Multiple channel audio data and WAVE files

Article • 06/01/201725 minutes to read

Describes the standard for storing and transporting multiple channel audio data using the WAVE file format. You should have a basic understanding of multimedia file formats, and especially of audio file formats.

Introduction

The PC has become a good multimedia platform for today's applications. CD-quality audio (stereo, 16-bit, 44.1 kHz) is typically the highest audio quality achieved on the PC platform. As the industry moves forward, new content is being authored for standards that eclipse those currently defined. This includes higher sampling rates, greater bit depths, and multiple channel (greater than stereo) audio streams and playback systems. In order to keep pace with new data formats and delivery mechanisms, a standard must be produced to ensure consistency between applications and hardware.

Microsoft has extended formats that are traditionally mono or stereo and 8-bit or 16-bit. Basing WAVE_FORMAT_PCM and WAVE_FORMAT_IEEE_FLOAT on a structure named WAVEFORMATEXTENSIBLE does this. As a matter of fact, any currently registered format tag can be extended in the same way by using WAVEFORMATEXTENSIBLE. Creating a GUID (globally unique identifier) for the SubFormat field of WAVEFORMATEXTENSIBLE for existing format tags is done using the macro DEFINE_WAVEFORMATEX_GUID(x) defined in ksmedia.h. Some of the commonly used GUIDs have static names that can be used instead of the macro. These names include KSDATAFORMAT_SUBTYPE_PCM, KSDATAFORMAT_SUBTYPE_IEEE_FLOAT, KSDATAFORMAT_SUBTYPE_ALAW, KSDATAFORMAT_SUBTYPE_MULAW, KSDATAFORMAT_SUBTYPE_ADPCM, and KSDATAFORMAT_SUBTYPE_MPEG.

New formats that do not have a registered format tag just need to define a new GUID for their format type. Examples of other format GUIDs that have been defined and do not have a matching registered format tag are MEDIASUBTYPE_DOLBY_AC3, MEDIASUBTYPE_DVD_LPCM_AUDIO, and MEDIASUBTYPE_MPEG2_AUDIO.

Using WAVEFORMATEXTENSIBLE allows for specifying certain channels in the order of speaker configuration defined in this article, and/or content that indicates the number of actual significant bits in a sample container. This structure is intended for use in situations where WAVEFORMATEX is not satisfactory.

The WAVE_FORMAT_EXTENSIBLE format tag defined for the wFormatTag field of the WAVEFORMATEXTENSIBLE structure indicates that the SubFormat field of the WAVEFORMATEXTENSIBLE is to be used when determining the format of the data that the structure describes. Since this format struct includes a GUID, sub-formats representing vendor-specific wave protocols or data formats can be defined without registering new wave format tags with Microsoft.

Multiple Channel Configurations

Ambiguity within WAVE_FORMAT_PCM

In the traditional interpretation of uncompressed PCM wave formats, there was no way to connect a given channel to a given speaker in a multi-speaker configuration. The stereo-speaker configuration was always assumed, and since nChannels was always one or two, it was easy to map these to a pair of speakers. Files with nChannels > 2 were dealt with through proprietary, undocumented mappings, but the range of possibilities was small because only two speakers (hence two wave outputs) could be assumed.

With the onset of enhanced audio output configurations such as quadraphonic (four-corner), 3.1 (front left, front center, front right, low frequency enhance), 5.1 (front left, front center, front right, back left, back right, low frequency enhance), and so on, this situation has changed. The many possible speaker configurations available for both playback and authoring must be taken into account.

Today, a collection of streams intended for 5.1 delivery would most likely be streamed as a six-channel file. There still exist, however, multiple-channel files that are intended as packages of synchronized mono files. It was deemed impossible to define a default multi-channel speaker configuration for WAVE_FORMAT_PCM without causing functionality to break in one of these file types. Therefore, a channel-to-speaker specification for WAVE_FORMAT_PCM will continue to be undocumented for the case of nChannels > 2.

Default Channel Ordering

The way to deterministically link channel numbers to speaker locations, thus providing consistency among multiple channel audio files, is to define the order in which the channels are laid out in the audio file. Several external standards define parts of the following master channel layout:

1. Front Left - FL
2. Front Right - FR
3. Front Center - FC
4. Low Frequency - LF
5. Back Left - BL
6. Back Right - BR
7. Front Left of Center - FLC
8. Front Right of Center - FRC
9. Back Center - BC
10. Side Left - SL
11. Side Right - SR
12. Top Center - TC
13. Top Front Left - TFL
14. Top Front Center - TFC
15. Top Front Right - TFR
16. Top Back Left - TBL
17. Top Back Center - TBC
18. Top Back Right - TBR

The channels in the interleaved stream corresponding to these spatial positions must appear in the order specified above. This holds true even in the case of a non-contiguous subset of channels. For example, if a stream contains left, bass enhance and right, then channel 1 is left, channel 2 is right, and channel 3 is bass enhance. This enables the linkage of multi-channel streams to well-defined multi-speaker configurations.

Warning: Content intended for the Low Frequency channel may not be rendered on the speaker that the data is sent to. This is because there is no way to guarantee the frequency range of the low frequency speaker in a user's system. For this reason, a speaker that is receiving low frequency audio might filter the frequencies that it cannot handle.

Representing High-Resolution Audio

Ambiguity within WAVE_FORMAT_PCM

It is important to enumerate the assumptions made by WAVE_FORMAT_PCM. One assumption is that the nBlockAlign field contains exactly one set of samples (one block). Also, each sample must be byte-aligned within the block. The byte-aligned space that each sample consumes is called the sample's "container." Additionally, there cannot be any extra space between the end of the last sample container and the actual end of the block. In other words, nBlockAlign must be an integer multiple of nChannels, and that multiple is considered the "container size" for the sample.

Although it was not necessarily the original intent, some entities treat wBitsPerSample as equivalent to the container size. In other words, the number of bits of valid data was assumed equal to the size of the container. This assumption held well for 8-bit and 16-bit audio. With the coming of high fidelity audio equipment for render and capture, 20-bit and 24-bit streams emerged. Specifying the actual bit resolution of the samples in a byte-aligned stream became important; the accepted interpretation of wBitsPerSample started to unravel, because its value is not rigidly enforced by all software. In many cases, it continued to be the container size. In other cases, however, the field was used to indicate the bits of actual valid data, while the container size was inferred from nBlockAlign and nChannels (an example would be wBitsPerSample = 20, nBlockAlign = 8, nChannels = 2).

It was not considered feasible to resolve this issue either way without causing many legacy problems for WAVE_FORMAT_PCM, so the new formats include an additional field that clarifies this ambiguity. The new formats are intended to handle those cases in which the actual bits of precision are not equal to the container size, as well as all cases in which the container size is greater than sixteen bits.

It is not the purpose of this article to resolve the ambiguity of wBitsPerSample for WAVE_FORMAT_PCM. However, this same field will be used unambiguously in the new formats.

Specifying the Actual Bit Depth

In the new formats, wBitsPerSample is strictly defined as the container size. Containers must be byte-aligned, so wBitsPerSample must be a multiple of 8. A new field conveys exactly how many of those bits contain actual data, regardless of the size of the container. As audio goes through a processing system, the number of valid data bits per sample can change. Each sample is justified to the most significant bit, so it is easy to add additional precision on the bottom of a sample, or dither to a lower precision. The container size can be manipulated independently without implying any change in the data precision. A 24-bit stream in 32-bit containers (for efficient processing) can safely be transferred into 24-bit containers (for efficient storage space) without any data loss.

Using WAVE_FORMAT_EXTENSIBLE

To solve the problem of multiple channel ordering and high precision data, Microsoft has defined the new structure WAVE_FORMAT_EXTENSIBLE. This new structure not only has the benefit of solving these problems, it also provides a mechanism for self-registering new data formats. The SubFormat field is set to the GUID that specifies the type of data described by the WAVE_FORMAT_EXTENSIBLE structure.

Definition of WAVE_FORMAT_EXTENSIBLE

The definition (in MMREG.H and KSMEDIA.H) of WAVE_FORMAT_EXTENSIBLE is included below:

```
typedef struct {
    WAVEFORMATEX  Format;
    union {
        WORD wValidBitsPerSample; /* bits of precision */
        WORD wSamplesPerBlock;   /* valid if wBitsPerSample==0 */
        WORD wReserved;         /* If neither applies, set to zero. */
    } Samples;
    DWORD    dwChannelMask; /* which channels are present in stream */
    GUID            SubFormat;
} WAVEFORMATEXTENSIBLE, *PWAVEFORMATEXTENSIBLE;
```

Definition of WAVEFORMATPCMEX

The wave format WAVEFORMATEX is too under-specified to be used for high bit-depth samples or multiple channel streams. For cases in which the spatial locations of the channels are linked to the standard speaker locations, WAVEFORMATPCMEX is appropriate. For the WAVEFORMATEXTENSIBLE structure, the Format.cbSize field must be set to 22 and the SubFormat field must be set to KSDATAFORMAT_SUBTYPE_PCM.

The definition (in KSMEDIA.H) of KSDATAFORMAT_SUBTYPE_PCM is included below:

```
#define STATIC_KSDATAFORMAT_SUBTYPE_PCM\
    DEFINE_WAVEFORMATEX_GUID(WAVE_FORMAT_PCM)
DEFINE_GUIDSTRUCT("00000001-0000-0010-8000-00aa00389b71", KSDATAFORMAT_SUBTYPE_PCM);
#define KSDATAFORMAT_SUBTYPE_PCM DEFINE_GUIDNAMED(KSDATAFORMAT_SUBTYPE_PCM)
```

PWAVEFORMATIEEEFLOATEX can be safely cast to PWAVEFORMATEXTENSIBLE or PWAVEFORMATEX.

```
typedef WAVEFORMATEXTENSIBLE    WAVEFORMATPCMEX;
typedef WAVEFORMATPCMEX       *PWAVEFORMATPCMEX;
typedef WAVEFORMATPCMEX NEAR *NPWAVEFORMATPCMEX;
typedef WAVEFORMATPCMEX FAR  *LPWAVEFORMATPCMEX;
```

Details about WAVEFORMATEX Fields

What follows are interpretations of the fields of the WAVEFORMATEX structure, descriptions of the new fields, and numerous examples that seek to clarify the use of this format.

wFormatTag is WAVE_FORMAT_EXTENSIBLE

In the new structure WAVEFORMATEXTENSIBLE, the wFormatTag field must be set to WAVE_FORMAT_EXTENSIBLE (defined in MMREG.H). KSDATAFORMAT_SUBTYPE_PCM and KSDATAFORMAT_SUBTYPE_IEEE_FLOAT are the sub-format itself, not the actual tag.

nChannels, nSamplesPerSec, nAvgBytesPerSec Unchanged

The meanings of nChannels, nSamplesPerSec, and nAvgBytesPerSec have not been altered from WAVE_FORMAT_PCM. nChannels is the number of interleaved samples per block--the number of individual channels in the stream. nSamplesPerSec is the intended sample rate for the stream--the number of blocks that should be processed in exactly one second. nAvgBytesPerSec is used for buffer size estimation, so this number is calculated on gross block size. In fact, nAvgBytesPerSec will always be the product of nBlockAlign and nSamplesPerSec, as it was in WAVE_FORMAT_PCM.

wBitsPerSample Rigidly Defined as Container Size

In WAVEFORMATEXTENSIBLE, wBitsPerSample is defined unambiguously as the size of the container for each sample. Individual samples must be byte-aligned, so this value must be an integer multiple of 8. A case such as (nChannels = 2; wBitsPerSample = 20; nBlockAlign = 5) is explicitly disallowed in the new format.

A special case is necessary because of the nature of variable bit rate formats where a static wBitsPerSample cannot be provided. These types of formats should specify 0 in the wBitsPerSample field.

nBlockAlign and Channel Alignment

For PCM and IEEE float formats based on WAVEFORMATEXTENSIBLE, nBlockAlign must be the product of nChannels and wBitsPerSample (divided by eight). This maintains compatibility with the definition of nBlockAlign in WAVE_FORMAT_PCM (for those cases in which wBitsPerSample was assumed to be the container size).

cbSize is at Least 22

For WAVEFORMATEXTENSIBLE, cbSize must always be set to at least 22. This is the sum of the sizes of the Samples union (2), DWORD dwChannelMask (4), and GUID guidSubFormat (16). This is appended to the initial WAVEFORMATEX Format (size 18), so a WAVEFORMATPCMEX and WAVEFORMATIEEEFLOATEX structure is 64-bit aligned.

Details about the Samples Union

To keep the WAVEFORMATEXTENSIBLE structure under the 64-bit limit, wValidBitsPerSample and wSamplesPerBlock were joined together in a union called Samples. An extra field named wReserved was also added for future use.

Details about wValidBitsPerSample

The field wValidBitsPerSample is used to explicitly indicate how many bits of precision are present in the signal. Most of the time this value will be equal to wBitsPerSample. If, however, wave data originated from a 20-bit A/D, then wValidBitsPerSample could be set to 20, even though wBitsPerSample might be 24 or 32. Examples are included later in this article.

If wValidBitsPerSample is less than wBitsPerSample, then the actual PCM data is "left-aligned" within the container. The sample itself is justified most significant; all extra bits are at the least-significant portion of the container. All non-valid data bits must be set to 0.

The value of wValidBitsPerSample should never exceed that of wBitsPerSample. If this is encountered, the proper action is to reject the data format.

An entity can change wValidBitsPerSample as it processes the data. For example, an application would know that a stream with wValidBitsPerSample = 24 must be dithered to 16 bits if the output driver indicated that it supported wValidBitsPerSample = 16 only.

Although this can be very expensive from the standpoint of memory bandwidth, wBitsPerSample can be changed as well. wValidBitsPerSample indicates whether the container size (wBitsPerSample) can be reduced without data loss. A stream with wValidBitsPerSample = 20; wBitsPerSample = 32 (for processing by a 32-bit CPU) could safely be compressed to wBitsPerSample = 24 (for archiving to disk). Without wValidBitsPerSample, one would not know whether this was lossless.

Details about wSamplesPerBlock

It is often useful to know how many samples are contained in one compressed block of audio data. The wSamplesPerBlock is used in compressed formats that have a fixed number of samples within each block. This value aids in buffer estimation and position information. If wSamplesPerBlock is 0, a variable amount of samples is contained in each block of compressed audio data. In this case, buffer estimation and position information need to be obtained in other ways.

Details about wReserved

If neither wValidBitsPerSample or wSamplesPerBlock apply to the audio data being described by the WAVEFORMATEXTENSIBLE structure, set the wReserved field to 0.

Specifying Channel Locations Using dwChannelMask

```
#define SPEAKER_FRONT_LEFT             0x1
#define SPEAKER_FRONT_RIGHT            0x2
#define SPEAKER_FRONT_CENTER           0x4
#define SPEAKER_LOW_FREQUENCY          0x8
#define SPEAKER_BACK_LEFT              0x10
#define SPEAKER_BACK_RIGHT             0x20
#define SPEAKER_FRONT_LEFT_OF_CENTER   0x40
#define SPEAKER_FRONT_RIGHT_OF_CENTER  0x80
#define SPEAKER_BACK_CENTER            0x100
#define SPEAKER_SIDE_LEFT              0x200
#define SPEAKER_SIDE_RIGHT             0x400
#define SPEAKER_TOP_CENTER             0x800
#define SPEAKER_TOP_FRONT_LEFT         0x1000
#define SPEAKER_TOP_FRONT_CENTER       0x2000
#define SPEAKER_TOP_FRONT_RIGHT        0x4000
#define SPEAKER_TOP_BACK_LEFT          0x8000
#define SPEAKER_TOP_BACK_CENTER        0x10000
#define SPEAKER_TOP_BACK_RIGHT         0x20000
#define SPEAKER_RESERVED               0x80000000
```

These values correspond exactly to the master channel layout defined in various external standards.

As an example, assume nChannels = 4; dwChannelMask = 0x00000033. This indicates that the audio channels are intended for playback to the Front Left, Front Right, Back Left and Back Right speakers. The channel data should be interleaved in that order within each block.

When using WAVEFORMATEXTENSIBLE, channel locations beyond this predefined set of 18 are considered reserved. One should make no assumptions regarding ordering of channels beyond these, other than to assume that Microsoft will conform to additional standards.

Details about dwChannelMask

The field dwChannelMask indicates which channels are present in the multi-channel stream. The least significant bit corresponds with the Front Left speaker, the next least significant bit corresponds to the Front Right speaker, and so on, continuing in the order defined in Section 2. The channels specified in dwChannelMask must be present in the prescribed order (from least significant bit up). In other words, if only Front Left and Front Center are specified, then Front Left should come first in the interleaved stream.

Should nChannels be less than the number of bits set in dwChannelMask, then the extra (most significant) bits in dwChannelMask are ignored. Should nChannels exceed the number of bits set in dwChannelMask, then the remaining channels are not assigned to any particular speaker location. An audio device would render the remaining channel data to output ports not in use.

If an audio sink, such as WDM Audio's built-in kernel mixer, does not know to process the extra channels without speaker locations, the data is not rendered. Having nChannels exceed the number of bits set in dwChannelMask can produce inconsistent results and should be avoided if possible.

If, for example in a multi-channel audio authoring application, no speaker location is desired on any of the mono streams, the dwChannelMask should explicitly be set to 0. A dwChannelMask of 0 tells the audio device to render the first channel to the first port on the device, the second channel to the second port on the device, and so on. This also means that if the device doesn't know how to process the raw audio streams, it should not accept the multi-channel stream with a dwChannelMask of 0. A device like a digital mixer or a digital audio storage device (hard disk, ADAT, and so on) might want to accept formats without particular speaker locations.

The dwChannelMask value 0xFFFFFFFF (or any value in which the most significant bit of the DWORD is set) is pre-defined to indicate that an entity supports all possible channel configurations. An example would be WDM Audio's built-in kernel mixer. There will never be a location corresponding to the most significant bit in dwChannelMask, so this value is unambiguous.

This format does not deal with speaker locations that do not correspond to the defined values, although Microsoft reserves the right to define them in the future.

Examples

Multi-Channel 16-bit

The following WAVEFORMATPCMEX structure indicates a 16-bit four-channel audio rendering with Front Left, Front Right, Back Left and Back Right:

```
WAVEFORMATPCMEX  waveFormatPCMEx;
waveFormatPCMEx.Format.wFormatTag = WAVE_FORMAT_EXTENSIBLE;
waveFormatPCMEx.Format.nChannels = 4;
waveFormatPCMEx.Format.nSamplesPerSec = 44100L;
waveFormatPCMEx.Format.nAvgBytesPerSec = 352800L;
waveFormatPCMEx.Format.nBlockAlign = 8;  /* Same as the usual */
waveFormatPCMEx.Format.wBitsPerSample = 16;
waveFormatPCMEx.Format.cbSize = 22;  /* After this to GUID */
waveFormatPCMEx.wValidBitsPerSample = 16;  /* All bits have data */
waveFormatPCMEx.dwChannelMask = SPEAKER_FRONT_LEFT |
                                SPEAKER_FRONT_RIGHT |
                                SPEAKER_BACK_LEFT |
                                SPEAKER_BACK_RIGHT;
                                // Quadraphonic = 0x00000033
waveFormatPCMEx.SubFormat = KSDATAFORMAT_SUBTYPE_PCM;  // Specify PCM
```

The four channels of audio data are packed into memory as follows, and a pointer to that memory is stored in the lpData member of the WAVEHDR structure.

Byte 1 - Channel 1, Front Left, Low Order Byte

Byte 2 - Channel 1, Front Left, High Order Byte

Byte 3 - Channel 2, Front Right, Low Order Byte

Byte 4 - Channel 2, Front Right, High Order Byte

Byte 5 - Channel 3, Back Left, Low Order Byte

Byte 6 - Channel 3, Back Left, High Order Byte

Byte 7 - Channel 4, Back Right, Low Order Byte

Byte 8 - Channel 4, Back Right, High Order Byte

Byte 9 - Channel 1, Front Left, Low Order Byte, Sample 2

Byte 10 - Channel 1, Front Left, High Order Byte, Sample 2 etc.

Again, PWAVEFORMATPCMEX can be safely cast to PWAVEFORMATEXTENSIBLE or PWAVEFORMATEX for portability.

Stereo 20-bit, Padded wBitsPerSample

The following WAVEFORMATPCMEX structure indicates 2 channels of 20-bit resolution, packed into 24-bit containers on disk:

```
WAVEFORMATPCMEX    waveFormatPCMEx;
waveFormatPCMEx.Format.wFormatTag = WAVE_FORMAT_EXTENSIBLE;
waveFormatPCMEx.Format.nChannels = 2;
waveFormatPCMEx.Format.nSamplesPerSec = 44100L;
waveFormatPCMEx.Format.nAvgBytesPerSec = 264600L; // Compute using nBlkAlign * nSamp/Sec
waveFormatPCMEx.Format.nBlockAlign = 6;
waveFormatPCMEx.Format.wBitsPerSample = 24; //Container has 3 bytes waveFormatPCMEx.Format.cbSize = 22;
waveFormatPCMEx.wValidBitsPerSample = 20;  // Top 20 bits have data
waveFormatPCMEx.dwChannelMask = SPEAKER_FRONT_LEFT |
                                SPEAKER_FRONT_RIGHT |
                                // Stereo = 0x00000003
waveFormatPCMEx.SubFormat = KSDATAFORMAT_SUBTYPE_PCM;  // Specify PCM
```

The two channels of three-byte audio data are packed into memory as follows, and a pointer to that memory is stored in the lpData member of the WAVEHDR structure.

Byte 1 - Channel 1, Front Left, Low Order Byte, only top four bits have valid data

Byte 2 - Channel 1, Front Left, Mid Order Byte, all valid data

Byte 3 - Channel 1, Front Left, High Order Byte, all valid data

Byte 4 - Channel 2, Front Right, Low Order Byte, top four bits have valid data

Byte 5 - Channel 2, Front Right, Mid Order Byte, all valid data

Byte 6 - Channel 2, Front Right, High Order Byte, all valid data

Byte 7 - Channel 1, Front Left, Low Order Byte, top four bits have valid data, Sample 2

Byte 8 - Channel 1, Front Left, Mid Order Byte, all valid data, Sample 2 etc.

NOTE: nAvgBytesPerSec is computed using the block size (nBlockAlign), as opposed to the container size (wBitsPerSample) or the actual number of significant bits (wValidBitsPerSample).

6 Channels in 5.1 Format

The following WAVEFORMATPCMEX structure is an example of a structure that might be specified as the output of a decoder producing audio streams for a 5.1 speaker layout.

```
WAVEFORMATPCMEX    waveFormatPCMEx;
waveFormatPCMEx.Format.wFormatTag = WAVE_FORMAT_EXTENSIBLE;
waveFormatPCMEx.Format.nChannels = 6;
waveFormatPCMEx.Format.nSamplesPerSec = 48000L;
waveFormatPCMEx.Format.nAvgBytesPerSec = 864000L; // Compute using nBlkAlign * nSamp/Sec
waveFormatPCMEx.Format.nBlockAlign = 18;
waveFormatPCMEx.Format.wBitsPerSample = 24; //Container has 3 bytes waveFormatPCMEx.Format.cbSize = 22;
waveFormatPCMEx.wValidBitsPerSample = 20;  // Top 20 bits have data
waveFormatPCMEx.dwChannelMask = KSAUDIO_SPEAKER_5POINT1;
                             // SPEAKER_FRONT_LEFT | SPEAKER_FRONT_RIGHT |
                             // SPEAKER_FRONT_CENTER | SPEAKER_LOW_FREQUENCY |
                             // SPEAKER_BACK_LEFT  | SPEAKER_BACK_RIGHT
waveFormatPCMEx.SubFormat =  KSDATAFORMAT_SUBTYPE_PCM;  // Specify PCM
```

The two channels of three-byte audio data are packed into memory as follows, and a pointer to that memory is stored in the lpData member of the WAVEHDR structure.

Byte 1 - Channel 1, Front Left, Low Order Byte, only top four bits have valid data

Byte 2 - Channel 1, Front Left, Mid Order Byte, all valid data

Byte 3 - Channel 1, Front Left, High Order Byte, all valid data

Byte 4 - Channel 2, Front Right, Low Order Byte, top four bits have valid data

Byte 5 - Channel 2, Front Right, Mid Order Byte, all valid data

Byte 6 - Channel 2, Front Right, High Order Byte, all valid data

Byte 7 - Channel 3, Front Center, Low Order Byte, only top four bits have valid data

Byte 8 - Channel 3, Front Center, Mid Order Byte, all valid data

Byte 9 - Channel 3, Front Center, High Order Byte, all valid data

Byte 10 - Channel 4, Low Frequency, Low Order Byte, top four bits have valid data

Byte 11 - Channel 4, Low Frequency, Mid Order Byte, all valid data

Byte 12 - Channel 4, Low Frequency, High Order Byte, all valid data

Byte 13 - Channel 5, Back Left, Low Order Byte, only top four bits have valid data

Byte 14 - Channel 5, Back Left, Mid Order Byte, all valid data

Byte 15 - Channel 5, Back Left, High Order Byte, all valid data

Byte 16 - Channel 6, Back Right, Low Order Byte, top four bits have valid data

Byte 17 - Channel 6, Back Right, Mid Order Byte, all valid data

Byte 18 - Channel 6, Back Right, High Order Byte, all valid data

Byte 19 - Channel 1, Front Left, Low Order Byte, top four bits have valid data, Sample 2

Byte 20 - Channel 1, Front Left, Mid Order Byte, all valid data, Sample 2 etc.

Unspecified location channel, DWORD Container

The following WAVEFORMATPCMEX structure indicates three channels of 23-bit resolution, packed into 32-bit containers on disk, of which only the first two channels are localized:

```
WAVEFORMATPCMEX    waveFormatPCMEx;
waveFormatPCMEx.Format.wFormatTag = WAVE_FORMAT_EXTENSIBLE;
waveFormatPCMEx.Format.nChannels = 3;
waveFormatPCMEx.Format.nSamplesPerSec = 48000L;
waveFormatPCMEx.Format.nAvgBytesPerSec = 576000L; // Compute using nBlkAlign * nSamp/Sec
waveFormatPCMEx.Format.nBlockAlign = 12;
waveFormatPCMEx.Format.wBitsPerSample = 32; //Container has 4 bytes waveFormatPCMEx.Format.cbSize = 22;
waveFormatPCMEx.wValidBitsPerSample = 23; // Top 23 bits have data
waveFormatPCMEx.dwChannelMask =
                         SPEAKER_FRONT_LEFT_OF_CENTER |
                         SPEAKER_FRONT_RIGHT_OF_CENTER
                     // Left, Right of Center = 0x000000C0
WAVEFORMATPCMEX.SUBFORMAT =  KSDATAFORMAT_SUBTYPE_PCM; // SPECIFY PCM
```

The two channels of four-byte audio data are packed into memory as follows, and a pointer to that memory is stored in the lpData member of the WAVEHDR structure.

Byte 1 - Channel 1, Left of Center, Low Order Byte, no valid data

Byte 2 - Channel 1, Left of Center, Mid Low Order Byte, only top seven bits have valid data

Byte 3 - Channel 1, Left of Center, Mid High Order Byte, all valid data

Byte 4 - Channel 1, Left of Center, High Order Byte, all valid data

Byte 5 - Channel 2, Right of Center, Low Order Byte, no valid data

Byte 6 - Channel 2, Right of Center, Mid Low Order Byte, top seven bits have valid data

Byte 7 - Channel 2, Right of Center, Mid High Order Byte, all valid data

Byte 8 - Channel 2, Right of Center, High Order Byte, all valid data

Byte 9 - Channel 3, Unspecified Location, Low Order Byte, no valid data

Byte 10 - Channel 3, Unspecified Location, Mid Low Order Byte, top seven bits have valid data

Byte 11 - Channel 3, Unspecified Location, Mid High Order Byte, all valid data

Byte 12 - Channel 3, Unspecified Location, High Order Byte, all valid data

Byte 13 - Channel 1, Left of Center, Low Order Byte, no valid data, Sample 2

Byte 14 - Channel 1, Left of Center, Mid Low Order Byte, top seven bits have valid data, Sample 2 etc.

These streams could be sent through a scaling algorithm that puts fractional values into the lower 9 bits as well, and the only change to the values in the wave format structure would be:

```
// All 32 bits have datawaveFormat
PCMEx.wValidBitsPerSample = 32;
```

An Example Using WAVEFORMATIEEEFLOATEX

The following WAVEFORMATIEEEFLOATEX structure indicates seven channels of 18-bit data, in 32-bit containers. Channel seven contains valid audio data that simply is not assigned a spatial location.

```
PWAVEFORMATIEEEFLOATEX   waveFormatIEEEFloatEx;
waveFormatIEEEFloatEx.Format.wFormatTag = WAVE_FORMAT_EXTENSIBLE;
waveFormatIEEEFloatEx.Format.nChannels = 7;
waveFormatIEEEFloatEx.Format.nSamplesPerSec = 48000L;
waveFormatIEEEFloatEx.Format.nAvgBytesPerSec = 1344000L;
waveFormatIEEEFloatEx.Format.nBlockAlign = 28;
waveFormatIEEEFloatEx.Format.wBitsPerSample = 32;
waveFormatIEEEFloatEx.Format.cbSize = 22;
waveFormatIEEEFloatEx.wValidBitsPerSample = 18; //Top 18 bits have data
waveFormatIEEEFloatEx.dwChannelMask = SPEAKER_FRONT_LEFT |
                                      SPEAKER_FRONT_RIGHT |
                                      SPEAKER_FRONT_CENTER |
                                      SPEAKER_LOW_FREQUENCY |
                                      SPEAKER_BACK_LEFT |
                                      SPEAKER_BACK_RIGHT;
waveFormatPCMEx.SubFormat = KSDATAFORMAT_SUBTYPE_IEEE_FLOAT; // 5.1 + unassigned chan
```

Byte 1 - Channel 1, Front Left, Low Order Byte, no valid data

Byte 2 - Channel 1, Front Left, Mid Low Order Byte, only top two bits have valid data

Byte 3 - Channel 1, Front Left, Mid High Order Byte, all valid data

Byte 4 - Channel 1, Front Left, High Order Byte, all valid data

Byte 5 - Channel 2, Front Right, Low Order Byte, no valid data

Byte 6 - Channel 2, Front Right, Mid Low Order Byte, top two bits have valid data

Byte 7 - Channel 2, Front Right, Mid High Order Byte, all valid data

Byte 8 - Channel 2, Front Right , High Order Byte, all valid data

Byte 9 - Channel 3, Front Center, Low Order Byte, no valid data

Byte 10 - Channel 3, Front Center, Mid Low Order Byte, top two bits have valid data

Byte 11 - Channel 3, Front Center, Mid High Order Byte, all valid data

Byte 12 - Channel 3, Front Center, High Order Byte, all valid data

Byte 13 - Channel 4, Low Frequency Enhance, Low Order Byte, no valid data

Byte 14 - Channel 4, Low Frequency Enhance, Mid Low Order Byte, top two bits have valid data

Byte 15 - Channel 4, Low Frequency Enhance, Mid High Order Byte, all valid data

Byte 16 - Channel 4, Low Frequency Enhance, High Order Byte, all valid data

Byte 17 - Channel 5, Back Left, Low Order Byte, no valid data

Byte 18 - Channel 5, Back Left, Mid Low Order Byte, top two bits have valid data

Byte 19 - Channel 5, Back Left, Mid High Order Byte, all valid data

Byte 20 - Channel 5, Back Left, High Order Byte, all valid data

Byte 21 - Channel 6, Back Right, Low Order Byte, no valid data

Byte 22 - Channel 6, Back Right, Mid Low Order Byte, top two bits have valid data

Byte 23 - Channel 6, Back Right, Mid High Order Byte, all valid data

Byte 24 - Channel 6, Back Right, High Order Byte, all valid data

Byte 25 - Channel 7, Unspecified Location, Low Order Byte, no valid data

Byte 26 - Channel 7, Unspecified Location, Mid Low Order Byte, top two bits have valid data

Byte 27 - Channel 7, Unspecified Location, Mid High Order Byte, all valid data

Byte 28 - Channel 7, Unspecified Location, High Order Byte, all valid data

Byte 29 - Channel 1, Front Left, Low Order Byte, no valid data, Sample 2

Byte 30 - Channel 1, Front Left, Mid Low Order Byte, top two bits have valid data, Sample 2 etc.

Multiple mono streams (no speaker location)

The following WAVEFORMATIEEEFLOATEX structure indicates six channels of 32-bit floating point data, in 32-bit containers. None of the mono channels are intended for a speaker, so the audio device sends the streams to the device-defined ports in ascending order.

```
PWAVEFORMATIEEEFLOATEX     waveFormatIEEEFloatEx;
waveFormatIEEEFloatEx.Format.wFormatTag = WAVE_FORMAT_EXTENSIBLE;
waveFormatIEEEFloatEx.Format.nChannels = 6;
waveFormatIEEEFloatEx.Format.nSamplesPerSec = 96000L;
waveFormatIEEEFloatEx.Format.nAvgBytesPerSec = 1152000L;
waveFormatIEEEFloatEx.Format.nBlockAlign = 24;
waveFormatIEEEFloatEx.Format.wBitsPerSample = 32;
waveFormatIEEEFloatEx.Format.cbSize = 22;
waveFormatIEEEFloatEx.wValidBitsPerSample = 32;
waveFormatIEEEFloatEx.dwChannelMask = 0;
waveFormatPCMEx.SubFormat = KSDATAFORMAT_SUBTYPE_IEEE_FLOAT;
```

Byte 1 - Mono Channel 1, Low Order Byte, valid floating point data

Byte 2 - Mono Channel 1, Mid Low Order Byte, valid floating point data

Byte 3 - Mono Channel 1, Mid High Order Byte, valid floating point data

Byte 4 - Mono Channel 1, High Order Byte, valid floating point data

Byte 5 - Mono Channel 2, Low Order Byte, valid floating point data

Byte 6 - Mono Channel 2, Mid Low Order Byte, valid floating point data

Byte 7 - Mono Channel 2, Mid High Order Byte, valid floating point data

Byte 8 - Mono Channel 2, High Order Byte, valid floating point data

Byte 9 - Mono Channel 3, Low Order Byte, valid floating point data

Byte 10 - Mono Channel 3, Mid Low Order Byte, valid floating point data

Byte 11 - Mono Channel 3, Mid High Order Byte, valid floating point data

Byte 12 - Mono Channel 3, High Order Byte, valid floating point data

Byte 13 - Mono Channel 4, Low Order Byte, valid floating point data

Byte 14 - Mono Channel 4, Mid Low Order Byte, valid floating point data

Byte 15 - Mono Channel 4, Mid High Order Byte, valid floating point data

Byte 16 - Mono Channel 4, High Order Byte, valid floating point data

Byte 17 - Mono Channel 5, Low Order Byte, valid floating point data

Byte 18 - Mono Channel 5, Mid Low Order Byte, valid floating point data

Byte 19 - Mono Channel 5, Mid High Order Byte, valid floating point data

Byte 20 - Mono Channel 5, High Order Byte, valid floating point data

Byte 21 - Mono Channel 6, Low Order Byte, valid floating point data

Byte 22 - Mono Channel 6, Mid Low Order Byte, valid floating point data

Byte 23 - Mono Channel 6, Mid High Order Byte, valid floating point data

Byte 24 - Mono Channel 6, High Order Byte, valid floating point data

Byte 25 - Mono Channel 1, Low Order Byte, valid floating point data, Sample 2

Byte 26 - Mono Channel 1, Mid Low Order Byte, valid floating point data, Sample 2 etc.

Send comments about this topic to Microsoft