the complete sub-block bytes:

42 03 57 57 47 56 48 00

we have a total of 5 term/delta pairs whose values are as follows:

Decorrelation Term ₅	0x17 - 5 = 18	Decorrelation Delta ₅	0x2 = 2
Decorrelation Term ₄	0x17 - 5 = 18	Decorrelation Delta ₄	0x2 = 2
Decorrelation Term ₃	0x07 - 5 = 2	Decorrelation Delta ₃	0x2 = 2
Decorrelation Term ₂	0x16 - 5 = 17	Decorrelation Delta ₂	0x2 = 2
Decorrelation Term ₁	0x08 - 5 = 3	Decorrelation Delta ₁	0x2 = 2

Remember that this is a little-endian stream and that the least-significant bits (the ‘Decorrelation Delta’ value) are on the left side of each byte.

8.3.3 the Decorrelation Weights Sub-Block

0	Metadata Function (3)	4	0	5	Actual Size 1 Less	6	Large Block	7	Block Size	8	15/31
	...		0		Decorrelation Weight ₂	7		8	Decorrelation Weight ₁		15

As with the decorrelations terms sub-block, the decorrelation weights are stored in reverse order. The number of weights stored can be determined from the sub-block's size. Each is stored in a signed, 8-bit field and interleaved between channels in the case of 2 channel blocks. For example, Decorrelation Weight₁ is for channel 'B', Decorrelation Weight₂ is for channel 'A'¹, Decorrelation Weight₃ is for channel 'B' and so on such that the first weight value in the sub-block will be for the highest 'Decorrelation Weight A'. Converting the 8-bit values to the actual decorrelation weights requires the following formula:

$$\text{Decorrelation Weight} = \begin{cases} \text{value} \times 2^3 + \left\lfloor \frac{\text{value} \times 2^3 + 2^6}{2^7} \right\rfloor & \text{if value} > 0 \\ 0 & \text{if value} = 0 \\ \text{value} \times 2^3 & \text{if value} < 0 \end{cases}$$

For example, given a 2 channel block with 5 decorrelation terms and the sub-block bytes:

03 05 06 06 06 06 04 04 06 06 02 03

our 'Decorrelation Weights' are as follows:

$$\begin{array}{ll} \text{Weight A}_5 & 6 \times 2^3 + \left\lfloor \frac{6 \times 2^3 + 2^6}{2^7} \right\rfloor = 48 \\ \text{Weight A}_4 & 6 \times 2^3 + \left\lfloor \frac{6 \times 2^3 + 2^6}{2^7} \right\rfloor = 48 \\ \text{Weight A}_3 & 4 \times 2^3 + \left\lfloor \frac{4 \times 2^3 + 2^6}{2^7} \right\rfloor = 32 \\ \text{Weight A}_2 & 6 \times 2^3 + \left\lfloor \frac{6 \times 2^3 + 2^6}{2^7} \right\rfloor = 48 \\ \text{Weight A}_1 & 2 \times 2^3 + \left\lfloor \frac{2 \times 2^3 + 2^6}{2^7} \right\rfloor = 16 \\ \text{Weight B}_5 & 6 \times 2^3 + \left\lfloor \frac{6 \times 2^3 + 2^6}{2^7} \right\rfloor = 48 \\ \text{Weight B}_4 & 6 \times 2^3 + \left\lfloor \frac{6 \times 2^3 + 2^6}{2^7} \right\rfloor = 48 \\ \text{Weight B}_3 & 4 \times 2^3 + \left\lfloor \frac{4 \times 2^3 + 2^6}{2^7} \right\rfloor = 32 \\ \text{Weight B}_2 & 6 \times 2^3 + \left\lfloor \frac{6 \times 2^3 + 2^6}{2^7} \right\rfloor = 48 \\ \text{Weight B}_1 & 3 \times 2^3 + \left\lfloor \frac{3 \times 2^3 + 2^6}{2^7} \right\rfloor = 24 \end{array}$$

Note that decoding a WavPack file requires having the same number of 'Decorrelation Weight' values, per channel, as 'Decorrelation Terms' values. However, this block may contain less. In that event, those low weight values are set to 0.

¹Why 'A' and 'B'? Since a block may be only two channels out of many, it makes sense not to number them to avoid ambiguity.

8.3.4 the Decorrelation Samples Sub-Block

0	Metadata Function (4)	4	0	Actual Size 1 Less	6	Large Block	7	Block Size	8	15/31
0	Decorrelation Sample ₁	15	16	Decorrelation Sample ₂	31	...				

The decorrelation samples values are stored as signed, 16-bit values. Converting them to sample values requires the following formula:

$$\text{Sample} = \begin{cases} \lfloor \text{wv_exp2}(\text{value} \bmod 256) \div 2^{9-\lfloor \text{value} \div 2^8 \rfloor} \rfloor & \text{if } 0 \leq \text{value} \leq 2304 \\ \text{wv_exp2}(\text{value} \bmod 256) \times 2^{\lfloor \text{value} \div 2^8 \rfloor - 9} & \text{if } 2304 < \text{value} \leq 32767 \\ -\lfloor \text{wv_exp2}(-\text{value} \bmod 256) \div 2^{9-\lfloor -\text{value} \div 2^8 \rfloor} \rfloor & \text{if } -2304 \leq \text{value} < 0 \\ -(\text{wv_exp2}(-\text{value} \bmod 256) \times 2^{\lfloor -\text{value} \div 2^8 \rfloor - 9}) & \text{if } -32768 \leq \text{value} < -2304 \end{cases}$$

where ‘wv_exp2’ is defined from the following base-16 table:

	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?	100	101	101	102	103	103	104	105	106	106	107	108	108	109	10A	10B
0x1?	10B	10C	10D	10E	10E	10F	110	110	111	112	113	113	114	115	116	116
0x2?	117	118	119	119	11A	11B	11C	11D	11D	11E	11F	120	120	121	122	123
0x3?	124	124	125	126	127	128	128	129	12A	12B	12C	12C	12D	12E	12F	130
0x4?	130	131	132	133	134	135	135	136	137	138	139	13A	13A	13B	13C	13D
0x5?	13E	13F	140	141	141	142	143	144	145	146	147	148	148	149	14A	14B
0x6?	14C	14D	14E	14F	150	151	151	152	153	154	155	156	157	158	159	15A
0x7?	15B	15C	15D	15E	15E	15F	160	161	162	163	164	165	166	167	168	169
0x8?	16A	16B	16C	16D	16E	16F	170	171	172	173	174	175	176	177	178	179
0x9?	17A	17B	17C	17D	17E	17F	180	181	182	183	184	185	187	188	189	18A
0xA?	18B	18C	18D	18E	18F	190	191	192	193	195	196	197	198	199	19A	19B
0xB?	19C	19D	19F	1A0	1A1	1A2	1A3	1A4	1A5	1A6	1A8	1A9	1AA	1AB	1AC	1AD
0xC?	1AF	1B0	1B1	1B2	1B3	1B4	1B6	1B7	1B8	1B9	1BA	1BC	1BD	1BE	1BF	1C0
0xD?	1C2	1C3	1C4	1C5	1C6	1C8	1C9	1CA	1CB	1CD	1CE	1CF	1D0	1D2	1D3	1D4
0xE?	1D6	1D7	1D8	1D9	1DB	1DC	1DD	1DE	1E0	1E1	1E2	1E4	1E5	1E6	1E8	1E9
0xF?	1EA	1EC	1ED	1EE	1F0	1F1	1F2	1F4	1F5	1F6	1F8	1F9	1FA	1FC	1FD	1FF

For example, given the sub-frame bytes:

04 04 CF F8 B7 F8 CF 05 B3 05

our ‘Decorrelation Sample’ values are:

$$\begin{aligned} \text{Sample}_1 &= 0xF8CF = -1841 = -\lfloor \text{wv_exp2}(1841 \bmod 256) \div 2^{9-\lfloor 1841 \div 2^8 \rfloor} \rfloor \\ &= -\lfloor \text{wv_exp2}(49) \div 2^{9-7} \rfloor = -\lfloor 292 \div 4 \rfloor = \mathbf{-73} \end{aligned}$$

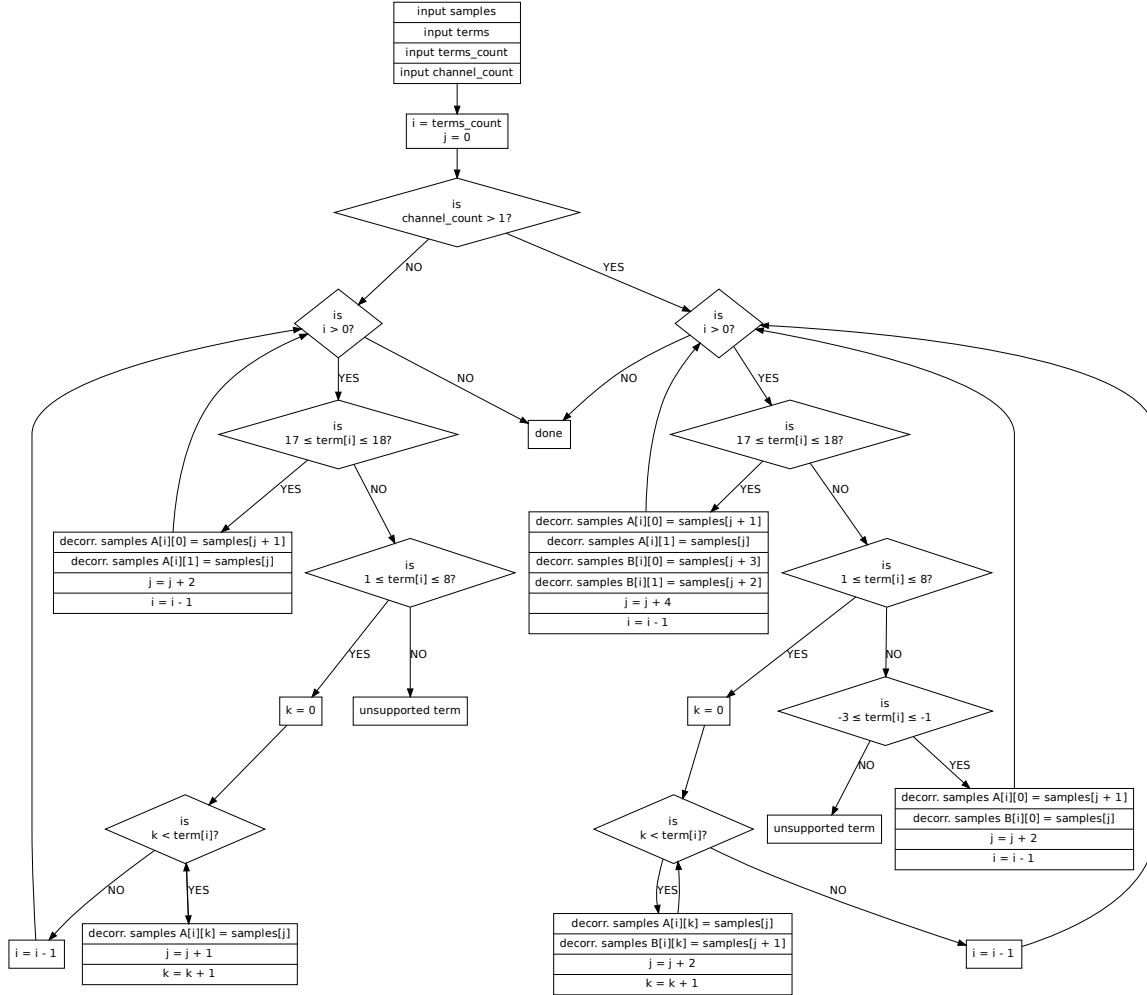
$$\begin{aligned} \text{Sample}_2 &= 0xF8B7 = -1865 = -\lfloor \text{wv_exp2}(1865 \bmod 256) \div 2^{9-\lfloor 1865 \div 2^8 \rfloor} \rfloor \\ &= -\lfloor \text{wv_exp2}(73) \div 2^{9-7} \rfloor = -\lfloor 312 \div 4 \rfloor = \mathbf{-78} \end{aligned}$$

$$\begin{aligned} \text{Sample}_3 &= 0x05CF = 1487 = \lfloor \text{wv_exp2}(1487 \bmod 256) \div 2^{9-\lfloor 1487 \div 2^8 \rfloor} \rfloor \\ &= \lfloor \text{wv_exp2}(207) \div 2^{9-5} \rfloor = \lfloor 448 \div 16 \rfloor = \mathbf{28} \end{aligned}$$

$$\begin{aligned} \text{Sample}_4 &= 0x05B3 = 1459 = \lfloor \text{wv_exp2}(1459 \bmod 256) \div 2^{9-\lfloor 1459 \div 2^8 \rfloor} \rfloor \\ &= \lfloor \text{wv_exp2}(179) \div 2^{9-5} \rfloor = \lfloor 416 \div 16 \rfloor = \mathbf{26} \end{aligned}$$

We're not done yet, however. The next step is to determine which set of 'Decorrelation Sample' values correspond to which 'Decorrelation Term'².

As with 'Decorrelation Weights', 'Decorrelation Sample' values are stored in reverse order, alternate between channels and depend on the corresponding 'Decorrelation Term' values.



It's likely that the decorrelation sample assignment process will request more samples than this sub-block contains. In that event, treat those samples as 0.

²As extracted on page 72

8.3.5 the Entropy Variables Sub-Block

Whereas the three preceding sub-blocks are for performing decorrelation passes, this sub-block is required for decoding the ‘Bitstream’ sub-block’s data. These entropy variables are median values which are stored as fractions of integers.

0	Metadata Function (5)	4	0	Actual Size 1 Less	6	Large Block	7	Block Size	15/31
0	Entropy Variable A ₁	15	16	Entropy Variable A ₂	31	32	Entropy Variable A ₃	47	
48	Entropy Variable B ₁	63	64	Entropy Variable B ₂	79	80	Entropy Variable B ₃	95	

If a block is mono, this sub-block contains 3 ‘Entropy Variables’. If a block is stereo, this sub-block contains 6. Each is stored as a signed, 16-bit value which is packed in the same fashion as ‘Decorrelation Samples’³. For example, given a 2 channel block with the sub-block bytes:

```
05 06 e2 07 9b 08 55 09 e2 07 76 08 ba 08
```

our ‘Entropy Variables’ are:

$$\begin{aligned} \text{Entropy Variable } A_1 &= 0x07E2 = 2018 = \lfloor \text{wv_exp2}(2018 \bmod 256) \div 2^{9-\lfloor 2018 \div 2^8 \rfloor} \rfloor \\ &= \lfloor \text{wv_exp2}(226) \div 2^{9-7} \rfloor = \lfloor 472 \div 4 \rfloor = \mathbf{118} \end{aligned}$$

$$\begin{aligned} \text{Entropy Variable } A_2 &= 0x089B = 2203 = \lfloor \text{wv_exp2}(2203 \bmod 256) \div 2^{9-\lfloor 2203 \div 2^8 \rfloor} \rfloor \\ &= \lfloor \text{wv_exp2}(155) \div 2^{9-8} \rfloor = \lfloor 389 \div 2 \rfloor = \mathbf{194} \end{aligned}$$

$$\begin{aligned} \text{Entropy Variable } A_3 &= 0x0955 = 2389 = \text{wv_exp2}(2389 \bmod 256) \times 2^{\lfloor 2389 \div 2^8 \rfloor - 9} \\ &= \text{wv_exp2}(85) \times 2^{9-9} = 322 \times 1 = \mathbf{322} \end{aligned}$$

$$\begin{aligned} \text{Entropy Variable } B_1 &= 0x07E2 = 2018 = \lfloor \text{wv_exp2}(2018 \bmod 256) \div 2^{9-\lfloor 2018 \div 2^8 \rfloor} \rfloor \\ &= \lfloor \text{wv_exp2}(226) \div 2^{9-7} \rfloor = \lfloor 472 \div 4 \rfloor = \mathbf{118} \end{aligned}$$

$$\begin{aligned} \text{Entropy Variable } B_2 &= 0x0876 = 2166 = \lfloor \text{wv_exp2}(2166 \bmod 256) \div 2^{9-\lfloor 2166 \div 2^8 \rfloor} \rfloor \\ &= \lfloor \text{wv_exp2}(118) \div 2^{9-8} \rfloor = \lfloor 352 \div 2 \rfloor = \mathbf{176} \end{aligned}$$

$$\begin{aligned} \text{Entropy Variable } B_3 &= 0x08BA = 2234 = \lfloor \text{wv_exp2}(2234 \bmod 256) \div 2^{9-\lfloor 2234 \div 2^8 \rfloor} \rfloor \\ &= \lfloor \text{wv_exp2}(186) \div 2^{9-8} \rfloor = \lfloor 424 \div 2 \rfloor = \mathbf{212} \end{aligned}$$

³As described on page 74.

8.3.6 the Bitstream Sub-Block

This sub-block contains the block's residual values. Once decoded, these will ultimately become our file's raw PCM values.

Metadata Function (0xA) 0	0 5	Actual Size 1 Less 6	Large Block 7	Block Size 8	15/31
Residual Data 32					

Decoding the 'Bitstream' sub-block requires the 'Entropy Variables' data which it combines with this sub-block's bitstream to yield a set of signed values totaling 'Block Samples' (times 2 if the block is stereo).

Decoding each value is a complex process which I'll divide into three separate steps.

As an example, we'll use a 1-channel block with the entropy variables:

- Entropy Variable $A_1 = 111$
- Entropy Variable $A_1 = 159$
- Entropy Variable $A_1 = 299$

and the partial bitstream sub-block bytes:

8a 0e 00 00 a1 77 e9

The first four bytes are the header and block size values. The remaining three are as follows as a little-endian stream:

1 0 0 0 0 1 0 1 1 1 1 0 1 1 1 0 1 0 0 1 0 1 1 1

Determining t

The first step is taking two boolean values called ‘Holding One’ and ‘Holding Zero’ and determining ‘t’. These holding values can be thought of as registers in a sort of Wav-Pack bitstream virtual machine whose values will change over the course of decoding. Their initial values are both false.

‘limited unary’ means counting the number of 1 bits until the next 0 bit, to a maximum of 33, 1 bits in a row.

In our example, to decode the first ‘t’ value:

- **is holding_zero?** no
- **t = limited_unary** = ‘1 0’ = 1
- **is t = 16?** no
- **is holding_one?** no
- **is t odd?** yes
- **holding_one = true**
- **holding_zero = false**
- **t = $\lfloor t \div 2 \rfloor = \lfloor 1 \div 2 \rfloor = 0$**

So our ‘t’ value is 0, our ‘Holding One’ value is true, our ‘Holding Zero’ value is false and we’ve consumed 2 bits from the sub-block’s bitstream.

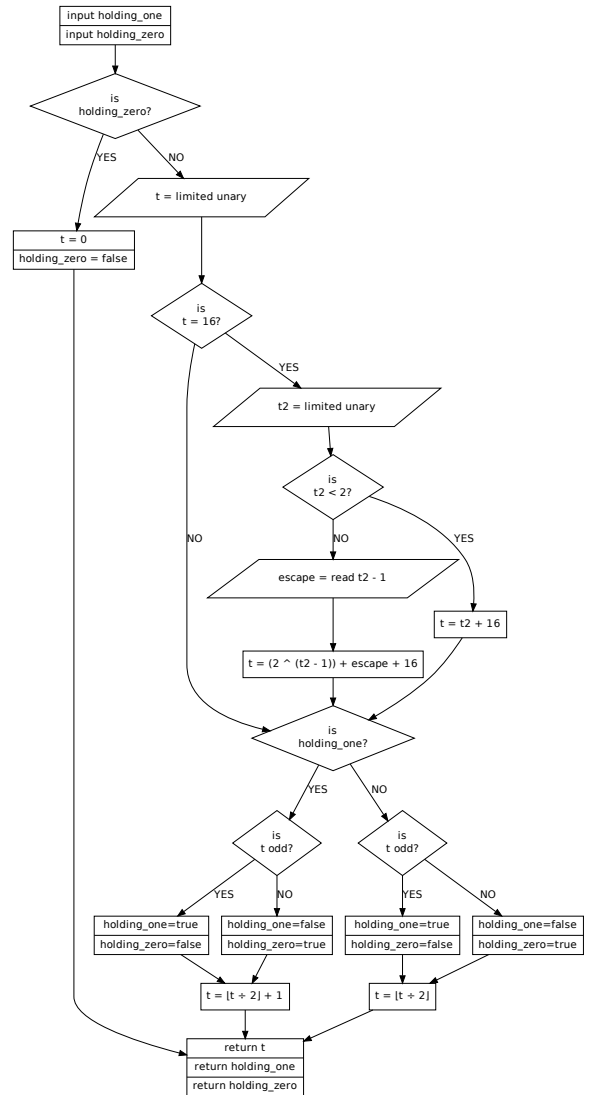


Figure 8.1: Step 1: determining t

Calculating Base/Add

The next step is taking ‘t’ and calculating ‘Base’ and ‘Add’ from our entropy variables, updating our entropy variables in the process.

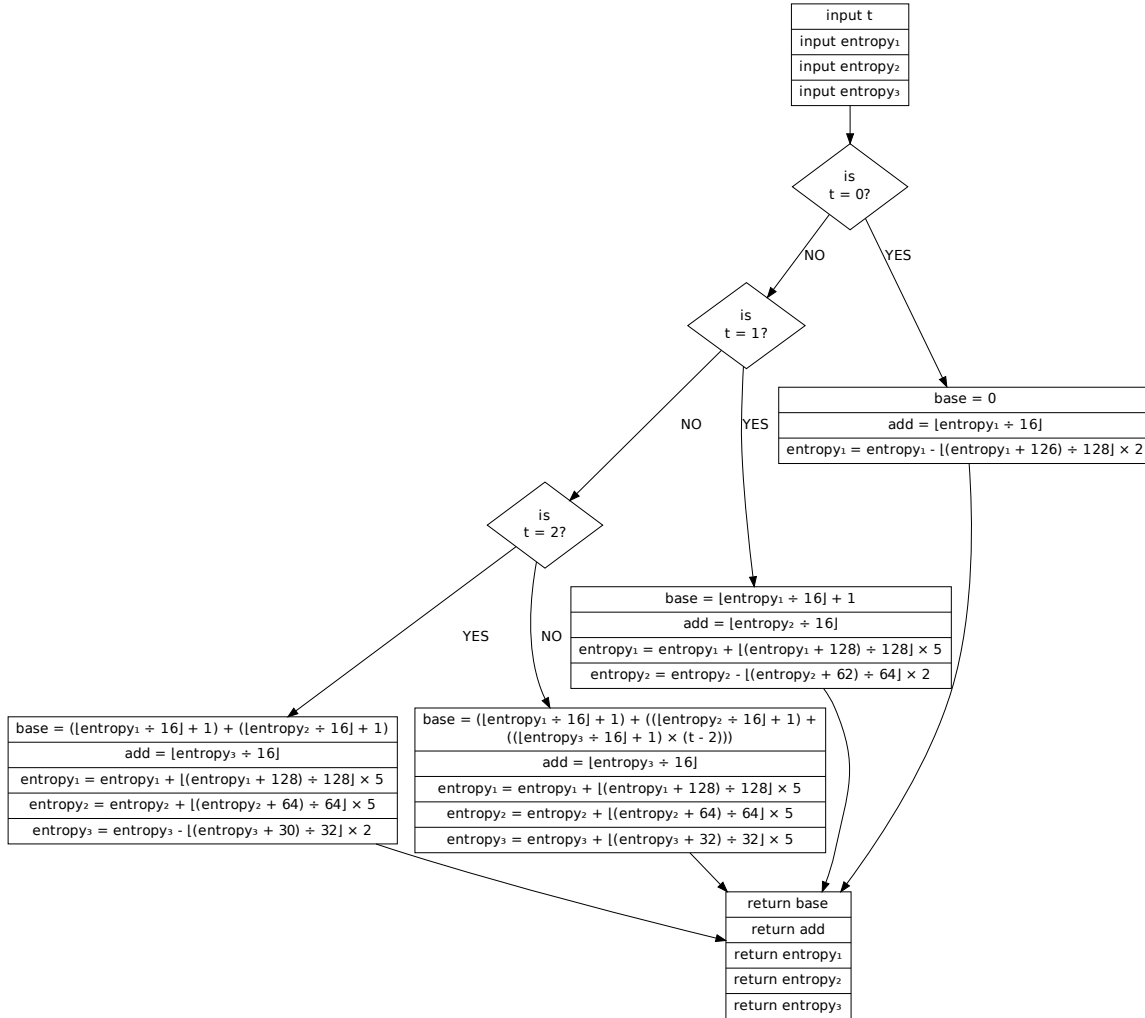


Figure 8.2: Step 2: determining base/add

So to continue our example:

- **is t = 0?** yes
- **base** = 0
- **add** = $\lfloor \text{Entropy}_1 \div 16 \rfloor = \lfloor 111 \div 16 \rfloor = 6$
- **Entropy₁** = $\text{Entropy}_1 - \lfloor (\text{Entropy}_1 + 126) \div 128 \rfloor \times 2 = 111 - 2 = 109$

Determining Value

Finally, given our ‘Base’ and ‘Add’ values, we determine the final residual value as follows:

- **is add < 1?** no
- **p = $\log_2(\text{add})$** = $\log_2(6)$ = 2
- **e = $2^{p+1} - \text{add} - 1$** = $2^3 - 6 - 1 = 1$
- **is p > 0?** yes
- **result = read 2** = ‘0 0’ = 0
- **is result > e?** no
- **sign = read 1** = ‘0’ = 0
- **is sign = 1?** no
- **value = base + result** = 0 + 0 = 0

Thus, this stage consumes an additional 3 bits and our first residual value is 0.

Determining the second residual value requires going through all three steps again, with our freshly updated ‘Holding One’, ‘Holding Zero’ and ‘Entropy’ values.

Note that in a 2 channel (non-mono) block, the ‘Entropy’ values alternate between residuals. For example, Residual₀ uses Entropy A, Residual₁ uses Entropy B, Residual₃ uses Entropy A, and so forth. However, ‘Holding One’ and ‘Holding Zero’ are shared between channels.

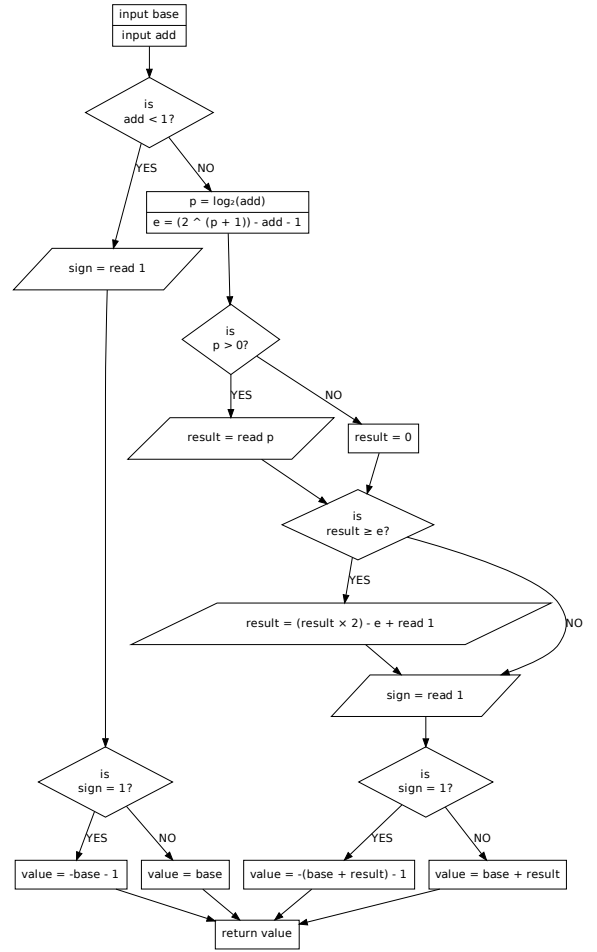


Figure 8.3: Step 3: determining value

Now, let's run through the next residual on our remaining bits:

1 0 1 1 1 1 0 1 1 1 0 1 0 0 1 0 1 1 1

- **is holding_zero?** no
- **t = limited_unary** = '1 0' = 1
- **is t = 16?** no
- **is holding_one?** yes
- **is t odd?** yes
- **holding_one** = true
- **holding_zero** = false
- **t** = $\lfloor t \div 2 \rfloor + 1 = \lfloor 1 \div 2 \rfloor + 1 = 1$
- **is t = 0?** no
- **is t = 1?** yes
- **base** = $\lfloor \text{Entropy}_1 \div 16 \rfloor + 1 = \lfloor 109 \div 16 \rfloor + 1 = 7$
- **add** = $\lfloor \text{Entropy}_2 \div 16 \rfloor = \lfloor 159 \div 16 \rfloor = 9$
- **Entropy₁** = $\text{Entropy}_1 + \lfloor (\text{Entropy}_1 + 128) \div 128 \rfloor \times 5 = 114$
- **Entropy₂** = $\text{Entropy}_2 - \lfloor (\text{Entropy}_2 + 62) \div 64 \rfloor \times 2 = 153$
- **is add < 1?** no
- **p** = $\log_2(\text{add}) = \log_2(9) = 3$
- **e** = $2^{p+1} - \text{add} - 1 = 2^4 - 9 - 1 = 6$
- **is p > 0?** yes
- **result = read 3** = '1 1 1' = 7
- **is result > e?** yes
- **result** = $(\text{result} \times 2) - \text{e} + \text{read 1} = (7 \times 2) - 6 + 1 = 9$
- **sign = read 1** = 0
- **is sign = 1?** no
- **value = base + result** = $7 + 9 = 16$

Which returns the value 16 and consumes 7 bits in total.

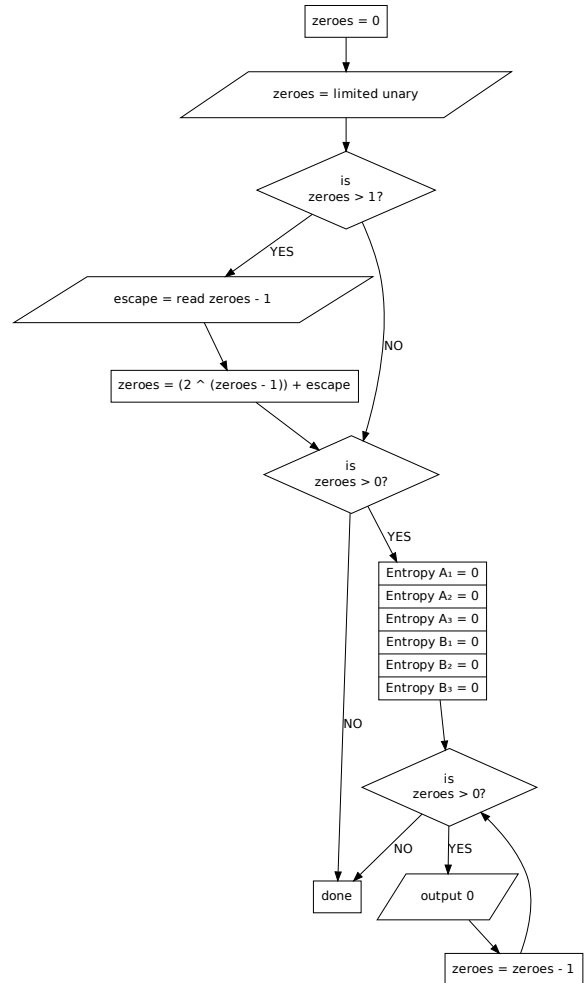
Zero Residuals

As with most other lossless codecs, WavPack features a special case to handle a large number of 0 samples in a row. This is triggered when Entropy A_1 is less than 2, Entropy B_1 is less than 2 (for non-mono blocks), and ‘Holding One’ and ‘Holding Zero’ are both false.

In that instance, we read a residual-like value to determine how many 0 values follow. If any, we set the block’s six ‘Entropy’ variables to 0 and output the necessary number of 0 values just as regular residuals.

Therefore, for non-mono blocks, these values alternative between channels just as regular residual values do. In addition, they also count against the block’s total number of samples.

Once all of the 0 values have been output, if any ‘Block Samples’ remain, we return to the regular residual reading process.



8.3.7 Sample Decorrelation

Once the bitstream sub-block has been decoded into a set of samples values (alternating between ‘Channel A’ and ‘Channel B’ if the block is not mono), we then apply decorrelation passes to those samples - one pass per decorrelation term value,⁴ per channel.

Each decorrelation pass requires a ‘Decorrelation Term’, a ‘Decorrelation Delta’, one ‘Decorrelation Weight’ per channel, and one or more ‘Decorrelation Sample’ values - in addition to our set of input samples we’re running the pass over. These passes are applied in *incrementing* order (i.e. Term₁ first, Term₂ next, and so on). The function for each pass depends on its ‘Decorrelation Term’:

Decorrelation Term = 18:

$$\begin{aligned} \text{Temp}_i &= \lfloor ((3 \times \text{Output}_{i-1}) - \text{Output}_{i-2}) \div 2 \rfloor \\ \text{Output}_i &= \lfloor ((\text{Weight}_{i-1} \times \text{Temp}_i) + 512) \div 1024 \rfloor + \text{Input}_i \\ \text{Weight}_i &= \begin{cases} \text{Weight}_{i-1} & \text{if } \text{Temp}_i = 0 \text{ or } \text{Input}_i = 0 \\ \text{Weight}_{i-1} + \text{Delta} & \text{if } (\text{Temp}_i \mathbf{xor} \text{Input}_i) \geq 0 \\ \text{Weight}_{i-1} - \text{Delta} & \text{if } (\text{Temp}_i \mathbf{xor} \text{Input}_i) < 0 \end{cases} \end{aligned}$$

Decorrelation Term = 17:

$$\begin{aligned} \text{Temp}_i &= (2 \times \text{Output}_{i-1}) - \text{Output}_{i-2} \\ \text{Output}_i &= \lfloor ((\text{Weight}_{i-1} \times \text{Temp}_i) + 512) \div 1024 \rfloor + \text{Input}_i \\ \text{Weight}_i &= \begin{cases} \text{Weight}_{i-1} & \text{if } \text{Temp}_i = 0 \text{ or } \text{Input}_i = 0 \\ \text{Weight}_{i-1} + \text{Delta} & \text{if } (\text{Temp}_i \mathbf{xor} \text{Input}_i) \geq 0 \\ \text{Weight}_{i-1} - \text{Delta} & \text{if } (\text{Temp}_i \mathbf{xor} \text{Input}_i) < 0 \end{cases} \end{aligned}$$

$1 \leq \text{Decorrelation Term} \leq 8$:

$$\begin{aligned} \text{Output}_i &= \lfloor ((\text{Weight}_{i-1} \times \text{Output}_{i-\text{term}}) + 512) \div 1024 \rfloor + \text{Input}_i \\ \text{Weight}_i &= \begin{cases} \text{Weight}_{i-1} & \text{if } \text{Output}_{i-\text{term}} = 0 \text{ or } \text{Input}_i = 0 \\ \text{Weight}_{i-1} + \text{Delta} & \text{if } (\text{Output}_{i-\text{term}} \mathbf{xor} \text{Input}_i) \geq 0 \\ \text{Weight}_{i-1} - \text{Delta} & \text{if } (\text{Output}_{i-\text{term}} \mathbf{xor} \text{Input}_i) < 0 \end{cases} \end{aligned}$$

Note that each function uses previously output samples for its calculation. This is where ‘Decorrelation Samples’ are used; those are our Output₋₁, Output₋₂, etc. which are used for decorrelation but not actually output.

⁴As decoded on page 72.

For 1 or 2 channel blocks, positive decorrelation terms are applied on a per-channel basis with the weight A values being applied to channel A and the weight B values being applied to channel B (if present). However, the three negative correlation terms are only valid for 2 channel blocks.

Decorrelation Term = -1:

$$\begin{aligned}
 \text{Output } A_i &= \lfloor ((\text{Weight } A_{i-1} \times \text{Output } B_{i-1}) + 512) \div 1024 \rfloor + \text{Input } A_i \\
 \text{Weight } A_i &= \begin{cases} \text{Weight } A_{i-1} & \text{if Output } B_{i-1} = 0 \text{ or Input } A_i = 0 \\ \text{Weight } A_{i-1} + \text{Delta} & \text{if } (\text{Output } B_{i-1} \mathbf{xor} \text{ Input } A_i) \geq 0 \\ \text{to a maximum of 1024} & \\ \text{Weight } A_{i-1} - \text{Delta} & \text{if } (\text{Output } B_{i-1} \mathbf{xor} \text{ Input } A_i) < 0 \\ \text{to a minimum of -1024} & \end{cases} \\
 \text{Output } B_i &= \lfloor ((\text{Weight } B_{i-1} \times \text{Output } A_i) + 512) \div 1024 \rfloor + \text{Input } B_i \\
 \text{Weight } B_i &= \begin{cases} \text{Weight } B_{i-1} & \text{if Output } A_i = 0 \text{ or Input } B_i = 0 \\ \text{Weight } B_{i-1} + \text{Delta} & \text{if } (\text{Output } A_i \mathbf{xor} \text{ Input } B_i) \geq 0 \\ \text{to a maximum of 1024} & \\ \text{Weight } B_{i-1} - \text{Delta} & \text{if } (\text{Output } A_i \mathbf{xor} \text{ Input } B_i) < 0 \\ \text{to a minimum of -1024} & \end{cases}
 \end{aligned}$$

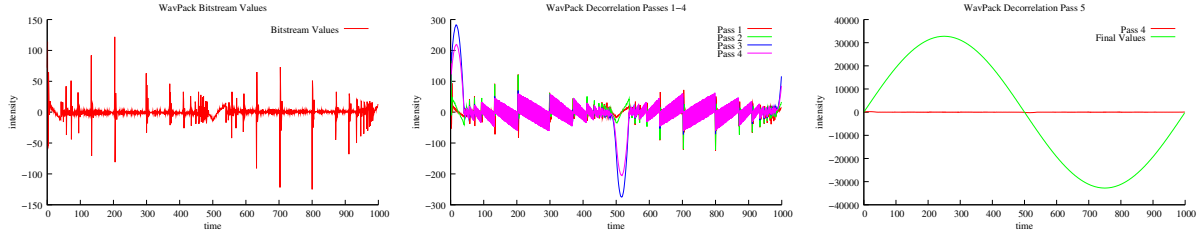
Decorrelation Term = -2:

$$\begin{aligned}
 \text{Output } B_i &= \lfloor ((\text{Weight } B_{i-1} \times \text{Output } A_{i-1}) + 512) \div 1024 \rfloor + \text{Input } B_i \\
 \text{Weight } B_i &= \begin{cases} \text{Weight } B_{i-1} & \text{if Output } A_{i-1} = 0 \text{ or Input } B_i = 0 \\ \text{Weight } B_{i-1} + \text{Delta} & \text{if } (\text{Output } A_{i-1} \mathbf{xor} \text{ Input } B_i) \geq 0 \\ \text{to a maximum of 1024} & \\ \text{Weight } B_{i-1} - \text{Delta} & \text{if } (\text{Output } A_{i-1} \mathbf{xor} \text{ Input } B_i) < 0 \\ \text{to a minimum of -1024} & \end{cases} \\
 \text{Output } A_i &= \lfloor ((\text{Weight } A_{i-1} \times \text{Output } B_i) + 512) \div 1024 \rfloor + \text{Input } A_i \\
 \text{Weight } A_i &= \begin{cases} \text{Weight } A_{i-1} & \text{if Output } B_i = 0 \text{ or Input } A_i = 0 \\ \text{Weight } A_{i-1} + \text{Delta} & \text{if } (\text{Output } B_i \mathbf{xor} \text{ Input } A_i) \geq 0 \\ \text{to a maximum of 1024} & \\ \text{Weight } A_{i-1} - \text{Delta} & \text{if } (\text{Output } B_i \mathbf{xor} \text{ Input } A_i) < 0 \\ \text{to a minimum of -1024} & \end{cases}
 \end{aligned}$$

Decorrelation Term = -3:

$$\begin{aligned} \text{Output } A_i &= \lfloor ((\text{Weight } A_{i-1} \times \text{Output } B_{i-1}) + 512) \div 1024 \rfloor + \text{Input } A_i \\ \text{Weight } A_i &= \begin{cases} \text{Weight } A_{i-1} & \text{if Output } B_{i-1} = 0 \text{ or Input } A_i = 0 \\ \text{Weight } A_{i-1} + \text{Delta} & \text{if } (\text{Output } B_{i-1} \mathbf{xor} \text{ Input } A_i) \geq 0 \\ & \text{to a maximum of 1024} \\ \text{Weight } A_{i-1} - \text{Delta} & \text{if } (\text{Output } B_{i-1} \mathbf{xor} \text{ Input } A_i) < 0 \\ & \text{to a minimum of -1024} \end{cases} \\ \text{Output } B_i &= \lfloor ((\text{Weight } B_{i-1} \times \text{Output } A_{i-1}) + 512) \div 1024 \rfloor + \text{Input } B_i \\ \text{Weight } B_i &= \begin{cases} \text{Weight } B_{i-1} & \text{if Output } A_{i-1} = 0 \text{ or Input } B_i = 0 \\ \text{Weight } B_{i-1} + \text{Delta} & \text{if } (\text{Output } A_{i-1} \mathbf{xor} \text{ Input } B_i) \geq 0 \\ & \text{to a maximum of 1024} \\ \text{Weight } B_{i-1} - \text{Delta} & \text{if } (\text{Output } A_{i-1} \mathbf{xor} \text{ Input } B_i) < 0 \\ & \text{to a minimum of -1024} \end{cases} \end{aligned}$$

The effect of applying these passes cumulatively is interesting when visualized on a 1 channel sine wave example stream:



Now it's time to put all this together into an example. Given a 1 channel block with the sub-block decorrelation values:

Term ₁	3	Delta ₁	2	Weight ₁	16	Sample _{1 1}	0	Sample _{1 2}	0	Sample _{1 3}	0
Term ₂	17	Delta ₂	2	Weight ₂	48			Sample _{2 1}	0	Sample _{2 2}	0
Term ₃	2	Delta ₃	2	Weight ₃	32			Sample _{3 1}	0	Sample _{3 2}	0
Term ₄	18	Delta ₄	2	Weight ₄	48			Sample _{4 1}	0	Sample _{4 2}	0
Term ₅	18	Delta ₅	2	Weight ₅	48			Sample _{5 1}	-78	Sample _{5 2}	-73

and the residual values:

$$\text{Residual}_1 = -61$$

$$\text{Residual}_2 = -33$$

then decorrelation pass 1 applies the Term₁ formula 3, Delta₁ value of 2, Weight_{1 0} value of 16 and initial sample values of 0 (Sample_{1 1}, Sample_{1 2}, Sample_{1 3}) to the residual input values of -61 and -33 (Channel₁ and Channel₂).

$$\begin{aligned} \text{Output}_1 &= \lfloor ((\text{Weight}_{1 0} \times \text{Output}_{-2}) + 512) \div 1024 \rfloor + \text{Input}_1 \\ &= \lfloor ((16 \times 0) + 512) \div 1024 \rfloor - 61 = \mathbf{-61} \end{aligned}$$

$$\text{Weight}_{1 1} = \text{Weight}_{1 0} = \mathbf{16}$$

$$\begin{aligned} \text{Output}_2 &= \lfloor ((\text{Weight}_{1 1} \times \text{Output}_{-1}) + 512) \div 1024 \rfloor + \text{Input}_2 \\ &= \lfloor ((16 \times 0) + 512) \div 1024 \rfloor - 33 = \mathbf{-33} \end{aligned}$$

$$\text{Weight}_{1 2} = \text{Weight}_{1 1} = \mathbf{16}$$

Decorrelation pass 2 applies the Term₂ formula 17, Delta₂ value of 2, Weight_{2 0} value of 48 and the initial sample values of 0 (Sample_{2 1}, Sample_{2 2}). Note that the inputs to pass 2 are the outputs from pass 1.

$$\text{Temp}_1 = (2 \times \text{Output}_0) - \text{Output}_{-1} = (2 \times 0) - 0 = \mathbf{0}$$

$$\begin{aligned} \text{Output}_1 &= \lfloor ((\text{Weight}_{2 0} \times \text{Temp}_1) + 512) \div 1024 \rfloor + \text{Input}_1 \\ &= \lfloor ((48 \times 0) + 512) \div 1024 \rfloor - 61 = \mathbf{-61} \end{aligned}$$

$$\text{Weight}_{2 1} = \text{Weight}_{2 0} = \mathbf{48}$$

$$\text{Temp}_2 = (2 \times \text{Output}_1) - \text{Output}_0 = (2 \times -61) - 0 = \mathbf{-122}$$

$$\begin{aligned} \text{Output}_2 &= \lfloor ((\text{Weight}_{2 1} \times \text{Temp}_2) + 512) \div 1024 \rfloor + \text{Input}_2 \\ &= \lfloor ((48 \times -122) + 512) \div 1024 \rfloor - 33 = -6 - 33 = \mathbf{-39} \end{aligned}$$

$$\text{Weight}_{2 2} = \text{Weight}_{2 1} + \text{Delta}_2 = 48 + 2 = \mathbf{50}$$

Decorrelation pass 3 applies the Term₃ formula 2, Delta₃ value of 2, Weight_{3 0} value of 32

and initial samples of 0 (Sample_{3 1}, Sample_{3 2}).

$$\begin{aligned}\text{Output}_1 &= \lfloor ((\text{Weight}_{3 0} \times \text{Output}_{-1}) + 512) \div 1024 \rfloor + \text{Input}_1 \\ &= \lfloor ((32 \times 0) + 512) \div 1024 \rfloor - 61 = \mathbf{-61}\end{aligned}$$

$$\text{Weight}_{3 1} = \text{Weight}_{3 0} = \mathbf{32}$$

$$\begin{aligned}\text{Output}_2 &= \lfloor ((\text{Weight}_{3 1} \times \text{Output}_0) + 512) \div 1024 \rfloor + \text{Input}_2 \\ &= \lfloor ((32 \times 0) + 512) \div 1024 \rfloor - 39 = \mathbf{-39}\end{aligned}$$

$$\text{Weight}_{3 2} = \text{Weight}_{3 1} = \mathbf{32}$$

Decorrelation pass 4 applies the Term₄ formula 18, Delta₄ value of 2, Weight_{4 0} value of 48 and initial samples of 0 (Sample_{4 1}, Sample_{4 2}).

$$\text{Temp}_1 = \lfloor ((3 \times \text{Output}_0) - \text{Output}_{-1}) \div 2 \rfloor = \lfloor ((3 \times 0) - 0) \div 2 \rfloor = \mathbf{0}$$

$$\begin{aligned}\text{Output}_1 &= \lfloor ((\text{Weight}_{4 0} \times \text{Temp}_1) + 512) \div 1024 \rfloor + \text{Input}_1 \\ &= \lfloor ((48 \times 0) + 512) \div 1024 \rfloor - 61 = \mathbf{-61}\end{aligned}$$

$$\text{Weight}_{4 1} = \text{Weight}_{4 0} = \mathbf{48}$$

$$\text{Temp}_2 = \lfloor ((3 \times \text{Output}_1) - \text{Output}_0) \div 2 \rfloor = \lfloor ((3 \times -61) - 0) \div 2 \rfloor = \mathbf{-92}$$

$$\begin{aligned}\text{Output}_2 &= \lfloor ((\text{Weight}_{4 1} \times \text{Temp}_2) + 512) \div 1024 \rfloor + \text{Input}_2 \\ &= \lfloor ((48 \times -92) + 512) \div 1024 \rfloor - 39 = -4 - 39 = \mathbf{-43}\end{aligned}$$

$$\text{Weight}_{4 2} = \text{Weight}_{4 1} + \text{Delta}_4 = 48 + 2 = \mathbf{50}$$

Finally, decorrelation pass 5 applies the Term₅ formula 18, Delta₅ value of 2, Weight_{5 0} value of 48 and initial samples of -78, -73 (Sample_{5 1}, Sample_{5 2}).

$$\text{Temp}_1 = \lfloor ((3 \times \text{Output}_0) - \text{Output}_{-1}) \div 2 \rfloor = \lfloor ((3 \times -73) + 78) \div 2 \rfloor = \mathbf{-71}$$

$$\begin{aligned}\text{Output}_1 &= \lfloor ((\text{Weight}_{5 0} \times \text{Temp}_1) + 512) \div 1024 \rfloor + \text{Input}_1 \\ &= \lfloor ((48 \times -71) + 512) \div 1024 \rfloor - 61 = -3 - 61 = \mathbf{-64}\end{aligned}$$

$$\text{Weight}_{5 1} = \text{Weight}_{5 0} + \text{Delta}_5 = 48 + 2 = \mathbf{50}$$

$$\text{Temp}_2 = \lfloor ((3 \times \text{Output}_1) - \text{Output}_0) \div 2 \rfloor = \lfloor ((3 \times -64) + 73) \div 2 \rfloor = \mathbf{-60}$$

$$\begin{aligned}\text{Output}_2 &= \lfloor ((\text{Weight}_{5 1} \times \text{Temp}_2) + 512) \div 1024 \rfloor + \text{Input}_2 \\ &= \lfloor ((50 \times -60) + 512) \div 1024 \rfloor - 43 = \mathbf{-46}\end{aligned}$$

$$\text{Weight}_{5 2} = \text{Weight}_{5 1} + \text{Delta}_5 = 50 + 2 = \mathbf{52}$$

So, after running through all five passes, our samples are now -64 and -46.

8.3.8 Joint Stereo

If the block is not mono and the ‘Joint Stereo’ bit is set in the block header, our channels require one more stage of processing to transform their mid-side values back into left and right sample values.⁵

$$\begin{aligned}\text{Left}_i &= \left\lceil \frac{\text{Channel } A_i + (\text{Channel } B_i \times 2)}{2} \right\rceil \\ \text{Right}_i &= (\text{Channel } B_i \times 2) - \left\lfloor \frac{\text{Channel } A_i + (\text{Channel } B_i \times 2)}{2} \right\rfloor\end{aligned}$$

For example, given the Channel A samples of -64 and -46, and the Channel B samples of 32 and 39, we convert them to left and right samples as follows:

$$\begin{aligned}\text{Left}_1 &= \left\lceil \frac{\text{Channel } A_1 + (\text{Channel } B_1 \times 2)}{2} \right\rceil = \left\lceil \frac{-64 + (32 \times 2)}{2} \right\rceil = \mathbf{0} \\ \text{Right}_1 &= (\text{Channel } B_1 \times 2) - \left\lfloor \frac{\text{Channel } A_1 + (\text{Channel } B_1 \times 2)}{2} \right\rfloor \\ &= (32 \times 2) - \left\lfloor \frac{-64 + (32 \times 2)}{2} \right\rfloor = 64 - 0 = \mathbf{64} \\ \text{Left}_2 &= \left\lceil \frac{-46 + (39 \times 2)}{2} \right\rceil = \mathbf{16} \\ \text{Right}_2 &= (\text{Channel } B_2 \times 2) - \left\lfloor \frac{\text{Channel } A_2 + (\text{Channel } B_2 \times 2)}{2} \right\rfloor \\ &= (39 \times 2) - \left\lfloor \frac{-46 + (39 \times 2)}{2} \right\rfloor = 78 - 16 = \mathbf{62}\end{aligned}$$

Thus, our left samples are 0 and 16, and our right samples are 64 and 62.

8.3.9 the CRC

Verifying the block’s CRC is quite simple:

$$\text{CRC}_i = (3 \times \text{CRC}_{i-1}) + \text{Decoded Sample}_i$$

where Decoded Sample_i alternates between channels if necessary, CRC_{-1} begins with a value of 0xFFFFFFFF and each CRC_i value is truncated to 32 bits.

The CRC is calculated *after* the joint stereo transformation, but *before* handling extended/shifted integers and false stereo.

⁵In the case of multi-channel audio, these aren’t necessarily *front* left and right; they might be side left and right or rear left and right channels.

8.3.10 Extended/Shifted Integers

If ‘Extended Size Integers’ is set in the block header, there should be an ‘Int32 Info’ sub-block present whose layout is as follows:

0	Metadata Function (9)	4	0	5	Actual Size 1 Less	6	Large Block (0)	7	8	Block Size (2)	15
16	Sent Bits	23	24	Zero Bits	31	32	One Bits	39	40	Duplicate Bits	47

Curiously, these values are exclusive; if ‘Zero Bits’ is present, ‘One Bits’ is ignored and so forth. If ‘Zero Bits’ is non-zero, we pad each sample’s least-significant bits with that many 0 bits. If ‘One Bits’ is non-zero, we pad each sample’s least-significant bits with that many 1 bits. If ‘Duplicate Bits’ is non-zero, we pad each sample’s least-significant bits with that sample’s own least-significant bit, ‘Duplicate Bits’ number of times.

This can be summarized as follows:

$$\text{Extended}_i = \begin{cases} \text{Original}_i \times 2^{\text{Zero Bits}} & \text{if 'Zero Bits' } > 0 \\ \text{Original}_i \times 2^{\text{One Bits}} + (2^{\text{One Bits}} - 1) & \text{if 'One Bits' } > 0 \\ \text{Original}_i \times 2^{\text{Duplicate Bits}} & \text{if 'Duplicate Bits' } > 0 \text{ and } \text{Original}_i \bmod 2 = 0 \\ \text{Original}_i \times 2^{\text{Duplicate Bits}} + (2^{\text{Duplicate Bits}} - 1) & \text{if 'Duplicate Bits' } > 0 \text{ and } \text{Original}_i \bmod 2 = 1 \end{cases}$$

8.3.11 False Stereo

If the ‘False Stereo’ bit is set in the block header, we’ve been treating the block as being mono thus far. At this point, we duplicate Channel A’s values to Channel B just prior to returning the from the block.

8.3.12 RIFF WAVE Header/Footer

These sub-blocks are typically found in the first and last WavPack block, respectively. The header must always be present in the file while the footer is optional.

<small>0</small>	Metadata Function (1)	<small>4</small>	Nondecoder Data (1)	<small>5</small>	Actual Size 1 Less	<small>6</small>	Large Block	<small>7</small>
<small>8</small>	Block Size	<small>15/31</small>	<small>16/32</small>	RIFF WAVE Header				

<small>0</small>	Metadata Function (2)	<small>4</small>	Nondecoder Data (1)	<small>5</small>	Actual Size 1 Less	<small>6</small>	Large Block	<small>7</small>
<small>8</small>	Block Size	<small>15/31</small>	<small>16/32</small>	RIFF WAVE Footer				

One can think of them as halves of a ‘PCM sandwich’ of which our decoded data comprises the ‘meat’:

RIFF WAVE Header	Decoded PCM Data	RIFF WAVE Footer
------------------	------------------	------------------

8.3.13 MD5

This optional sub-block is typically found in the final WavPack block.

<small>0</small>	Metadata Function (6)	<small>4</small>	Nondecoder Data (1)	<small>5</small>	Actual Size 1 Less (0)	<small>6</small>	Large Block (0)	<small>7</small>
<small>8</small>	Block Size (16)	<small>15</small>	<small>16</small>	MD5 sum				

The MD5 is the hash of all the samples over the entire file. It is calculated by running the hashing algorithm⁶ over the raw input samples in little-endian format and signed if their bits-per-sample are greater than 8.

⁶As described by RFC1321

8.4 WavPack Encoding

For WavPack encoding, one needs a stream of input PCM values along with the stream's sample rate, number of channels, bits per sample and channel mask.

We first split our input samples into chunks containing 'Block Size' number of PCM frames. Since WavPack's headers are relatively large and its adaptive algorithm is quite good over long stretches of samples, it makes sense to use a large block size. The reference encoder defaults to 44100 PCM frames.

The next step is to split those chunks of PCM frames into WavPack blocks containing 1 or 2 channels each. For a one channel input stream, the blocks are sent as follows:

Block	First Block Bit	Last Block Bit	Is Mono	Channel A	Channel B
Block ₁	1	1	1	Front Center	

For a two channel input stream, the blocks are sent as follows:

Block	First Block Bit	Last Block Bit	Is Mono	Channel A	Channel B
Block ₁	1	1	0	Front Left	Front Right

However, for multi-channel input streams, we need to split its channels into a set of blocks with 1 or 2 channels per block. By using the channel mask⁷ we can split the stream into 2 channel blocks with left-right channel pairs and 1 channel blocks for everything else.

For example, given a 6-channel audio stream with the channel mask `0x3F`, we have the channels 'Front Left', 'Front Right', 'Front Center', 'LFE', 'Back Left' and 'Back Right' - in that order. So, a good way to split our channels into blocks is as follows:

Block	First Block Bit	Last Block Bit	Is Mono	Channel A	Channel B
Block ₁	1	0	0	Front Left	Front Right
Block ₂	0	0	1	Front Center	
Block ₃	0	0	1	LFE	
Block ₄	0	1	0	Back Left	Back Right

⁷As explained on page 19.

8.4.1 False Stereo

If the block is stereo and Channel A's samples are identical to Channel B's samples, one can set the 'False Stereo' bit in the block header and treat the block as having only one channel for the rest of its encoding. Note that the block's 'Is Mono' bit is still **false** in this case.

8.4.2 Extended/Shifted Integers

If the following condition holds:

$$0 = \sum_{i=0}^{\text{block size}-1} \text{Channel}_i \bmod 2^{\text{bits}}$$

for Channel A and, if present, Channel B where *bits* > 0, then the highest value of *bits* if what's used for the 'Zero Bits' field in an 'Extended Size Integers' sub-block, as described on page 89. Each channel's samples are then divided by 2^{bits} for the remainder of encoding and the 'Extended Size Integers' bit is set in the block header.

8.4.3 the CRC

After the audio samples have been processed for false stereo and wasted bits, it's best to perform the block header CRC calculation before starting to encode them, as follows:

$$\text{CRC}_i = (3 \times \text{CRC}_{i-1}) + \text{Sample}_i$$

where Sample_i alternates between channels if necessary. CRC_{-1} begins with a value of 0xFFFFFFFF and each CRC_i value is truncated to 32 bits.

8.4.4 Joint Stereo

Next, for two channel blocks, one typically converts both channels to joint stereo. This involves transforming independent left and right channels to mid and side channels.

$$\begin{aligned} \text{Mid}_i &= \text{Channel A}_i - \text{Channel B}_i \\ \text{Side}_i &= \left\lfloor \frac{\text{Channel A}_i + \text{Channel B}_i}{2} \right\rfloor \end{aligned}$$

Where 'Mid' is the new 'Channel A' and 'Side' is the new 'Channel B'. For example, given the 'Channel A' value of 16 and the 'Channel B' value of 62, our conversion is as follows:

$$\begin{aligned} \text{Mid}_0 &= 16 - 62 = \mathbf{-46} \\ \text{Side}_0 &= \left\lfloor \frac{16 + 62}{2} \right\rfloor = \mathbf{39} \end{aligned}$$

One must also set the 'Joint Stereo' bit in the block header.

8.4.5 Block Header

Once the ‘False Stereo’, ‘Extended Size Integers’, ‘Joint Stereo’ and ‘CRC’ values are decided, we can finally write a block header based on our input:

Block ID `wvpk' (0x6B707677)				Block Size			
0	31	32	63				
Version		Track Number		Index Number		Total Samples	
64	79	80	87	88	95	96	127
Block Index				Block Samples			
128	159	160	191				
Bits per Sample		Mono Output		Hybrid Mode		Joint Stereo	
192	193	194	195	196	197	198	199
Channel Decorrelation		Hbd. Noise Shaping		Floating Point Data		Extended Size Integers	
197	198	199	200	201	202	203	204
Hbd. Controls Bitrate		Hbd. Noise Balanced		Initial Block		Final Block	
201	202	203	204	205	206	207	208
Left Shift Data		Maximum Magnitude		Sample Rate			
205	209	210	214	215	216	217	218
Reserved		Use IIR		False Stereo		Reserved	
219	220	221	222	223	224	225	226
CRC							
224	225	226	227	228	229	230	231

The remaining fields are as follows:

Block Size 24 + byte length of sub blocks

Version 0x407

Track Number 0

Index Number 0

Block Index total PCM frames written thus far

Block Samples total PCM frames of block

Hybrid Mode 0

Channel Decorrelation 1 if stereo, 0 if mono

Hybrid Noise Shaping 0

Floating Point Data 0

Hybrid Controls Bitrate 0

Hybrid Noise Balanced 0

Left Shift Data 0

sample rate	value
6000	0000
8000	0001
9600	0010
11025	0011
12000	0100
16000	0101
22050	0110
24000	0111
32000	1000
44100	1001
48000	1010
64000	1011
88200	1100
96000	1101
192000	1110
bits per sample	value
8	00
16	01
24	10
32	11

Maximum Magnitude maximum sample size, in bits

Use IIR 0

Note that the ‘Block Size’ and ‘Total Samples’ fields can’t be known in advance; all the block’s sub-blocks must be generated before we’ll know the former, and the entire file must be written before we’ll know the latter.

8.4.6 Decorrelation Terms/Deltas

These are typically defined by the number of decorrelation passes to use:

Terms	Decorrelation Passes					Deltas	Decorrelation Passes				
	1	2	5	10	16		1	2	5	10	16
Term ₁	18	17	3	4	2	Delta ₁	2	2	2	2	2
Term ₂		18	17	17	18	Delta ₂		2	2	2	2
Term ₃			2	-1	-1	Delta ₃			2	2	2
Term ₄			18	5	8	Delta ₄			2	2	2
Term ₅			18	3	6	Delta ₅			2	2	2
Term ₆				2	3	Delta ₆				2	2
Term ₇				-2	5	Delta ₇				2	2
Term ₈				18	7	Delta ₈				2	2
Term ₉				18	4	Delta ₉				2	2
Term ₁₀				18	2	Delta ₁₀				2	2
Term ₁₁					18	Delta ₁₁					2
Term ₁₂					-2	Delta ₁₂					2
Term ₁₃					3	Delta ₁₃					2
Term ₁₄					2	Delta ₁₄					2
Term ₁₅					18	Delta ₁₅					2
Term ₁₆					18	Delta ₁₆					2

They are placed in a sub-block as follows:

Metadata Function (2)				0	Actual Size 1 Less				Large Block				Block Size			
0			4	5			6			7		8			15/31	
...		Decorr. Term ₂ + 5			Decorr. Delta ₂			Decorr. Term ₁ + 5			Decorr. Delta ₁					
			4	5			7	8			12	13			15	

Since each term/delta pair is 8 bits, ‘Actual Size 1 Less’ is set when the number of terms is odd, ‘Large Block’ is always going to be 0 and ‘Block Size’ equals the number of terms, divided by 2.

8.4.7 Decorrelation Passes

Once our number of decorrelation passes is decided, we must also generate decorrelation weights, decorrelation samples and entropy variables sub-blocks before moving on to the residuals sub-block. So where do we get those values? They actually come from the *previous* block.⁸ Since encoding will modify decorrelation weights and entropy variables as it progresses, the final values for Block_{*i*} become the initial values for Block_{*i*+1}. As for the decorrelation values, the final few decorrelated samples (whose quantity depends on the decorrelation term) are ‘wrapped’ from the previous decorrelation pass into our ‘Decorrelation Samples’ sub-block as its starting point.

However, we can’t store the previous block’s final values as-is. Remember that the values for decorrelation weights are multiplied by 2³ and the values for decorrelation samples and entropy variables are stored logarithmically. Therefore, we must ‘round-trip’ the previous block’s output samples before using them as input samples since they’ll be parsed the same way during decoding. This process will be explained in the sub-block sections to follow.

For Block₀, we’ll set our initial decorrelation weights, decorrelation samples and entropy variables to 0.

The application of each pass requires a ‘Decorrelation Term’, a ‘Decorrelation Delta’, one ‘Decorrelation Weight’ per channel and one or more ‘Decorrelation Sample’ values - in addition to the set of processed input samples we’re running the pass over.

Decorrelation Term = 18:

$$\begin{aligned} \text{Temp}_i &= \lfloor ((3 \times \text{Input}_{i-1}) - \text{Input}_{i-2}) \div 2 \rfloor \\ \text{Output}_i &= \text{Input}_i - \lfloor ((\text{Weight}_{i-1} \times \text{Temp}_i) + 512) \div 1024 \rfloor \\ \text{Weight}_i &= \begin{cases} \text{Weight}_{i-1} & \text{if } \text{Temp}_i = 0 \text{ or } \text{Output}_i = 0 \\ \text{Weight}_{i-1} + \text{Delta} & \text{if } (\text{Temp}_i \mathbf{xor} \text{Output}_i) \geq 0 \\ \text{Weight}_{i-1} - \text{Delta} & \text{if } (\text{Temp}_i \mathbf{xor} \text{Output}_i) < 0 \end{cases} \end{aligned}$$

Decorrelation Term = 17:

$$\begin{aligned} \text{Temp}_i &= (2 \times \text{Input}_{i-1}) - \text{Input}_{i-2} \\ \text{Output}_i &= \text{Input}_i - \lfloor ((\text{Weight}_{i-1} \times \text{Temp}_i) + 512) \div 1024 \rfloor \\ \text{Weight}_i &= \begin{cases} \text{Weight}_{i-1} & \text{if } \text{Temp}_i = 0 \text{ or } \text{Output}_i = 0 \\ \text{Weight}_{i-1} + \text{Delta} & \text{if } (\text{Temp}_i \mathbf{xor} \text{Output}_i) \geq 0 \\ \text{Weight}_{i-1} - \text{Delta} & \text{if } (\text{Temp}_i \mathbf{xor} \text{Output}_i) < 0 \end{cases} \end{aligned}$$

⁸More precisely, the previous block covering the same set of channels - in the case of multi-channel audio.

$1 \leq \text{Decorrelation Term} \leq 8$:

$$\begin{aligned} \text{Output}_i &= \text{Input}_i - \lfloor ((\text{Weight}_{i-1} \times \text{Input}_{i-\text{term}}) + 512) \div 1024 \rfloor \\ \text{Weight}_i &= \begin{cases} \text{Weight}_{i-1} & \text{if } \text{Input}_{i-\text{term}} = 0 \text{ or } \text{Output}_i = 0 \\ \text{Weight}_{i-1} + \text{Delta} & \text{if } (\text{Input}_{i-\text{term}} \mathbf{xor} \text{Output}_i) \geq 0 \\ \text{Weight}_{i-1} - \text{Delta} & \text{if } (\text{Input}_{i-\text{term}} \mathbf{xor} \text{Output}_i) < 0 \end{cases} \end{aligned}$$

Similar to decoding, each function uses previous input samples for its calculation. This is where ‘Decorrelation Samples’ are used; those are our Input_{-1} , Input_{-2} , etc. which are used for decorrelation but not actually output.

For 1 or 2 channel blocks, positive decorrelation terms are applied on a per-channel basis with the weight A values being applied to channel A and the weight B values being applied to channel B (if present). However, the three negative correlation terms are only valid for 2 channel blocks:

Decorrelation Term = -1:

$$\begin{aligned} \text{Temp A}_i &= \text{Input B}_{i-1} \\ \text{Temp B}_i &= \text{Input A}_i \\ \text{Output A}_i &= \text{Input A}_i - \lfloor ((\text{Weight A}_{i-1} \times \text{Temp A}_i) + 512) \div 1024 \rfloor \\ \text{Weight A}_i &= \begin{cases} \text{Weight A}_{i-1} & \text{if } \text{Temp A}_i \text{ or } \text{Output A}_i = 0 \\ \text{Weight A}_{i-1} + \text{Delta} & \text{if } (\text{Temp A}_i \mathbf{xor} \text{Output A}_i) \geq 0 \\ \text{to a maximum of 1024} & \\ \text{Weight A}_{i-1} - \text{Delta} & \text{if } (\text{Temp A}_i \mathbf{xor} \text{Output A}_i) < 0 \\ \text{to a minimum of -1024} & \end{cases} \\ \text{Output B}_i &= \text{Input B}_i - \lfloor ((\text{Weight B}_{i-1} \times \text{Temp B}_i) + 512) \div 1024 \rfloor \\ \text{Weight B}_i &= \begin{cases} \text{Weight B}_{i-1} & \text{if } \text{Temp B}_i \text{ or } \text{Output B}_i = 0 \\ \text{Weight B}_{i-1} + \text{Delta} & \text{if } (\text{Temp B}_i \mathbf{xor} \text{Output B}_i) \geq 0 \\ \text{to a maximum of 1024} & \\ \text{Weight B}_{i-1} - \text{Delta} & \text{if } (\text{Temp B}_i \mathbf{xor} \text{Output B}_i) < 0 \\ \text{to a minimum of -1024} & \end{cases} \end{aligned}$$

Decorrelation Term = -2:

$$\begin{aligned}
& \text{Temp } A_i = \text{Input } B_i \\
& \text{Temp } B_i = \text{Input } A_{i-1} \\
& \text{Output } A_i = \text{Input } A_i - \lfloor ((\text{Weight } A_{i-1} \times \text{Temp } A_i) + 512) \div 1024 \rfloor \\
& \text{Weight } A_i = \begin{cases} \text{Weight } A_{i-1} & \text{if Temp } A_i \text{ or Output } A_i = 0 \\ \text{Weight } A_{i-1} + \text{Delta} & \text{if (Temp } A_i \text{ } \mathbf{xor} \text{ Output } A_i) \geq 0 \\ \text{to a maximum of 1024} & \\ \text{Weight } A_{i-1} - \text{Delta} & \text{if (Temp } A_i \text{ } \mathbf{xor} \text{ Output } A_i) < 0 \\ \text{to a minimum of -1024} & \end{cases} \\
& \text{Output } B_i = \text{Input } B_i - \lfloor ((\text{Weight } B_{i-1} \times \text{Temp } B_i) + 512) \div 1024 \rfloor \\
& \text{Weight } B_i = \begin{cases} \text{Weight } B_{i-1} & \text{if Temp } B_i \text{ or Output } B_i = 0 \\ \text{Weight } B_{i-1} + \text{Delta} & \text{if (Temp } B_i \text{ } \mathbf{xor} \text{ Output } B_i) \geq 0 \\ \text{to a maximum of 1024} & \\ \text{Weight } B_{i-1} - \text{Delta} & \text{if (Temp } B_i \text{ } \mathbf{xor} \text{ Output } B_i) < 0 \\ \text{to a minimum of -1024} & \end{cases}
\end{aligned}$$

Decorrelation Term = -3:

$$\begin{aligned}
& \text{Temp } A_i = \text{Input } B_{i-1} \\
& \text{Temp } B_i = \text{Input } A_{i-1} \\
& \text{Output } A_i = \text{Input } A_i - \lfloor ((\text{Weight } A_{i-1} \times \text{Temp } A_i) + 512) \div 1024 \rfloor \\
& \text{Weight } A_i = \begin{cases} \text{Weight } A_{i-1} & \text{if Temp } A_i \text{ or Output } A_i = 0 \\ \text{Weight } A_{i-1} + \text{Delta} & \text{if (Temp } A_i \text{ } \mathbf{xor} \text{ Output } A_i) \geq 0 \\ \text{to a maximum of 1024} & \\ \text{Weight } A_{i-1} - \text{Delta} & \text{if (Temp } A_i \text{ } \mathbf{xor} \text{ Output } A_i) < 0 \\ \text{to a minimum of -1024} & \end{cases} \\
& \text{Output } B_i = \text{Input } B_i - \lfloor ((\text{Weight } B_{i-1} \times \text{Temp } B_i) + 512) \div 1024 \rfloor \\
& \text{Weight } B_i = \begin{cases} \text{Weight } B_{i-1} & \text{if Temp } B_i \text{ or Output } B_i = 0 \\ \text{Weight } B_{i-1} + \text{Delta} & \text{if (Temp } B_i \text{ } \mathbf{xor} \text{ Output } B_i) \geq 0 \\ \text{to a maximum of 1024} & \\ \text{Weight } B_{i-1} - \text{Delta} & \text{if (Temp } B_i \text{ } \mathbf{xor} \text{ Output } B_i) < 0 \\ \text{to a minimum of -1024} & \end{cases}
\end{aligned}$$

8.4.8 Decorrelation Weights

Once the decorrelation passes for Block₀ have been completed (with its initial decorrelation weight values of 0), we should store its final updated weight values to be used as the initial decorrelation weights for Block₁, as so on through the rest of the file.

There is one decorrelation weight value per decorrelation pass, per channel. Each has a minimum value of -1024 and a maximum value of 1024. Converting their values to 8 bits requires the following formula:

$$\text{value} = \begin{cases} \lfloor (\text{Weight} - \lfloor \frac{\text{Weight} + 2^6}{7} \rfloor + 4) \div 2^3 \rfloor & \text{if Weight} > 0 \\ 0 & \text{if Weight} = 0 \\ \lfloor (\text{Weight} + 4) \div 2^3 \rfloor & \text{if Weight} < 0 \end{cases}$$

Weights are placed in a sub-block in reverse order as follows:

₀ Metadata Function (3) ₄	₅ 0	₆ Actual Size 1 Less	₇ Large Block	₈ Block Size _{15/31}
...	₀	₇ Decorrelation Weight ₂	₈	₁₅ Decorrelation Weight ₁

Since each decorrelation weight value is stored in 8 bits, 'Actual Size 1 Less' is set if the total number of weights is odd, 'Large Block' is always going to be 0 and 'Block Size' is the total number of weights divided by 2.

After the initial weights for Block_{*i*} have been stored, the 'round-trip' formula to retrieve those weight values for Block_{*i*}'s decorrelation passes is as follows:

$$\text{Decorrelation Weight} = \begin{cases} \text{value} \times 2^3 + \lfloor \frac{\text{value} \times 2^3 + 2^6}{2^7} \rfloor & \text{if value} > 0 \\ 0 & \text{if value} = 0 \\ \text{value} \times 2^3 & \text{if value} < 0 \end{cases}$$

8.4.9 Decorrelation Samples

We apply the following formulas to convert our 32-bit, signed decorrelation values to 16-bit signed sub-block values:

$$\begin{aligned}
 asample &= |sample| + \left\lfloor \frac{|sample|}{2^9} \right\rfloor \\
 bitcount &= \text{count_bits}(asample) \\
 \text{Value} &= \begin{cases} (bitcount \times 2^8) + \text{wv_log2}((asample \times 2^{9-bitcount}) \bmod 256) & \text{if } 0 \leq asample < 256 \text{ and } sample \geq 0 \\ (bitcount \times 2^8) + \text{wv_log2}(\lfloor asample \div 2^{bitcount-9} \rfloor \bmod 256) & \text{if } 256 \leq asample \text{ and } sample \geq 0 \\ -((bitcount \times 2^8) + \text{wv_log2}((asample \times 2^{9-bitcount}) \bmod 256)) & \text{if } 0 \leq asample < 256 \text{ and } sample < 0 \\ -((bitcount \times 2^8) + \text{wv_log2}(\lfloor asample \div 2^{bitcount-9} \rfloor \bmod 256)) & \text{if } 256 \leq asample \text{ and } sample < 0 \end{cases}
 \end{aligned}$$

where ‘count_bits’ is defined as follows:

$$\text{count_bits}(x) = \begin{cases} 0 & \text{if } x = 0 \\ 1 + \text{count_bits}(\lfloor x \div 2 \rfloor) & \text{if } x \neq 0 \end{cases}$$

and ‘wv_log2’ is defined from the following base-16 table:

	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	0x01	0x03	0x04	0x06	0x07	0x09	0x0a	0x0b	0x0d	0x0e	0x10	0x11	0x12	0x14	0x15
0x1?	0x16	0x18	0x19	0x1a	0x1c	0x1d	0x1e	0x20	0x21	0x22	0x24	0x25	0x26	0x28	0x29	0x2a
0x2?	0x2c	0x2d	0x2e	0x2f	0x31	0x32	0x33	0x34	0x36	0x37	0x38	0x39	0x3b	0x3c	0x3d	0x3e
0x3?	0x3f	0x41	0x42	0x43	0x44	0x45	0x47	0x48	0x49	0x4a	0x4b	0x4d	0x4e	0x4f	0x50	0x51
0x4?	0x52	0x54	0x55	0x56	0x57	0x58	0x59	0x5a	0x5c	0x5d	0x5e	0x5f	0x60	0x61	0x62	0x63
0x5?	0x64	0x66	0x67	0x68	0x69	0x6a	0x6b	0x6c	0x6d	0x6e	0x6f	0x70	0x71	0x72	0x74	0x75
0x6?	0x76	0x77	0x78	0x79	0x7a	0x7b	0x7c	0x7d	0x7e	0x7f	0x80	0x81	0x82	0x83	0x84	0x85
0x7?	0x86	0x87	0x88	0x89	0x8a	0x8b	0x8c	0x8d	0x8e	0x8f	0x90	0x91	0x92	0x93	0x94	0x95
0x8?	0x96	0x97	0x98	0x99	0x9a	0x9b	0x9c	0x9d	0x9e	0x9f	0xa0	0xa1	0xa2	0xa3	0xa4	0xa5
0x9?	0xa6	0xa7	0xa8	0xa9	0xaa	0xab	0xac	0xad	0xae	0xaf	0xb0	0xb1	0xb2	0xb3	0xb4	0xb5
0xA?	0xb6	0xb7	0xb8	0xb9	0xba	0xbb	0xbc	0xbd	0xbe	0xbf	0xc0	0xc1	0xc2	0xc3	0xc4	0xc5
0xB?	0xc6	0xc7	0xc8	0xc9	0xca	0xcb	0xcc	0xcd	0xce	0xcf	0xd0	0xd1	0xd2	0xd3	0xd4	0xd5
0xC?	0xd6	0xd7	0xd8	0xd9	0xda	0xdb	0xdc	0xdd	0xde	0xdf	0xe0	0xe1	0xe2	0xe3	0xe4	0xe5
0xD?	0xe6	0xe7	0xe8	0xe9	0xea	0xeb	0xec	0xed	0xee	0xef	0xf0	0xf1	0xf2	0xf3	0xf4	0xf5
0xE?	0xf6	0xf7	0xf8	0xf9	0xfa	0xfb	0xfc	0xfd	0xfe	0xff						

For example, given a sample value of 28:

$$asample = |28| + \left\lfloor \frac{|28|}{2^9} \right\rfloor = 28 + 0 = \mathbf{28}$$

$$bitcount = \mathbf{5}$$

$$\begin{aligned}
 value &= (5 \times 2^8) + \text{wv_log2}((28 \times 2^{9-5}) \bmod 256) \\
 &= 1280 + \text{wv_log2}(448 \bmod 256) \\
 &= 1280 + \text{wv_log2}(192) = \mathbf{1487}
 \end{aligned}$$

These samples are then placed in a sub-block as follows:

0	Metadata Function (4)	4	0	Actual Size 1 Less	6	Large Block	7	Block Size	8	15/31
0	Decorrelation Sample ₁	15	16	Decorrelation Sample ₂	31	...				

where ‘Actual Size 1 Less’ and ‘Large Block’ are 0, and ‘Block Size’ is the total number of decorrelation samples.

Writing the values themselves requires traversing the decorrelation samples lists in *reverse* order, from $i = \text{‘Decorrelation Passes’} - 1$ to 0.

For Stereo Block

- If $17 \leq \text{Decorrelation Term}_i \leq 18$
 1. Write Sample A_{i-1}
 2. Write Sample A_{i-0}
 3. Write Sample B_{i-1}
 4. Write Sample B_{i-0}
- If $1 \leq \text{Decorrelation Term}_i \leq 8$
 1. For $j = 0$ to $\text{Decorrelation Term}_i - 1$
 - a) Write Sample A_{i-j}
 - b) Write Sample B_{i-j}
- If $-3 \leq \text{Decorrelation Term}_i \leq -1$
 1. Write Sample B_{i-0}
 2. Write Sample A_{i-0}

For Mono Block

- If $17 \leq \text{Decorrelation Term}_i \leq 18$
 1. Write Sample A_{i-1}
 2. Write Sample A_{i-0}
- If $1 \leq \text{Decorrelation Term}_i \leq 8$
 1. For $j = 0$ to $\text{Decorrelation Term}_i - 1$
 - a) Write Sample A_{i-j}

Round-tripping these values back to decorrelation samples for the next block requires applying the same formula as decoding:

$$\text{Sample} = \begin{cases} \lfloor \text{wv_exp2}(\text{value} \bmod 256) \div 2^{9-\lfloor \text{value} \div 2^8 \rfloor} \rfloor & \text{if } 0 \leq \text{value} \leq 2304 \\ \text{wv_exp2}(\text{value} \bmod 256) \times 2^{\lfloor \text{value} \div 2^8 \rfloor - 9} & \text{if } 2304 < \text{value} \leq 32767 \\ -\lfloor \text{wv_exp2}(-\text{value} \bmod 256) \div 2^{9-\lfloor -\text{value} \div 2^8 \rfloor} \rfloor & \text{if } -2304 \leq \text{value} < 0 \\ -(\text{wv_exp2}(-\text{value} \bmod 256) \times 2^{\lfloor -\text{value} \div 2^8 \rfloor - 9}) & \text{if } -32768 \leq \text{value} < -2304 \end{cases}$$

where ‘wv_exp2’ is defined from the following base-16 table:

	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?	100	101	101	102	103	103	104	105	106	106	107	108	108	109	10A	10B
0x1?	10B	10C	10D	10E	10E	10F	110	110	111	112	113	113	114	115	116	116
0x2?	117	118	119	119	11A	11B	11C	11D	11D	11E	11F	120	120	121	122	123
0x3?	124	124	125	126	127	128	128	129	12A	12B	12C	12C	12D	12E	12F	130
0x4?	130	131	132	133	134	135	135	136	137	138	139	13A	13A	13B	13C	13D
0x5?	13E	13F	140	141	141	142	143	144	145	146	147	148	148	149	14A	14B
0x6?	14C	14D	14E	14F	150	151	151	152	153	154	155	156	157	158	159	15A
0x7?	15B	15C	15D	15E	15E	15F	160	161	162	163	164	165	166	167	168	169
0x8?	16A	16B	16C	16D	16E	16F	170	171	172	173	174	175	176	177	178	179
0x9?	17A	17B	17C	17D	17E	17F	180	181	182	183	184	185	187	188	189	18A
0xA?	18B	18C	18D	18E	18F	190	191	192	193	195	196	197	198	199	19A	19B
0xB?	19C	19D	19F	1A0	1A1	1A2	1A3	1A4	1A5	1A6	1A8	1A9	1AA	1AB	1AC	1AD
0xC?	1AF	1B0	1B1	1B2	1B3	1B4	1B6	1B7	1B8	1B9	1BA	1BC	1BD	1BE	1BF	1C0
0xD?	1C2	1C3	1C4	1C5	1C6	1C8	1C9	1CA	1CB	1CD	1CE	1CF	1D0	1D2	1D3	1D4
0xE?	1D6	1D7	1D8	1D9	1DB	1DC	1DD	1DE	1E0	1E1	1E2	1E4	1E5	1E6	1E8	1E9
0xF?	1EA	1EC	1ED	1EE	1F0	1F1	1F2	1F4	1F5	1F6	1F8	1F9	1FA	1FC	1FD	1FF

8.4.10 the Entropy Variables Sub-Block

Metadata Function (5)	0	Actual Size 1 Less	Large Block	Block Size
0	4	5	6	7
Entropy Variable A ₁	15	16	31	32
0	15	16	31	32
Entropy Variable B ₁	48	63	64	79
48	63	64	79	80
Entropy Variable A ₂	15	16	31	32
0	15	16	31	32
Entropy Variable B ₂	48	63	64	79
48	63	64	79	80
Entropy Variable A ₃	15	16	31	32
0	15	16	31	32
Entropy Variable B ₃	48	63	64	79
48	63	64	79	80

‘Actual Size 1 Less’ and ‘Large Block’ are 0. ‘Block Size’ is 3 for mono blocks and 6 for stereo blocks. The samples themselves are converted and round-tripped the same way that ‘Decorrelation Sample’ values are, as explained on pages 99 and 74.

8.4.11 the Bitstream Sub-Block

Given a set of residual values and one set of 3 entropy variables per channel, the final encoding step for a WavPack block is generating the bitstream sub-block. Since the subframe header requires a size, we must either write it in advance with a size of 0 and rewrite it once the sub-block is finished, or first write the residuals to temporary space before writing the sub-block header.

Metadata Function (0xA) 0	4	0 5	Actual Size 1 Less 6	Large Block 7	Block Size 8	15/31
Residual Data 32						

As with decoding, writing each residual value is a multi-stage process which involves calculating a unary value, referencing and updating the channel's set of entropy variables, and calculating a fixed collection of bits.

However, this procedure is complicated by the 'holding_one' and 'holding_zero' boolean values. As you'll recall, when 'holding_zero' is false, the decoder skips reading a unary value entirely. This means that before we can output Residual_{i-1} 's values, we must determine the values for Residual_i so that the 'holding_one' and 'holding_zero' boolean values can be set properly. Note that holding_one_{-1} and holding_zero_{-1} are both **false**.

In practice, we'll first need to handle the special case of many zero residuals in a row as discussed on page 106. But for clarity, it's best to understand the general case first.

Calculate Unsigned Value and Sign Bit

$$\text{value}_i = \begin{cases} \text{Residual}_i & \text{if } \text{Residual}_i \geq 0 \\ -\text{Residual}_i - 1 & \text{if } \text{Residual}_i < 0 \end{cases}$$

$$\text{sign}_i = \begin{cases} 0 & \text{if } \text{Residual}_i \geq 0 \\ 1 & \text{if } \text{Residual}_i < 0 \end{cases}$$

Calculate Unary Value, High, Low and Next Medians

$$\text{Median}_{i\ 0} = \lfloor \text{Entropy}_{i\ 0} \div 2^4 \rfloor + 1$$

$$\text{Median}_{i\ 1} = \lfloor \text{Entropy}_{i\ 1} \div 2^4 \rfloor + 1$$

$$\text{Median}_{i\ 2} = \lfloor \text{Entropy}_{i\ 2} \div 2^4 \rfloor + 1$$

If $\text{value}_i < \text{Median}_{i\ 0}$:

$$\begin{aligned}
\text{unary}_i &= 0 \\
\text{low}_i &= 0 \\
\text{high}_i &= \text{Median}_{i\ 0} - 1 \\
\text{Entropy}_{(i+1)\ 0} &= \text{Entropy}_{i\ 0} - \left\lfloor \frac{\text{Entropy}_{i\ 0} + 126}{128} \right\rfloor \times 2 \\
\text{Entropy}_{(i+1)\ 1} &= \text{Entropy}_{i\ 1} \\
\text{Entropy}_{(i+1)\ 2} &= \text{Entropy}_{i\ 2}
\end{aligned}$$

If $(\text{value}_i - \text{Median}_{i\ 0}) < \text{Median}_{i\ 1}$:

$$\begin{aligned}
\text{unary}_i &= 1 \\
\text{low}_i &= \text{Median}_{i\ 0} \\
\text{high}_i &= \text{Median}_{i\ 0} + \text{Median}_{i\ 1} - 1 \\
\text{Entropy}_{(i+1)\ 0} &= \text{Entropy}_{i\ 0} + \left\lfloor \frac{\text{Entropy}_{i\ 0} + 128}{128} \right\rfloor \times 5 \\
\text{Entropy}_{(i+1)\ 1} &= \text{Entropy}_{i\ 1} - \left\lfloor \frac{\text{Entropy}_{i\ 1} + 62}{64} \right\rfloor \times 2 \\
\text{Entropy}_{(i+1)\ 2} &= \text{Entropy}_{i\ 2}
\end{aligned}$$

If $(\text{value}_i - (\text{Median}_{i\ 0} + \text{Median}_{i\ 1})) < \text{Median}_{i\ 2}$:

$$\begin{aligned}
\text{unary}_i &= 2 \\
\text{low}_i &= \text{Median}_{i\ 0} + \text{Median}_{i\ 1} \\
\text{high}_i &= \text{Median}_{i\ 0} + \text{Median}_{i\ 1} + \text{Median}_{i\ 2} - 1 \\
\text{Entropy}_{(i+1)\ 0} &= \text{Entropy}_{i\ 0} + \left\lfloor \frac{\text{Entropy}_{i\ 0} + 128}{128} \right\rfloor \times 5 \\
\text{Entropy}_{(i+1)\ 1} &= \text{Entropy}_{i\ 1} + \left\lfloor \frac{\text{Entropy}_{i\ 1} + 64}{64} \right\rfloor \times 5 \\
\text{Entropy}_{(i+1)\ 2} &= \text{Entropy}_{i\ 2} - \left\lfloor \frac{\text{Entropy}_{i\ 2} + 30}{32} \right\rfloor \times 2
\end{aligned}$$

Otherwise:

$$\begin{aligned}
 \text{unary}_i &= \left\lfloor \frac{\text{value}_i - (\text{Median}_{i\ 0} + \text{Median}_{i\ 1})}{\text{Median}_{i\ 2}} \right\rfloor + 2 \\
 \text{low}_i &= \text{Median}_{i\ 0} + \text{Median}_{i\ 1} + ((\text{unary}_i - 2) \times \text{Median}_{i\ 2}) \\
 \text{high}_i &= \text{low}_i + \text{Median}_{i\ 2} - 1 \\
 \text{Entropy}_{(i+1)\ 0} &= \text{Entropy}_{i\ 0} + \left\lfloor \frac{\text{Entropy}_{i\ 0} + 128}{128} \right\rfloor \times 5 \\
 \text{Entropy}_{(i+1)\ 1} &= \text{Entropy}_{i\ 1} + \left\lfloor \frac{\text{Entropy}_{i\ 1} + 64}{64} \right\rfloor \times 5 \\
 \text{Entropy}_{(i+1)\ 2} &= \text{Entropy}_{i\ 2} + \left\lfloor \frac{\text{Entropy}_{i\ 2} + 32}{32} \right\rfloor \times 5
 \end{aligned}$$

Calculate Fixed Size, Fixed Value and Optional Extra Bit

If $\text{low}_i = \text{high}_i$:

fixed size_i = 0
has extra_i = **false**

Otherwise:

$$\text{extras}_i = 2^{\text{count_bits}(\text{high}_i - \text{low}_i)} - (\text{high}_i - \text{low}_i) - 1$$

If $\text{low}_i \neq \text{high}_i$ and $(\text{value}_i - \text{low}_i) < \text{extras}_i$:

fixed size_i = value_i - low_i
fixed value_i = count_bits(high_i - low_i) - 1
has extra_i = **false**

If $\text{low}_i \neq \text{high}_i$ and $(\text{value}_i - \text{low}_i) \geq \text{extras}_i$:

fixed size_i = $\lfloor (\text{value}_i - \text{low}_i + \text{extras}_i) \div 2 \rfloor$
fixed value_i = count_bits(high_i - low_i) - 1
has extra_i = **true**
extra bit_i = $(\text{value}_i - \text{low}_i + \text{extras}_i) \bmod 2$

Update Previous Residual Based On Current Residual

If $\text{unary}_{i-1} > 0$ and $\text{unary}_i > 0$:

$$\text{unary}_{i-1} \leftarrow \begin{cases} (\text{unary}_{i-1} \times 2) + 1 & \text{if holding one}_{i-1} = \text{false} \\ (\text{unary}_{i-1} \times 2) - 1 & \text{if holding one}_{i-1} = \text{true} \end{cases}$$

holding zero_{*i*} = **false**
 holding one_{*i*} = **true**

If $\text{unary}_{i-1} = 0$ and $\text{unary}_i > 0$:

$$\text{unary}_{i-1} \leftarrow \begin{cases} 1 & \text{if holding zero}_{i-1} = \text{false} \\ \text{not output} & \text{if holding zero}_{i-1} = \text{true} \end{cases}$$

holding zero_{*i*} = **false**
 holding one_{*i*} = **not** holding zero_{*i-1*}

If $\text{unary}_{i-1} > 0$ and $\text{unary}_i = 0$:

$$\text{unary}_{i-1} \leftarrow \begin{cases} \text{unary}_{i-1} \times 2 & \text{if holding one}_{i-1} = \text{false} \\ (\text{unary}_{i-1} - 1) \times 2 & \text{if holding one}_{i-1} = \text{true} \end{cases}$$

holding zero_{*i*} = **true**
 holding one_{*i*} = **false**

If $\text{unary}_{i-1} = 0$ and $\text{unary}_i = 0$:

$$\text{unary}_{i-1} \leftarrow \begin{cases} 0 & \text{if holding zero}_{i-1} = \text{false} \\ \text{not output} & \text{if holding zero}_{i-1} = \text{true} \end{cases}$$

holding zero_{*i*} = **not** holding zero_{*i-1*}
 holding one_{*i*} = **false**

Output Previous Residual

Once the previous residual's unary value has been determined, its fields can now be output.

- If unary output, write unary_{i-1} number of 1 bits followed by a 0 bit (if $\text{unary}_{i-1} < 16$)
- Write 'fixed size_{*i-1*}' number of bits with the value 'fixed value_{*i-1*}'
- If 'has extra_{*i-1*}', write a single bit with the value 'extra bit_{*i-1*}'

- Write a single bit with the value ‘sign_{*i*-1}’

Note that if unary_{*i*-1} ≥ 16, we write an escape code instead.

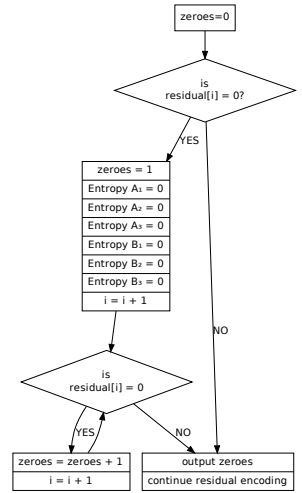
- If unary_{*i*-1} = 16, write 18 bits with the value 0xFFFF (unary 16 plus unary 0)
- If unary_{*i*-1} = 17, write 19 bits with the value 0x2FFFF (unary 16 plus unary 1)
- If unary_{*i*-1} ≥ 18, write 17 bits with the value 0xFFFF (unary 16)
 - Write count_bits(unary_{*i*-1} - 16) number of 1 bits followed by a 0 bit
 - Write count_bits(unary_{*i*-1} - 16) - 1 number of bits with the value ((unary_{*i*-1} - 16) mod 2^{count_bits(unary_{*i*-1} - 16) - 1})

Handle Groups Of Zero Residuals

This is necessary when Entropy A_{*i* 0} < 2 and, for 2 channel blocks, Entropy B_{*i* 0} < 2, holding_zero_{*i*} = **false** and holding_one_{*i*} = **false**. In that event, whether the current Residual_{*i*} is 0 or not, we must generate a zeroes block after outputting Residual_{*i*-1} but before outputting Residual_{*i*}.

Once the number of zeroes has been determined, their output is quite similar to an escaped unary value.

- If zeroes = 0, write a single 0 bit.
- If zeroes > 0
 - Write count_bits(zeroes) number of 1 bits followed by a 0 bit
 - Write count_bits(zeroes) - 1 number of bits with the value (zeroes mod 2^{count_bits(zeroes)-1})



8.4.12 Extended Integers

In the rare case that a block has ‘wasted bits’, as explained on page 92, we generate the following sub-block to store our ‘Zero Bits’ value:

0	Metadata Function (9)	4	0	Actual Size 1 Less	6	Large Block (0)	7	Block Size (2)	15
16	Sent Bits	23	24	Zero Bits	31	One Bits	39	Duplicate Bits	47

8.4.13 RIFF WAVE Header

WavPack expects to find a RIFF WAVE header sub-block in the first block within the file. This sub-block is laid out as follows:

0	Metadata Function (1)	4	Nondecoder Data (1)	5	Actual Size 1 Less	6	Large Block	7
8	Block Size	15/31	RIFF WAVE Header					
				16/32				

The RIFF WAVE header is everything from the start of a RIFF WAVE file to the end of its **data** chunk's header. For non WAVEFORMATEXTENSIBLE files, this is typically the first 36 bytes. For WAVEFORMATEXTENSIBLE files, this is typically the first 60 bytes.

8.4.14 the Footer Block

Though not required, WavPack files often contain a trailing block after the audio has been exhausted. This block contains only an MD5 sum sub-block and optional RIFF WAVE footer sub-block for wave files with additional chunks of data after the **data** chunk.

RIFF WAVE Footer

0	Metadata Function (2)	4	Nondecoder Data (1)	5	Actual Size 1 Less	6	Large Block	7
8	Block Size	15/31	RIFF WAVE Footer					
				16/32				

MD5 Sum

0	Metadata Function (6)	4	Nondecoder Data (1)	5	Actual Size 1 Less (0)	6	Large Block (0)	7
8	Block Size (16)	15	MD5 sum					
				16				271

The MD5 is the hash of all the samples over the entire file. It is calculated by running the hashing algorithm⁹ over the raw input samples in little-endian format and signed if their bits-per-sample are greater than 8.

⁹As described by RFC1321

9 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.

9.1 the Monkey's Audio File Stream

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

9.2 the Monkey's Audio Descriptor

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

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

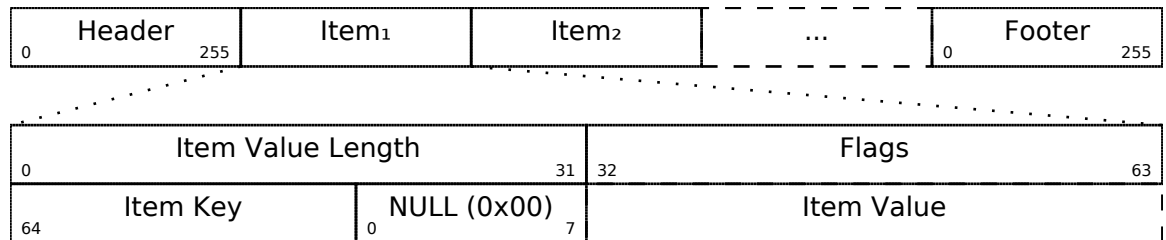
9.3 the Monkey's Audio header

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

$$\text{Length in Seconds} = \frac{((\text{Total Frames} - 1) \times \text{Blocks Per Frame}) + \text{Final Frame Blocks}}{\text{Sample Rate}} \quad (9.1)$$

9.4 the APEv2 Tag

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



'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.

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 characteristic 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

9.4.1 the APEv2 Tag Header/Footer

Preamble ('APETAGEX' 0x4150455441474558)				0	63
Version (0xD0070000)		95	Tag Size		127
64			96		
Item Count		Flags		Reserved	
128	159	160	191	192	255

The format of the APEv2 header and footer are identical except for the 'Is Header' tag. '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.

9.4.2 the APEv2 Flags

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

This flags field is used by both the APEv2 header/footer and the individual tag items. The 'Encoding' field indicates the encoding of its value:

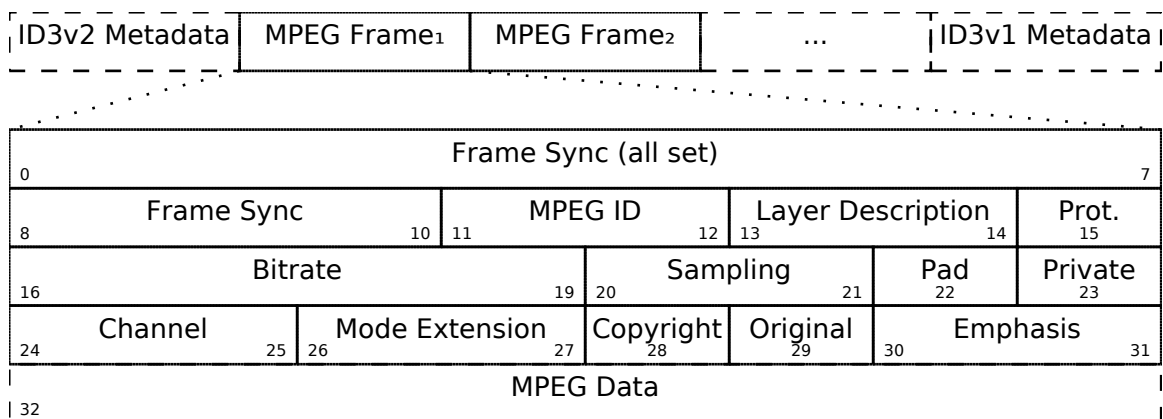
00 = UTF-8, 01 = Binary, 10 = External Link, 11 = Reserved .

10 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.

10.1 the MP3 File Stream



bits	MPEG ID	Description	Sample Rate			Channels
			MPEG-1	MPEG-2	MPEG-2.5	
00	MPEG-2.5	reserved	44100	22050	11025	Stereo
01	reserved	Layer III	48000	24000	12000	Joint stereo
10	MPEG-2	Layer II	32000	16000	8000	Dual channel stereo
11	MPEG-1	Layer I	reserved	reserved	reserved	Mono

Layer I frames always contain 384 samples. Layer II and Layer III frames always contain 1152 samples. If the 'Protection' bit is 0, the frame header is followed by a 16 bit CRC.

bits	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

Table 10.1: Bitrate in 1000 bits per second

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

Layer I:

$$\text{Byte Length} = \left(\frac{12 \times \text{Bitrate}}{\text{Sample Rate}} + \text{Pad} \right) \times 4 \quad (10.1)$$

Layer II/III:

$$\text{Byte Length} = \frac{144 \times \text{Bitrate}}{\text{Sample Rate}} + \text{Pad} \quad (10.2)$$

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.

$$\frac{144 \times 128000}{44100} + 1 = 418 \quad (10.3)$$

10.1.1 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.

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

10.2 ID3v1 Tags

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.

10.2.1 ID3v1

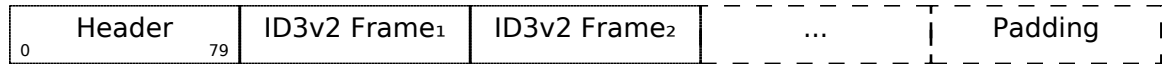
Header (`TAG' 0x544147)		Track Title		Artist Name		Album Name	
0	23	24	263	264	503	504	743
Year		Comment					
744	775	776					1015
Genre							
1016							1023

10.2.2 ID3v1.1

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

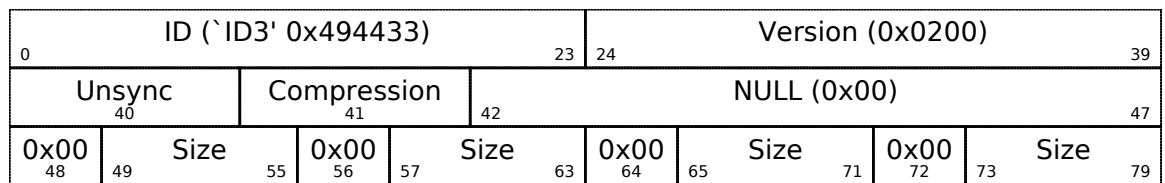
10.3 ID3v2 Tags

The ID3v2 tag was invented to address the deficiencies in the original ID3v1 tag. ID3v2 comes in three similar but not entirely compatible variants: ID3v2.2, ID3v2.3 and ID3v2.4. All of its fields are big-endian.



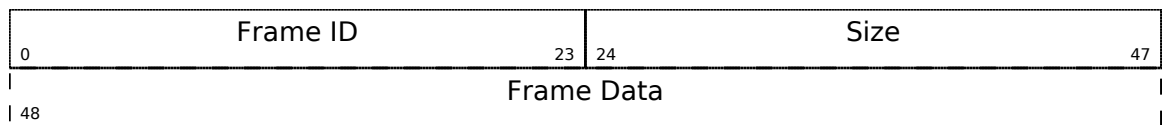
10.3.1 ID3v2.2

ID3v2.2 Header



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). This size field is the length of the entire tag, not including the header.

ID3v2.2 Frame



Frame IDs 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 may be terminated by a null character (0x00 or 0x0000, depending on the encoding).

Frame ID `TXX' (0x54XXXX)										Size														
0					23					24					47									
48					55					56Text														
Encoding Byte					Text Encoding																			
0x00					ISO-8859-1																			
0x01					UCS-16																			

ID3v2.2 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.

Frame ID `PIC' (0x504943)										Size									
0					23					24					47				
Text Encoding					55					Image Format					79				
48					56														
Picture Type					87					Description									
80					88														
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. 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.

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 colored fish
18	Illustration	19	Band / Artist logotype
20	Publisher / Studio logotype		

Table 10.2: PIC image types

ID3v2.2 Frame IDs

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

10.3.2 ID3v2.3

ID3v2.3 Header

ID ('ID3' 0x494433)										Version (0x0300)																			
Unsync										Extended					Experimental					Footer					NULL (0x00)				
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'. This size field is the length of the entire tag, not including the header.

ID3v2.3 Frame

Frame ID										Size									
031										3263									
Tag Alter			File Alter			Read Only			NULL (0x00)										
64			65			66			6771										
Compression			Encryption			Grouping			NULL (0x00)										
72			73			74			7579										
Frame Data																			
80-----																			

Frame IDs 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 may be terminated by a null character (0x00 or 0x0000, depending on the encoding).

Frame ID 'TXXX' (0x54XXXXXX)										Size									
031										3263									
Tag Alter			File Alter			Read Only			NULL (0x00)										
64			65			66			67			71							
Compression			Encryption			Grouping			NULL (0x00)										
72			73			74			75			79							
Encoding			Text																
80			87			80													
Encoding Byte			Text Encoding																
0x00			ISO-8859-1																
0x01			UCS-16																

ID3v2.3 APIC Frame

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

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 colored fish
18	Illustration	19	Band / Artist logotype
20	Publisher / Studio logotype		

Table 10.3: APIC image types

ID3v2.3 Frame IDs

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

10.3.3 ID3v2.4**ID3v2.4 Header**

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

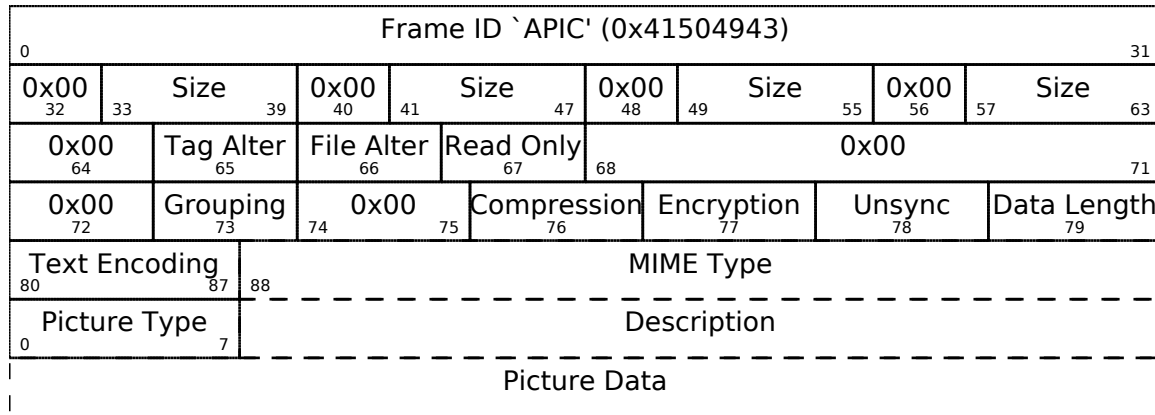
ID3v2.4 Frame

Frame ID																31
0	0x00 32		Size 33 39		0x00 40		41 Size 47		0x00 48		49 Size 55		0x00 56		57 Size 63	
0x00 64			Tag Alter 65		File Alter 66		Read Only 67		68		0x00 71					
0x00 72			Grouping 73		0x00 74 75		Compression 76		Encryption 77		Unsync 78		Data Length 79			
Frame Data																
80																

Frame IDs 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 may be terminated by a null character (0x00 or 0x0000, depending on the encoding).

Frame ID 'TXXX' (0x54XXXXXX)																31
0x00 32	Size 39		0x00 40	Size 47		0x00 48	Size 55		0x00 56	Size 63						
0x00 64		Tag Alter 65	File Alter 66		Read Only 67		0x00 71									
0x00 72		Grouping 73	0x00 74		Compression 76		Encryption 77		Unsync 78		Data Length 79					
Encoding 80			Text 80													
Encoding Byte			Text Encoding													
0x00			ISO-8859-1													
0x01			UTF-16													
0x02			UTF-16BE													
0x03			UTF-8													

ID3v2.4 APIC Frame



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.

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 colored fish
18	Illustration	19	Band / Artist logotype
20	Publisher / Studio logotype		

Table 10.4: APIC image types

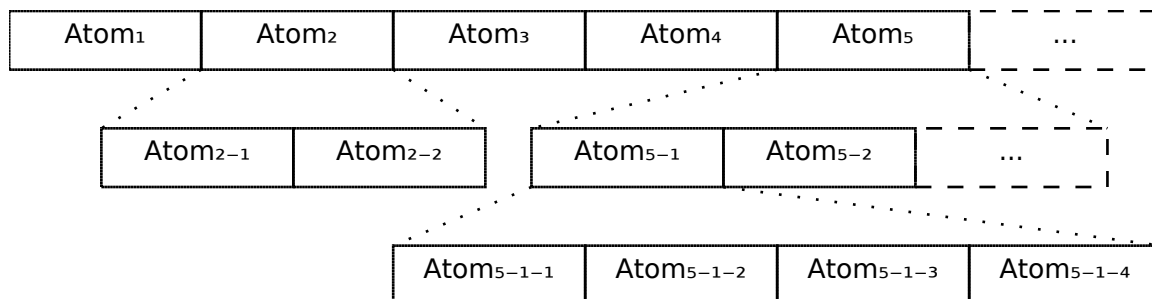
ID3v2.4 Frame IDs

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

11 M4A

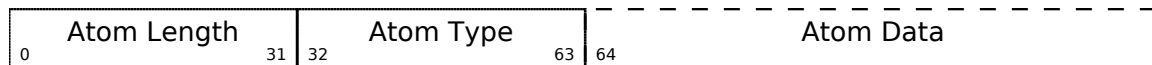
M4A is typically AAC audio in a QuickTime container stream, though it may also contain other formats such as MPEG-1 audio.

11.1 the QuickTime File Stream

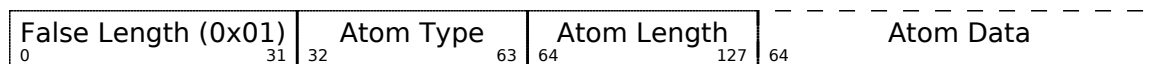


Unlike other chunked formats such as RIFF WAVE, QuickTime's atom chunks may be containers for other atoms. All of its fields are big-endian.

11.1.1 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.



11.1.2 Container Atoms

There is no flag or field to tell a QuickTime parser which of its atoms are containers and which ones are not. 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.

11.2 M4A Atoms

A typical M4A begins with an ‘ftyp’ atom indicating its file type, followed by a ‘moov’ atom containing a copious amount of file metadata, an optional ‘free’ atom with nothing but empty space (so that metadata can be resized, if necessary) and an ‘mdat’ atom containing the song data itself.

11.2.1 the ftyp Atom

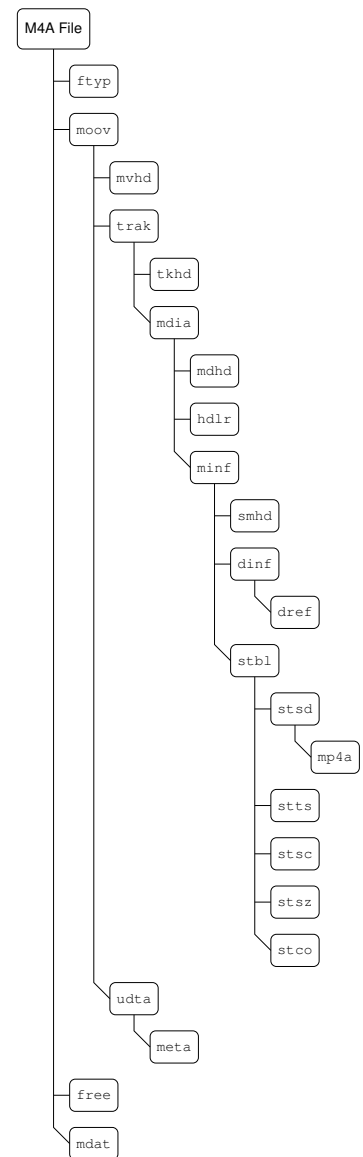
0	ftyp Length	31	32	`ftyp' (0x66747970)	63
64	Major Brand	95	96	Major Brand Version	127
128	Compatible Brand	159	160	...	

The ‘Major Brand’ and ‘Compatible Brand’ fields are ASCII strings. ‘Major Brand Version’ is an integer.

11.2.2 the mvhd Atom

0	mvhd Length	31	32	`mvhd' (0x6D766864)	63
64	Version	71	72	Flags (0x000000)	95
96	Created Mac UTC Date	127/159	128/160	Modified Mac UTC Date	159/223
160/224	Time Scale	191/255	192/256	Duration	223/319
224/320	Playback Speed	255/351	256/352	User Volume	271/367
				Reserved (0)	351/447
352/448	WGM A	383/479	384/480	WGM B	415/511
				WGM U	447/543
				WGM C	479/575
480/576	WGM D	511/607	512/608	WGM V	543/639
				WGM X	575/671
				WGM Y	607/703
608/704	WGM W	639/735	640/736	QuickTime Preview	703/799
704/800	QuickTime Still Poster	735/831	736/832	QuickTime Selection Time	799/895
800/896	QuickTime Current Time	831/927	832/928	next/new track ID	863/959

If ‘Version’ is 0, ‘Created Mac UTC Date’, ‘Modified Mac UTC Date’ and ‘Duration’ are 32-bit fields. If it is 1, they are 64-bit fields.



11.2.3 the tkhd Atom

tkhd Length				`tkhd' (0x746B6864)			
0			31	32			63
Version	Reserved (0)	Track in Poster	Track in Preview	Track in Movie	Track Enabled		
64	71	72	91	92	93	94	95
Created Mac UTC Date				Modified Mac UTC Date			
96			127/159	128/160			159/223
Track ID		Reserved (0)					
160/224	191/255	192/256					255/319
Duration				Reserved (0)			
256/320			287/383	288/384			319/415
Video Layer		QuickTime Alt		Audio Volume		Reserved (0)	
320/415	335/431	336/432	351/447	352/448	367/463	368/464	383/479
VGM value A		VGM value B		VGM value U		VGM value C	
384/480	415/511	416/512	447/543	448/544	479/575	480/576	511/607
VGM value D		VGM value V		VGM value X		VGM value Y	
512/608	543/639	544/640	575/671	576/672	607/703	608/704	639/735
VGM value W		Video Width			Video Height		
640/736	671/767	672/768		703/799	704/800		735/831

As with ‘mvhd’, if ‘Version’ is 0, ‘Created Mac UTC Date’, ‘Modified Mac UTC Date’ and ‘Duration’ are 32-bit fields. If it is 1, they are 64-bit fields.

11.2.4 the mdhd Atom

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

mdhd Length				`mdhd' (0x6D646864)			
0			31	32			63
Version		Flags (0x000000)					
64	71	72					95
Created Mac UTC Date				Modified Mac UTC Date			
96			127/159	128/160			159/223
Sample Rate				Track Length			
160/224			191/255	192/256			223/319
Pad		Language		Quality			
224/320	225/321	239/335	240/336				255/351

As with ‘mvhd’, if ‘Version’ is 0, ‘Created Mac UTC Date’, ‘Modified Mac UTC Date’ and ‘Track Length’ are 32-bit fields. If it is 1, they are 64-bit fields.

11.2.5 the hdlr Atom

hdlr Length			31	`hdlr' (0x68646C72)			63	
Version		71	72	Flags (0x000000)				95
QuickTime type			127	Subtype/media type			159	
Quicktime manufacturer								191
QuickTime flags			223	Quicktime flags mask			255	
Component Name Length		263	264	Component Name				

‘QuickTime flags’, ‘QuickTime flags mask’ and ‘Component Name Length’ are integers. The rest are ASCII strings.

11.2.6 the smhd Atom

smhd Length				`smhd' (0x736D6864)			
0		31		32		63	
Version		Flags (0x000000)		Audio Balance		Reserved (0x0000)	
64	71	72	95	96	111	112	127

11.2.7 the dref Atom

0				dref Length				31				32				`dref' (0x64726566)				63															
64				Version				71				72				Flags (0x000000)				95				96				Number of References				127			
128				Reference Atom ₁								Reference Atom ₂								...															

11.2.8 the stsd Atom

stsd Length		`stsd' (0x73747364)	
0	31	32	63
Version		Flags (0x000000)	
64	71	72	95
Description Atom ₁		Description Atom ₂	
128			
Description Atom ₁		Description Atom ₂	
128			

11.2.9 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.

mp4a Length		`mp4a' (0x6D703461)	
0	31	32	63
Reserved (0x000000000000)		Reference Index	
64	111	112	127
QuickTime Version		QuickTime Revision Level	
128	143	144	159
QuickTime Audio Encoding Vendor			
160			191
Channels		Bits per Sample	
192	207	208	223
QuickTime Compression ID		Audio Packet Size	
224	239	240	255
Audio Sample Rate		`esds' atom	
256	287	288	
.....			
esds Length		`esds' (0x65736473)	
0	31	32	63
Version		Flags (0x000000)	
64	71	72	95
ESDS Atom Data			
96			

11.2.10 the stts Atom

0		stts Length		31	32		`stts' (0x73747473)		63					
64		Version		71	72		Flags (0x000000)		95	96		Number of Times		127
128		Frame Count ₁		159		160		Duration ₁		191				
192		Frame Count ₂		223		224		Duration ₂		255				
...														
256														

11.2.11 the stsc Atom

stsc Length				`stsc' (0x73747363)			
0	31			32	63		
Version		Flags (0x000000)		Number of Blocks			
64	71	72	95	96	127		
First Chunk ₁		Samples per Chunk ₁		Sample Duration Index ₁			
128	159	160	191	192	223		
First Chunk ₂		Samples per Chunk ₂		Sample Duration Index ₂			
224	255	256	287	288	319		
...							
320							

11.2.12 the stsz Atom

stsz Length			`stsz' (0x7374737A)		
0	31	32	63		
Version		Flags (0x000000)		Number of Block Sizes	
64	71	72	95	96	127
Block Size ₁		Block Size ₂		...	
128	159	160	191	192	

11.2.13 the stco Atom

stco Length				`stco' (0x7374636F)			
0	31	32	63				
Version		Flags (0x000000)		Number of Offsets			
64	71	72	95	96	127		
Offset ₁		Offset ₂		...			
128	159	160	191	192	---	---	---

Offsets point to an absolute position in the M4A file of AAC data in the `mdat` atom. Therefore, if the `moov` atom size changes (which can happen by writing new metadata in its `meta` child atom) the `mdat` atom may move and these absolute offsets will change. In that instance, they **must** be re-adjusted in the `stco` atom or the file may become unplayable.

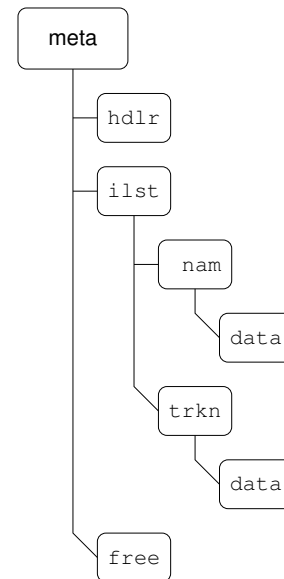
11.2.14 the meta Atom

meta Length				`meta' (0x6D657461)			
0	31	32	63				
Version		Flags (0x000000)					
64	71	72	95				
`hdlr' atom		`ilst' atom		`free' atom		...	
96							

The atoms within the `ilst` container are all containers themselves, each with a `data` atom of its own. Notice that many of `ilst`'s sub-atoms begin with the non-ASCII 0xA9 byte.

data Length				`data' (0x64617461)			
0	31	32	63				
Type		Reserved (0x00000000)					
64	95	96	127				
Data							
128							

Text data atoms have a 'Type' of 1. Binary data atoms typically have a 'Type' of 0.

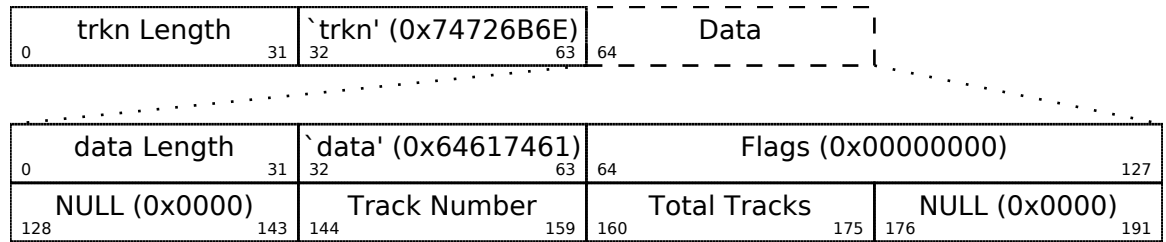


Atom	Description	Atom	Description	Atom	Description
alb	Album Name	ART	Track Artist	cmt	Comments
covr	Cover Image	cpil	Compilation	cpri	Copyright
day	Year	disk	Disc Number	gnre	Genre
grp	Grouping	----	iTunes-specific	nam	Track Name
rtng	Rating	tmpo	BPM	too	Encoder
trkn	Track Number	wrt	Composer		

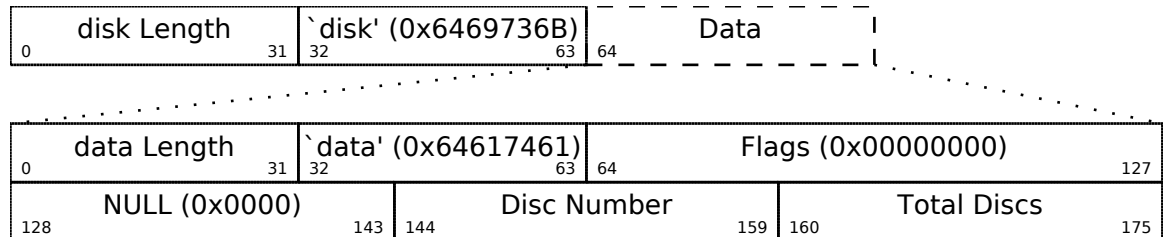
Table 11.1: Known `ilst` sub-atoms

the trkn Sub-Atom

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

**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.



12 Apple Lossless

Apple's lossless audio codec, informally referred to as "ALAC", is lossless audio inside a QuickTime container - similar to M4A. Its stream is the same collection of atoms as covered on page 125. The key difference is the contents of its `mdat` atom.

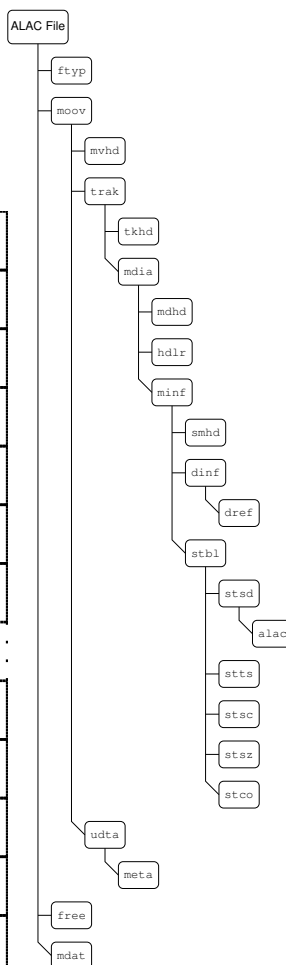
12.1 the ALAC File Stream

This is the typical arrangements of ALAC atoms as encoded by iTunes. As you can see, it is almost identical to the layout of AAC audio. One of the key differences is that ALAC's `stsd` atom contains an `alac` description sub-atom rather than an `mp4a` description atom.

Its layout is as follows:

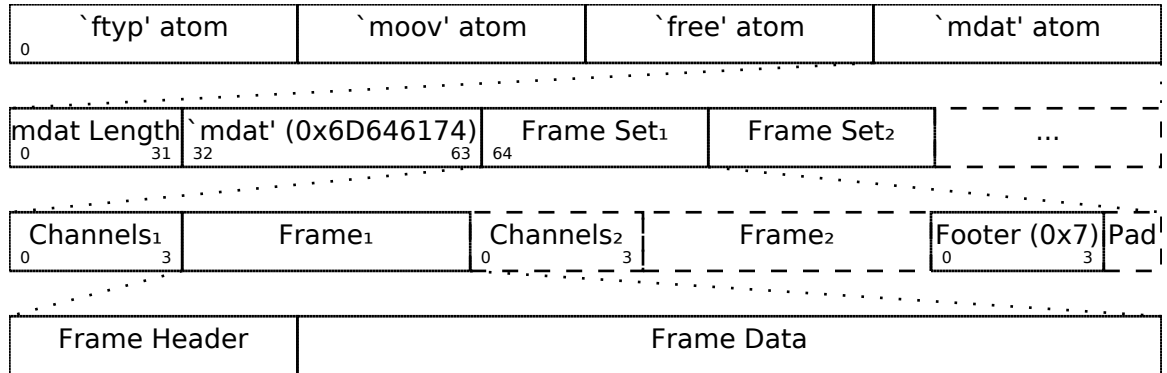
0	alac Length	31	32	`alac' (0x616C6163)	63
64	Reserved (0x000000000000)	111	112	Reference Index	127
128	QuickTime Version	143	144	QuickTime Revision Level	159
160	QuickTime Audio Encoding Vendor				191
192	Channels	207	208	Bits per Sample	223
224	QuickTime Compression ID	239	240	Audio Packet Size	255
256	Audio Sample Rate	287	288	`alac' atom	

0	alac Length	31	32	`alac' (0x616C6163)	63
64	Padding (0x00000000)	95	96	Max Samples Per Frame	127
128	Padding (0)	135	136	Bits per Sample	143
152	Initial History	159	160	Maximum K	167
168	Channels	175	176	Unknown	191
192	Max Coded Frame Size	223	224	Bitrate	255
256	Sample Rate				287



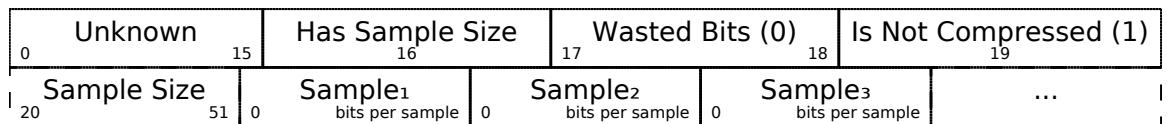
12.2 ALAC Decoding

An ALAC stream is made up of individual framesets within the `mdat` atom, as follows:



Note that the channels field is the total number of channels, offset by 1; 0 is mono and 1 is stereo. The frames within a frameset continue until the value 0x7 is encountered in the channel field. Each frameset will contain only a single frame in most cases. However, multichannel audio may contain more. For example, a 6 channel stream may have framesets with 4 frames each where Frame₁ has channels 1-2, Frame₂ has channel 3, Frame₃ has channel 4 and Frame₄ has channel 5-6.

ALAC frames come in two varieties: compressed and uncompressed, depending on the 'Is Not Compressed' bit in the frame header. An uncompressed frame is laid out as follows:



The number of PCM frames in an ALAC frame depends on the 'Has Sample Size' bit. If set, the number of PCM frames equals the 32-bit 'Sample Size' value. If not set, the total number of PCM frames equals the 'Max Coded Frame Size' value in the `alac` atom and the 'Sample Size' value is omitted. ALAC streams typically use the same number of samples per frame until the end of the stream, at which point the leftover samples are placed in a different-sized frame.

Uncompressed frames interleave samples between channels during decoding. For example, a 2 channel frame places Sample₁ on Channel₁, Sample₂ on Channel₂, Sample₃ on Channel₁, Sample₄ on Channel₂ and so on.

Finally, note that all ALAC framesets have padding as needed such that each new frameset begins on an aligned byte boundary.

A compressed frame is laid out as follows:

Unknown		Has Sample Size		Wasted Bits (0)		Is Not Compressed (1)	
0	15	16	17	18	19		
Sample Size		Interlacing Shift			Interlacing Leftweight		
20	51	0	7	8			15
Subframe Header ₁				Subframe Header ₂			
Wasted Bits Sample ₁		Wasted Bits Sample ₂		Wasted Bits Sample ₃		...	
Residual Block ₁				Residual Block ₂			

‘Interlacing Shift’ and ‘Interlacing Leftweight’ are used for channel decorrelation after the subframes have been decoded.

There is one subframe header and one residual block per channel. If ‘Wasted Bits’ is greater than 0, there is also a block of wasted bits samples after the subframe headers but before the residuals.

‘Wasted Bits’ is an attempt to store the least significant bits of each sample more efficiently at high bits-per-sample, where that data will often be indistinguishable from random noise. In effect, it’s a block of interlaced uncompressed samples (similar to an uncompressed ALAC frame) each $8 \times$ ‘Wasted Bits’ bits large. Those wasted bits are then prepended to each sample after channel decorrelation. This process is explained in more detail on page 142.

Each subframe header is laid out as follows:

0	Prediction Type	3	4	Prediction Quantitization	7
8	Rice Modifier	10	11	Coefficient Count	15
16	Coefficient ₁	31	32	Coefficient ₂	47
...					

There are ‘Coefficient Count’ number of coefficients in each subframe header, each a 16-bit signed value.

12.2.1 Residual Decoding

There are ‘Sample Size’ number of residuals per residual block. Decoding a residual block requires knowing the ‘Initial History’, ‘History Multiplier’, ‘Maximum K’, ‘Channels’, ‘Wasted Bits’ and ‘Bits per Sample’ values, from the **alac** atom and frame header. Fortunately, most of these values are rarely used; we’ll mostly be concerned with the ‘History’ and ‘History Multiplier’.

For each residual block, ‘History’ starts with the value of ‘Initial History’ and will change during residual decoding. We use it to calculate κ using the following formula:

$$\kappa = \left\lfloor \log_2 \left(\frac{\text{history}}{2^9} + 3 \right) \right\rfloor \quad (12.1)$$

Note that if κ exceeds the ‘Maximum K’ value from the **alac** atom, ‘Maximum K’ is used instead.

We then need to know the ‘Bits per Sample’ value of the residual block, which is equal to:

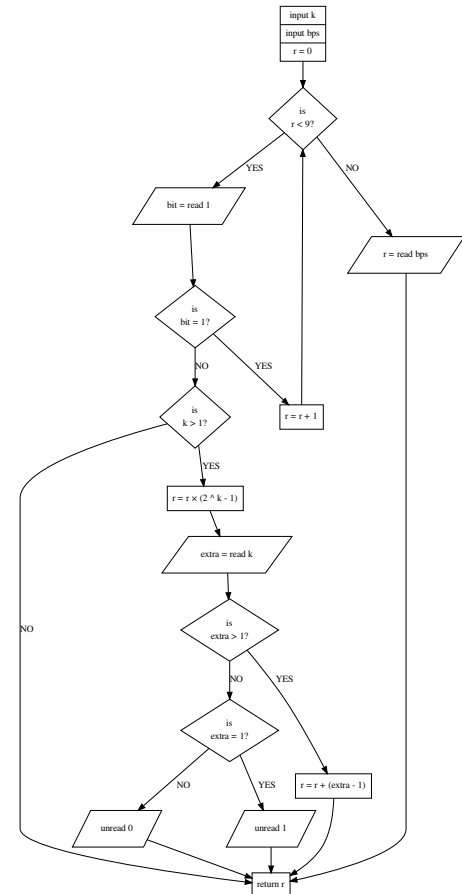
$$\text{Bits per Sample}_{\text{Residual}} = \text{Bits per Sample}_{\text{ALAC}} - (\text{Wasted Bits} \times 8) + \text{Channels} - 1 \quad (12.2)$$

For mono streams, this is typically equal to the stream’s ‘Bits per Sample’, while for stereo streams it’s often the ‘Bits per Sample’ plus 1.

The κ and ‘Bits per Sample’ values are used to read a single unsigned residual value in the following way:

The initial bit reading portion of the process involves reading a unary value with a stop bit of ‘0’ and a maximum value of 8. If the maximum value is exceeded, we read ‘Bits per Sample’ number of bits as our final value (the only place ‘Bits per Sample’ is used throughout the residual decoding process).

Otherwise, we read κ number of ‘extra’ bits if $\kappa > 1$. If ‘extra’ is greater than 1, we return $(\text{‘unary’} \times (2^k - 1)) + (\text{‘extra’} - 1)$. If not, we push a single ‘extra’ bit back on the stream and return our $\text{‘unary’} \times (2^k - 1)$ value.



We perform the following to convert unsigned residuals to signed residuals which are returned to the subframe decoder:

$$\text{signed} = \begin{cases} \lfloor (\text{unsigned} + 1) \div 2 \rfloor & \text{if unsigned is even} \\ -\lfloor (\text{unsigned} + 1) \div 2 \rfloor & \text{if unsigned is odd} \end{cases} \quad (12.3)$$

Finally, we use our unsigned value to update ‘history’ before reading the next residual:

$$\text{history} = \begin{cases} \text{history} + (\text{unsigned} \times \text{history multiplier}) - \lfloor \frac{\text{history} \times \text{history multiplier}}{2^9} \rfloor & \text{if unsigned} \leq 65535 \\ 65535 & \text{if unsigned} > 65535 \end{cases}$$

Thus far, residual decoding isn’t overly complex. We simply calculate κ from ‘history’, read a residual, update ‘history’ from its unsigned value and repeat the process until we’ve read an entire subframe’s worth of residuals. The bulk of an ALAC file’s residuals will be read in this way. However, ALAC also features an “escape code” for large chunks of 0 residuals (which may happen during a long stretch of digital silence, for example).

If ‘history’ ever falls below 128, this special case is triggered. First, we read a special ‘block size’ residual value with a ‘Bits per Sample’ value of 16 and a κ^1 value of:

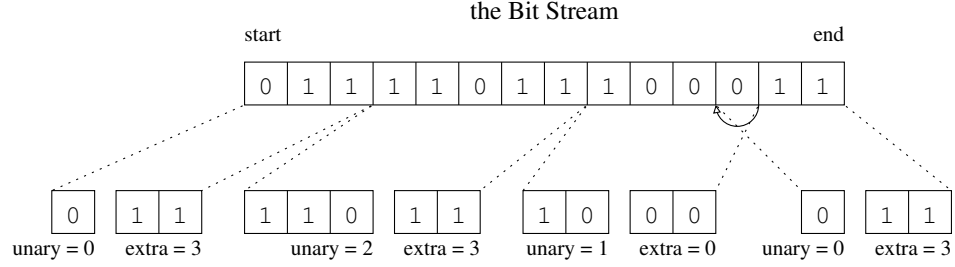
$$\kappa_{\text{blocksize}} = 7 - \log_2(\text{history}) + \frac{\text{history} + 16}{64} \quad (12.4)$$

This ‘block size’ is how many 0 residuals to send outright - which may be 0, indicating no 0 residuals to send. Either way, if ‘block size’ ≤ 65535 , we add 1 to the unsigned value of the next residual in the block (if any). Finally, ‘history’ is automatically set to 0.

¹Again, κ cannot exceed ‘Maximum K’ as encoded in the `alac` atom.

Residual Decoding Example

In this example, we'll decode a group of residuals in which our 'Initial History' is 1130 and our 'History Multiplier' is 40. As a spoiler, κ (the amount of non-unary bits to read after each unary value) will remain 2 for this batch of residuals, but why that is so will be explained below.



- Residual 1
 - $\kappa = \lfloor \log_2((1130 \div 2^9) + 3) \rfloor = \lfloor \log_2(5) \rfloor = 2$
 - $\text{unsigned}_1 = (0 \times (2^2 - 1)) + (3 - 1) = 2$
 - $\text{residual}_1 = \lfloor (2 + 1) \div 2 \rfloor = 1$
 - $\text{history} = 1130 + (2 \times 40) - \lfloor (1130 \times 40) \div 2^9 \rfloor = 1130 + 80 - 88 = 1122$
- Residual 2
 - $\kappa = \lfloor \log_2((1122 \div 2^9) + 3) \rfloor = \lfloor \log_2(5) \rfloor = 2$
 - $\text{unsigned}_2 = (2 \times (2^2 - 1)) + (3 - 1) = 8$
 - $\text{residual}_2 = \lfloor (8 + 1) \div 2 \rfloor = 4$
 - $\text{history} = 1122 + (8 \times 40) - \lfloor (1122 \times 40) \div 2^9 \rfloor = 1122 + 320 - 87 = 1355$
- Residual 3
 - $\kappa = \lfloor \log_2((1355 \div 2^9) + 3) \rfloor = \lfloor \log_2(5) \rfloor = 2$
 - $\text{unsigned}_3 = (1 \times (2^2 - 1)) = 3$ (note that we “unread” one 0 bit here)
 - $\text{residual}_3 = -\lfloor (3 + 1) \div 2 \rfloor = -2$
 - $\text{history} = 1355 + (3 \times 40) - \lfloor (1355 \times 40) \div 2^9 \rfloor = 1355 + 120 - 105 = 1370$
- Residual 4
 - $\kappa = \lfloor \log_2((1370 \div 2^9) + 3) \rfloor = \lfloor \log_2(5) \rfloor = 2$
 - $\text{unsigned}_4 = (0 \times (2^2 - 1)) + (3 - 1) = 2$
 - $\text{residual}_4 = \lfloor (2 + 1) \div 2 \rfloor = 1$
 - $\text{history} = 1370 + (2 \times 40) - \lfloor (1370 \times 40) \div 2^9 \rfloor = 1370 + 80 - 107 = 1343$

Thus, our batch of signed residual values are 1, 4, -2 and 1.

12.2.2 Subframe Calculation

Given our list of decoded residual values; along with a list of coefficients, a ‘Coefficient Count’ and a ‘Prediction Quantitization’ value (all from the subframe header), we can now generate a list of signed subframe samples for a given channel.

The first residual is always the first output sample:

$$\text{Sample}_0 = \text{Residual}_0$$

Then, for the next ‘Coefficient Count’ number of residuals:

$$\text{Sample}_i = \text{Residual}_i + \text{Sample}_{i-1}$$

For example, given that we have a ‘Coefficient Count’ of 4 and our first five residuals are 16, 1, 9, -1 and -1; our first five sample values are:

$$\begin{aligned} \text{Sample}_0 &= \mathbf{16} \\ \text{Sample}_1 &= 16 + 1 = \mathbf{17} \\ \text{Sample}_2 &= 17 + 9 = \mathbf{26} \\ \text{Sample}_3 &= 26 - 1 = \mathbf{25} \\ \text{Sample}_4 &= 25 - 1 = \mathbf{24} \end{aligned}$$

These are our “starting point” samples upon which the remainder of the subframe will be built.

Subsequent samples are calculated in the following way:

$$\begin{aligned} \text{LPC Sum}_i &= \sum_{j=0}^{\text{coeffs}-1} \text{Coefficient}_j \times (\text{Sample}_{i-j-1} - \text{Sample}_{i-\text{coeffs}-1}) \\ \text{Sample}_i &= \left\lfloor \frac{\text{LPC Sum}_i + 2^{\text{Predictor Quantitization} - 1}}{2^{\text{Predictor Quantitization}}} \right\rfloor + \text{Residual}_i + \text{Sample}_{i-\text{coeffs}-1} \end{aligned}$$

For example, given $\text{Residual}_5 = 1$, ‘Predictor Quantitization’ = 9 and the coefficients 1122, -766, 107 and 122:

$$\begin{aligned} \text{LPC Sum}_5 &= (1122 \times (24 - 16)) + (-766 \times (25 - 16)) + (107 \times (26 - 16)) + (122 \times (17 - 16)) \\ &= (1122 \times 8) + (-766 \times 9) + (107 \times 10) + (122 \times 1) \\ &= 8976 + -6894 + 1070 + 122 = \mathbf{3274} \\ \text{Sample}_5 &= \left\lfloor \frac{\mathbf{3274} + 2^8}{2^9} \right\rfloor + 1 + 16 = \left\lfloor \frac{3530}{512} \right\rfloor + 1 + 16 = \mathbf{23} \end{aligned}$$

But before calculating Sample_6 , we need to adjust our coefficient list.

Updating the coefficient list first requires the ‘sign’ function:

$$\text{sign}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases} \quad (12.5)$$

We then take our ‘samples’, index i , ‘coefficients’, coefficient ‘count’, ‘residual’, ‘predictor quantitization’ values and go through the following routine if residual $\neq 0$:

Step 1.	original sign	=	sign(residual)
Step 2.	j	=	0
Step 3.	v	=	Sample _{$i-\text{count}-1$} – Sample _{$i-\text{count}+j$}
Step 4.	$sign$	=	$\begin{cases} \text{sign}(v) & \text{if original sign} > 0 \\ -\text{sign}(v) & \text{if original sign} \leq 0 \end{cases}$
Step 5.	Coefficient _{count-j-1}	=	Coefficient _{count-j-1} – $sign$
Step 6.	$residual$	=	$residual - \left(\left\lfloor \frac{v \times sign}{2^{\text{Predictor Quantitization}}} \right\rfloor \times (j + 1) \right)$
Step 7.	j	=	$j + 1$
Step 8.	if ($j < \text{count}$) and ($\text{sign}(residual) = \text{original sign}$), goto step 3		

To continue our example:

$$\begin{aligned}
 \text{original sign} &= \text{sign}(1) = \mathbf{1} \\
 j &= 0 \\
 v &= \text{Sample}_0 - \text{Sample}_1 = 16 - 17 = \mathbf{-1} \\
 sign &= \text{sign}(-1) = \mathbf{-1} \\
 \text{Coefficient}_3 &= \text{Coefficient}_3 - -1 = 122 + 1 = \mathbf{123} \\
 residual &= residual - \left(\left\lfloor \frac{-1 \times -1}{2^9} \right\rfloor \times (0 + 1) \right) = 1 - (0 \times 1) = \mathbf{1} \\
 j &= 1 \\
 v &= \text{Sample}_0 - \text{Sample}_2 = 16 - 26 = \mathbf{-10} \\
 sign &= \text{sign}(-10) = \mathbf{-1} \\
 \text{Coefficient}_2 &= \text{Coefficient}_2 - -1 = 107 + 1 = \mathbf{108} \\
 residual &= residual - \left(\left\lfloor \frac{-10 \times -1}{2^9} \right\rfloor \times (1 + 1) \right) = 1 - (0 \times 2) = \mathbf{1} \\
 j &= 2 \\
 v &= \text{Sample}_0 - \text{Sample}_3 = 16 - 25 = \mathbf{-9} \\
 sign &= \text{sign}(-9) = \mathbf{-1} \\
 \text{Coefficient}_1 &= \text{Coefficient}_1 - -1 = -766 + 1 = \mathbf{-765} \\
 residual &= residual - \left(\left\lfloor \frac{-9 \times -1}{2^9} \right\rfloor \times (2 + 1) \right) = 1 - (0 \times 3) = \mathbf{1} \\
 j &= 3 \\
 v &= \text{Sample}_0 - \text{Sample}_4 = 16 - 25 = \mathbf{-8} \\
 sign &= \text{sign}(-8) = \mathbf{-1} \\
 \text{Coefficient}_0 &= \text{Coefficient}_0 - -1 = 1122 + 1 = \mathbf{1123} \\
 residual &= residual - \left(\left\lfloor \frac{-8 \times -1}{2^9} \right\rfloor \times (3 + 1) \right) = 1 - (0 \times 4) = \mathbf{1} \\
 j &= 4 \text{ and stop, since 4 equals our coefficient count}
 \end{aligned}$$

Given that Residual_6 is -4, the calculation of Sample_6 is as follows:

$$\begin{aligned}\text{LPC Sum}_6 &= (1123 \times (23 - 17)) + (-765 \times (24 - 17)) + (108 \times (25 - 17)) + (123 \times (26 - 17)) \\ &= (1123 \times 6) + (-765 \times 7) + (108 \times 8) + (123 \times 9) \\ &= 6738 + -5355 + 864 + 1107 = \mathbf{3354}\end{aligned}$$

$$\text{Sample}_6 = \left\lfloor \frac{\mathbf{3354} + 2^8}{2^9} \right\rfloor + -4 + 17 = \left\lfloor \frac{3610}{512} \right\rfloor + -4 + 17 = \mathbf{20}$$

$$\text{original sign} = \text{sign}(-4) = -1$$

$$j = 0$$

$$v = \text{Sample}_1 - \text{Sample}_2 = 17 - 26 = \mathbf{-9}$$

$$\text{sign} = -\text{sign}(-9) = \mathbf{1}$$

$$\text{Coefficient}_3 = \text{Coefficient}_3 - 1 = 123 - 1 = \mathbf{122}$$

$$\text{residual} = \text{residual} - \left(\left\lfloor \frac{-9 \times 1}{2^9} \right\rfloor \times (0 + 1) \right) = -4 - (-1 \times 1) = \mathbf{-3}$$

$$j = 1$$

$$v = \text{Sample}_1 - \text{Sample}_3 = 17 - 25 = \mathbf{-8}$$

$$\text{sign} = -\text{sign}(-8) = \mathbf{1}$$

$$\text{Coefficient}_2 = \text{Coefficient}_2 - 1 = 108 - 1 = \mathbf{107}$$

$$\text{residual} = \text{residual} - \left(\left\lfloor \frac{-8 \times 1}{2^9} \right\rfloor \times (1 + 1) \right) = -3 - (-1 \times 2) = \mathbf{-1}$$

$$j = 2$$

$$v = \text{Sample}_1 - \text{Sample}_4 = 17 - 24 = \mathbf{-7}$$

$$\text{sign} = -\text{sign}(-7) = \mathbf{1}$$

$$\text{Coefficient}_1 = \text{Coefficient}_1 - 1 = -765 - 1 = \mathbf{-766}$$

$$\text{residual} = \text{residual} - \left(\left\lfloor \frac{-7 \times 1}{2^9} \right\rfloor \times (2 + 1) \right) = -1 - (-1 \times 3) = \mathbf{2}$$

and stop, since $\text{sign}(2) \neq \text{sign}(-4)$ (our ‘original sign’ value)

So, the coefficients for Sample_7 are 1123, -766, 107 and 122.

12.2.3 Channel Decorrelation

If we have more than one channel of output, the next step is performing channel decorrelation. If our ‘Interlacing Leftweight’ value from the frame header is 0, our channels are stored independently. In that case, the Channel₁ is our left samples and Channel₂ is our right samples.

If ‘Interlacing Leftweight’ is greater than zero, we calculate samples as follows:

$$\text{Right}_i = \text{Channel}_1 - \left\lfloor \frac{\text{Channel}_2 \times \text{Interlacing Leftweight}}{2^{\text{Interlacing Shift}}} \right\rfloor$$

$$\text{Left}_i = \text{Channel}_2 + \text{Right}_i$$

For example, given the Channel₁ samples of 14, 15, 19, 17, 18; the Channel₂ samples of 16, 17, 26, 25, 24, an ‘Interlacing Shift’ value of 2 and an ‘Interlacing Leftweight’ values of 3, we calculate output samples as follows:

Sample	Channel ₁	Channel ₂	Right _i	Left _i
0	14	16	$14 - \lfloor (16 \times 3) \div 2^2 \rfloor = \mathbf{2}$	$16 + \mathbf{2} = \mathbf{18}$
1	15	17	$15 - \lfloor (17 \times 3) \div 2^2 \rfloor = \mathbf{3}$	$17 + \mathbf{3} = \mathbf{20}$
2	19	26	$19 - \lfloor (26 \times 3) \div 2^2 \rfloor = \mathbf{0}$	$26 + \mathbf{0} = \mathbf{26}$
3	17	25	$17 - \lfloor (25 \times 3) \div 2^2 \rfloor = \mathbf{-1}$	$25 + \mathbf{-1} = \mathbf{24}$
4	18	24	$18 - \lfloor (24 \times 3) \div 2^2 \rfloor = \mathbf{0}$	$24 + \mathbf{0} = \mathbf{24}$

12.2.4 Wasted Bits

A compressed ALAC frame with ‘Wasted Bits’ stores them interleaved between channels. Then, after channel decorrelation, these verbatim values are prepended to each sample. For example, given a 2 channel stream with 24 bits-per-sample and a ‘Wasted Bits’ value of 1 (meaning our “wasted” samples are 8 bits large), our final output is as follows:

Left Channel			Right Channel		
0	Left ₀	15 16	0	Right ₀	15 16
		Wasted ₀			Wasted ₁
		23			23
0	Left ₁	15 16	0	Right ₁	15 16
		Wasted ₂			Wasted ₃
		23			23
0	Left ₂	15 16	0	Right ₂	15 16
		Wasted ₄			Wasted ₅
		23			23
0	Left ₃	15 16	0	Right ₃	15 16
		Wasted ₆			Wasted ₇
		23			23
0	Left ₄	15 16	0	Right ₄	15 16
		Wasted ₈			Wasted ₉
		23			23

12.3 ALAC Encoding

To encode an ALAC file, we need a stream of PCM sample integers along with that stream's sample rate, bits-per-sample and number of channels. We'll start by encoding all of the non-audio ALAC atoms, most of which are contained within the **moov** atom. There's over twenty atoms in a typical ALAC file, most of which are packed with seemingly redundant or nonessential data, so it will take awhile before we can move on to the actual audio encoding process.

Remember, all of an ALAC's fields are big-endian.

12.3.1 ALAC Atoms

We'll encode our ALAC file in iTunes order, which means it contains the **ftyp**, **moov**, **free** and **mdat** atoms, in that order.

the ftyp Atom

Field	Size	Value
atom length	32	32
atom type	32	'ftyp' (0x66747970)
major brand	32	'M4A ' (0x4d344120)
major brand version	32	0
compatible brand	32	'M4A ' (0x4d344120)
compatible brand	32	'mp42' (0x6d703432)
compatible brand	32	'isom' (0x69736f6d)
compatible brand	32	0x00000000

the moov Atom

Field	Size	Value
atom length	32	mvhd size + trak size + udta size + 8
atom type	32	'moov' (0x6d6f6676)
mvhd atom	mvhd size	mvhd data
trak atom	trak size	trak data
udta atom	udta size	udta data

the mvhd Atom

Field	Size	Value
atom length	32	108/120
atom type	32	'mvhd' (0x6d766864)
version	8	0x00
flags	24	0x000000
created date	32/64	creation date as Mac UTC
modified date	32/64	modification date as Mac UTC
time scale	32	sample rate
duration	32/64	total PCM frames
playback speed	32	0x10000
user volume	16	0x100
padding	80	0x000000000000000000000000
window geometry matrix a	32	0x10000
window geometry matrix b	32	0
window geometry matrix u	32	0
window geometry matrix c	32	0
window geometry matrix d	32	0x10000
window geometry matrix v	32	0
window geometry matrix x	32	0
window geometry matrix y	32	0
window geometry matrix w	32	0x40000000
QuickTime preview	64	0
QuickTime still poster	32	0
QuickTime selection time	64	0
QuickTime current time	32	0
next track ID	32	2

If 'version' is 0, 'created date', 'modified date' and 'duration' are 32 bit fields. Otherwise, they are 64 bit fields. The 'created date' and 'modified date' are seconds since the Macintosh Epoch, which is 00:00:00, January 1st, 1904.² To convert a Unix Epoch timestamp (seconds since January 1st, 1970) to a Macintosh Epoch, one needs to add 24,107 days - or 2082844800 seconds.

²Why 1904? It's the first leap year of the 20th century.

the trak Atom

Field	Size	Value
atom length	32	tkhd size + mdia size + 8
atom type	32	'trak' (0x7472616b)
tkhd atom	tkhd size	tkhd data
mdia atom	mdia size	mdia data

the tkhd Atom

Field	Size	Value
atom length	32	92/104
atom type	32	'tkhd' (0x746b6864)
version	8	0x00
padding	20	0x00000000
track in poster	1	0
track in preview	1	1
track in movie	1	1
track enabled	1	1
created date	32/64	creation date as Mac UTC
modified date	32/64	modification date as Mac UTC
track ID	32	1
padding	32	0x00000000
duration	32/64	total PCM frames
padding	64	0x0000000000000000
video layer	16	0
QuickTime alternate	16	0
volume	16	0x1000
padding	16	0x0000
video geometry matrix a	32	0x10000
video geometry matrix b	32	0
video geometry matrix u	32	0
video geometry matrix c	32	0
video geometry matrix d	32	0x10000
video geometry matrix v	32	0
video geometry matrix x	32	0
video geometry matrix y	32	0
video geometry matrix w	32	0x40000000
video width	32	0
video height	32	0

the mdia Atom

Field	Size	Value
atom length	32	mdhd size + hdlr size + minf size + 8
atom type	32	'mdia' (0x6d646961)
mdhd atom	mdhd size	mdhd data
hdlr atom	hdlr size	hdlr data
minf atom	minf size	minf data

the mdhd Atom

Field	Size	Value
atom length	32	32/44
atom type	32	'mdhd' (0x6d646864)
version	8	0x00
flags	24	0x000000
created date	32/64	creation date as Mac UTC
modified date	32/64	modification date as Mac UTC
time scale	32	sample rate
duration	32/64	total PCM frames
padding	1	0
language	5	language value as ISO 639-2
language	5	
language	5	
QuickTime quality	16	0

Note the three, 5-bit 'language' fields. By adding 0x60 to each value and converting the result to ASCII characters, the result is an ISO 639-2 string of the file's language representation. For example, given the values 0x15, 0x0E and 0x04:

$$\text{language}_0 = 0x15 + 0x60 = 0x75 = \text{u}$$

$$\text{language}_1 = 0x0E + 0x60 = 0x6E = \text{n}$$

$$\text{language}_2 = 0x04 + 0x60 = 0x64 = \text{d}$$

Which is the code 'und', meaning 'undetermined' - which is typical.

the hdlr Atom

Field	Size	Value
atom length	32	33 + component
atom type	32	'hdlr' (0x68646c72)
version	8	0x00
flags	24	0x0000000
QuickTime type	32	0x00000000
QuickTime subtype	32	'soun' (0x736f756e)
QuickTime manufacturer	32	0x00000000
QuickTime component reserved flags	32	0x00000000
QuickTime component reserved flags mask	32	0x00000000
component name length	8	0x00
component name	component name length \times 8	

the minf Atom

Field	Size	Value
atom length	32	smhd size + dinf size + stbl size + 8
atom type	32	'minf' (0x6d696e66)
smhd atom	smhd size	smhd data
dinf atom	dinf size	dinf data
stbl atom	stbl size	stbl data

the smhd Atom

Field	Size	Value
atom length	32	16
atom type	32	'smhd' (0x736d6864)
version	8	0x00
flags	24	0x000000
audio balance	16	0x0000
padding	16	0x0000

the dinf Atom

Field	Size	Value
atom length	32	dref size + 8
atom type	32	'dinf' (0x64696e66)
dref atom	dref size	dref data

the dref Atom

Field	Size	Value
atom length	32	28
atom type	32	'dref' (0x64726566)
version	8	0x00
flags	24	0x000000
number of references	32	1
reference atom size	32	12
reference atom type	32	'url' (0x75726c20)
reference atom data	32	0x00000001

the stbl Atom

Field	Size	Value
atom length	32	stsd size + stts size + stsc size + stsz size + stco size + 8
atom type	32	'stbl' (0x7374626c)
stsd atom	stsd size	stsd data
stts atom	stts size	stts data
stsc atom	stsc size	stsc data
stsz atom	stsz size	stsz data
stco atom	stco size	stco data

the stsd Atom

Field	Size	Value
atom length	32	alac size + 16
atom type	32	'stsd' (0x73747364)
version	8	0x00
flags	24	0x000000
number of descriptions	32	1
alac atom	alac size	alac data

the alac Atom

Field	Size	Value
atom length	32	72
atom type	32	'alac' (0x616c6163)
reserved	48	0x00000000000000
reference index	16	1
version	16	0
revision level	16	0
vendor	32	0x00000000
channels	16	channel count
bits per sample	16	bits per sample
compression ID	16	0
audio packet size	16	0
sample rate	16	sample rate
padding	16	0x0000
atom length	32	36
atom type	32	'alac' (0x616c6163)
padding	32	0x00000000
max samples per frame	32	largest number of PCM frames per ALAC frame
padding	8	0x00
sample size	8	bits per sample
history multiplier	8	40
initial history	8	10
maximum K	8	14
channels	8	channel count
unknown	16	0x00FF
max coded frame size	32	largest ALAC frame size, in bytes
bitrate	32	$((\text{mdat size} \times 8) \div (\text{total PCM frames} \div \text{sample rate}))$
sample rate	32	sample rate

The 'history multiplier', 'initial history' and 'maximum K' values are encode-time options, typically set to 40, 10 and 14, respectively.

Note that the 'bitrate' field can't be known in advance; we must fill that value with 0 for now and then return to this atom once encoding is completed and its size has been determined.

the stts Atom

Field	Size	Value
atom length	32	number of times $\times 8 + 16$
atom type	32	'stts' (0x73747473)
version	8	0x00
flags	24	0x000000
number of times	32	
frame count 1	32	number of occurrences
frame duration 1	32	PCM frame count
...		

This atom keeps track of how many different sizes of ALAC frames occur in the ALAC file, in PCM frames. It will typically have only two “times”, the block size we’re using for most of our samples and the final block size for any remaining samples.

For example, let’s imagine encoding a 1 minute audio file at 44100Hz with a block size of 4096 frames. This file has a total of 2,646,000 PCM frames ($60 \times 44100 = 2646000$). 2,646,000 PCM frames divided by a 4096 block size means we have 645 ALAC frames of size 4096, and 1 ALAC frame of size 4080.

Therefore:

$$\begin{aligned}
 \text{number of times} &= 2 \\
 \text{frame count}_1 &= 645 \\
 \text{frame duration}_1 &= 4096 \\
 \text{frame count}_2 &= 1 \\
 \text{frame duration}_2 &= 4080
 \end{aligned}$$

the stsc Atom

Field	Size	Value
atom length	32	entries $\times 12 + 16$
atom type	32	'stsc' (0x73747363)
version	8	0x00
flags	24	0x000000
number of entries	32	
first chunk	32	
ALAC frames per chunk	32	
description index	32	1
...		

This atom stores how many ALAC frames are in a given “chunk”. In this instance a “chunk” represents an entry in the **stco** atom table, used for seeking backwards and forwards through the file. ‘First chunk’ is the starting offset of its frames-per-chunk value, beginning at 1.

As an example, let’s take a one minute, 44100Hz audio file that’s been broken into 130 chunks (each with an entry in the **stco** atom). Its **stsc** entries would typically be:

$$\begin{aligned}\text{first chunk}_1 &= 1 \\ \text{frames per chunk}_1 &= 5 \\ \text{first chunk}_2 &= 130 \\ \text{frames per chunk}_2 &= 1\end{aligned}$$

What this means is that chunks 1 through 129 have 5 ALAC frames each while chunk 130 has 1 ALAC frame. This is a total of 646 ALAC frames, which matches the contents of the **stts** atom.

the stsz Atom

Field	Size	Value
atom length	32	sizes \times 4 + 20
atom type	32	‘stsz’ (0x7374737a)
version	8	0x00
flags	24	0x000000
block byte size	32	0x00000000
number of sizes	32	
frame size	32	
...		

This atom is a list of ALAC frame sizes, each in bytes. For example, our 646 frame file would have 646 corresponding **stsz** entries.

the stco Atom

Field	Size	Value
atom length	32	offset \times 4 + 16
atom type	32	‘stco’ (0x7374636f)
version	8	0x00
flags	24	0x000000
number of offsets	32	
frame offset	32	
...		

This atom is a list of absolute file offsets for each chunk, where each chunk is typically 5 ALAC frames large.

the udta Atom

Field	Size	Value
atom length	32	meta size + 8
atom type	32	'udta' (0x75647461)
meta atom	meta size	meta data

the meta Atom

Field	Size	Value
atom length	32	hdlr size + ilst size + free size + 12
atom type	32	'meta' (0x6d657461)
version	8	0x00
flags	24	0x000000
hdlr atom	hdlr size	hdlr data
ilst atom	ilst size	ilst data
free atom	free size	free data

the hdlr atom (revisited)

Field	Size	Value
atom length	32	34
atom type	32	'hdlr' (0x68646c72)
version	8	0x00
flags	24	0x000000
QuickTime type	32	0x00000000
QuickTime subtype	32	'mdir' (0x6d646972)
QuickTime manufacturer	32	'appl' (0x6170706c)
QuickTime component reserved flags	32	0x00000000
QuickTime component reserved flags mask	32	0x00000000
component name length	8	0x00
component name	0	

This atom is laid out identically to the ALAC file's primary **hdlr** atom (described on page 147). The only difference is the contents of its fields.

the ilst Atom

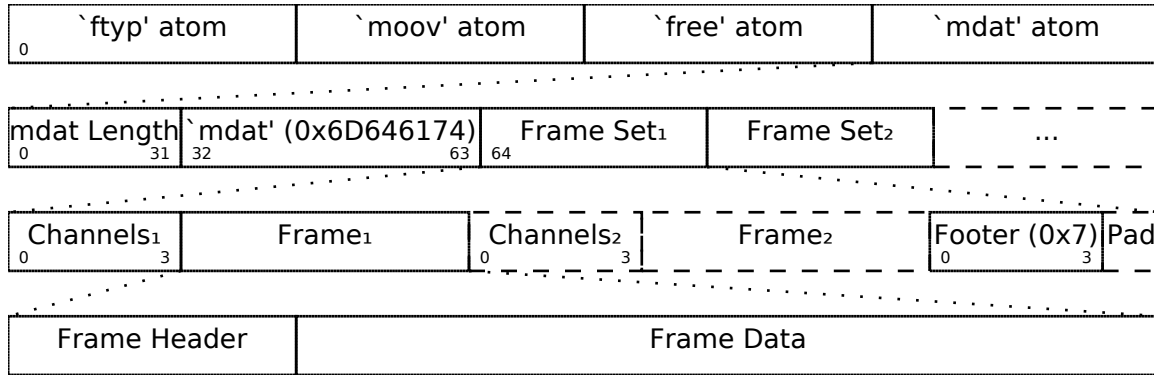
This atom is a collection of **data** sub-atoms and is described on page 131.

the free Atom

These atoms are simple collection of NULL bytes which can easily be resized to make room for other atoms without rewriting the entire file.

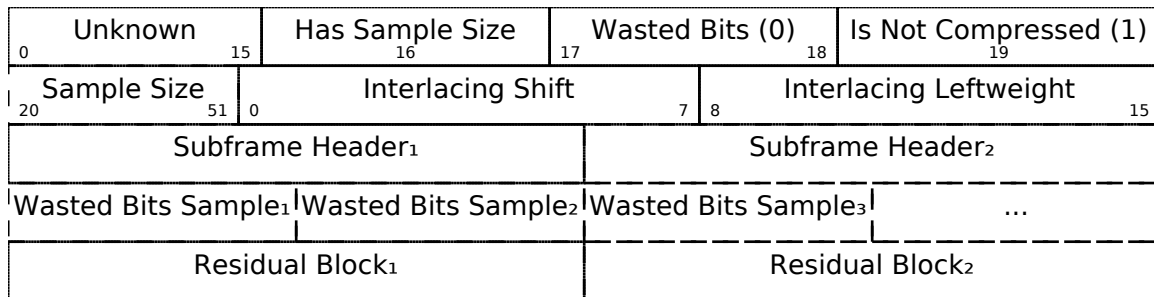
12.3.2 Compressed and Uncompressed Frames

Now that we've built a set of non-audio ALAC atoms, the next step is to break our audio into a set of PCM frames of a certain block size - typically 4096. When encoding more than 2 channels, these PCM frames then need to be broken apart into 1-2 channel sets such that each set becomes an ALAC frame and all the frames become an ALAC frameset.



We then have to decide whether to turn a set of 1-2 channel PCM frames into a compressed or uncompressed ALAC frame. This is done by first attempting a compressed frame while keeping track of its size. If that compressed frame's size is greater than what an uncompressed frame would be $((\text{block size} \times \text{channels} \times \text{bits per sample} \div 8) + 7)$, use an uncompressed frame instead.³ For any audio data that isn't random noise, compressed frames will be the better choice.

As you'll recall, a compressed frame is laid out as follows:



'Has Sample Size' will be 0 so long as we have enough remaining samples to fill our consistent block size. But, at the end of the stream, this value will be 1 to cover any remaining samples whose size will be stored in the 'Sample Size' field. Next, we use a 'Wasted Bits' value of 1 if our stream's bits-per-sample is greater than 16.

In the event we have wasted bits, we simply chop off the bottom 8 bits of each sample and store them as a block of wasted bits samples (whose values are typically random noise,

³See page 134

which does not compress well) and then treat the stream as having 16 bits per sample throughout the remainder of compression.

12.3.3 Channel Correlation

For stereo streams, we must determine good ‘Interlacing Shift’ and ‘Interlacing Leftweight’ values to exploit similarities between the left and right channels.

In this case, we’ll use an ‘Interlacing Shift’ value of 2 and try all all ‘Interlacing Leftweight’ values between 0 and 4 (inclusive), using the one that generates the smallest subframes.

Correlating our left and right channel samples using ‘Interlacing Shift’ and ‘Interlacing Leftweight’ is done using the inverse of decorrelation:

$$\begin{aligned}\text{Channel}_1 &= \text{Right}_i + \left\lfloor \frac{(\text{Left}_i - \text{Right}_i) \times \text{Interlacing Leftweight}}{2^{\text{Interlacing Shift}}} \right\rfloor \\ \text{Channel}_2 &= \text{Left}_i - \text{Right}_i\end{aligned}$$

For example, given the left channels 18, 20, 26, 24, 24; the right channels of 2, 3, 0, -1, 0; an ‘Interlacing Shift’ value of 2 and an ‘Interlacing Leftweight’ value of 3, we calculate ALAC channels as follows:

Sample	Left _i	Right _i	Channel ₁		Channel ₂
0	18	2	$2 + \left\lfloor \frac{(18-2) \times 3}{2^2} \right\rfloor$	$= 2 + 12 = \mathbf{14}$	$18 - 2 = \mathbf{16}$
1	20	3	$3 + \left\lfloor \frac{(20-3) \times 3}{2^2} \right\rfloor$	$= 3 + 12 = \mathbf{15}$	$20 - 3 = \mathbf{17}$
2	26	0	$0 + \left\lfloor \frac{(26-0) \times 3}{2^2} \right\rfloor$	$= 0 + 19 = \mathbf{19}$	$26 - 0 = \mathbf{26}$
3	24	-1	$-1 + \left\lfloor \frac{(24+1) \times 3}{2^2} \right\rfloor$	$= -1 + 18 = \mathbf{17}$	$24 - -1 = \mathbf{25}$
4	24	0	$0 + \left\lfloor \frac{(24-0) \times 3}{2^2} \right\rfloor$	$= 0 + 18 = \mathbf{18}$	$24 - 0 = \mathbf{24}$

12.3.4 Coefficient Calculation

Given a list of correlated samples, we need to generate one subframe header per channel by performing coefficient calculation to generate ‘Prediction Quantitization’, ‘Coefficient Count’ and a set of ‘Coefficient’ values. We’ll set ‘Prediction Type’ to 0 and ‘Rice Modifier’ to 4.

Prediction Type			Prediction Quantitization		
0	3	4	7		
Rice Modifier			Coefficient Count		
8	10	11	15		
Coefficient ₁		Coefficient ₂		...	
16	31	32	47

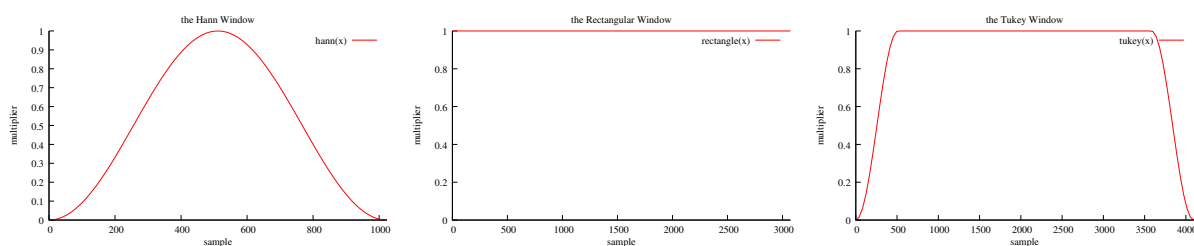
Windowing

The first step in ALAC 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.

$$\text{hann}(n) = \frac{1}{2} \left(1 - \cos \left(\frac{2\pi n}{\text{sample count} - 1} \right) \right) \quad (12.6)$$

$$\text{rectangle}(n) = 1.0 \quad (12.7)$$

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	14	×	0.0000	=	0.00
1	15	×	0.1464	=	2.20
2	19	×	0.5000	=	9.50
3	17	×	0.8536	=	14.51
4	18	×	1.0000	=	18.00
5	17	×	1.0000	=	17.00
6	16	×	1.0000	=	16.00
7	18	×	1.0000	=	18.00
8	17	×	1.0000	=	17.00
9	15	×	1.0000	=	15.00
10	13	×	1.0000	=	13.00
11	13	×	1.0000	=	13.00
12	12	×	1.0000	=	12.00
13	12	×	1.0000	=	12.00
14	15	×	1.0000	=	15.00
15	17	×	1.0000	=	17.00
16	16	×	0.8536	=	13.66
17	17	×	0.5000	=	8.50
18	16	×	0.1464	=	2.34
19	13	×	0.0000	=	0.00

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.

0.00	2.20	9.50	14.51	18.00	17.00	16.00	18.00	17.00	15.00	13.00	13.00	12.00	12.00	15.00	17.00	13.66	8.50	2.34	0.00
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
0.00	2.20	9.50	14.51	18.00	17.00	16.00	18.00	17.00	15.00	13.00	13.00	12.00	12.00	15.00	17.00	13.66	8.50	2.34	0.00
lag 0 sum = 3416.9513																			
0.00	2.20	9.50	14.51	18.00	17.00	16.00	18.00	17.00	15.00	13.00	13.00	12.00	12.00	15.00	17.00	13.66	8.50	2.34	
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	
2.20	9.50	14.51	18.00	17.00	16.00	18.00	17.00	15.00	13.00	13.00	12.00	12.00	15.00	17.00	13.66	8.50	2.34	0.00	
lag 1 sum = 3314.1450																			
0.00	2.20	9.50	14.51	18.00	17.00	16.00	18.00	17.00	15.00	13.00	13.00	12.00	12.00	15.00	17.00	13.66	8.50		
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		
9.50	14.51	18.00	17.00	16.00	18.00	17.00	15.00	13.00	13.00	12.00	12.00	15.00	17.00	13.66	8.50	2.34	0.00		
lag 2 sum = 3078.9564																			
0.00	2.20	9.50	14.51	18.00	17.00	16.00	18.00	17.00	15.00	13.00	13.00	12.00	12.00	15.00	17.00	13.66			
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x			
14.51	18.00	17.00	16.00	18.00	17.00	15.00	13.00	13.00	12.00	12.00	15.00	17.00	13.66	8.50	2.34	0.00			
lag 3 sum = 2807.4600																			
0.00	2.20	9.50	14.51	18.00	17.00	16.00	18.00	17.00	15.00	13.00	13.00	12.00	12.00	15.00	17.00				
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x				
18.00	17.00	16.00	18.00	17.00	15.00	13.00	13.00	12.00	12.00	15.00	17.00	13.66	8.50	2.34	0.00				
lag 4 sum = 2554.6000																			
0.00	2.20	9.50	14.51	18.00	17.00	16.00	18.00	17.00	15.00	13.00	13.00	12.00	12.00	15.00					
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x					
17.00	16.00	18.00	17.00	15.00	13.00	13.00	12.00	12.00	15.00	17.00	13.66	8.50	2.34	0.00					
lag 5 sum = 2325.5300																			
0.00	2.20	9.50	14.51	18.00	17.00	16.00	18.00	17.00	15.00	13.00	13.00	12.00	12.00						
x	x	x	x	x	x	x	x	x	x	x	x	x	x						
16.00	18.00	17.00	15.00	13.00	13.00	12.00	12.00	15.00	17.00	13.66	8.50	2.34	0.00						
lag 6 sum = 2107.9100																			
0.00	2.20	9.50	14.51	18.00	17.00	16.00	18.00	17.00	15.00	13.00	13.00	12.00							
x	x	x	x	x	x	x	x	x	x	x	x	x							
18.00	17.00	15.00	13.00	13.00	12.00	12.00	15.00	17.00	13.66	8.50	2.34	0.00							
lag 7 sum = 1903.3500																			
0.00	2.20	9.50	14.51	18.00	17.00	16.00	18.00	17.00	15.00	13.00	13.00								
x	x	x	x	x	x	x	x	x	x	x	x								
17.00	15.00	13.00	13.00	12.00	12.00	15.00	17.00	13.66	8.50	2.34	0.00								
lag 8 sum = 1701.2700																			

In this example the autocorrelation values are: 3416.9513, 3314.1450, 3078.9564, 2807.4600, 2554.6000, 2325.5300, 2107.9100, 1903.3500 and 1701.2700.

LP Coefficient Calculation

Calculating the LP coefficients uses the Levinson-Durbin recursive method.⁴ Our inputs are M , the maximum coefficient count, 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 $m \leftarrow m + 1$ and goto step 1. If $m = M$ then stop.

Let's run through an example in which $M = 9$, $r(0) = 3417$, $r(1) = 3314$, $r(2) = 3079$, $r(3) = 2807$, $r(4) = 2555$, $r(5) = 2326$, $r(6) = 2108$, $r(7) = 1903$ and $r(8) = 1701$:

⁴This algorithm is taken from <http://www.engineer.tamuk.edu/SPark/chap7.pdf>

12 Apple Lossless

$$\begin{aligned}
E_0 &= r(0) = 3417 \\
a_{11} &= \kappa_1 = \frac{r(1)}{E_0} = \frac{3314}{3417} = 0.97 \\
E_1 &= E_0(1 - \kappa_1^2) = 3417(1 - .97^2) = 201.95 \\
q_2 &= r(2) - \sum_{i=1}^1 a_{i1}r(2-i) = 3079 - (0.97)(3314) = -135.58 \\
\kappa_2 &= \frac{q_2}{E_1} = \frac{-135.58}{201.95} = -0.671 \\
a_{22} &= \kappa_2 = -0.671 \\
a_{12} &= a_{11} - \kappa_2 a_{11} = 0.97 - (-0.671)(0.97) = 1.621 \\
E_2 &= E_1(1 - \kappa_2^2) = 201.95(1 - (-0.671)^2) = 111.024 \\
q_3 &= r(3) - \sum_{i=1}^2 a_{i2}r(3-i) = 2807 - ((1.621)(3079) + (-0.671)(3314)) = 39.635 \\
\kappa_3 &= \frac{q_3}{E_2} = \frac{39.635}{111.024} = 0.357 \\
a_{33} &= \kappa_3 = 0.357 \\
a_{13} &= a_{12} - \kappa_3 a_{22} = 1.621 - (0.357)(-0.671) = 1.861 \\
a_{23} &= a_{22} - \kappa_3 a_{12} = -0.671 - (0.357)(1.621) = -1.25 \\
E_3 &= E_2(1 - \kappa_3^2) = 111.024(1 - 0.357^2) = 96.874 \\
q_4 &= r(4) - \sum_{i=1}^3 a_{i3}r(4-i) = 2555 - ((1.861)(2807) + (-1.25)(3079) + (0.357)(3314)) = -3.175 \\
\kappa_4 &= \frac{q_4}{E_3} = \frac{-3.175}{96.874} = -0.033 \\
a_{44} &= \kappa_4 = -0.033 \\
a_{14} &= a_{13} - \kappa_4 a_{33} = 1.861 - (-0.033)(0.357) = 1.873 \\
a_{24} &= a_{23} - \kappa_4 a_{23} = -1.25 - (-0.033)(-1.25) = -1.291 \\
a_{34} &= a_{33} - \kappa_4 a_{33} = 0.357 - (-0.033)(1.861) = 0.418 \\
E_4 &= E_3(1 - \kappa_4^2) = 96.874(1 - (-0.033)^2) = 96.769
\end{aligned}$$

Calculating E_5 through E_8 and a_{15} though a_{88} will be left as an exercise for the reader. Our final values are:

$$\begin{aligned}
a_{11} &= 0.970 \\
a_{12} &= 1.621 & a_{22} &= -0.67 \\
a_{13} &= 1.861 & a_{23} &= -1.25 & a_{33} &= 0.357 \\
a_{14} &= 1.873 & a_{24} &= -1.29 & a_{34} &= 0.418 & a_{44} &= -0.03 \\
a_{15} &= 1.868 & a_{25} &= -1.28 & a_{35} &= 0.340 & a_{45} &= 0.122 & a_{55} &= -0.14 \\
a_{16} &= 1.874 & a_{26} &= -1.28 & a_{36} &= 0.341 & a_{46} &= 0.119 & a_{56} &= -0.09 & a_{66} &= 0.00 \\
a_{17} &= 1.875 & a_{27} &= -1.27 & a_{37} &= 0.332 & a_{47} &= 0.091 & a_{57} &= 0.011 & a_{67} &= -0.15 & a_{77} &= 0.078 \\
a_{18} &= 1.892 & a_{28} &= -1.30 & a_{38} &= 0.338 & a_{48} &= 0.112 & a_{58} &= 0.082 & a_{68} &= -0.42 & a_{78} &= 0.479 & a_{88} &= -0.214 \\
E_1 &= 202.0 & E_2 &= 111.0 & E_3 &= 96.9 & E_4 &= 96.8 & E_5 &= 96.1 & E_6 &= 96.1 & E_7 &= 95.6 & E_8 &= 91.2
\end{aligned}$$

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

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. Making an 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:

$$\text{Error Scale} = \frac{\ln(2)^2}{2 \times \text{Total Samples}} \quad (12.8)$$

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:

$$\text{Bits per Residual(LPC Error)} = \frac{\ln(\text{Error Scale} \times \text{LPC Error})}{2 \times \ln(2)} \quad (12.9)$$

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

$$\text{Total Bits(LPC Error, Order)} = \text{Bits per Residual(LPC Error)} \times (\text{Total Samples} - \text{Order}) + (\text{Order} \times \text{Overhead bits})$$

Continuing with our example, we have 20 samples which gives us an error scale of: $\frac{\ln(2)^2}{2 \times 20} = \frac{.6931^2}{40} = .01201$

Now, we’ll estimate the bits used by order 4 and order 8, which use LPC Error values 96.8 and 91.2, respectively:

At LPC order 4, our bits per residual are:

$$\frac{\ln(.01201 \times 96.8)}{2 \times \ln(2)} = \frac{\ln(1.163)}{1.386} = 0.1089$$

And our total bits are:

$$(0.1089 \times (20 - 1)) + (1 \times (16 + 5)) = 2.069 + 21 = 23.069$$

At LPC order 8, our bits per residual are:

$$\frac{\ln(.01201 \times 91.2)}{2 \times \ln(2)} = \frac{\ln(1.095)}{1.386} = 0.065$$

And our total bits are:

$$(0.065 \times (20 - 2)) + (2 \times (16 + 5)) = 1.17 + 42 = 43.17$$

Therefore, since the total bits for order 4 are the smallest, the best order for this group of samples is 4.

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 Coefficients and a Prediction Quantitization value. The first step is determining the upper and lower limits of the Coefficients, which will set to the upper and lower bounds that iTunes supports:

$$\text{Coefficient Maximum} = 2^{12-1} - 1 = 2047 \quad (12.10)$$

$$\text{Coefficient Minimum} = -2^{12-1} = -2048 \quad (12.11)$$

Prediction Quantitization is always 9 in iTunes, so we'll use that value also:

$$\text{Prediction Quantitization} = 9 \quad (12.12)$$

We determine the Coefficient values themselves via a small recursive routine:

$$X(i) = E(i-1) + (\text{LP Coefficient}_i \times 2^{\text{quantitization}}) \quad (12.13)$$

$$\text{Coefficient}_i = \text{round}(X(i)) \quad (12.14)$$

$$E(i) = X(i) - \text{Coefficient}_i \quad (12.15)$$

where $E(0) = 0$ and each Coefficient is adjusted prior to calculating the next $E(i)$ value such that:

$$\text{Coefficient Minimum} \leq \text{Coefficient}_i \leq \text{Coefficient Maximum} \quad (12.16)$$

So to finish our example in which we're quantizing the LP coefficients 1.873, -1.29, 0.418 and -0.03:

$$X(1) = E(0) + (1.873 \times 2^9) = 0 + 958.976 = \mathbf{958.976}$$

$$\text{QLP Coefficient}_1 = \text{round}(958.976) = \mathbf{959}$$

$$E(1) = X(1) - \text{QLP Coefficient}_1 = 958.976 - 959 = \mathbf{-0.024}$$

$$X(2) = E(1) + (-1.29 \times 2^9) = -0.024 + -660.48 = \mathbf{-660.504}$$

$$\text{QLP Coefficient}_2 = \text{round}(-660.504) = \mathbf{-661}$$

$$E(2) = X(2) - \text{QLP Coefficient}_2 = -660.504 - -661 = \mathbf{0.496}$$

$$X(3) = E(2) + (0.418 \times 2^9) = 0.496 + 214.015 = \mathbf{214.511}$$

$$\text{QLP Coefficient}_3 = \text{round}(214.511) = \mathbf{215}$$

$$E(3) = X(3) - \text{QLP Coefficient}_3 = 214.511 - 215 = \mathbf{-0.489}$$

$$X(4) = E(3) + (-0.03 \times 2^9) = -0.489 + -15.36 = \mathbf{-15.849}$$

$$\text{QLP Coefficient}_4 = \text{round}(-15.849) = \mathbf{-16}$$

Therefore, the 'Coefficient Count' is 4, the 'Coefficient' values are 959, -661, 215, -16, and the 'Prediction Quantitization' value is 9.

12.3.5 Subframe Calculation

Given our list of correlated samples; along with a list of coefficients, a ‘Coefficient Count’ and a ‘Prediction Quantitization’ value, we can now generate a list of signed residuals values for a given channel.

The first residual is always the first input sample:

$$\text{Residual}_0 = \text{Sample}_0$$

Then, for the next ‘Coefficient Count’ number of samples:

$$\text{Residual}_i = \text{Sample}_i - \text{Sample}_{i-1}$$

For example, given that we have a ‘Coefficient Count’ of 4 and our first five samples are 16, 17, 26, 25 and 24; our first five residual values are:

$$\begin{aligned}\text{Residual}_0 &= \mathbf{16} \\ \text{Residual}_1 &= 17 - 16 = \mathbf{1} \\ \text{Residual}_2 &= 26 - 17 = \mathbf{9} \\ \text{Residual}_3 &= 25 - 26 = \mathbf{-1} \\ \text{Residual}_4 &= 24 - 25 = \mathbf{-1}\end{aligned}$$

These are our “starting point” residuals upon which the remainder of the residuals will be calculated from.

Subsequent residuals are calculated in the following way:

$$\begin{aligned}\text{LPC Sum}_i &= \sum_{j=0}^{\text{coeffs}-1} \text{Coefficient}_j \times (\text{Sample}_{i-j-1} - \text{Sample}_{i-\text{coeffs}-1}) \\ \text{Residual}_i &= \text{Sample}_i - \left(\left\lfloor \frac{\text{LPC Sum}_i + 2^{\text{Predictor Quantitization} - 1}}{2^{\text{Predictor Quantitization}}} \right\rfloor + \text{Sample}_{i-\text{coeffs}-1} \right)\end{aligned}$$

For example, given $\text{Sample}_5 = 23$, ‘Predictor Quantitization’ = 9 and the coefficients 1122, -766, 107 and 122:

$$\begin{aligned}\text{LPC Sum}_5 &= (1122 \times (24 - 16)) + (-766 \times (25 - 16)) + (107 \times (26 - 16)) + (122 \times (17 - 16)) \\ &= (1122 \times 8) + (-766 \times 9) + (107 \times 10) + (122 \times 1) \\ &= 8976 + -6894 + 1070 + 122 = \mathbf{3274}\end{aligned}$$

$$\text{Residual}_5 = 23 - \left(\left\lfloor \frac{\mathbf{3274} + 2^8}{2^9} \right\rfloor + 16 \right) = 23 - \left(\left\lfloor \frac{3530}{512} \right\rfloor + 16 \right) = 23 - (6 + 16) = \mathbf{1}$$

But before calculation Residual_6 , we need to adjust our coefficient list in the same way as residual decoding.

As described on page 140, we run through the following process:

Step 1.	original sign	=	sign(<i>residual</i>)
Step 2.	<i>j</i>	=	0
Step 3.	<i>v</i>	=	Sample _{<i>i</i>-count-1} - Sample _{<i>i</i>-count+<i>j</i>}
Step 4.	<i>sign</i>	=	$\begin{cases} \text{sign}(v) & \text{if original sign} > 0 \\ -\text{sign}(v) & \text{if original sign} \leq 0 \end{cases}$
Step 5.	Coefficient _{count-<i>j</i>-1}	=	Coefficient _{count-<i>j</i>-1} - <i>sign</i>
Step 6.	<i>residual</i>	=	<i>residual</i> - ($\lfloor \frac{v \times \text{sign}}{2^{\text{Predictor Quantitization}}} \rfloor \times (j + 1)$)
Step 7.	<i>j</i>	=	<i>j</i> + 1
Step 8.	if (<i>j</i> < <i>count</i>) and (sign(<i>residual</i>) = original sign), goto step 3		

To continue our example:

$$\text{original sign} = \text{sign}(1) = \mathbf{1}$$

$$j = 0$$

$$v = \text{Sample}_0 - \text{Sample}_1 = 16 - 17 = \mathbf{-1}$$

$$\text{sign} = \text{sign}(-1) = \mathbf{-1}$$

$$\text{Coefficient}_3 = \text{Coefficient}_3 - -1 = 122 + 1 = \mathbf{123}$$

$$\text{residual} = \text{residual} - \left(\left\lfloor \frac{-1 \times -1}{2^9} \right\rfloor \times (0 + 1) \right) = 1 - (0 \times 1) = \mathbf{1}$$

$$j = 1$$

$$v = \text{Sample}_0 - \text{Sample}_2 = 16 - 26 = \mathbf{-10}$$

$$\text{sign} = \text{sign}(-10) = \mathbf{-1}$$

$$\text{Coefficient}_2 = \text{Coefficient}_2 - -1 = 107 + 1 = \mathbf{108}$$

$$\text{residual} = \text{residual} - \left(\left\lfloor \frac{-10 \times -1}{2^9} \right\rfloor \times (1 + 1) \right) = 1 - (0 \times 2) = \mathbf{1}$$

$$j = 2$$

$$v = \text{Sample}_0 - \text{Sample}_3 = 16 - 25 = \mathbf{-9}$$

$$\text{sign} = \text{sign}(-9) = \mathbf{-1}$$

$$\text{Coefficient}_1 = \text{Coefficient}_1 - -1 = -766 + 1 = \mathbf{-765}$$

$$\text{residual} = \text{residual} - \left(\left\lfloor \frac{-9 \times -1}{2^9} \right\rfloor \times (2 + 1) \right) = 1 - (0 \times 3) = \mathbf{1}$$

$$j = 3$$

$$v = \text{Sample}_0 - \text{Sample}_4 = 16 - 25 = \mathbf{-8}$$

$$\text{sign} = \text{sign}(-8) = \mathbf{-1}$$

$$\text{Coefficient}_0 = \text{Coefficient}_0 - -1 = 1122 + 1 = \mathbf{1123}$$

$$\text{residual} = \text{residual} - \left(\left\lfloor \frac{-8 \times -1}{2^9} \right\rfloor \times (3 + 1) \right) = 1 - (0 \times 4) = \mathbf{1}$$

$$j = 4 \text{ and stop, since 4 equals our coefficient count}$$

Which, you'll notice, is identical to the procedure used during residual decoding.

Given that Sample_6 is 20, the calculation of Residual_6 is as follows:

$$\begin{aligned}\text{LPC Sum}_6 &= (1123 \times (23 - 17)) + (-765 \times (24 - 17)) + (108 \times (25 - 17)) + (123 \times (26 - 17)) \\ &= (1123 \times 6) + (-765 \times 7) + (108 \times 8) + (123 \times 9) \\ &= 6738 + -5355 + 864 + 1107 = \mathbf{3354}\end{aligned}$$

$$\text{Residual}_6 = 20 - \left(\left\lfloor \frac{\mathbf{3354} + 2^8}{2^9} \right\rfloor + 17 \right) = 20 - \left(\left\lfloor \frac{3610}{512} \right\rfloor + 17 \right) = 20 - (7 + 17) = \mathbf{-4}$$

$$\text{original sign} = \text{sign}(-4) = -1$$

$$j = 0$$

$$v = \text{Sample}_1 - \text{Sample}_2 = 17 - 26 = \mathbf{-9}$$

$$\text{sign} = -\text{sign}(-9) = \mathbf{1}$$

$$\text{Coefficient}_3 = \text{Coefficient}_3 - 1 = 123 - 1 = \mathbf{122}$$

$$\text{residual} = \text{residual} - \left(\left\lfloor \frac{-9 \times 1}{2^9} \right\rfloor \times (0 + 1) \right) = -4 - (-1 \times 1) = \mathbf{-3}$$

$$j = 1$$

$$v = \text{Sample}_1 - \text{Sample}_3 = 17 - 25 = \mathbf{-8}$$

$$\text{sign} = -\text{sign}(-8) = \mathbf{1}$$

$$\text{Coefficient}_2 = \text{Coefficient}_2 - 1 = 108 - 1 = \mathbf{107}$$

$$\text{residual} = \text{residual} - \left(\left\lfloor \frac{-8 \times 1}{2^9} \right\rfloor \times (1 + 1) \right) = -3 - (-1 \times 2) = \mathbf{-1}$$

$$j = 2$$

$$v = \text{Sample}_1 - \text{Sample}_4 = 17 - 24 = \mathbf{-7}$$

$$\text{sign} = -\text{sign}(-7) = \mathbf{1}$$

$$\text{Coefficient}_1 = \text{Coefficient}_1 - 1 = -765 - 1 = \mathbf{-766}$$

$$\text{residual} = \text{residual} - \left(\left\lfloor \frac{-7 \times 1}{2^9} \right\rfloor \times (2 + 1) \right) = -1 - (-1 \times 3) = \mathbf{2}$$

and stop, since $\text{sign}(2) \neq \text{sign}(-4)$ (our ‘original sign’ value)

So, the coefficients for Residual_7 are 1123, -766, 107 and 122.

12.3.6 Residual Encoding

The final step of frame encoding is writing our list of residual values. As with decoding, this also requires knowing several additional values whose defaults are as follows:

value	default
Initial History	10
History Multiplier	40
Maximum K	14
Bits per Sample	Stream's BPS – (Wasted Bits × 8) + Channels – 1

For each residual block, ‘History’ starts with the value of ‘Initial History’ and will change during residual encoding. We use it to calculate κ using the following formula:

$$\kappa = \left\lfloor \log_2 \left(\frac{\text{history}}{2^9} + 3 \right) \right\rfloor \quad (12.17)$$

Note that if κ exceeds the ‘Maximum K’ value from the **alac** atom, ‘Maximum K’ is used instead.

The κ and ‘Bits per Sample’ values are used to write a single unsigned residual value r in the following way:

First, we convert our signed residual to an unsigned value:

$$\text{unsigned} = \begin{cases} \text{signed} \times 2 & \text{if signed} \geq 0 \\ (-\text{signed} \times 2) - 1 & \text{if signed} < 0 \end{cases}$$

We then divide our unsigned residual into quotient and remainder values:

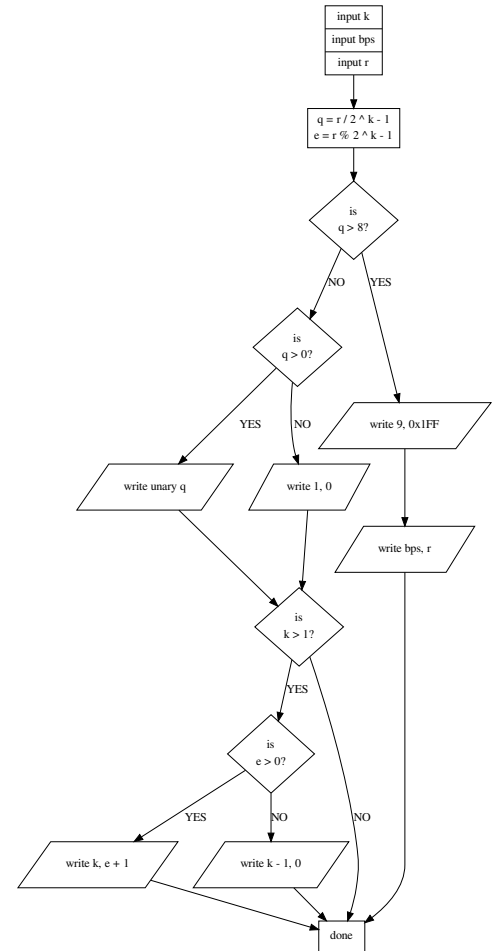
$$q = \lfloor r \div (2^k - 1) \rfloor$$

$$e = r \bmod 2^k - 1$$

If q is greater than 8, we write a 9 bit value 0x1FF (i.e. the bits ‘1 1 1 1 1 1 1 1 1’) before writing the value r verbatim using ‘Bits per Sample’ number of bits.

If q is greater than 0, we write its value in unary. Otherwise, we write a single 0 bit.

Finally, if e is greater than 0, we write the value $e + 1$ using κ of bits. Otherwise, we write the value 0 using $\kappa - 1$ number of bits. Assuming κ is greater than 1, of course.



As with decoding, we then use our unsigned value to update ‘history’ before encoding the next residual:

$$\text{history} = \begin{cases} \text{history} + (\text{unsigned} \times \text{history multiplier}) - \lfloor \frac{\text{history} \times \text{history multiplier}}{2^9} \rfloor & \text{if unsigned} \leq 65535 \\ 65535 & \text{if unsigned} > 65535 \end{cases}$$

Should history ever fall below 128, we generate a special residual value. That residual’s ‘Bits per Sample’ value is 16, its κ value is:

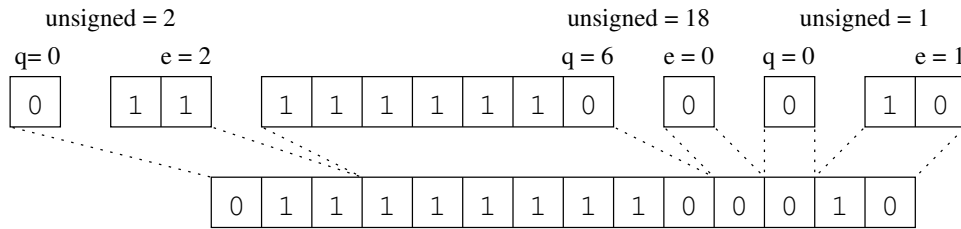
$$\kappa_{\text{blocksize}} = 7 - \log_2(\text{history}) + \frac{\text{history} + 16}{64} \quad (12.18)$$

and its value is how many consecutive ‘0’ residuals follow - which may be 0 if the next residual is not ‘0’. Either way, if ‘block size’ ≤ 65535 , we subtract 1 to the next unsigned value of the next residual in the block (if any). Finally, ‘history’ is automatically set to 0.

Residual Encoding Example

In this example, we’ll encode a group of residuals in which our ‘Initial History’ is 1290, our ‘History Multiplier’ is 40 and our signed residual values are 1, 9 and -1:

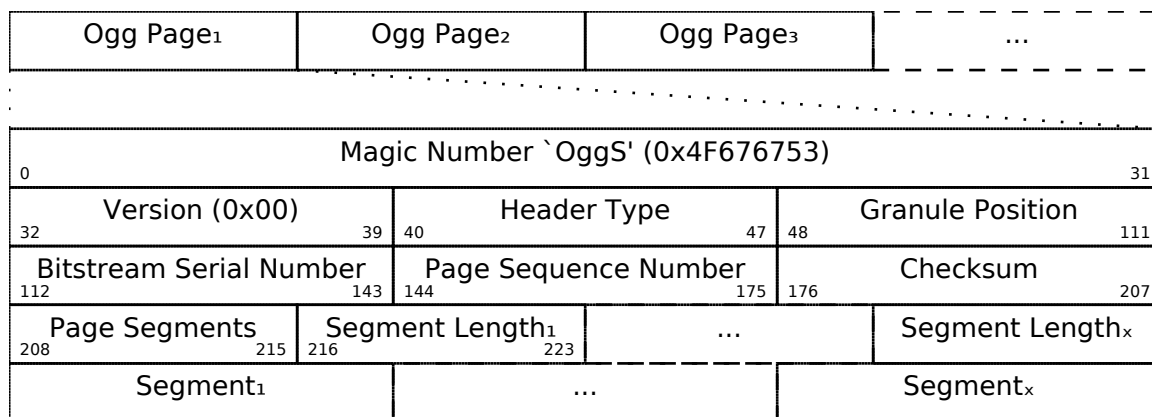
- Residual 1
 - $\kappa = \lfloor \log_2((1290 \div 2^9) + 3) \rfloor = \lfloor \log_2(5) \rfloor = 2$
 - $\text{unsigned}_1 = 2 \times 1 = \mathbf{2}$
 - $\text{history} = 1290 + (2 \times 40) - \lfloor (1290 \times 40) \div 2^9 \rfloor = 1370 - 100 = 1270$
- Residual 2
 - $\kappa = \lfloor \log_2((1270 \div 2^9) + 3) \rfloor = \lfloor \log_2(5) \rfloor = 2$
 - $\text{unsigned}_2 = 2 \times 9 = \mathbf{18}$
 - $\text{history} = 1270 + (18 \times 40) - \lfloor (1270 \times 40) \div 2^9 \rfloor = 1990 - 99 = 1891$
- Residual 3
 - $\kappa = \lfloor \log_2((1891 \div 2^9) + 3) \rfloor = \lfloor \log_2(6) \rfloor = 2$
 - $\text{unsigned}_3 = (1 \times 2) - 1 = \mathbf{1}$
 - $\text{history} = 1891 + (1 \times 40) - \lfloor (1891 \times 40) \div 2^9 \rfloor = 1931 - 147 = 1784$



13 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.

13.1 Ogg File 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.

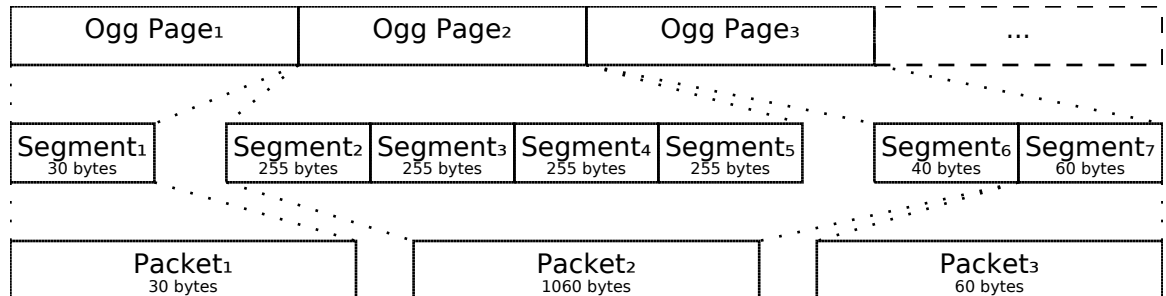
‘Checksum’ is a 32-bit checksum of the entire Ogg page.

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

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

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.

13.1.1 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.

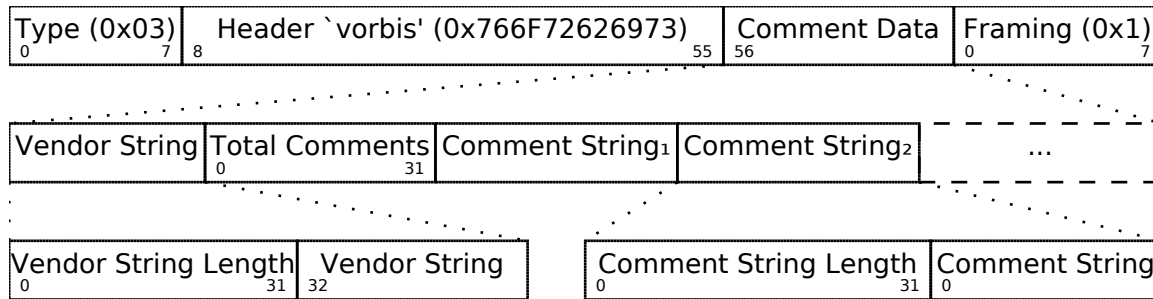
13.2 the Identification Packet

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)										55											
0	7	8																					
Vorbis version (0x00000000)												Channels											
56												87		88		95							
Sample Rate						Maximum Bitrate																	
96						127						128						159					
Nominal Bitrate						Minimum Bitrate																	
160						191						192						223					
Blocksize ₁				Blocksize ₂				Framing flag (0x01)															
224				227				228				231				232				239			

13.3 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**.

ALBUM album name

ARTIST artist name, band name, composer, author, etc.

CATALOGNUMBER* CD spine number

COMPOSER* the work’s author

CONDUCTOR* performing ensemble’s leader

COPYRIGHT copyright attribution

DATE recording date

DESCRIPTION a short description

DISCNUMBER* disc number for multi-volume work

ENGINEER* the recording masterer

ENSEMBLE* performing group

GENRE a short music genre label

GUEST ARTIST* collaborating artist

ISRC ISRC number for the track

LICENSE license information

LOCATION recording location

OPUS* number of the work

ORGANIZATION record label

PART* track’s movement title

PERFORMER performer name, orchestra, actor, etc.

PRODUCER* person responsible for the project

PRODUCTNUMBER* UPC, EAN, or JAN code

PUBLISHER* album’s publisher

RELEASE DATE* date the album was published

REMIXER* person who created the remix

SOURCE ARTIST* artist of the work being performed

SOURCE MEDIUM* CD, radio, cassette, vinyl LP, etc.

SOURCE WORK* a soundtrack’s original work

SPARS* DDD, ADD, AAD, etc.

SUBTITLE* for multiple track names in a single file

TITLE track name

TRACKNUMBER track number

VERSION track version

Fields marked with * are proposed extension fields and not part of the official Vorbis comment specification.

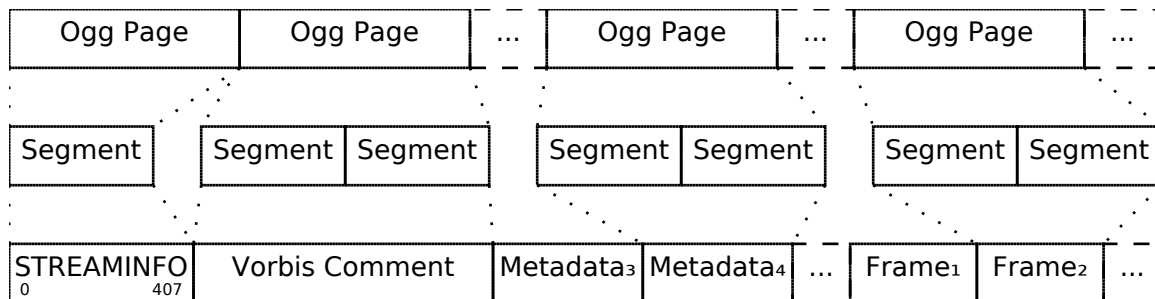
13.4 Channel Assignment

channel count	channel 1	channel 2	channel 3	channel 4	channel 5	channel 6	channel 7	channel 8
1	front center							
2	front left	front right						
3	front left	front center	front right					
4	front left	front right	back left	back right				
5	front left	front center	front right	back left	back right			
6	front left	front center	front right	back left	back right	LFE		
7	front left	front center	front right	side left	side right	back center	LFE	
8	front left	front center	front right	side left	side right	back left	back right	LFE
8+	defined by application							

14 Ogg FLAC

Ogg FLAC is a FLAC audio stream in an Ogg container.

14.1 the Ogg FLAC File Stream



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

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.

15 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 on page 167. Therefore, I shall move directly to the Ogg Speex packets themselves.

15.1 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			Speex Version ID		
64	223	224			255
Header Size		Sampling Rate		Mode	
256	287	288	319	320	351
Mode Bitstream Version		Number of Channels		Bitrate	
352	383	384	415	416	447
Frame Size		VBR		Frames Per Packet	
448	479	480	511	512	543
Extra Headers		Reserved ₁		Reserved ₂	
544	575	576	607	608	639

15.2 the Comment Packet

The second packet within a Speex stream is the Comment packet. This is identical to the comments used by Ogg Vorbis which is detailed on page 169.

16 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.

16.1 the SV7 File Stream

This is the earliest version of Musepack with wide support. All of its fields are little-endian. Each frame contains 1152 samples per channel. Therefore:

Header		Frame ₁		Frame ₂		...		APEv2 tag	
0	223	224							
.....									
Signature ('MP+' 0x4D502B)				Version (0x07)		Frame Count		Max Level	
0	23	24	31	32	63	64	79		
Profile		Link		Sample Rate		Intensity Stereo		Midside Stereo	
80	83	84	85	86	87	88	89	90	95
Title Gain			Title Peak			Album Gain			Album Peak
96	111	112	127	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

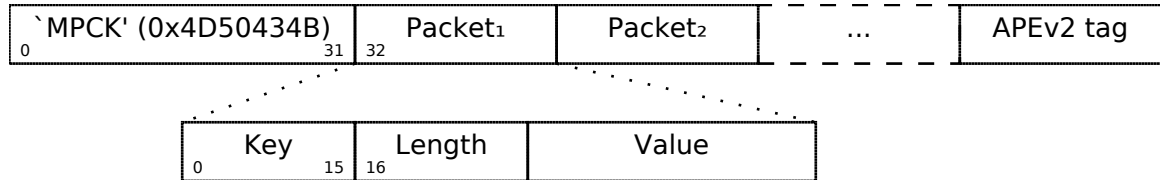
$$\text{Total Samples} = ((\text{Frame Count} - 1) \times 1152) + \text{Last Frame Samples} \quad (16.1)$$

Musepack files always have exactly 2 channels and its lossy samples are stored as floating point. Its sampling rate is one of four values:

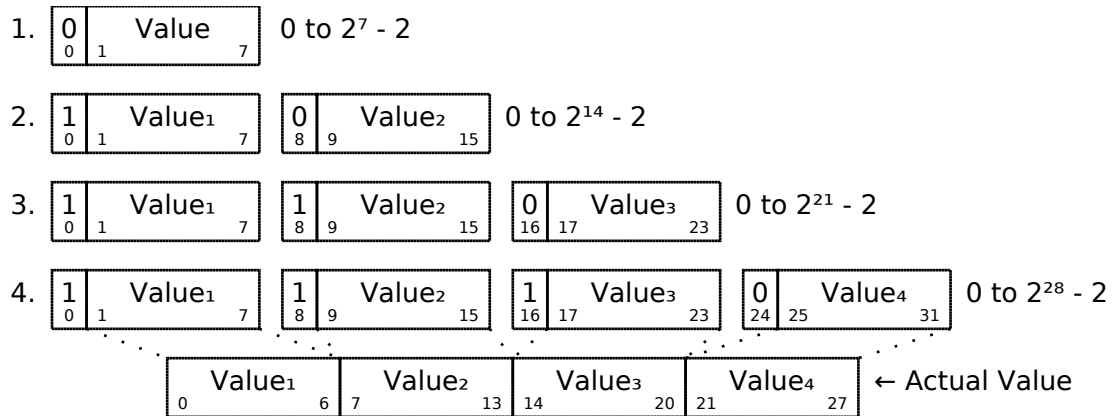
00 = 44100Hz, 01 = 48000Hz, 10 = 37800Hz, 11 = 32000Hz .

16.2 the SV8 File Stream

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

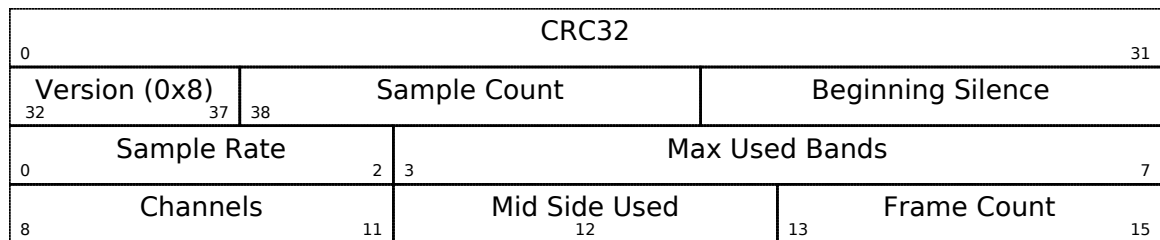


‘Key’ is a two character uppercase ASCII string (i.e. each digit must be between the characters 0x41 and 0x5A, inclusive). ‘Length’ is a variable length field indicating the size of the entire packet, including the header. This is a Nut-encoded field whose total size depends on whether the eighth bit 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.



16.2.1 the SH Packet

This is the Stream Header, which must be found before the first audio packet in the 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:

$$\text{Number of Frames} = 4^{\text{Frame Count}} \quad (16.2)$$

‘Sample Rate’ is one of four values:

000 = 44100Hz, 001 = 48000Hz, 010 = 37800Hz, 011 = 32000Hz .

16.2.2 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.

16.2.3 the RG Packet

This is ReplayGain information about the file.

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

16.2.4 the EI Packet

This is information about the Musepack encoder.

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

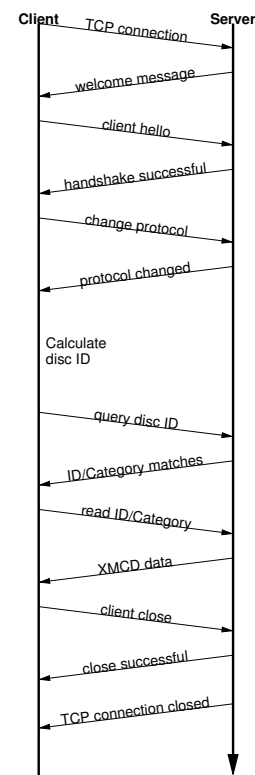
17 FreeDB

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 metadata 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.

17.1 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.



17.1.1 the Disc ID

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

Offset Seconds Digit Sum				Total Length in Seconds				Track Count			
0			7	8			23	24			31

‘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.

Track Number	Length			Offset		
	in M:SS	in seconds	in frames	in M:SS	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 $\frac{16340+15294+16340+15045}{75} = \mathbf{840}$

There are 75 frames per second, and 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 $2 + 2 + 1 + 9 + 4 + 2 + 3 + 6 + 4 + 1 = \mathbf{34}$

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

17.1.2 Initial Greeting

<i>From Server</i>	
<code><code></code>	<code><host>_CDDBP_server_<version>_ready_at_<datetime></code>

```

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

```

17.1.3 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_<client>_<version>
```

<code> 200 handshake successful
 <code> 402 already shook hands
 <code> 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

17.1.4 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>
```

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

17.1.5 Query Database

To Server

```
cddb_query<disc_id><track_count><offset_1><...><offset_n><seconds>
```

<disc_id> 32-bit disc ID
 <track_count> number of tracks in CD
 <offset> frame offset of each track
 <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>
<...>
.
```

	200	Found exact match
	211	Found inexact matches, list follows
<code>	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

17.1.6 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

17.1.7 Close Connection

To Server

```
quit
```

From Server

```
<code>_<hostname>_<message>
```

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

17.2 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 are no separate `hello`, `proto` and `quit` commands; these are issued along with the primary server command or are implied.

key	value
<code>hello</code>	<code><username> <hostname> <clientname> <version></code>
<code>proto</code>	<code><protocol></code>
<code>cmd</code>	<code><command></code>

Table 17.1: POST arguments

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
```

17.3 XMCD

XMCD files are text files encoded either in UTF-8, ISO-8859-1 or US-ASCII. All 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
<code>DISCID</code>	a comma-separated list of 32-bit disc IDs
<code>DTITLE</code>	an artist name and album name, separated by <code>' / '</code>
<code>DYEAR</code>	a 4 digit disc release year
<code>DGENRE</code>	the disc's FreeDB category string
<code>TITLEX</code>	the track title, or the track artist name and track title, separated by <code>' / '</code> X is an integer starting from 0
<code>EXTD</code>	extended data about the disc
<code>EXTTX</code>	extended data about the track X is an integer starting from 0
<code>PLAYORDER</code>	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.

18 MusicBrainz

MusicBrainz is another CD metadata retrieval service similar to FreeDB, but designed to eliminate many of FreeDB's limitations. For example, MusicBrainz has a more robust disc ID calculation mechanism, it has an easier way to disambiguate database entries in case of collision, and its XML metadata format is less prone to errors (track names with '/' characters are a particular problem for FreeDB).

However, because it is a newer service, it's common to find disc entries that are on FreeDB but do not yet have a MusicBrainz entry - whereas the converse is much more rare. Therefore, a metadata looking program would be wise to check both services if possible.

18.1 Searching Releases

This is analogous to FreeDB's search routine in which one calculates a CD's disc ID, submits it to MusicBrainz via an HTTP get query and receives information such as album name, artist name, track names and so forth as an XML file.

18.1.1 the Disc ID

Calculating a MusicBrainz disc ID requires knowing a CD's first track number, last track number, track offsets (in CD frames) and lead out track offset (also in CD frames). For example, given the following CD:

Track Number	Length			Offset		
	in M:SS	in seconds	in frames	in M:SS	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

The first track number is 1, the last track number is 4, the track offsets are 150, 16490, 31784 and 48124, and the lead out track offset is 63169 (track 4's offset 48124 plus its length of 15045).

These numbers are then converted to 0-padded, big-endian hexadecimal strings with the track numbers using 2 digits and the offsets using 8 digits. In this example, the first track number becomes 01, the last track number becomes 04, the track offsets become 00000096, 0000406A, 00007C28 and 0000BBFC, and the lead out track offset becomes 0000F6C1.

These individual strings are then combined into a single 804 byte string:

First Track Number	Last Track Number	Lead Out Offset	Offset ₁	Offset ₂	...	Offset ₉₉
015	0416	6316931	0000009695	0000406A127	00007C28128	0000BBFC191

Excess track offsets are treated as having an offset value of 0, or a string value of 00000000. Our string starts with 01040000F6C1000000960000406A00007C280000BBFC and is padded with an additional 760 '0' characters which I'll omit for brevity.

That string is then passed through the SHA-1 hashing algorithm¹ which results in a 20 byte hash value. Remember to use the binary hash value, not its 40 byte ASCII hexadecimal one.

In our example, this yields the hash: 0xDA3D930462773DD57BBE43B535AD6A457138F079

The resulting hash value is then encoded to a 28 byte Base64² string. However, unlike standard Base64, MusicBrainz's disc ID replaces the characters '=', '+' and '/' with '-', '.', and '_' respectively to make the value better suited to HTTP requests. So to complete our example, the hash value becomes a disc ID of 2j2TBGJ3PdV7vk01Na1qRXE48Hk-

¹This is described in RFC3174

²This is described in RFC3548 and RFC4648

18.1.2 Server Query

MusicBrainz runs as a service on HTTP port 80. To retrieve Release information, one can make a GET request to `/ws/1/release` using the following fields:

key	value
type	xml
discid	<disc ID string>

For example, to retrieve the Release data for disc ID `2jnmj7l5rSw0yVb_vlWAYkK_YBwk-` one sends the GET query:

```
type=xml&discid=2jnmj7l5rSw0yVb_vlWAYkK_YBwk-
```

Whether the Release is found in the MusicBrainz database or not, an XML file will always be generated.

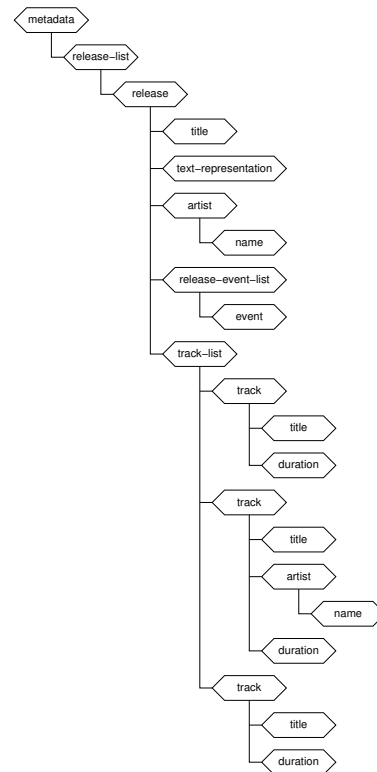
18.1.3 Release XML

All XML files returned by a MusicBrainz query consist of a `<metadata>` tag container. When making a Release query, it contains a `<release-list>` which is itself a container for zero or more `<release>` tags, depending on how many Release entries match the submitted disc ID.

The `<release>` tag typically contains a `<title>` which is the album's name, an `<artist>` tag which is the album's primary artist, a `<release-event-list>` tag containing information such as the album's release date and catalog number, and finally a `<track-list>` which contains all the track data.

The `<track>` tags are always listed in order of their appearance in the album. Each contains a `<title>` which is the track's name, a `<duration>` which is the track's length in milliseconds, and optionally an `<artist>` tag which is information about a track-specific artist, for instances where the track's artist differs from the album's artist.

In addition, the `<release>`, `<artist>`, and `<track>` tags all contain an 'id' attribute with 32 hex digits in the format `'12345678-9abc-def1-2345-6789abcdef1234'`. These uniquely identify the Release, Artist and Track information in the MusicBrainz database and can be used for direct lookups.



18.2 MusicBrainz XML

The following is the complete specification for MusicBrainz XML output in RELAX NG Compact syntax from <http://bugs.musicbrainz.org/browser/mmd-schema/trunk/schema> and converted to compact syntax for better readability.

```

Schema Start
-----

default namespace id3034801 = "http://musicbrainz.org/ns/mmd-1.0#"

namespace local = ""

namespace inh = inherit

start = def_metadata-element

def_metadata-element =
  element metadata
  {
    attribute generator { xsd:anyURI }?,
    attribute created { xsd:dateTime }?,
    def_artist-element?,
    def_release-element?,
    def_release-group-element?,
    def_track-element?,
    def_label-element?,
    def_artist-list?,
    def_release-list?,
    def_release-group-list?,
    def_track-list?,
    def_label-list?,
    def_metadata-element_extension
  }

def_artist-element =
  element artist
  {
    attribute id { xsd:anyURI }?,
    attribute type { xsd:anyURI }?,
    def_artist-attribute_extension,
    element name { text }?,
    element sort-name { text }?,
    element disambiguation { text }?,
    element life-span
    {
      attribute begin { def_incomplete-date }?,
      attribute end { def_incomplete-date }?
    }?,
    def_alias-list?,
    def_release-list?,
    def_release-group-list?,
    def_relation-list*,
    def_tag-list?,
    def_user-tag-list?,
    def_rating?,
    def_user-rating?,
    def_artist-element_extension
  }

```

```

def_release-element =
  element release
  {
    attribute id { xsd:anyURI }?,
    attribute type { def_URI-list }?,
    def_release-attribute_extension,
    element title { text }?,
    element text-representation
    {
      attribute language { def_iso-639 }?,
      attribute script { def_iso-15924 }?
    }?,
    element asin { xsd:string { pattern = "[A-Z0-9]{10}" } }?,
    def_artist-element?,
    def_release-group-element?,
    def_release-event-list?,
    def_disc-list?,
    def_puid-list?,
    def_track-list?,
    def_relation-list*,
    def_tag-list?,
    def_user-tag-list?,
    def_rating?,
    def_user-rating?,
    def_release-element_extension
  }

def_release-group-element =
  element release-group
  {
    attribute id { xsd:anyURI }?,
    attribute type { def_URI-list }?,
    def_release-group-attribute_extension,
    element title { text }?,
    def_artist-element?,
    def_release-list?,
    def_release-group-element_extension
  }

def_track-element =
  element track
  {
    attribute id { xsd:anyURI }?,
    def_track-attribute_extension,
    element title { text }?,
    element duration { xsd:nonNegativeInteger }?,
    element isrc-list { element isrc { attribute id { def_isrc } } * }?,
    def_artist-element?,
    def_release-list?,
    def_puid-list?,
    def_relation-list*,
    def_tag-list?,
    def_user-tag-list?,
    def_rating?,
    def_user-rating?,
    def_track-element_extension
  }

```

```

def_label-element =
  element label
  {
    attribute id { xsd:anyURI }?,
    attribute type { xsd:anyURI }?,
    def_label-attribute_extension,
    element name { text }?,
    element sort-name { text }?,
    element label-code { xsd:nonNegativeInteger }?,
    element disambiguation { text }?,
    element country { def_iso-3166 }?,
    element life-span
    {
      attribute begin { def_incomplete-date }?,
      attribute end { def_incomplete-date }?
    }?,
    def_alias-list?,
    def_release-list?,
    def_release-group-list?,
    def_relation-list*,
    def_tag-list?,
    def_user-tag-list?,
    def_rating?,
    def_user-rating?,
    def_label-element_extension
  }

def_relation-element =
  element relation
  {
    attribute type { xsd:anyURI },
    attribute target { xsd:anyURI },
    attribute direction { def_direction }?,
    attribute attributes { def_URI-list }?,
    attribute begin { def_incomplete-date }?,
    attribute end { def_incomplete-date }?,
    (
      def_artist-element
    | def_release-element
    | def_track-element
    | def_relation-element_extension
    )?
  }

def_alias =
  element alias
  {
    attribute type { xsd:anyURI }?,
    attribute script { def_iso-15924 }?,
    text
  }

def_tag = element tag { attribute count { xsd:nonNegativeInteger }?, text }

def_user-tag = element user-tag { text }

def_rating =
  element rating
  {

```

```

        attribute votes-count { xsd:nonNegativeInteger }?,
        xsd:float
    }

def_user-rating = element user-rating { xsd:nonNegativeInteger }

def_metadata-element_extension = def_extension_element?

def_artist-element_extension = def_extension_element*

def_release-element_extension = def_extension_element*

def_release-group-element_extension = def_extension_element*

def_track-element_extension = def_extension_element*

def_label-element_extension = def_extension_element*

def_relation-element_extension = def_extension_element

def_artist-attribute_extension = def_extension_attribute*

def_release-attribute_extension = def_extension_attribute*

def_release-group-attribute_extension = def_extension_attribute*

def_track-attribute_extension = def_extension_attribute*

def_label-attribute_extension = def_extension_attribute*

def_extension_element =
    element * - (id3034801:* | local:*)
    {
        ( attribute * { text } | text | def_anything )*
    }

def_extension_attribute = attribute * - (id3034801:* | local:*) { text }

def_anything =
    element * - local:* { ( attribute * { text } | text | def_anything )* }

def_artist-list =
    element artist-list { def_list-attributes, def_artist-element* }

def_release-list =
    element release-list { def_list-attributes, def_release-element* }

def_release-group-list =
    element release-group-list
    {
        def_list-attributes,
        def_release-group-element*
    }

def_alias-list = element alias-list { def_list-attributes, def_alias* }

def_track-list = element track-list { def_list-attributes, def_track-element* }

def_label-list = element label-list { def_list-attributes, def_label-element* }

```

```

def_release-event-list =
  element release-event-list
  {
    def_list-attributes,
    element event
    {
      attribute date { def_incomplete-date },
      attribute country { def_iso-3166 }?,
      attribute catalog-number { text }?,
      attribute barcode { text }?,
      attribute format { xsd:anyURI }?,
      def_label-element?
    }*
  }

def_disc-list =
  element disc-list
  {
    def_list-attributes,
    element disc
    {
      attribute id { xsd:string { pattern = "[a-zA-Z0-9._]{27}-" } },
      attribute sectors { xsd:nonNegativeInteger }?
    }*
  }

def_puid-list =
  element puid-list
  {
    def_list-attributes,
    element puid { attribute id { def_uuid } }*
  }

def_relation-list =
  element relation-list
  {
    attribute target-type { xsd:anyURI },
    def_list-attributes,
    def_relation-element*
  }

def_tag-list = element tag-list { def_list-attributes, def_tag* }

def_user-tag-list =
  element user-tag-list { def_list-attributes, def_user-tag* }

def_list-attributes =
  attribute count { xsd:nonNegativeInteger }?,
  attribute offset { xsd:nonNegativeInteger }?

def_URI-list = list { xsd:anyURI+ }

def_incomplete-date =
  xsd:string { pattern = "[0-9]{4}(-[0-9]{2})?(-[0-9]{2})?" }

def_iso-3166 = xsd:string { pattern = "[A-Z]{2}" }

def_iso-639 = xsd:string { pattern = "[A-Z]{3}" }

```



```
def_iso-15924 = xsd:string { pattern = "[A-Z][a-z]{3}" }  
def_isrc = xsd:string { pattern = "[A-Z]{2}[A-Z0-9]{3}[0-9]{2}[0-9]{5}" }  
def_uuid = xsd:string { pattern = "[0-9a-f]{8}(-[0-9a-f]{4}){3}-[0-9a-f]{12}" }  
def_direction = "both" | "forward" | "backward"
```

Schema End

19 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.

19.1 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:

$$\text{Output}_i = \text{Input}_i \times 10^{\frac{\text{gain}}{20}} \quad (19.1)$$

For example, if the gain is -2.19, each sample should be multiplied by $10^{\frac{-2.19}{20}}$ 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 ($0.9765625 \times 1.024 = 1.0$).

19.2 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:

$$\text{Output}_i = \frac{|\text{Input}_i|}{2^{\text{bits per sample}-1}} \quad (19.2)$$

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.

19.2.1 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:

$$\text{Output}_i = \left(\sum_{j=i-\text{order}}^i \text{Input}_j \times \text{Input Filter}_j \right) - \left(\sum_{k=i-\text{order}}^{i-1} \text{Output}_k \times \text{Output Filter}_k \right) \quad (19.3)$$

‘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.

a filtering example

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

sample	Input _{<i>i</i>}	Yule _{<i>i</i>}	Butter _{<i>i</i>}
89	-33	-14.90	
90	-32	-14.93	
91	-35	-14.65	
92	-32	-14.46	
93	-30	-14.15	
94	-32	-13.58	
95	-33	-13.18	
96	-30	-13.16	
97	-30	-13.12	0.41
98	-30	-12.89	0.61
99	-32	-12.81	0.66

If the value of sample 100 from the input stream is -30, here's how we calculate output sample 100:

sample	Input _{<i>i</i>}	Yule Input Filter _{<i>i</i>}	result	Yule _{<i>i</i>}	Yule Output Filter _{<i>i</i>}	result
90	-32 ×	-0.00187763777362 =	0.06	-14.93 ×	0.13149317958807999 =	-1.96
91	-35 ×	0.006746136822469999 =	-0.24	-14.65 ×	-0.75104302451432003 =	11.00
92	-32 ×	-0.0024087905158400001 =	0.08	-14.46 ×	2.1961168489077401 =	-31.76
93	-30 ×	0.016248649629749999 =	-0.49	-14.15 ×	-4.3947099607955904 =	62.19
94	-32 ×	-0.025963385129149998 =	0.83	-13.58 ×	6.8540154093699801 =	-93.08
95	-33 ×	0.022452932533390001 =	-0.74	-13.18 ×	-8.8149868137015499 =	116.18
96	-30 ×	-0.008349909049359996 =	0.25	-13.16 ×	9.4769360780128 =	-124.72
97	-30 ×	-0.0085116564546900003 =	0.26	-13.12 ×	-8.5475152747187408 =	112.14
98	-30 ×	-0.0084870937985100006 =	0.25	-12.89 ×	6.3631777756614802 =	-82.02
99	-32 ×	-0.029110078089480001 =	0.93	-12.81 ×	-3.4784594855007098 =	44.56
100	-30 ×	0.054186564064300002 =	-1.63			
input values sum =			-0.44	output values sum = 12.53		

Therefore, $Yule_{100} = -0.44 - 12.53 = -12.97$

We're not quite done yet. Remember, ReplayGain's equal loudness filter requires both a Yule *and* Butter filter, in that order. Notice how Butter's input samples are Yule's output samples. Thus, our next input sample to the Butter filter is -12.97. Calculating sample 100 is now a similar process:

sample	Yule _{<i>i</i>}		Butter Input Filter _{<i>i</i>}		result	Butter _{<i>i</i>}		Butter Output Filter _{<i>i</i>}		result	
98	-12.89	×	0.98500175787241995	=	-12.70	0.61	×	0.97022847566350001	=	0.59	
99	-12.81	×	-1.9700035157448399	=	25.24	0.66	×	-1.96977855582618	=	-1.30	
100	-12.97	×	0.98500175787241995	=	-12.78						
input values sum					=	-0.24	output values sum = -0.71				

Therefore, $Butter_{100} = -0.24 - -0.71 = 0.47$, which is the next sample from the equal loudness filter.

19.2.2 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.

$$\text{Block dB}_i = 10 \times \log_{10} \left(\frac{\left(\frac{\sum_{x=0}^{\text{Block Length}-1} \text{Left Sample}_x^2}{\text{Block Length}} \right) + \left(\frac{\sum_{y=0}^{\text{Block Length}-1} \text{Right Sample}_y^2}{\text{Block Length}} \right)}{2} + 10^{-10} \right) \quad (19.4)$$

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	

$$\frac{\left(\frac{7106715}{2205} \right) + \left(\frac{11642400}{2205} \right)}{2} = 4251 \quad (19.5)$$

$$10 \times \log_{10}(4251 + 10^{-10}) = 36.28 \quad (19.6)$$

Thus, the decibel value of this block is 36.28.

19.2.3 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 ($64.82 - 67.01 = -2.19$).

Appendices

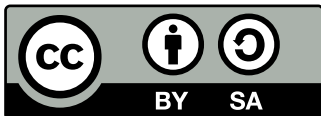
A References

- Wave File Format Specifications
<http://www-mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/WAVE.html>
- Audio File Format Specifications
<http://www-mmsp.ece.mcgill.ca/Documents/AudioFormats/AIFF/AIFF.html>
- AU Audio File Format
<http://www.opengroup.org/public/pubs/external/auformat.html>
- Shorten Research Paper
ftp://svr-ftp.eng.cam.ac.uk/pub/reports/robinson_tr156.ps.Z
- 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.datavoyage.com/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
- Proposals for Extending Ogg Vorbis Comments
<http://www.reallylongword.org/articles/vorbiscomment/>

A References

- 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
- ALAC decoder
<http://craz.net/programs/itunes/alac.html>
- ISO 14496-1 Media Format
<http://xhelmboyx.tripod.com/formats/mp4-layout.txt>
- FreeDB Information
http://www.freedb.org/en/download_miscellaneous.11.html
- ReplayGain
<http://replaygain.hydrogenaudio.org>

B License



THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE (“CCPL” OR “LICENSE”). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

B.1 Definitions

1. **“Adaptation”** means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image (“synching”) will be considered an Adaptation for the purpose of this License.
2. **“Collection”** means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(f) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the Work is included in its entirety in unmodified form along with one or more other contributions, each constituting separate and independent works in themselves, which

together are assembled into a collective whole. A work that constitutes a Collection will not be considered an Adaptation (as defined below) for the purposes of this License.

3. **“Creative Commons Compatible License”** means a license that is listed at <http://creativecommons.org/compatiblelicenses> that has been approved by Creative Commons as being essentially equivalent to this License, including, at a minimum, because that license: (i) contains terms that have the same purpose, meaning and effect as the License Elements of this License; and, (ii) explicitly permits the relicensing of adaptations of works made available under that license under this License or a Creative Commons jurisdiction license with the same License Elements as this License.
4. **“Distribute”** means to make available to the public the original and copies of the Work or Adaptation, as appropriate, through sale or other transfer of ownership.
5. **“License Elements”** means the following high-level license attributes as selected by Licensor and indicated in the title of this License: Attribution, ShareAlike.
6. **“Licensor”** means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.
7. **“Original Author”** means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in addition (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore; (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and, (iii) in the case of broadcasts, the organization that transmits the broadcast.
8. **“Work”** means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent

it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

9. **“You”** means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.
10. **“Publicly Perform”** means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.
11. **“Reproduce”** means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

B.2 Fair Dealing Rights.

Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

B.3 License Grant.

Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

1. to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections;
2. to create and Reproduce Adaptations provided that any such Adaptation, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked “The original work was translated from English to Spanish,” or a modification could indicate “The original work has been modified.”;

3. to Distribute and Publicly Perform the Work including as incorporated in Collections; and,
4. to Distribute and Publicly Perform Adaptations.
5. For the avoidance of doubt:
 - a) **Non-waivable Compulsory License Schemes.** In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;
 - b) **Waivable Compulsory License Schemes.** In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor waives the exclusive right to collect such royalties for any exercise by You of the rights granted under this License; and,
 - c) **Voluntary License Schemes.** The Licensor waives the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License.

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. Subject to Section 8(f), all rights not expressly granted by Licensor are hereby reserved.

B.4 Restrictions.

The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

1. You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License.

If You create a Collection, upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(c), as requested. If You create an Adaptation, upon notice from any Licensor You must, to the extent practicable, remove from the Adaptation any credit as required by Section 4(c), as requested.

2. You may Distribute or Publicly Perform an Adaptation only under the terms of: (i) this License; (ii) a later version of this License with the same License Elements as this License; (iii) a Creative Commons jurisdiction license (either this or a later license version) that contains the same License Elements as this License (e.g., Attribution-ShareAlike 3.0 US)); (iv) a Creative Commons Compatible License. If you license the Adaptation under one of the licenses mentioned in (iv), you must comply with the terms of that license. If you license the Adaptation under the terms of any of the licenses mentioned in (i), (ii) or (iii) (the “Applicable License”), you must comply with the terms of the Applicable License generally and the following provisions: (I) You must include a copy of, or the URI for, the Applicable License with every copy of each Adaptation You Distribute or Publicly Perform; (II) You may not offer or impose any terms on the Adaptation that restrict the terms of the Applicable License or the ability of the recipient of the Adaptation to exercise the rights granted to that recipient under the terms of the Applicable License; (III) You must keep intact all notices that refer to the Applicable License and to the disclaimer of warranties with every copy of the Work as included in the Adaptation You Distribute or Publicly Perform; (IV) when You Distribute or Publicly Perform the Adaptation, You may not impose any effective technological measures on the Adaptation that restrict the ability of a recipient of the Adaptation from You to exercise the rights granted to that recipient under the terms of the Applicable License. This Section 4(b) applies to the Adaptation as incorporated in a Collection, but this does not require the Collection apart from the Adaptation itself to be made subject to the terms of the Applicable License.
3. If You Distribute, or Publicly Perform the Work or any Adaptations or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution (“Attribution Parties”) in Licensor’s copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and (iv) , consistent with Ssection 3(b), in the case of an Adaptation, a credit identifying the use of the Work in the Adaptation

(e.g., “French translation of the Work by Original Author,” or “Screenplay based on original Work by Original Author”). The credit required by this Section 4(c) may be implemented in any reasonable manner; provided, however, that in the case of a Adaptation or Collection, at a minimum such credit will appear, if a credit for all contributing authors of the Adaptation or Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

4. Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Adaptations or Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author’s honor or reputation. Licensor agrees that in those jurisdictions (e.g. Japan), in which any exercise of the right granted in Section 3(b) of this License (the right to make Adaptations) would be deemed to be a distortion, mutilation, modification or other derogatory action prejudicial to the Original Author’s honor and reputation, the Licensor will waive or not assert, as appropriate, this Section, to the fullest extent permitted by the applicable national law, to enable You to reasonably exercise Your right under Section 3(b) of this License (right to make Adaptations) but not otherwise.

B.5 Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

B.6 Limitation on Liability.

EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

B.7 Termination

1. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Adaptations or Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.
2. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

B.8 Miscellaneous

1. Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.
2. Each time You Distribute or Publicly Perform an Adaptation, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.
3. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

B License

4. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.
5. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.
6. The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be enforced according to the corresponding provisions of the implementation of those treaty provisions in the applicable national law. If the standard suite of rights granted under applicable copyright law includes additional rights not granted under this License, such additional rights are deemed to be included in the License; this License is not intended to restrict the license of any rights under applicable law.