WavPack 5 Porting Guide For Developers

David Bryant
November 20, 2016

1.0 Introduction

Unlike the transition to WavPack 4, which was a complete rewrite with a totally new API, the transition to WavPack 5 is only an incremental step. Many new features and capabilities have been added, but for many common applications (e.g., CD archiving) these new features are probably irrelevant. Therefore, the legacy APIs are all retained and existing applications that worked with the previous version of the library will continue to work unmodified. It is only if the new features of WavPack are desired by an application will any modifications be required. This document will outline the new features introduced in WavPack 5 and list the new APIs, and is intended be used in conjunction with the *WavPack 5 Library Documentation* and the `wavpack.h` header file.

2.0 New Features

The purpose of WavPack 5 was to eliminate the last nagging, artificial limitations that existed in the previous WavPack versions, and to introduce a few extra capabilities that will allow WavPack to serve as a "universal" lossless audio format. Specifically, those enhancements are:

1. Eliminate the 2 GB size limit for seekable WavPack files. Related to this on the application side is support for the RF64 extension of the WAV format that allows files over 4 GB.
2. Increase the maximum number of samples in a seekable WavPack file from about 2^32 to about 2^40.
3. Allow alternate file formats in addition to Microsoft WAV/RF64 to be handled, including those with different audio data formats, such as big-endian and signed bytes. Currently on the application side this includes Apple's Core Audio Format and Sony's Wave64 format, plus two DSD formats (see #5 below).
4. Allow more flexible channel configurations, including reordered channels and channels not defined by the Microsoft channel mask.
5. Add two modes of lossless compression for 1-bit PCM audio (also known as DSD). On the application side this includes two new file formats (Philips DSDIFF and Sony's DSD Stream Format).
6. Update the WavPack stream version from `0x407` to `0x410`, thereby making the "mono optimization" feature enabled by default. This has possible implications for very old decoders, specifically the one in WinZip.
7. Add a block-level checksum to improve robustness with corrupt files. This has possible implications for streaming applications that might not provide verbatim blocks (Matroska).

3.0 File Size Limit

The 2 GB size limit for seekable WavPack files came from the original `WavpackStreamReader` callback functions that had 32-bit arguments. This has been replaced with a new set of callbacks that have 64-bit arguments where required, and also include the previously missing “truncate” and “close” functionalities. The new structure is:

```
typedef struct {
   int32_t (*read_bytes)(void *id, void *data, int32_t bcount);
   int32_t (*write_bytes)(void *id, void *data, int32_t bcount);
   int64_t (*get_pos)(void *id);
   int (*set_pos_abs)(void *id, int64_t pos);
   int (*set_pos_rel)(void *id, int64_t delta, int mode);
   int (*push_back_byte)(void *id, int c);
   int64_t (*get_length)(void *id);
   int (*can_seek)(void *id);
   int (*truncate_here)(void *id);
   int (*close)(void *id);
} WavpackStreamReader64;
```

This structure is used in the new file open API:

```
WavpackContext *WavpackOpenFileInputEx64 (
    WavpackStreamReader64 *reader,
    void *wv_id,
    void *wvc_id,
    char *error,
    int flags,
    int norm_offset
);
```

However, as mentioned above, the old API using the old callbacks is still available and now provides a translation layer between the two interfaces. Of course, seeking within files over 2 GB cannot be handled without moving to the new structure.

Note that the new set of callbacks includes a “close” function that did not previously exist. If provided, this callback is called for each file (wv and wvc) when `WavpackCloseFile()` is called. If the application wants to handle closing the files itself, simply provide a `NULL` pointer for this callback.

In addition, the function to obtain the total size of the WavPack file(s) was updated to handle large files:

```
int64_t WavpackGetFileSize64 (WavpackContext *wpc);
```

4.0 Number of Samples Limit

The existing sample count limit was probably sufficient for most applications, especially because it actually referred to frames of samples, which meant that multichannel files did not needlessly increase this number (e.g., at 48 kHz sampling rate, 2^32 samples is enough for over 24 hours of audio, regardless of the channel count or bit depth).

However, because high sample rates are becoming more common (especially with DSD) and some people were using WavPack for non-audio applications (e.g., RF sampling), it was decided that eliminating this arbitrary limit would be potentially useful. There were two unused bytes in the WavPack header, and by incorporating these it was possible to increase the maximum sample count to almost 2^40, which should be sufficient (the *smallest* possible WavPack file that could have 2^40 samples would be over 300 TB long).

Again, all the existing APIs still function and are sufficient for virtually any application, but these new APIs are provided to support the enhanced capabilities (and should probably be migrated to over time):

```
int64_t WavpackGetNumSamples64 (WavpackContext *wpc);
int64_t WavpackGetSampleIndex64 (WavpackContext *wpc);
int WavpackSeekSample64 (WavpackContext *wpc, int64_t sample);

int WavpackSetConfiguration64 (
    WavpackContext *wpc,
    WavpackConfig *config,
    int64_t total_samples,
    const unsigned char *chan_ids
);

#define MAX_WAVPACK_SAMPLES ((1LL << 40) - 257)
```

5.0 Alternate File Formats

Previous versions of WavPack could only handle audio in the Microsoft Waveform Audio format (WAV). Of course, raw audio was supported, but even this was restricted to WAV audio data formatting (i.e., little-endian, unsigned bytes) and raw files would be restored to WAV files unless specifically overridden. The 4 GB limit of WAV files was solved by two competing new formats (RF64 and Wave64) and Apple created a new, very flexible format called Core Audio Format (CAF). Several new items needed to be stored in WavPack files to accommodate handling these additional input file formats:

1. A numeric value indicating the source file format
2. The source file's extension (we could no longer assume WAV)
3. Information about non-WAV data formatting (e.g., big-endian, signed bytes, etc.)
4. Possible channel reordering (CAF-specific “layouts”)

It was important that existing applications be able to decode and play the new files, but obviously there were limitations on how complete the support for them could be. For example, we would not want an old decoder to restore a CAF file's headers, but put a .WAV extension on the file and and store the audio data as little-endian even though the header indicated big-endian!

The solution to this is that headers and trailers for non-wav files are stored with a different metadata ID than regular WAV headers, and old applications cannot access those headers. Also, the MD5 checksum for files that have modified data formats (e.g., big-endian, reordered channels) is stored with a different ID. Old applications will simply see a “raw” file, and may not be able to access the stored MD5, but otherwise will work fine.

A new flag `OPEN_ALT_TYPES` was added to the `WavpackOpenFileInput**` functions that indicates to the library that the application understands alternate file types and will use the following functions to obtain information about the file so that it can be restored (or MD5 summed) properly. Of course, most applications will not be concerned with this because they are generally interested in simply accessing the audio samples.

```
unsigned char WavpackGetFileFormat (WavpackContext *wpc);
char *WavpackGetFileExtension (WavpackContext *wpc);
int WavpackGetQualifyMode (WavpackContext *wpc);

uint32_t WavpackGetChannelLayout (
    WavpackContext *wpc,
    unsigned char *reorder
);
```

6.0 Alternate Channel Configurations

Previous versions of WavPack honored the multichannel limitations of the Microsoft Waveform File Format. Specifically, that there are only 18 defined channel identities, and that the channels in a file matching those channels must occur first in the file and be in a specific order. A 32-bit channel mask identifies the channels present in a file. A file could contain unidentified (or unassigned) channels, but these must be stored *after* the defined channels, and there was no standard way of identifying these channels.

For compatibility sake I have decided to retain the rule that the 18 Microsoft channels must come first in the file (and in that order), but two new capabilities have been added to support the features of the CAF file format. The first is that the channels may be reordered after decoding, either to match one of Core Audio's "layouts", or to an arbitrary layout. Note that this reordering is only required if an attempt is made to restore the original CAF file or to verify the MD5 checksum stored in a file, and that reordering is performed by the application, ***not the library***.

The second added capability is to allow non-Microsoft channels to have identities. These identities are numbered from 1 to 254, with 1-18 being the 18 Microsoft channels, 30 and 31 are reserved for the BWF "stereo downmix" (although I'm not sure that this has ever been used), and other values reserved for Core Audio channels and some Adobe Amio channel identities, and some values still free for future use (see `pack_utils.c` for the current assignments). A value of 255 indicates a truly undefined channel. These channels so defined may be in any order, but must still be *after* the ordered Microsoft channels.

These new functions are related to the new channel support:

```
void WavpackGetChannelIdentities (
    WavpackContext *wpc,
    unsigned char *identities
);

uint32_t WavpackGetChannelLayout (
    WavpackContext *wpc,
    unsigned char *reorder
);

int WavpackSetConfiguration64 (
    WavpackContext *wpc,
    WavpackConfig *config,
    int64_t total_samples,
    const unsigned char *chan_ids
);

int WavpackSetChannelLayout (
    WavpackContext *wpc,
    uint32_t layout_tag,
    const unsigned char *reorder
);
```

7.0 DSD Audio Compression

Sony developed DSD (Direct Stream Digital) to maximize the quality of audio transfers of analog masters, and later Sony and Philips incorporated this technology into the SACD format as a replacement for the CD (whose patents were running out). The format never really caught on, and there are serious debates as to whether it really offers any audible advantage over standard 16/44 PCM audio, but many recordings have been made available in DSD and many audiophiles swear by it. Therefore, it seemed reasonable to include lossless compression of DSD audio in WavPack.

Unfortunately, none of the regular audio container formats easily accommodated 1-bit PCM, and so both Sony and Philips developed their own new formats for storing DSD. Both of these are now supported in WavPack. The hybrid mode is not supported (DSD audio is always lossless in a single file), and there are only two modes: the default which is very fast and provides an average of about 45% data reduction, and a "high" mode that provides about 12% or so more compression, but requires considerably more CPU to encode and decode (although still not as much as the DST compression used in some SACDs).

From an application point of view under WavPack, DSD audio "samples" are actually a byte of 8 consecutive 1-bit samples. That means, for example, that seeking can only be performed to an 8-sample boundary, and that functions to obtain the sample count in a file will actually be querying the number of sample "bytes" in the file. This was chosen to eliminate the strange situation where a seek would be performed to a "mid-byte" position and all samples retrieved after that would be shuffled around, and also because WavPack is actually storing DSD samples as bytes. This also means that the sample rate returned by `WavpackGetSampleRate()` is the number of *bytes* per second, not bits per second (however there is a new function `WavpackGetNativeSampleRate()` that avoids this but should only be used for reporting the sample rate to the user).

An application has three options for dealing with WavPack files containing DSD audio. The first is to do nothing, in which case WavPack will report an error when attempting to open such a file (the error is "not configured to handle DSD WavPack files!"). This is what unmodified applications will see.

The second option is to include the flag `OPEN_AS_PCM` when opening the WavPack file. In this case, the DSD file will be appear as 24-bit PCM, and will automatically be decimated 8X during decoding. This allows applications that cannot natively handle DSD audio to do so. Note that this will still result in a very high sampling rate (352.8 kHz for DSD64) with lots of remaining quantization noise and should therefore be further down-sampled before use. This is how the WavPack Cool Edit and Audition filters and the winamp plugin handle DSD audio (the winamp plugin downsamples another 4x for playback while the Audition filters assume that the user will do this as they see fit). In this mode `WavpackGetBitsPerSample()` returns 24 and `WavpackGetBytesPerSample()` returns 3 when a DSD file is opened.

The third option is to include the flag `OPEN_DSD_NATIVE` when opening the WavPack file. In this case, the DSD file will be appear as 8-bit PCM, but each 8-bit PCM sample returned will actually be 8 consecutive 1-bit DSD samples, with the MSB being first temporally (same as DSDIFF). The channels are interleaved at the sample level (just like PCM) and these 8-bit DSD bytes are returned in the lower 8 bits of 32-bit integers (just like 8-bit PCM is returned). In this mode `WavpackGetBitsPerSample()` returns 8 and `WavpackGetBytesPerSample()` returns 1 when a DSD file is opened.

A DSD file is detected from the application side by checking for either of the `QMODE_DSD_xSB_FIRST` bits being set in the `WavpackGetQualifyMode()` mask, whether or not conversion to PCM is performed.

The WavPack library does not contain a DSD encoder, so DSD must be provided natively, and this is done in the exact same format as decoding (8 bits of DSD audio contained in the lower bits of each 32-bit integer). To signal DSD audio, the application must set either the `QMODE_DSD_LSB_FIRST` or `QMODE_DSD_MSB_FIRST` bit in the `qmode` field of the `WavpackContext` structure and set `bytes_per_sample = 1` and `bits_per_sample = 8`. Again, keep in mind that for setting sample rate, number of samples, etc., the DSD “samples” are considered an 8-bit byte's worth of audio data. Also, setting `QMODE_DSD_LSB_FIRST` does **not** affect how the WavPack library interprets the DSD bytes; this is there for informational purposes only (Sony's DSF files can have the data ordered either way, and we have to remember which it is).

8.0 WavPack Stream Version Update

The previous release of the library (4.80.0) generated by default a stream version (`0x407`) compatible with all decoders back to WavPack 4.0. If the `CONFIG_OPTIMIZE_MONO` flag was specified, that encoder would generate a newer stream version (`0x410`) that was not compatible with some very old decoders (before 4.3) but would more efficiently encode stereo streams that were actually mono (kind of an edge case, but not unheard of).

For 5.0.0, the new stream is the default and the `CONFIG_OPTIMIZE_FLAG` is ignored (because that optimization is always active). This is normally inconsequential, but could cause trouble with very old decoders, specifically the decoder built into WinZip, but perhaps others.

To fix this, a new configuration flag was added called `CONFIG_COMPATIBLE_WRITE` which forces the older stream and, of course, disables the mono optimization. To make sure that no application unintentionally generates newer streams (e.g., for WinZip) this flag is *automatically* added when the legacy function `WavpackSetConfiguration()` is called. But if an application is updated to use the new `WavpackSetConfiguration64()` then this should be considered.

9.0 WavPack Block Checksums

WavPack 5 adds a checksum to the end of each generated block, and this checksum is verified during decoding before a block is even parsed. This makes the format more robust to corrupt blocks and prevents even short bursts of noise from appearing when decoding damaged files. In virtually all situations this is transparent because even the oldest decoders simply ignore the checksums.

However, some streaming formats (Matroska) remove the header of the WavPack blocks to store them more efficiently and then fail to restore all fields of the header while decoding. This was of no consequence in the past, but now these “corrupt” blocks will fail the verification checksums. To remedy this there is a new flag used when opening files called `OPEN_NO_CHECKSUM` that causes the checksum verification to be bypassed. For existing streaming applications, the legacy function `WavpackOpenFileInputEx()` will *automatically* set the `OPEN_NO_CHECKSUM` flag if the `OPEN_STREAMING` flag is set, and thereby circumvent this problem. But if a streaming application is updated to use the new `WavpackOpenFileInputEx64()` then this should be considered.