WavPack 4 & 5 Binary File / Block Format

David Bryant
April 12, 2020

1.0 Introduction

A WavPack 4.0 or 5.0 file consists of a series of WavPack audio blocks. It may also contain tags and other information, but these must be outside the blocks (either before, in-between, or after) and are ignored for the purpose of unpacking audio data. The WavPack blocks are easy to identify by their unique header data, and by looking in the header it is very easy to determine the total size of the block (both in physical bytes and compressed samples) and the audio format stored. The first block containing audio determines the format of the entire file. There are no specialized seek tables.

The blocks are completely independent in that they can be decoded to mono or stereo audio all by themselves. The blocks may contain any number of samples (well, up to 131072), either stereo or mono. Obviously, putting more samples in each block is more efficient because of reduced header overhead, but they are reasonably efficient down to even a thousand samples. For version 5.0 the default number of samples stored in a block has been reduced by half to improve seeking performance. The max size is 1 MB for the whole block, but this is arbitrary (and blocks will generally be much smaller). The blocks may be lossless or lossy. Currently the hybrid/lossy modes are basically CBR, but the format can support quality-based VBR also.

For multichannel audio, the data is divided into some number of stereo and mono streams and multiplexed into separate blocks which repeat in sequence. A flag in the header indicates whether the block is the first or the last in the sequence (for simple mono or stereo files both of these would always be set). The speaker assignments are in standard Microsoft order and the `channel_mask` is transmitted in a separate piece of metadata. Channels that naturally belong together (i.e. left and right pairs) are put into stereo blocks for more efficient encoding. So, for example, a standard 5.1 audio stream would have a `channel_mask` of `0x3F` and be organized into 4 blocks in sequence:

1. stereo block (front left + front right) (`INITIAL_BLOCK`)
2. mono block (front center)
3. mono block (low frequency effects)
4. stereo block (back left + back right) (`FINAL_BLOCK`)

Correction files (.wvc) have an identical structure to the main file (.wv) and there is a one-to-one correspondence between main file blocks that contain audio and their correction file match (blocks that do not contain audio do not exist in the correction file). The only difference in the headers of main blocks and correction blocks is the size and the CRC value, although it is easy (if a little ugly) to tell the blocks apart by looking at the metadata ids.

The format is designed with hardware decoding in mind, and so it is possible to decode regular stereo (or mono) WavPack files without buffering an entire block, which allows the memory requirements to be reduced to only a few kilobytes if desired. This is not true of multichannel files, and this also restricts playback of high-resolution files to 24 bits of precision (although neither of these would be associated with low-cost playback equipment).

2.0 Block Header

Here is the 32-byte little-endian header at the front of every WavPack block:

```
typedef struct {
    char ckID [4];          // "wvpk"
    uint32_t ckSize;        // size of entire block (minus 8)
    uint16_t version;       // 0x402 to 0x410 are valid for decode
    uchar block_index_u8;   // upper 8 bits of 40-bit block_index
    uchar total_samples_u8; // upper 8 bits of 40-bit total_samples
    uint32_t total_samples; // lower 32 bits of total samples for
                            // entire file, but this is only valid
                            // if block_index == 0 and a value of -1
                            // indicates an unknown length
    uint32_t block_index;   // lower 32 bit index of the first sample
                            // in the block relative to file start,
                            // normally this is zero in first block
    uint32_t block_samples; // number of samples in this block, 0 =
                            // non-audio block
    uint32_t flags;         // various flags for id and decoding
    uint32_t crc;           // crc for actual decoded data
} WavpackHeader;
```

Note that in this context the meaning of "samples" refers to a complete sample for all channels (sometimes called a "frame"). Therefore, in a stereo or multichannel file the actual number of numeric samples is this value multiplied by the number of channels. For version 5.0, this was extended from 32 bits to 40 bits with the upper 8 bits placed in previously unused bytes in the header. Note that the 40-bit `total_samples` reserves values with the lower 32 bits all set to represent an unknown length, so loading and storing this is a little tricky (see `src/wavpack_local.h` for macros to do this).

Normally, the first block of a WavPack file that contains audio data (blocks may contain *only* metadata, especially at the beginning and end of a file) would have `block_index` == 0 and `total_samples` would be equal to the total number of samples in the file. However, there are exceptions to this rule. For example, a file may be created such that its total length is unknown (i.e. with pipes) and in this case the lower 32 bits of `total_samples` are 1. For these files, the WavPack decoder will attempt to seek to the end of the file to determine the actual length, and if this fails then the length is simply unknown.

Another case is where a WavPack file is created by cutting a portion out of a longer WavPack file (or from a WavPack stream). Since this file would start with a block that didn't have `block_index` == 0, the length would be unknown until a seek to end was performed.

It is also possible to have streamed WavPack data. In this case both the `block_index` and `total_samples` fields are ignored for every block and the decoder simply decodes every block encountered indefinitely.

Note that the first block that contains audio samples in a WavPack file determines the format of the entire file (i.e., the format may not change arbitrarily). This refers to sample bit depth and format (i.e., integer/float), sample rate, and channel count and layout. However, other characteristics that are transparent to the library client *may* change (e.g., lossless vs hybrid, hybrid bitrate, and speed modes).

The `flags` field contains information for decoding the block along with some general information including sample size and format, hybrid/lossless, mono/stereo and sampling rate (if one of 15 standard rates). Here are the (little-endian) bit assignments:

```
bits 1,0:     //  00 = 1 byte / sample (1-8 bits / sample)
              //  01 = 2 bytes / sample (9-16 bits / sample)
              //  10 = 3 bytes / sample (15-24 bits / sample)
              //  11 = 4 bytes / sample (25-32 bits / sample)
bit 2:        //  0 = stereo output; 1 = mono output
bit 3:        //  0 = lossless mode; 1 = hybrid mode
bit 4:        //  0 = true stereo; 1 = joint stereo (mid/side)
bit 5:        //  0 = independent channels; 1 = cross-channel decorrelation
bit 6:        //  0 = flat noise spectrum in hybrid; 1 = hybrid noise shaping
bit 7:        //  0 = integer data; 1 = floating point data
bit 8:        //  1 = extended size integers (> 24-bit) or shifted integers
bit 9:        //  0 = hybrid mode parameters control noise level (not used yet)
              //  1 = hybrid mode parameters control bitrate
bit 10:       //  1 = hybrid noise balanced between channels
bit 11:       //  1 = initial block in sequence (for multichannel)
bit 12:       //  1 = final block in sequence (for multichannel)
bits 17-13:   //  amount of data left-shift after decode (0-31 places)
bits 22-18:   //  maximum magnitude of decoded data
              //    (number of bits integers require minus 1)
bits 26-23:   //  sampling rate (1111 = unknown/custom)
bit 27:       //  reserved (but decoders should ignore if set)
bit 28:       //  block contains checksum in last 2 or 4 bytes (ver 5.0+)
bit 29:       //  1 = use IIR for negative hybrid noise shaping
bit 30:       //  1 = false stereo (data is mono but output is stereo)
bit 31:       //  0 = PCM audio; 1 = DSD audio (ver 5.0+)
```

3.0 Metadata Sub-Blocks

Following the 32-byte header to the end of the block are a series of "metadata" sub-blocks. These may be from 2 bytes long to the size of the entire block and are extremely easy to parse (even without knowing what they mean). These mostly contain extra information needed to decode the audio, but may also contain user information that is not required for decoding and that could be used in the future without breaking existing decoders. The final sub-block is usually the compressed audio bitstream itself, although this is not a strict rule. For version 5.0 a checksum block of 4 or 6 bytes (total) was added beyond that, although again this would be ignored by previous decoders.

The format of the metadata is:

```
uchar id;             // mask  meaning
                      // ----  -------
                      // 0x3f  unique metadata function id
                      // 0x20  decoder needn't understand metadata
                      // 0x40  actual data byte length is 1 less
                      // 0x80  large block (> 255 words)

uchar ws;             // if small block: data size in words
  ...or...
uchar ws [3];         // if large block: data size in words (le)

uint16_t data [ws];   // data, padded to an even number of bytes
```

The data portions are either “small” (<= 510 bytes) or “large” (can be very large). It is also possible to have no data at all in the sub-block (small, `ws` = 0), in which case the sub-block would occupy only 2 bytes but could still signal something by its presence. Because of the design of the sub-block, its total length will always be even and will always be aligned on an even address (even though its actual data length will be odd if the `0x40` mask is set in the id). The currently assigned metadata ids are:

```
ID_DUMMY              0x0   // could be used to pad WavPack blocks
ID_DECORR_TERMS       0x2   // decorrelation terms & deltas (fixed)
ID_DECORR_WEIGHTS     0x3   // initial decorrelation weights
ID_DECORR_SAMPLES     0x4   // decorrelation sample history
ID_ENTROPY_VARS       0x5   // initial entropy variables
ID_HYBRID_PROFILE     0x6   // entropy variables specific to hybrid mode
ID_SHAPING_WEIGHTS    0x7   // info needed for hybrid lossless (wvc) mode
ID_FLOAT_INFO         0x8   // specific info for floating point decode
ID_INT32_INFO         0x9   // specific info for decoding integers > 24
                            // bits, or data requiring shift after decode
ID_WV_BITSTREAM       0xa   // normal compressed audio bitstream (wv file)
ID_WVC_BITSTREAM      0xb   // correction file bitstream (wvc file)
ID_WVX_BITSTREAM      0xc   // special extended bitstream for floating
                            // point data or integers > 24 bit (can be
                            // in either wv or wvc file, depending...)
ID_CHANNEL_INFO       0xd   // contains channel count and channel_mask
ID_DSD_BLOCK          0xe   // contains compressed DSD audio (ver 5.0+)
```

```
// ids from here are “optional” so decoders should skip them if they don't understand them

ID_RIFF_HEADER          0x21   // RIFF header for .wav files (before audio)
ID_RIFF_TRAILER         0x22   // RIFF trailer for .wav files (after audio)
ID_CONFIG_BLOCK         0x25   // some encoding details for info purposes
ID_MD5_CHECKSUM         0x26   // 16-byte MD5 sum of raw audio data
ID_SAMPLE_RATE          0x27   // non-standard sampling rate info

// added with version 5.0 to handle non-wav files and block checksums:

ID_ALT_HEADER           0x23   // header for non-wav files (ver 5.0+)
ID_ALT_TRAILER          0x24   // trailer for non-wav files (ver 5.0+)
ID_ALT_EXTENSION        0x28   // target filename extension
ID_ALT_MD5_CHECKSUM     0x29   // 16-byte MD5 sum of raw audio data with non-
                               // wav standard (e.g., big-endian)
ID_NEW_CONFIG_BLOCK     0x2a   // new file configuration stuff including file
                               // type, non-wav formats (e.g., big endian),
                               // and CAF channel layouts and reordering
ID_CHANNEL_IDENTITIES   0x2b   // identities of non-MS channels
ID_BLOCK_CHECKSUM       0x2f   // 2- or 4-byte checksum of entire block
```

Note: unlisted ids are reserved.

The RIFF header and trailer are optional for most playback purposes, however older decoders (< 4.40) will not decode to .wav files unless at least the `ID_RIFF_HEADER` is present.

4.0 METADATA TAGS

These tags are not to be confused with the metadata sub-blocks described above but are specialized tags for storing user data on many formats of audio files. The tags recommended for use with WavPack files (and the ones that the WavPack supplied plugins and programs will work with) are ID3v1 and APEv2. The ID3v1 tags are somewhat primitive and limited, but are supported for legacy purposes. The more recommended tagging format is APEv2 because of its rich functionality and broad software support (it is also used on Monkey's Audio and Musepack files). Both the APEv2 tags and/or ID3v1 tags must come at the end of the WavPack file, with the ID3v1 coming last if both are present.

For the APEv2 tags, the following field names are officially supported and recommended by WavPack (although there are no restrictions on what field names may be used):

```
Artist
Title
Album
Track
Year
Genre
Comment
Cuesheet (note: may include replay gain info as remarks)
Encoder (note: can be automatically generated in ver 5.0+)
Settings (note: can be automatically generated in ver 5.0+)
Replaygain_Track_Gain
Replaygain_Track_Peak
Replaygain_Album_Gain
Replaygain_Album_Peak
Cover Art (Front)
Cover Art (Back)
Log
```