WavPack 5 Library Documentation

David Bryant
February 15, 2024

1.0 Introduction

This document describes the use of the WavPack library (libwavpack) from a programmer's viewpoint. The library is designed to make reading and writing WavPack files as easy as possible, however there are some subtleties involved (especially when *creating* WavPack files) that should be understood to get the most out of WavPack and to make sure the resulting files are standardized. It is assumed that this document will be used with the `wavpack.h` header file (which obviously takes precedence in the case of discrepancies).

Functionality that was introduced with version 5.0.0 will be noted, so this document can be used with older library versions as well. There is also a document called *WavPack 5 Porting Guide For Developers* that is specifically targeted at developers moving from a previous version of the library.

There is also functionality in the library to create, read and edit metadata tags that are often appended to WavPack files, and there is functionality to decode stand-alone WavPack blocks for streaming applications. The WavPack format was adopted by WinZip Computing to be used for compressing WAV files and important details of this are discussed at the end of the document for programmers integrating this library with Zip handling software.

To test decoders, there is also a test suite available that has many varieties of WavPack files to test the robustness of any decoder. This is constantly getting updated, but is as of this writing available here:

`http://www.rarewares.org/wavpack/test_suite.zip`

The WavPack library is written in C and has been ported to many platforms, and it should be possible to call it from many different programming languages. It is also written to contain no static storage and so can be used by any number of clients simultaneously.

Note that in this document the meaning of the word "sample" or "samples" refers to "samples per channel". So, a function like `WavpackUnpackSamples()` takes an integer parameter specifying the number of samples to unpack and this would mean the number of samples for one channel only (although all channels in the file would be unpacked and interleaved). In other words, the number of actual data values generated by the call would be the specified number of samples *times* the number of channels. On the other hand, in a function like `WavpackGetBitsPerSample()`, the value would be the number of bits in a single channel's sample (or 16, in the common case of CD audio).

DSD audio was introduced in WavPack 5, and this further complicated the definition of “sample”. In WavPack's DSD mode, a “sample” is 8 consecutive DSD bits stored in a byte, with the MSB first temporally. This is the minimum granularity of seeking and decoding in WavPack DSD files, and actually reflects how DSD audio is stored in the WavPack file.

For version 5.7.0, multithreading was added to libwavpack utilizing Windows native threads or POSIX pthreads depending on the platform. See section 8.0 at the end of this document for details and limitations of this feature.

2.0 General Purpose Functions

```
uint32_t WavpackGetLibraryVersion (void);
```

Return the WavPack library version in a packed integer. Bits 0-7 give the micro version, bits 8-15 give the minor version, and bits 16-23 give the major version. As of this writing the version is 5.0.0.

```
const char *WavpackGetLibraryVersionString (void);
```

Return the WavPack library version as a string. As of this writing this is "5.0.0".

```
char *WavpackGetErrorMessage (WavpackContext *wpc);
```

This function returns a pointer to a string describing the last error generated by WavPack. This may provide useful information about why something is not working the way it should.

```
void WavpackLittleEndianToNative (void *data, char *format);
void WavpackNativeToLittleEndian (void *data, char *format);
```

These are two helper functions that normally would not be needed by an application. However, if an application wanted to manually parse a WavPack file, a RIFF header, or a APEv2 tag, then these could come in handy. They transform structures back and forth between native-endian and little-endian (which is the format of all these particular structures) based on a format string that defines the layout of the structure. In the string, an 'S' indicates a 2-byte value, an 'L' indicates a 4-byte value, a 'D' indicates an 8-byte value, and a digit ('1' to '9') indicates that that many bytes should be skipped unchanged. In `wavpack.h` there are format strings for several structures that might be required. Of course, on little-endian computers they will do nothing (but can still be called). **Note that on machines that have field alignment requirements, it is important that the structures are properly aligned!**

3.0 Reading WavPack Files

The basic procedure for reading a WavPack file is this:

1. open file with `WavpackOpenFileInput()` or `WavpackOpenFileInputEx64()`
2. determine important characteristics for decoding using these (and other) functions:
   - `WavpackGetNumSamples()`
   - `WavpackGetBitsPerSample()`
   - `WavpackGetBytesPerSample()`
   - `WavpackGetSampleRate()`
3. read decoded samples with `WavpackUnpackSamples()`
4. optionally seek with `WavpackSeekSample()`
5. close file with `WavpackCloseFile()`

For opening existing WavPack files for decoding, the easiest function to use is:

```
WavpackContext *WavpackOpenFileInput (
    const char *infilename,
    char *error,
    int flags,
    int norm_offset
);
```

This function accepts the path/filename of a WavPack file and opens it for reading, returning a pointer to a `WavpackContext` on success. The filename can be simply the string '-' to specify `stdin` as the source. The returned pointer can essentially be treated as a handle by the calling function (it is actually typed as a void pointer in `wavpack.h`) and is used in all other calls to WavPack to refer to this file. If the function fails for some reason (such as the WavPack file is not found or is invalid) then `NULL` is returned and the string pointed to by "error" is set to a message describing the problem. Note that the string space is allocated by the caller and must be at least 80 chars. The "flags" parameter is a bitmask that provides the following options:

- `OPEN_WVC`: Attempt to open and read a corresponding "correction" file along with the standard WavPack file. No error is generated if this fails (although it is possible to find out which decoding mode is actually being used). **Note that if this flag is *not* set then lossy decoding will occur even when a correction file is available, therefore this flag *should normally be set!***
- `OPEN_TAGS`: Attempt to read any ID3v1 or APEv2 tags appended to the end of the file. This obviously requires a seekable file to succeed because it takes place before decoding audio.
- `OPEN_WRAPPER`: Normally all the information required to decode the file will be available from native WavPack information. However, if the purpose is to restore the actual source file verbatim (or the file header is needed for some other reason) then this flag should be set. After opening the file, `WavpackGetWrapperData()` can be used to obtain the actual file header (which the caller must parse if desired). Note that some WavPack files might not contain headers, and some may contain headers for other formats besides WAV (which will only be presented if the `OPEN_ALT_TYPES` flags is specified).

- `OPEN_2CH_MAX`: This allows multichannel WavPack files to be opened with only one stream, which usually incorporates the front left and front right channels. This is provided to allow decoders that can only handle 2 channels to at least provide "something" when playing multichannel. It would be nice if this could downmix the multichannel audio to stereo instead of just using two channels, but that exercise is left for the student. :)
- `OPEN_NORMALIZE`: Most floating point audio data is normalized to the range of +/-1.0 (especially the floating point data in Microsoft WAV files) and this is what WavPack normally stores. However, WavPack is a lossless compressor, which means that is should (and does) work with floating point data that is normalized to some other range (or even with crazy values all over the place). However, if an application simply wants to play the audio, then it probably wants the data normalized to the same range regardless of the source. This flag is provided to accomplish that, and when set simply tells the decoder to provide floating point data normalized to +/-1.0 even if the source had some other range. The "norm_offset" parameter can be used to select a different range if that is desired (see below).

  Keep in mind that floating point audio (unlike integer audio) is not required to stay within its normalized limits. In fact, it can be argued that this is one of the advantages of floating point audio (i.e. no danger of clipping)! However, when this is decoded for playback (which, of course, must eventually involve a conversion back to the integer domain) it is important to consider this possibility and (at a minimum) perform hard clipping.
- `OPEN_STREAMING`: This is essentially a "raw" or "blind" mode where the library will simply decode any blocks fed it through the reader callback (or file), regardless of where those blocks came from in a stream. The only requirement is that complete WavPack blocks are fed to the decoder (and this will require multiple blocks in multichannel mode) and that complete blocks are decoded (even if all samples are not actually required). All the blocks must contain the same number of channels and bit resolution, and the correction data must be either present or not. All other parameters may change from block to block (like lossy/lossless). Obviously, in this mode any seeking must be performed by the application (and again, decoding must start at the beginning of the block containing the seek sample).

  Starting with WavPack 5, all blocks include a checksum to improve performance with corrupted streams. However, this can cause trouble with streaming audio because the blocks may not be reassembled verbatim, so in this case the `OPEN_NO_CHECKSUM` flag should also be set to skip this verification. When the compatibility function `WavpackOpenFileInputEx()` is used, the `OPEN_NO_CHECKSUM` flag is *automatically* set if `OPEN_STREAMING` is set. This is done to retain compatibility with existing applications.
- `OPEN_EDIT_TAGS`: Open the file in read/write mode to allow editing of any APEv2 tags present, or appending a new APEv2 tag. Of course the file must have write permission and be seekable.
- `OPEN_FILE_UTF8`: On Windows platforms only, assume that the passed filename is UTF-8 encoded rather than local ANSI, allowing the WavPack library to be Unicode compliant. On non-Windows platforms this flag doesn't do anything; the filename is always passed directly to fopen(). Introduced with 4.80.0.
- `OPEN_DSD_NATIVE`: Open DSD files as bitstreams so that decoded audio is returned as 8-bit “samples” (which actually contain 8 discrete DSD samples, MSB first temporally) in 32-bit integers (like all WavPack audio data). If neither this flag nor the `OPEN_DSD_AS_PCM` flag are set, then DSD audio files will not be readable. Introduced with 5.0.0.

- `OPEN_DSD_AS_PCM`: Open DSD files as 24-bit PCM where DSD audio is decimated 8x. Note that this will still be at a very high sampling rate (352.8 kHz for DSD64) and contain lots of quantization noise, so they should be further downsampled before use. Introduced with 5.0.0.
- `OPEN_ALT_TYPES`: Indicates to the library that the application is aware of alternate file types, their extensions and the "qualify modes" returned by `WavpackGetQualifyMode()`. If this flag is *not* set then file wrappers for alternate file types and MD5 sums generated using "qualify modes" will *not* be returned to the application (because, presumably, it wouldn't handle them correctly). For simply accessing the audio of files this flag should not be required. Introduced with 5.0.0.
- `OPEN_NO_CHECKSUM`: Don't verify block checksums before decoding. This is useful in streaming applications where the blocks may have been regenerated without all the fields correct (Matroska). When using the compatibility function `WavpackOpenFileInputEx()` this flag is forced on when `OPEN_STREAMING` is set. Introduced with 5.0.0.
- `OPEN_THREADS_MASK & OPEN_THREADS_SHFT`: This 4-bit field controls how many additional worker threads (from 0 to 15) should be created by the library to speed up the decoding operation. In files with multiple streams (i.e., more than 2 channels) the streams will be decoded in parallel and nothing else is required of the client (although longer buffers will reduce thread context switching overhead). For single-stream files (i.e., mono and stereo) the client should request long buffers at each call to `WavpackUnpackSamples()` to allow temporally consecutive frames to be decoded in parallel (WavPack frames are generally 10,000 to 24,000 mono or stereo samples). This feature was officially added to libwavpack in version 5.7.0, and earlier versions will simply ignore these bits. See section 8.0 for more details and limitations.

The "norm_offset" parameter is used with the `OPEN_NORMALIZE` flag and floating point audio data to specify an alternate normalization range. The default is 0 and results in a standard range of +/-1.0; positive values increase the range and negative values decrease the range (by factors of two). For example, a value here of 15 will generate a range of +/-32768.0 (assuming no clipping samples).

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

This function is identical to `WavpackOpenFileInput()` except that instead of providing a filename to open, the caller provides a pointer to a set of reader callbacks and instances of up to two streams. The first of these streams is required and contains the regular WavPack data stream; the second contains the
"correction" file if desired. Unlike the standard open function which handles the correction file transparently, in this case it is the responsibility of the caller to be aware of correction files.

The advantage of this method is that the data doesn't necessarily need to be contained in a file. For

example, the data might be streamed from somewhere or it may already be in memory because it is being parsed by another program.

The prototype for the `WavpackStreamReader64` is in `wavpack.h` and an example implementation is provided in `open_filename.c` using standard stream I/O. **Pay close attention to the return values of the seek functions `set_pos_xxx()` because this has caused headaches on more than one occasion (0 means success)!** This 64-bit reader callback was introduced with WavPack 5.0.0 as was this function.

Note that the reader callbacks include a function to close the file. 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.

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

This function is identical to `WavpackOpenFileInputEx64()` except that it uses an older version of the reader callbacks that did not provide for 64-bit values and was therefore limited to 2 GB files. This function provides a translation layer between the old and new reader callbacks so that existing applications can continue to work, but new applications should use the new reader if possible. To maintain backward compatibility, this function also sets the `OPEN_NO_CHECKSUM` flag when the `OPEN_STREAMING` flag is set to that modified blocks are not discarded.

Once the WavPack file has been opened, the application will probably want to get some information about the file (like bitdepth, sampling rate, etc). This is accomplished with a series of several functions. The most basic is:

```
int WavpackGetMode (WavpackContext *wpc);
```

This returns a bitmask with the following values:

- `MODE_WVC`: A .wvc file has been found and will be used for lossless decoding.
- `MODE_LOSSLESS`: The file decoding is lossless (either pure or hybrid).
- `MODE_HYBRID`: The file is in hybrid mode (may be either lossy or lossless).
- `MODE_FLOAT`: The audio data is 32-bit ieee floating point.
- `MODE_VALID_TAG`: The file contains a valid ID3v1 or APEv2 tag (`OPEN_TAGS` must be set above to get this status).
- `MODE_HIGH`: The file was originally created in "high" mode (this is really only useful for

reporting to the user)

- MODE_FAST: The file was originally created in "fast" mode (this is really only useful for reporting to the user)
- MODE_EXTRA: The file was originally created with the "extra" mode (this is really only useful for reporting to the user). The MODE_XMODE below can sometimes allow determination of the exact extra mode level.
- MODE_XMODE: If the MODE_EXTRA bit above is set, this 3-bit field can sometimes allow the determination of the exact extra mode parameter specified by the user if the file was encoded with version 4.50 or later. If these three bits are zero then the extra mode level is unknown, otherwise is represents the extra mode level from 1-6.
- MODE_APETAG: The file contains a valid APEv2 tag (OPEN_TAGS must be set in the "open" call for this to be true). Note that only APEv2 tags can be edited by the library. If a file that has an ID3v1 tag needs to be edited then it must either be done with another library or it must be converted (field by field) into a APEv2 tag (see the wvgain.c program for an example of this).
- MODE_SFX: The file was created as a "self-extracting" executable (this is really only useful for reporting to the user). The creation of self-extracting files is no longer supported.
- MODE_VERY_HIGH: The file was created in the "very high" mode (or in the "high" mode prior to 4.40).
- MODE_MD5: The file contains an MD5 checksum.
- MODE_DNS: The hybrid file was encoded with the dynamic noise shaping feature which was introduced in the 4.50 version of WavPack.

```
int WavpackGetNumChannels (WavpackContext *wpc);
```

Returns the number of channels of the specified WavPack file. Note that this is the actual number of channels contained in the file even if the OPEN_2CH_MAX flag was specified when the file was opened.

```
int WavpackGetReducedChannels (WavpackContext *wpc);
```

If the OPEN_2CH_MAX flag is specified when opening the file, this function will return the actual number of channels decoded from the file (which may or may not be less than the actual number of channels, but will always be 1 or 2). Normally, this will be the front left and right channels of a multichannel file.

```
int WavpackGetChannelMask (WavpackContext *wpc);
```

Returns the standard Microsoft channel mask for the specified WavPack file. If present, these channels must come before any channels not defined by this mask, and be in the standard Microsoft order.

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

This function allows retrieving the Core Audio File channel layout, many of which do not conform to the Microsoft ordering standard that WavPack requires internally (at least for those channels present in the "channel mask"). In addition to the layout tag, this function returns the reordering string (if stored in the file) to allow the unpacker to reorder the channels back to the specified layout (if it wants to restore the CAF order). The number of channels in the layout is determined from the lower nybble of the layout word (and should probably match the number of channels in the file), and if a reorder string is requested then that much space must be allocated. Note that all the reordering is actually done outside of this library, and that if reordering is required then the appropriate bit from `WavpackGetQualifyMode()` will be set. Introduced in 5.0.0.

Note: Normally this function would *not* be used by an application unless it specifically wanted to restore a non-standard channel order (to check an MD5, for example) or obtain the Core Audio channel layout ID. For simple file decoding for playback, the channel_mask should provide all the information required unless there are non-Microsoft channels involved, in which case the following function, `WavpackGetChannelIdentities()`, will provide the identities of the other channels (if they are known).

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

This function provides the identities of **all** the channels in the file, including the standard Microsoft channels (which come first, in order, and are numbered 1-18) and also any non-Microsoft channels (which can be in any order and have values from 33-254). The value `0x00` is invalid and `0xFF` indicates an "unknown" or "unnassigned" channel. The string is `NULL` terminated so the caller must supply enough space for the number of channels indicated by `WavpackGetNumChannels()`, **plus one**. The current channel assignment values are listed in `pack_utils.c`. Introduced in 5.0.0.

Note that this function returns the actual order of the channels in the Wavpack file (i.e., the order returned by WavpackUnpackSamples()). If the file includes a "reordering" string because the source file was not in Microsoft order, that is **not** taken into account here and really only needs to be considered if doing an MD5 verification or if it's required to restore the original order/file (like wvunpack does).

```
uint32_t WavpackGetSampleRate (WavpackContext *wpc);
```

Returns the sample rate of the specified WavPack file in samples per second. For DSD audio, this represents the number of 8-bit “samples” per second, or 8 times the actual sample rate. Use the function `WavpackGetNativeSampleRate()` to obtain the true DSD sample rate to report to the user.

```
uint32_t WavpackGetNativeSampleRate (WavpackContext *wpc);
```

Returns the sample rate of the specified WavPack file in samples per second. For DSD audio, this returns the actual DSD sample rate that should be reported to the user (e.g., 2822400 for DSD64). Keep in mind that with respect to all other functions, a DSD “sample” is a byte that consist of 8 consecutive DSD bits, and the actual “DSD-byte” sample rate is returned by `WavpackGetSampleRate()`. This

function was introduced in 5.0.0.

```
int WavpackGetBitsPerSample (WavpackContext *wpc);
```

Returns the actual number of valid bits per sample contained in the original file, which may or may not be a multiple of 8. Floating data always has 32 bits, integers may be from 8 to 32 bits each. When this value is not a multiple of 8, then the "extra" zeroed bits are located in the LSBs of the result. So, values are left justified when placed into the number of bytes used by the original data, but these finished bytes are right-justified into each 4-byte buffer entry. For WavPack DSD audio files, this value will be 8 for files opened with `OPEN_DSD_NATIVE` or 24 for files opened with `OPEN_DSD_AS_PCM`.

```
int WavpackGetBytesPerSample (WavpackContext *wpc);
```

Returns the number of bytes used for each sample (1 to 4) in the original file. This is required information for the user of this module because the audio data is returned in the **lower** bytes of the 4-byte buffer and must be left-shifted 8, 16, or 24 bits if normalized 4-byte values are required. This value must be at least enough to store all the bits per sample, and of course the most natural and common case is when there is an exact fit. For WavPack DSD audio files, this value will be 1 for files opened with `OPEN_DSD_NATIVE` or 3 for files opened with `OPEN_DSD_AS_PCM`.

```
int WavpackGetVersion (WavpackContext *wpc);
```

This function returns the major version number of the WavPack program (or library) that created the open file. Currently, this can be from 1 to 5 if the library is built with legacy support, otherwise it's just 4 or 5. Minor and micro versions are not recorded directly in WavPack files.

```
unsigned char WavpackGetFileFormat (WavpackContext *wpc);
```

Return the file format specified in the call to `WavpackSetFileInformation()` when the file was created. For all files created prior to WavPack 5 this will be zero (`WP_FORMAT_WAV`). Introduced in WavPack 5.0.0. The following formats are currently defined in `wavpack.h`:

- `WP_FORMAT_WAV`: Microsoft Waveform Audio Format, including BWF and RF64
- `WP_FORMAT_W64`: Sony Wave64
- `WP_FORMAT_CAF`: Apple Core Audio
- `WP_FORMAT_DFF`: Philips DSDIFF
- `WP_FORMAT_DSF`: Sony DSD format
- `WP_FORMAT_AIF`: Apple AIFF format

```
char *WavpackGetFileExtension (WavpackContext *wpc);
```

Return a string representing the recommended file extension for the open WavPack file. For all files created prior to WavPack 5 this will be "wav", even for raw files with no RIFF into. This string is specified in the call to `WavpackSetFileInformation()` when the file was created. Introduced in WavPack 5.0.0.

```
int WavpackGetQualifyMode (WavpackContext *wpc);
```

This function obtains information about specific file features that were added for version 5, specifically qualifications added to support CAF and DSD files. These bits are meant to simply indicate the format of the data in the original source file and do **not** indicate how the library will return the data to the application (which is always the same). This means that in general an application that simply wants to play or process the audio data need not be concerned about these. Introduced with WavPack 5.0.0.

If the file is DSD audio, then either of the `QMODE_DSD_LSB_FIRST` or `QMODE_DSD_MSB_FIRST` bits will be set (but native DSD audio is always returned to the caller MSB first). Checking for one of these two bits is the proper way to check for DSD audio.

- `QMODE_BIG_ENDIAN`: big-endian data format (opposite of WAV format)
- `QMODE_SIGNED_BYTES`: 8-bit audio data is signed (opposite of WAV format)
- `QMODE_UNSIGNED_WORDS`: audio data (> 8-bit) is unsigned (opposite of WAV format)
- `QMODE_REORDERED_CHANS`: source channels were reordered
- `QMODE_DSD_LSB_FIRST`: DSD bytes, LSB first (most Sony .dsf files)
- `QMODE_DSD_MSB_FIRST`: DSD bytes, MSB first (Philips .dff files)
- `QMODE_DSD_IN_BLOCKS`: DSD data is blocked by channels (Sony .dsf only)

```
uint32_t WavpackGetNumSamples (WavpackContext *wpc);
```

Get total number of samples contained in the WavPack file, or -1 if unknown.

```
int64_t WavpackGetNumSamples64 (WavpackContext *wpc);
```

Get total number of samples contained in the WavPack file, or -1 if unknown. Introduced in 5.0.0 to handle files with over 2^32 samples.

```
uint32_t WavpackGetFileSize (WavpackContext *wpc);
```

Return the total size of the WavPack file(s) in bytes.

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

Return the total size of the WavPack file(s) in bytes. Introduced in 5.0.0 to handle files over 4 GB.

```
double WavpackGetRatio (WavpackContext *wpc);
```

Calculate the ratio of the specified WavPack file size to the size of the original audio data as a double greater than 0.0 and (usually) smaller than 1.0. A value greater than 1.0 represents "negative" compression and a return value of 0.0 indicates that the ratio cannot be determined.

```
double WavpackGetAverageBitrate (
    WavpackContext *wpc,
    int count_wvc
);
```

Calculate the average bitrate of the WavPack file in bits per second. A return of 0.0 indicates that the bitrate cannot be determined. An option is provided to use (or not use) any attendant .wvc file.

```
int WavpackGetFloatNormExp (WavpackContext *wpc);
```

Return the normalization value for floating point data (valid only if floating point data is present). A value of 127 indicates that the floating point range is +/- 1.0. Higher values indicate a larger floating point range. Note that if the OPEN_NORMALIZE flag is set when the WavPack file is opened, then floating data will be returned normalized to +/-1.0 regardless of this value (and can also be offset from that by using the "norm_offset" field of the "open" call).

```
int WavpackGetMD5Sum (WavpackContext *wpc, uchar data [16]);
```

Get any MD5 checksum stored in the metadata (should be called after reading last sample or an extra seek will occur). A return value of `FALSE` indicates that no MD5 checksum was stored.

```
uint32_t WavpackGetWrapperBytes (WavpackContext *wpc);
uchar *WavpackGetWrapperData (WavpackContext *wpc);
void WavpackFreeWrapper (WavpackContext *wpc);
```

These three routines are used to access (and free) header and trailer data that was retrieved from the WavPack file. The header will be available before the samples are decoded. The trailer will be available after all samples have been read and an attempt is made to read at least one sample past the end of the available samples (it is also possible to use the `WavpackSeekTrailingWrapper()` function described below). Note that the `OPEN_WRAPPER` flag must be set in the "open" call for this information to be available. Most applications will not need this data because everything required to decode and play a WavPack file can be determined without this information, and some valid WavPack files might not even actually have this wrapper.

The wrapper information is simply a verbatim copy of everything that was in the original source file right up to the start of the audio data (and anything after the audio data in the case of the "trailer"). In RIFF terms, this means that the data chunk will be last and only include the "data" ID and the size (as it was in the WAV file). Other file formats have different headers. The code was implemented like this so that the wvunpack program could simply copy this information directly to the output file (before and after the audio data) to create verbatim copies of the source file (like an archiver).

The WavPack library accumulates this information when parsing the WavPack file and only purges this when `WavpackFreeWrapper()` is called. Therefore, if the application unpacks an entire WavPack file without freeing anything, then the entire wrapper data (both header and trailer) will be returned.

```
void WavpackSeekTrailingWrapper (WavpackContext *wpc);
```

Normally the trailing wrapper will not be available when a WavPack file is first opened for reading because it is stored in the final block of the file. This function forces a seek to the end of the file to pick up any trailing wrapper stored there (then use `WavPackGetWrapper**()` to obtain). This can obviously only be used for seekable files (not pipes) and is not available for pre-4.0 WavPack files.

These are the only three functions directly involved in decoding WavPack audio data:

```
uint32_t WavpackUnpackSamples (
    WavpackContext *wpc,
    int32_t *buffer,
    uint32_t samples
);
```

Unpack the specified number of samples from the current file position. Note that "samples" here refers to "complete" samples, which would be 2 integers for stereo files or even more for multichannel files, so the required memory at "buffer" is (4 * samples * num_channels) bytes. The audio data is returned right-justified in 32-bit integers in the endian mode native to the executing processor. So, if the original data was 16-bit in 2-bytes, then the values returned would be +/-32k. Floating point data can also be returned if the source was floating point data (and this can be optionally normalized to +/-1.0 by using the appropriate flag in the call to `WavpackOpenFileInput()`). The actual number of samples unpacked is returned, which should be equal to the number requested unless the end of file is encountered or an error occurs. If all samples have been unpacked then 0 will be returned.

Note that if the WavPack file contains floating-point data (as indicated by the MODE_FLOAT bit being set in the value returned from `WavpackGetMode()`) then 32-bit float values are returned in the buffer despite the defined type of the pointer. If integers are desired (e.g. for writing to a DAC) then this conversion must be performed by the caller, and it is important to keep in mind that clipping is probably required. Assuming that OPEN_NORMALIZE is used to ensure that the normalized range is +/- 1.0, then this C code sample will perform the conversion to 16-bit:

```
    int32_t *lptr = buffer;

    while (num_samples--) {
        float fdata = * (float*) lptr;

        if (fdata >= 1.0)
            *lptr++ = 32767;
        else if (fdata <= -1.0)
            *lptr++ = -32768;
        else
            *lptr++ = floor (fdata * 32768.0);
    }
```

For highest quality when converting to 16-bit it would be advisable to also perform dithering and/or noise shaping, but that is beyond the scope of this document. For converting to 24-bit this would probably not be required.

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

Seek to the specified sample index, returning `TRUE` on success. Note that files generated with version 4.0 or newer will seek almost immediately. Older files can take quite long if required to seek through

unplayed portions of the file, but will create a seek map so that reverse seeks (or forward seeks to already scanned areas) will be very fast. After a FALSE return the file should not be accessed again (other than to close it); this is a fatal error. Introduced in 5.0.0.

```
int WavpackSeekSample (WavpackContext *wpc, uint32_t sample);
```

Legacy function for seeking that handles files with less that 2^32 samples, provided for compatibility with existing applications.

Finally, there are several functions which provide extra information during decoding:

```
int64_t WavpackGetSampleIndex64 (WavpackContext *wpc);
```

Get the current sample index position, or -1 if unknown. Introduced with 5.0.0.

```
uint32_t WavpackGetSampleIndex (WavpackContext *wpc);
```

Legacy function for obtaining sample index in files with less than 2^32 samples, provided for compatibility with existing application.

```
double WavpackGetInstantBitrate (WavpackContext *wpc);
```

Calculate the bitrate of the current WavPack file block in bits per second. This can be used for an "instant" bit display and gets updated from about 1 to 4 times per second. A return of 0.0 indicates that the bitrate cannot be determined.

```
int WavpackGetNumErrors (WavpackContext *wpc);
```

Get the number of errors encountered so far. These are probably CRC errors, but could also be missing blocks.

```
int WavpackLossyBlocks (WavpackContext *wpc);
```

Return TRUE if any uncorrected lossy blocks were actually written or read. This can be used to determine if the lossy bitrate specified was so high that the compression was nevertheless lossless.

```
double WavpackGetProgress (WavpackContext *wpc);
```

Calculate the progress through the file as a double from 0.0 (for begin) to 1.0 (for done). A return value of -1.0 indicates that the progress is unknown.

Finally, this function is used when no more access to a file is needed:

```
WavpackContext *WavpackCloseFile (WavpackContext *wpc);
```

Close the specified WavPack file and release all resources used by it. Returns `NULL`. If the 64-bit stream reader is being used and a “close” function has been provided, it will be called for each file instance.

4.0 Writing WavPack Files

To use the library to create WavPack files from raw PCM audio, the user must provide a `WavpackBlockOutput` function that is used by the library to write finished WavPack blocks to the output. Unlike the read case, there is no facility to write directly to named files. Here is the function required:

```
typedef int (*WavpackBlockOutput)(
    void *id,
    void *data,
    int32_t bcount
);
```

where the "id" is used to differentiate the regular WavPack data "wv" from the correction data "wvc" (or for the case of multiple streams running at the same time). The return value is simply `TRUE` for success and `FALSE` for error. An example of this function can be found in `wavpack.c` called `write_block()`.

Note that the function should keep track of the first block written, which is simply the first call to this function, because that block might need to be rewritten when packing is done to update the length fields. The library takes care of updating the fields with `WavpackUpdateNumSamples()`, but the application is responsible for rereading the first block and writing it back.

The basic procedure for creating WavPack files is this (with mandatory steps in **bold**):

1. **get a context and set block output function with `WavpackOpenFileOutput()`**
2. optionally specify the file type and extension with `WavpackSetFileInformation()`
3. optionally write a file header with `WavpackAddWrapper()`
4. **set the data format and specify modes with `WavpackSetConfiguration64()`**
5. optionally call `WavpackSetChannelLayout()` to specify Core Audio layouts
6. **allocate buffers and prepare for packing with `WavpackPackInit()`**
7. **actually compress audio and write blocks with `WavpackPackSamples()`**
8. **flush final samples into blocks with `WavpackFlushSamples()`**
9. optionally write MD5 sum with `WavpackStoreMD5Sum()`
10. optionally write file trailer with `WavpackAddWrapper()`
11. if MD5 sum or file trailer written, call `WavpackFlushSamples()` again
12. optionally append metadata tag with functions in next section
13. optionally update number of samples with `WavpackUpdateNumSamples()`
14. **close the context with `WavpackCloseFile()`**

Note that this does not show opening and closing the output files which is done by the application itself. What follows is a description of the functions involved.

```
WavpackContext *WavpackOpenFileOutput (
    WavpackBlockOutput blockout,
    void *wv_id,
    void *wvc_id
);
```

Open context for writing WavPack files. The returned context pointer is used in all following calls to the library. The "blockout" function will be used to store the actual completed WavPack blocks and will be called with the id pointers containing user defined data (one for the wv file and one for the wvc file). A return value of `NULL` indicates that memory could not be allocated for the context.

```
void WavpackSetFileInformation (
    WavpackContext *wpc,
    char *file_extension,
    unsigned char file_format
);
```

This function allows the application to store a file extension and a file format identification. The extension would be used by the unpacker if the user had not specified the target filename, and specifically handles the case where the original file had the "wrong" extension for the file format (e.g., a Wave64 file having a WAV extension) or an alternative (e.g., AMB or BWF) or where the file format is not known. Specifying a file format besides the default `WP_FORMAT_WAV` will ensure that old decoders will not be able to see the non-WAV wrapper provided with `WavpackAddWrapper()` (which they would not understand or end up putting on a file with a .wav extension). For a list of the currently defined file formats, see `wavpack.h` or `WavpackGetFileFormat()`. Introduced in 5.0.0.

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

Set configuration for writing WavPack files. This must be done before sending any actual samples, however it is okay to send wrapper or other metadata before calling this. It's a good idea to clear the `WavpackConfig` structure before setting it to avoid enabling unintentional settings. The "config" structure contains the following **required** information:

- `config->bytes_per_sample`: see `WavpackGetBytesPerSample()` for info
- `config->bits_per_sample`: see `WavpackGetBitsPerSample()` for info
- `config->channel_mask`: Microsoft standard (mono = 4, stereo = 3)
- `config->num_channels`: self evident
- `config->sample_rate`: self evident

Be particularly careful with the "channel_mask" field. If this is not set to the correct value (3 or 4 for stereo or mono) then everything will still appear to work correctly, but the resulting WavPack file

will have *undefined* channel assignments, which could cause trouble with some decoder or players, and will not compress as well.

Specifying these 5 parameters alone would create a default lossless WavPack file, identical to the one produced by using the command-line program without options. For optional configuration, the following fields and flags may be set:

- `config->flags`:
  - `CONFIG_HYBRID_FLAG`: select hybrid mode (must set bitrate)
  - `CONFIG_JOINT_STEREO`: select joint stereo (must set override also)
  - `CONFIG_JOINT_OVERRIDE`: override default joint stereo selection
  - `CONFIG_HYBRID_SHAPE`: select hybrid noise shaping (set override & shaping_weight != 0.0)
  - `CONFIG_SHAPE_OVERRIDE`: override default hybrid noise shaping (set `CONFIG_HYBRID_SHAPE` and `shaping_weight`)
  - `CONFIG_DYNAMIC_SHAPING`: force dynamic noise shaping even when WavPack would not use it (no need to set any of the other shaping flags when using this one)
  - `CONFIG_FAST_FLAG`: "fast" compression mode (same as -f)
  - `CONFIG_HIGH_FLAG`: "high" compression mode (same as -h)
  - `CONFIG_VERY_HIGH_FLAG`: "very high" compression mode (same as -hh)
  - `CONFIG_BITRATE_KBPS`: hybrid bitrate is kbps, not bits / sample
  - `CONFIG_CREATE_WVC`: create correction file
  - `CONFIG_OPTIMIZE_WVC`: maximize hybrid compression (same as -cc)
  - `CONFIG_CALC_NOISE`: calc noise in hybrid mode
  - `CONFIG_EXTRA_MODE`: extra processing mode (same as -x)
  - `CONFIG_SKIP_WVX`: no wvx stream for floats & large ints (same as -p)
  - `CONFIG_MD5_CHECKSUM`: specify if you plan to store MD5 signature (the sum is calculated by the application, NOT by the library)
  - `CONFIG_CREATE_EXE`: no longer used (ignored)
  - `CONFIG_OPTIMIZE_MONO`: “on” by default from 5.0.0 (flag ignored)
  - `CONFIG_COMPATIBLE_WRITE:` write streams compatible back to WavPack 4.0 and WinZip decoder (disables “mono optimization”)
  - `CONFIG_CROSS_DECORR`: force cross-channel decorrelation even in hybrid mode
- `config->bitrate`: hybrid bitrate in either bits/sample or kbps
- `config->shaping_weight`: hybrid noise shaping coefficient override
- `config->block_samples`: force samples per WavPack block (0 = use default, else 1-131072)
- `config->float_norm_exp`: select floating-point data (127 for +/-1.0)
- `config->xmode`: extra mode processing value override (1-6)
- `config->qmode`: `non-WAV` qualification modes (see `WavpackGetQualifyMode()` for details)
- `config->worker_threads`: specify number of additional worker threads to utilize for increased speed, from 0 (none) to 15 max (see section 8.0 for more details and limitations).

If the number of samples to be written is known then it should be passed here. If the duration is not known then pass -1. In the case that the size is not known (or the writing is terminated early) then it is suggested that the application retrieve the first block written and let the library update the total samples

indication. A function is provided to do this update and it should be done to the "correction" file also. If this cannot be done (because a pipe is being used, for instance) then a valid WavPack will still be created (assuming the initial duration was set to -1), but when applications want to access that file they will have to seek all the way to the end to determine the actual duration (the library takes care of this). Also, if a header has been included then it might need to be updated as well or the WavPack file will not be directly unpackable to a valid file (although it will still be usable by itself).

The “chan_ids” argument allows setting the identities of any channels that are **not** standard Microsoft channels and are therefore not represented in the channel mask. WavPack files require that all the Microsoft channels come first (and in Microsoft order) and these are followed by any other channels (which can be in any order).

The identities are provided in a `NULL`-terminated string (`0x00` is not an allowed channel ID). The Microsoft channels may be provided as well (and will be checked) but it is really only necessary to provide the "unknown" channels. Any truly unknown channels are indicated with a `0xFF`. The current channel assignment values are listed in `pack_utils.c`. Simply pass a `NULL` pointer if there are no extra channels to define (the most common case by far).

This function was introduced with WavPack 5.0.0, and results in a newer version of the stream being generated that is incompatible with some very old decoders (before 4.3). This new stream allows for more efficient encoding of stereo streams that are actually mono, and in previous versions of the library was generated when the `CONFIG_OPTIMIZE_MONO` flag was set. Now that flag is ignored and the newer stream is generated by default. The older stream can still be generated if the compatibility function `WavpackSetConfiguration()` is used instead, or the `CONFIG_COMPATIBLE_WRITE` flag is specified.

A return of `FALSE` indicates an error (use `WavpackGetErrorMessage()` to find out what happened).

```
int WavpackSetConfiguration (
    WavpackContext *wpc,
    WavpackConfig *config,
    uint32_t total_samples
);
```

This legacy function provides the same capability as `WavpackSetConfiguration64()` above except the total samples must be < 2^32, no extra non-Microsoft channels can be defined, and the stream generated will be compatible with all decoders back to 4.0. It it provided for compatibility with existing applications and legacy decoders (e.g., WinZip).

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

This function allows setting the Core Audio File channel layout, many of which do not conform to the

Microsoft ordering standard that Wavpack requires internally (at least for those channels present in the "channel mask"). In addition to the layout tag, this function allows a reordering string to be stored in the file to allow the unpacker to reorder the channels back to the specified layout (if it is aware of this feature and wants to restore the CAF order). The number of channels in the layout is specified in the lower nybble of the layout word, and if a reorder string is specified it must be that long.

The reorder string (if supplied) is first scanned for the lowest value, and that is used as the base. This is done so that the string can be binary zero based or can be a text string based on “1” (which is what the `caff.c` module does). Either way, the values (once normalized) represent the destination position when packing (reordering) and the source position when unpacking (restoring original order, or “unreordering”).

All the reordering is actually done outside of this library, and if reordering is done then the `QMODE_REORDERED_CHANS` bit in `qmode` bit must be set to ensure that any MD5 sum is stored with a new ID so that existing decoders don't try to verify it (and to let new decoders know that a reorder might be required).

Note: This function should only be used to encode Core Audio files in such a way that a verbatim archive can be created. Applications can just include the “chan_ids” parameter in the call to `WavpackSetConfiguration64()` if there are non-Microsoft channels to specify, or do nothing special if only Microsoft channels are present (the vast majority of cases).

```
int WavpackPackInit (WavpackContext *wpc);
```

Prepare to actually pack samples by determining the size of the WavPack blocks and allocating sample buffers and initializing each stream. Call after `WavpackSetConfiguration64()` and before `WavpackPackSamples()`. A return of `FALSE` indicates an error.

```
int WavpackPackSamples (
    WavpackContext *wpc,
    int32_t *sample_buffer,
    uint32_t sample_count
);
```

Pack the specified samples. Samples must be stored in 32-bit integers in the native endian format of the executing processor, and should be in the numerical range corresponding to config->bytes_per_sample, i.e. any extra high-order bytes in the input data should be a pure sign-extension of the low (significant) bytes. Otherwise, since samples are sign-extended during encode, out-of-range values will be aliased to a valid value. The number of samples specified indicates composite samples (sometimes called "frames"). So, the actual number of data points would be this "sample_count" times the number of channels. Note that samples are accumulated here until enough exist to create a complete WavPack block (or several blocks for multichannel audio). If an application wants to break a block at a specific sample, it just calls `WavpackFlushSamples()` to force an early termination. Completed WavPack blocks are sent to the function provided in the initial call to `WavpackOpenFileOutput()`. A return of `FALSE` indicates an error (which most likely indicates the that user-supplied blockout function returned an error).

```
int WavpackFlushSamples (WavpackContext *wpc);
```

Flush all accumulated samples into WavPack blocks. This is normally called after all samples have been sent to `WavpackPackSamples()`, but can also be called to terminate a WavPack block at a specific sample (in other words it is possible to continue after this operation). This also must be called to dump non-audio blocks like those holding metadata for MD5 sums or file trailers. A return of `FALSE` indicates an error.

```
void WavpackUpdateNumSamples (
    WavpackContext *wpc,
    void*first_block
);
```

Given the pointer to the first block written (to either a .wv or .wvc file), update the block with the actual number of samples written. If the wav header was generated by the library, then it is updated also. This should be done if `WavpackSetConfiguration64()` was called with an incorrect number of samples (or -1). This should also be done in the case where the application did not provide the RIFF header but did add some chunks for a RIFF trailer (see the section on adding RIFF metadata). It is the responsibility of the application to read and rewrite the block. An example of this can be found in the Audition filter or in the command-line packer when the -i option is used. **On machines with alignment requirements, be sure that the passed pointer is properly aligned!**

```
int WavpackStoreMD5Sum (WavpackContext *wpc, uchar data [16]);
```

Store computed MD5 sum in WavPack metadata. Note that the user must compute the 16 byte sum; it is not done here. It is also required that `WavpackFlushSamples()` be called after this to make sure the block containing the MD5 sum is actually written. A return of `FALSE` indicates an error.

```
WavpackContext *WavpackCloseFile (WavpackContext *wpc);
```

Close the specified WavPack file and release all resources used by it. Returns NULL.

With respect to the file wrapper that is normally appended to the WavPack file, there are three options available for the application to handle this.

The first is the simplest, and that is for the application to simply do nothing. The library will automatically create a RIFF header appropriate for the audio data (including RF64 and/or WAVEFORMATEXTENSIBLE) and store this in the WavPack file. If the application does not know the actual number of samples beforehand and needs to reread the first block and have the library update it, then the library will handle updating its own RIFF header also (even converting it to an RF64). In this scenario, there will be no trailing RIFF data. For the vast majority of applications, this will be all that is required, and the resulting WavPack file will be unpacked as a valid WAV file.

However, some applications may want to store a custom RIFF header (and trailer) instead, or they may even want to attach headers for a different file format (like the Core Audio Format). An example of a program that does this is the WavPack command-line program itself because it wants to work as a file

archiver for several different formats. To accomplish this, functions are provided (see below) to add headers and trailers to a WavPack file being written. In this case, the application must provide a verbatim copy of **all** the header data from the beginning of the file right up to the start of the audio data. If any data is desired to be come after the audio data (a “trailer”), then this is appended after all samples have been packed and flushed. This wrapper data will be **exactly** what wvunpack will use to restore the original file. In no circumstance does the WavPack library parse, verify, or otherwise use this data.

If the header information needs to be updated after packing (because the total file size was not known before packing the audio) then the first block must be reread and the function `WavpackGetWrapperLocation()` must be called to find the wrapper in the block (it would be possible to find it by searching for it in the last block, but using this function is significantly less ugly and is guaranteed to work in the future). The WavPack library will not touch a wrapper that it did not create when calling `WavpackUpdateNumSamples()`.

The third option for including RIFF data chunks is a hybrid between these two methods and is useful if an application wants to add a few specific RIFF chunks and does not mind if the chunks appear at the *end* of the WAV file. In this method, which only applies to WAV files, the application does **not** send any RIFF header before packing the audio data which forces the library to create and store a standard RIFF header. However, when all samples have been packed, the application sends the RIFF chunks that it wants to add at the end of the file (RIFF trailer). Then, after flushing, it must reread the first block and call `WavpackUpdateNumSamples()` so that the library can update the RIFF header to reflect the added chunks (even if the number of samples was correct). Because the library created the header, it will update it, and will take into account the RIFF chunks added to the end.

All of this wrapper stuff can be a little confusing. It is a good idea to test that the final application is working correctly and creating WavPack files that will unpack to valid WAV files (even if the RIFF info is totally wrong, the files will still work perfectly well as WavPack files because, again, the RIFF wrapper is just informational). The first test is to use the -ss option on WvUnpack to make sure the RIFF wrapper is reported correctly. Then, unpack the WavPack file into a WAV using WvUnpack and repack it again with the standard WavPack command-line program (without options). If it doesn't complain about anything then there is a good chance that all the wrapper information is valid (although trailing data is **not** parsed, it is simply copied).

Here are the appropriate functions:

```
int WavpackAddWrapper (
    WavpackContext *wpc,
    void *data,
    uint32_t bcount
);
```

Add wrapper to store verbatim in WavPack blocks. This should be called before sending any audio samples in the case of the header or after all samples have been sent (and flushed) for any trailer. It is also required that `WavpackFlushSamples()` be called again after specifying a trailer to make sure it is actually written to the file.

If the exact contents of the header written above are not known because, for example, the file duration

is uncertain or trailing chunks are possible, simply write a "dummy" header of the correct length. When all data has been written it will be possible to read the first block written and update the header directly. An example of this can be found in the Audition filter. A return of FALSE indicates an error.

```
void *WavpackGetWrapperLocation (
    void *first_block,
    uint32_t *size
);
```

Given the pointer to the first block written to a WavPack file, this function returns the location of the stored header that was originally written with WavpackAddWrapper(). This would normally be used to update the wav header to indicate that a different number of samples was actually written or if additional chunks are written at the end of the file. The "size" parameter can be set to non-NULL to obtain the exact size of the header, and the function will return FALSE if the header is not found in the block's metadata (or it is not a valid WavPack block). Note that the size of the RIFF header cannot be changed and it is the responsibility of the application to read and rewrite the block. An example of this can be found in the Audition filter.

5.0 Tagging Functions

The WavPack library supports reading and writing metadata tags on WavPack files. This includes creating new APEv2 tags during WavPack file creation, reading text fields from both ID3v1 and APEv2 tags on existing WavPack files, and editing data in APEv2 tags.

Users should be aware of the following limitations of this functionality:

1. ID3v1 tags are read-only, and cannot be accessed if there is an APEv2 tag prior to them in the file (because the APEv2 tag takes priority). ID3v1 tags are lost if the prior APEv2 tag is edited.
2. The text items in APEv2 tags are UTF-8 encoded. The functionality of converting to/from any local or multi-byte encoding must be handled by the calling application.
3. The binary items of APEv2 tags can now (version 4.60+) be read and written. The functions that handle binary items have `Binary` in their name; unless noted otherwise all other functions refer to text tags only.

   The convention for binary tag items in APEv2 tags is that the data starts with a `NULL`-terminated string representing a filename. After the terminating `NULL`, the actual binary data starts. In the WavPack code this filename has only the extension of the actual file; the name portion is made up of the tag item name. **Note that this functionality is not handled in the library. The library only stores and retrieves a binary image and it is up to the calling application to append and handle this filename.**
4. When APEv2 tags are edited and the resulting tags are shorter than the original tags (or APEv2 tags are deleted altogether), the library uses the `truncate_here()` callback that was added to the `WavpackStreaReader64` structure to reduce the file size. If that callback is not defined (`NULL`), or the old `WavpackStreamReader` is used, the tag is instead padded with zeros at the front rather than having the file shortened, and this padding cannot be reclaimed by future editing. The net result of this is that repeated editing of tags will cause the file to grow indefinitely (although this will only happen when the tag is actually made smaller).

In these descriptions the meaning of the word "tag" refers to the whole bundle that is appended to the end of the WavPack file. This bundle may contain many individual items, each consisting of a key/value pair. The key is referred to here as the "item", meaning the item's name (like "artist"). Some people refer to the individual items as "tags", but that usage is not used here. Also note that APEv2 tags store the case of tag item names and values, but are not case sensitive when locating tag item names (and this is carried here into the lookup of ID3v1 tag item names).

```
int WavpackGetNumTagItems (WavpackContext *wpc);
int WavpackGetNumBinaryTagItems (WavpackContext *wpc);
```

Count and return the total number of tag items (either text or binary) in the specified file. This works with either ID3v1 tags or APEv2 tags (although ID3V1 tags do not have binary items).

```
int WavpackGetTagItem (
    WavpackContext *wpc,
    const char *item,
    char *value,
    int size
);

int WavpackGetBinaryTagItem (
    WavpackContext *wpc,
    const char *item,
    char *value,
    int size
);
```

Attempt to get the specified item (either text or binary) from the specified file's ID3v1 or APEv2 tag. The "size" parameter specifies the amount of space available at "value", if the desired text item will not fit in this space then ellipses (...) will be appended and the string terminated (binary tag data is simply truncated). The actual length of the string (or binary data) is returned (or 0 if no matching value found). Note that with APEv2 text tags the length might not be the same as the number of characters because UTF-8 encoding is used. Also, APEv2 text tags can have multiple (`NULL` separated) strings for a single value (this is why the length is returned). If this function is called with a `NULL` "value" pointer (or a zero "length") then only the actual length of the value data is returned (**not** counting the terminating `NULL` of text tag items). This can be used to determine the actual memory to be allocated beforehand.

For ID3v1 tags the only "item" names supported are "title", "artist", "album", "year", "comment" and "track" (which is converted to numeric text by the library).

```
int WavpackGetTagItemIndexed (
    WavpackContext *wpc,
    int index,
    char *item,
    int size
);

int WavpackGetBinaryTagItemIndexed (
    WavpackContext *wpc,
    int index,
    char *item,
    int size
);
```

These functions look up the tag item name by index and is used when the application wants to access all the items in the file's ID3v1 or APEv2 tag. Keep in mind that text and binary items are totally separate and that there is a different count and index for each. Note that this function accesses only the item's name; `WavpackGetTagItem()` still must be called to get the actual value. The "size" parameter specifies the amount of space available at "item", if the desired item will not fit in this space then ellipses (...) will be appended and the string terminated. The actual length of the string is returned

(or 0 if no item exists for index). If this function is called with a NULL "value" pointer (or a zero "length") then only the actual length of the item name is returned (**not** counting the terminating NULL). This can be used to determine the actual memory to be allocated beforehand.

```
int WavpackAppendTagItem (
    WavpackContext *wpc,
    const char *item,
    const char *value,
    int vsize
);

int WavpackAppendBinaryTagItem (
    WavpackContext *wpc,
    const char *item,
    const char *value,
    int vsize
);
```

This function is used to append (or replace) the specified field to the tag being created or edited. If no tag has been started, then an empty one will be allocated first. When finished adding all the items to the tag, use WavpackWriteTag() to write the completed tag to the file. Note that ID3 tags are not supported for writing. A size parameter is included so that text values containing multiple (NULL separated) strings (and binary data) can be written. A FALSE return indicates an error.

```
int WavpackDeleteTagItem (
    WavpackContext *wpc,
    const char *item
);
```

Delete the specified tag item from the APEv2 tag being created or edited. This function works with either text or binary items. Returns TRUE to indicate that an item was actually deleted from the tag.

```
int WavpackWriteTag (WavpackContext *wpc);
```

Once a APEv2 tag has been created (or edited) using WavpackAppendTagItem() (and WavpackDeleteTagItem()), this function is used to write the completed tag to the end of the WavPack file. Note that this is **not** done for **each** item in the tag, but only after **all** items have been added to the tag.

If this function is called when creating a WavPack file, then it uses the same blockout function that is used to write regular WavPack blocks (and should be called after flushing all the audio data and writing any WavPack metadata like trailers and MD5 sums). Note that this function may call the blockout function multiple times to write the tag.

If this function is called when editing an existing APEv2 tag, then it will seek to the correct position and write the tag using the WavpackStreamReader64 functions for this purpose or using its own standard I/O functions if the file was opened by filename. If using the old WavpackStreamReader,

or if using the new `WavpackStreamReader64` with the `truncate_here()` function pointer `NULL`, this function will pad the file with 0's in front of a tag that had been edited to a shorter length.

6.0 Handling WavPack Streams

In some applications (for example streaming applications and some filters, or cases where WavPack data might be embedded into another multimedia container) it is required for the audio file parsing functions to be separated from the decoding functions. This is accomplished in two steps with the WavPack library.

First, the parsing functions are implemented outside the WavPack library. It is very straightforward to parse WavPack files because the WavPack block header is easy to recognize, and contains easy to parse and interpret information about the block's contents and its relation to the whole WavPack file (or stream). The exact file and block formats are described in detail in the `file_format.txt` document. The stand-alone WavPack parser `wvparser.c` could be used as a starting point for a parser. `WavpackLittleEndianToNative()` and `WavpackNativeToLittleEndian()` might come in handy (see descriptions above in section 2.0).

Next, to actually decode the individual WavPack blocks into interleaved PCM, this new function introduced in 5.0.0 can be used:

```
WavpackContext *WavpackOpenRawDecoder (
    void *main_data,
    int32_t main_size,
    void *corr_data,
    int32_t corr_size,
    int16_t version,
    char *error,
    int flags,
    int norm_offset
);
```

This function is somewhat similar to `WavpackOpenFileInput()` except that instead of providing a filename to open, the caller provides a pointer to a complete buffered WavPack frame (or two frames if a correction frame is available). It decodes only a single frame (or wv/wvc pair). The “version” parameter is simply copied from the WavPack frame header (and its not required of the WavPack blocks still have their headers), and the “error”, “flags” and “norm_offset” parameters have the same meaning as the other open functions.

After creating this context, all of the applicable regular WavPack functions can be used to obtain information about the audio data contained and unpack it to channel-interleaved PCM. Obviously seeking will not work (only the passed frame is available for decoding). When the frame has been decoded, then `WavpackCloseFile()` should be called to free the resources.

Note that in this context, a "frame" is a collection of WavPack blocks that represent all the channels present. In the case of mono or [most] stereo streams, this is the same thing, but for multichannel streams each frame consists of several WavPack blocks (which can contain only 1 or 2 channels each). The first block of a multichannel stream has the `INITIAL_BLOCK` flag set and the last block has the `FINAL_BLOCK` flag set.

It is also possible to convert WavPack blocks from Matroska streams that have had their headers removed and this case is detected and done automatically. This is the reason that the “version” parameter is provided (in Matroska it is stored in the CodecPrivate area) and can actually be left zero when decoding WavPack blocks with their headers still present. For more information about WavPack in Matroska, see here:

https://www.matroska.org/technical/specs/codecid/wavpack.html

The length of the frame in samples cannot be determined with existing WavPack functions, so this additional function is provided (although a parser might know this already):

```
uint32_t WavpackGetNumSamplesInFrame (WavpackContext *wpc);
```

As mentioned above, this raw decoding functionality was introduced in WavPack 5.0.0, but a somewhat less convenient method is available using the regular `WavpackOpenFileInputEx()` function. Using that method, the specified stream reader feeds the raw WavPack block's bytes into the library when requested. The flags parameter should have the `OPEN_STREAMING` flag set so that the decoder will ignore the position and any other "whole file" information in the block headers. Also, the `OPEN_NO_CHECKSUM` flag should be set if the blocks are being reassembled and not identical to the originals. The decoder will suck up the first block (through the stream reader) that actually contains audio and stop, ready to decode. The next step is to call `WavpackUnpackSamples()` to unpack the actual number of samples in the block (which should be known by the parser).

Normally, the entire block would be unpacked and then the decoder would be ready for the next block. If a single additional sample is requested past the size of the current block the decoder will attempt to read the next block, so it is important to request the exact number of samples (unless this behavior is okay). If it is not desired to finish decoding the block then there are two options. The easiest would be to simply decode the rest of the block anyway and discard the results. Another option would be to close the context with the function `WavpackCloseFile()` and then open another context when needed.

This procedure will also work fine for multichannel WavPack files. The decoder will have to suck up all the blocks for the various channels before decoding may begin.

The downside of this method is that the application is responsible for putting headers back on Matroska “headerless” frames, and is more like to get “out of sync” with the application if corrupted data is decoded. On the other hand, it does not require the context to be closed and opened again for every frame.

7.0 Zip Format Usage

With version 11.0, WinZip Computing added WavPack Audio to the official ZIP file format standard as compression method 97, as described here:

http://www.winzip.com/ppmd_info.htm

The WavPack library can easily be used to create or decode the WavPack images stored in the ZIP files. Some issues to keep in mind:

1. To generate streams compatible with WinZip it is important that the legacy function `WavpackSetConfiguration()` be called to set configuration, **not** the new `WavpackSetConfiguration64()`. The newest stream format is not compatible with WinZip, but unmodified applications should automatically get the older, compatible streams.
2. Only lossless mode is used. WinZip's implementation uses the "very high" mode with no extra processing, although there is no reason that a different profile could not be employed as they would still be fully compatible (for example, the new "high" mode and/or the new "extra" mode could be used).
3. All bitdepths (including 32-bit floating-point) are supported. However, bitdepths that are not multiples of 8 should be rounded up to the next multiple of 8 to ensure that all samples (even illegal ones) are encoded losslessly (this is described in more detail in the WinZip document).
4. Multichannel data (with or without WAVEFORMATEXTENSIBLE) is fully supported.
5. `CONFIG_OPTIMIZE_MONO` is not available during decoding and therefore should not be used for encoding. This only applies when using libwavpack versions prior to 5.0.0 (see #1).
6. The WavPack data must have RIFF headers (to generate .wav files) and may optionally have RIFF trailers. It would not be appropriate to have WavPack generate the RIFF headers (either during encode or decode) because of the obvious danger of generating files that don't match exactly.
7. The WavPack data should probably **not** have metadata tags or MD5 sums added to it. This information would be discarded during decoding anyway and could possibly trigger an error condition in a decoder.

The easiest way of using the WavPack library to decode the embedded WavPack data would be to open a WavPack context using `WavpackOpenFileInputEx()` and provide a `WavpackStreamReader` that would read the appropriate part of the ZIP file by using an offset. If only the standard unpacking operations are used, then the WavPack library will not attempt to seek during a decode. The only flag to use for the "open" call would be `OPEN_WRAPPER`.

The most critical aspect of creating WavPack images to embed in Zip files is making sure that the decoded data will exactly match the source. This is in contrast to the case of WavPack command-line programs where some invalid WAV files may not encode (or exactly decode) for one reason or another. For this reason it is important to check the parameters in the WAV header carefully and allow only a well-defined set of known good combinations. The size of the audio data should be checked to make sure it contains the correct number of whole composite samples (or if it doesn't then this is properly handled). Since WavPack cannot properly decode WavPack files that contain no audio data (i.e. zero

samples), this case should also be avoided. In all situations where some question exists, the prudent choice would be to default to some other (more generalized) data compression method.

8.0 Multithreading

For WavPack version 5.7.0 the ability to utilize multiple threads was added to libwavpack. Internally, this is implemented as either Windows native threads or POSIX pthreads (configured via macros), so the feature should be available in virtually all platforms. Up to 15 worker threads can be requested, but of course on single-core CPUs the multithreading will not provide any significant performance increase.

There are two fundamental types of multithreading in libwavpack. The first one developed was *spacial* multithreading where the independent streams of a multichannel file are processed in parallel. This type of multithreading is available for almost all compression and decompression modes, assuming of course that the file has multiple streams (which applies to all files with more than 2 channels), and does not require anything specific from the client to utilize (other than enabling it).

The second type of multithreaded operation is called *temporal* multithreading. This is only applicable to files containing a single stream (i.e., stereo or mono) and involves processing the audio from sequential frames in parallel. This is not quite as simple as spacial multithreading for several reasons. First, because WavPack blocks are relatively long (generally 10,000 to 24,000 samples) the audio must be processed by libwavpack in large chunks so that multiple sequential frames are visible (the library does not do work in the background between client calls). For this reason, to achieve the performance boost that the spacial multithreading can provide, large buffers must be passed to the encoding and decoding functions.

The other complication of temporal multithreading only applies to encoding, but is based on the fact that during encoding some information is carried from one frame to the next (WavPack's encoding model is "continuously adaptive"). Obviously, when sequential frames are processed in parallel, information from the completion of a given frame will not usually be available when the next frame is started. In the hybrid mode there are additionally noise shaping terms that must be carried over between frames to avoid possibly audible glitches in the shaped noise.

These complications have been compensated for to some degree in the library. For the pure lossless mode, the extra overhead of starting a frame from scratch is simply absorbed and is more than offset by the gain from multithreading (but the gain is not as great as it would have been). This is less and less a factor at the higher *extra* modes (which is nice because those modes tend to much slower anyway). Note that this means that in pure lossless mode the encoder will usually *not* generate the same binary output for different numbers of threads, but the compression ratio will be approximately the same and, of course, the operation will always be lossless. This differs from the spacial multithreading mode which *always* generates the same binary output regardless of threading.

Unfortunately there was no easy fix for the noise shaping discontinuities in the hybrid modes (either lossless or lossy) and so for now these modes will not work with temporal multithreaded encoding, but will still work fine with spacial multithreaded encoding and all decoding modes. The client does not have to be aware of these specific limitations; if multithreading is enabled but not available for the specific operation, it will simply be ignored.

Enabling multithreading is very straightforward. For decoding, there are 4 bits in the "flags" parameter

of the various file open functions (i.e., `WavpackOpenFileInput...()`) that control the number of worker threads. If this field is zero then no additional threads are created; otherwise up to 15 worker threads can be requested.

For encoding, there is a field in the `WavpackConfig` structure called `worker_threads` (it used to be called `extra_flags` and was not used). When configuring a WavPack packing context using `WavpackSetConfiguration...()` set this field to the number of desired additional worker threads (from 0 up to 15 maximum).

Note that for spacial multithreading the number of additional threads will be limited by the number of streams (a 5.1 file, for example, will have 4 streams). The client does not need to be aware of this and can simply request as many threads as desired. Also note that for temporal multithreading (i.e., stereo and mono) the number of threads that can be effectively used is based on the size of the buffers being passed in to the pack or unpack functions.

Finally, there is an effective limit to the number of useful worker threads because as the number of threads goes up, the amount of time the CPU uses switching context goes up. Also, some systems, especially on battery power, will limit the CPU frequency as the number of running cores goes up, partially canceling the advantage of multithreading. All this varies with the platform and lots of other factors, so experimentation is always key. A useful starting point, and the default of the command-line programs, is to request 4 worker threads.