International Telecommunication Union

ITU-R

Radiocommunication Sector of ITU

Recommendation ITU-R BS.2088-0

(10/2015)

Long-form file format for the international exchange of audio programme materials with metadata

BS Series

Broadcasting service (sound)

International Telecommunication Union

Foreword

The role of the Radiocommunication Sector is to ensure the rational, equitable, efficient and economical use of the radio-frequency spectrum by all radiocommunication services, including satellite services, and carry out studies without limit of frequency range on the basis of which Recommendations are adopted.

The regulatory and policy functions of the Radiocommunication Sector are performed by World and Regional Radiocommunication Conferences and Radiocommunication Assemblies supported by Study Groups.

Policy on Intellectual Property Right (IPR)

ITU-R policy on IPR is described in the Common Patent Policy for ITU-T/ITU-R/ISO/IEC referenced in Annex 1 of Resolution ITU-R 1. Forms to be used for the submission of patent statements and licensing declarations by patent holders are available from http://www.itu.int/ITU-R/go/patents/en where the Guidelines for Implementation of the Common Patent Policy for ITU-T/ITU-R/ISO/IEC and the ITU-R patent information database can also be found.

Series of ITU-R Recommendations

(Also available online at http://www.itu.int/publ/R-REC/en)

Series	Title
BO	Satellite delivery
BR	Recording for production, archival and play-out; film for television
BS	**Broadcasting service (sound)**
BT	Broadcasting service (television)
F	Fixed service
M	Mobile, radiodetermination, amateur and related satellite services
P	Radiowave propagation
RA	Radio astronomy
RS	Remote sensing systems
S	Fixed-satellite service
SA	Space applications and meteorology
SF	Frequency sharing and coordination between fixed-satellite and fixed service systems
SM	Spectrum management
SNG	Satellite news gathering
TF	Time signals and frequency standards emissions
V	Vocabulary and related subjects

***Note**: This ITU-R Recommendation was approved in English under the procedure detailed in Resolution ITU-R 1.*

Electronic Publication
Geneva, 2017

RECOMMENDATION ITU-R BS.2088-0*

Long-form file format for the international exchange of audio programme materials with metadata

(2015)

Scope

This Recommendation contains the specification of the BW64 (Broadcast Wave 64Bit) audio file format including the new chunks <ds64>, <axml> and <chna> which enable the file to carry large multichannel files and metadata including the Audio Definition Model (ADM) specified in Recommendation ITU-R BS.2076.

Keywords

File, file format, metadata, wave, BW64, exchange, audio programme, WAV, BWF, RIFF, RF64, wave-file, Immersive

The ITU Radiocommunication Assembly,

considering

a) that storage media based on Information Technology, including data disks and tapes, have penetrated all areas of audio production for radio broadcasting, namely non-linear editing, on-air play-out and archives;

b) that this technology offers significant advantages in terms of operating flexibility, production flow and station automation and it is therefore attractive for the up-grading of existing studios and the design of new studio installations;

c) that the adoption of a single file format for signal interchange would greatly simplify the interoperability of individual equipment and remote studios, it would facilitate the desirable integration of editing, on-air play-out and archiving;

d) that a minimum set of broadcast related information must be included in the file to document the metadata related to the audio signal;

e) that, to ensure the compatibility between applications with different complexity, a minimum set of functions, common to all the applications able to handle the recommended file format must be agreed;

f) that Recommendation ITU-R BS.646 defines the digital audio format used in audio production for radio and television broadcasting;

g) that the compatibility with currently available commercial file formats could minimize the industry efforts required to implement this format in the equipment;

h) that a standard format for the coding history information and other related metadata would simplify the use of the information after programme exchange;

i) that the quality of an audio signal is influenced by signal processing experienced by the signal, particularly by the use of non-linear coding and decoding during bit-rate reduction processes;

j) that future audio systems will require metadata associated with the audio to be carried in the file;

---

* Radiocommunication Study Group 6 made editorial amendments to this Recommendation in the year 2016, in March 2017 and in October 2017 in accordance with Resolution ITU-R 1..

k) that future audio systems will use a variety of multichannel configurations including channel, object and scene-based audio such as specified in Recommendation ITU-R BS.2051;

l) that Recommendation ITU-R BS.1352 has limitations with respect to file size and its ability to carry additional metadata;

m) that multichannel audio files could potentially be larger than 4 Gbytes in size,

recommends

1 that, for the exchange of audio programmes, the audio signal parameters, sampling frequency (part 1), bit depth (part 4 and 5) and pre-emphasis (part 6) should be set in agreement with the relevant parts of Recommendation ITU-R BS.646;

2 that the file format specified in Annex 1 should be used for the interchange of audio programmes in the following use-cases:

- in WAVE-file based environments, where WAVE-file based broadcast applications wish to upgrade to handle immersive content, while maintaining forward compatibility;
- in file-based workflows where a mixed library of legacy WAVE-file based content and immersive content will exist;
- in file-based workflows, where a single package data plus metadata wrapper is preferred;

Annex 1 (normative)

Specification of the BW64 File Format

1 Introduction

The BW64 format is based on the WAVE audio file format (described in Annex 2), which is a type of file specified in the Resource Interchange File Format (RIFF). WAVE files specifically contain audio data. The basic building block of the RIFF file format, called a chunk, contains a group of tightly related pieces of information. It consists of a chunk identifier, an integer value representing the length in bytes and the information carried. A RIFF file is made up of a collection of chunks. This BW64 format uses the core elements of the format as described in EBU Tech 3306.

The BWF file format, Recommendation ITU-R BS.1352, has a number of limitations, most notably:

- Maximum file size of less than 4Gbytes.
- No support for advanced multichannel audio.
- Inadequate support for technical metadata.

The BW64 format described in this Recommendation aims to overcome these limitations, and maintain as much compatibility as possible with the Recommendation ITU-R BS.1352 format with many of the core elements shared.

There is an increasing demand on the transfer of metadata, especially the transfer of Audio Definition Model (ADM) metadata according to Recommendation ITU-R BS.2076. This Recommendation includes a definition of the <axml> chunk for storing and transferring metadata as XML.

The primary purpose of the <chna> chunk described in this Recommendation is to provide the references from each track in a BW64 file to the IDs in the ADM metadata defined in Recommendation ITU-R BS.2076.

Apart from the primary purpose of linking each track in the file with its associated ADM metadata, the <chna> chunk also allows faster access to ADM IDs without having to gain access the XML metadata (if the IDs are within a range of values predefined for standard ADM configurations). As the <chna> chunk can be fixed in size, and is placed before the <data> and <axml> chunks, it is easier to access, generate or modify its contents on the fly.

Data types throughout this document are used in accordance with Annex 3.

2 BW64 format description

2.1 Contents of a BW64 format file

A BW64 format file should start with the mandatory "WAVE" header and at least the following chunks:

```
<WAVE-form> ->
    BW64('WAVE'
        <ds64-ck>       // ds64 chunk for 64-bit addressing
        <fmt-ck>        // Format of the audio signal: PCM/non-PCM
        <chna-ck>       // chna chunk for ADM look-up
        <axml-ck>       // axml chunk for carrying XML metadata
        <wave-data>)    // sound data
```

NOTE 1 – Additional chunks may be present in the file. Some of these may be outside the scope of this Recommendation. Applications may or may not interpret or make use of these chunks, so the integrity of the data contained in such unknown chunks cannot be guaranteed. However, compliant applications shall pass on unknown chunks transparently.

NOTE 2 – It would be permissible to place the <axml> chunk after the <data> chunk, as the XML metadata will likely to be of an unknown length and a known starting position of the audio samples in the file might be more practical.

NOTE 3 – This ordering of the <chna>, <axml> and <data> chunks is not strict, and it may be preferable to place the <axml> chunk after the <data> chunk when XML metadata is edited in post-production.

2.2 Existing chunks defined as part of the RIFF/WAVE standard

The RIFF/WAVE standard uses a number of chunks that are already defined. These are:

- <RIFF>
- <fmt>
- <data>

These chunks are described in § 2.4.1.

The RIFF/WAVE is a subset of the ITU-R BS.1352 format. Recommendation ITU-R BS.1352 contains these additional chunks:

- <bext>
- <ubxt>

These chunks will not be included in the BW64 format, which provides a more flexible solution carrying broadcast metadata.

2.3 New Chunks and Structs in the BW64 format

The new chunks introduced for BW64 are:

- <BW64>
- <ds64>
- <JUNK>
- <axml>
- <chna>

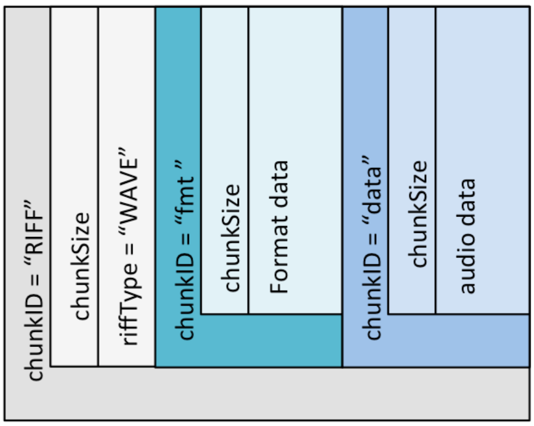
These chunks are described in §§ 3 to 6.

2.4 Using the <ds64> chunk to enable the use of files greater than 4 Gbyte in size

The reason for the 4 Gbyte barrier is the 32-bit addressing in RIFF/WAVE and BWF. With 32 bits a maximum of 4294967296 bytes = 4 Gbyte can be addressed. To solve this issue, 64-bit addressing is needed. The structure of a basic conventional RIFF/WAVE file is shown in Fig. 1, where the chunkSize fields are 32-bit numbers representing the sizes of their chunks.

FIGURE 1

Basic RIFF/WAVE file structure



Just changing the size of every field in a BWF to 64-bit would produce a file that is not compatible with the standard RIFF/WAVE format – an obvious but important observation.

The approach adopted is to define a new 64-bit based RIFF called BW64 that is identical to the original RIFF/WAVE format, except for the following changes:

- The ID ‘BW64’ is used instead of ‘RIFF’ in the first four bytes of the file
- A mandatory <ds64> (data size 64) chunk is added, which has to be the first chunk after the “BW64 chunk”.

The ‘ds64’ chunk has two mandatory 64-bit integer values, which replace two 32-bit fields of the RIFF/WAVE format:

- bw64Size (replaces the size field of the <RIFF> chunk)
- dataSize (replaces the size field of the <data> chunk)

For all two 32-bit fields of the RIFF/WAVE format the following rule applies:

If the 32-bit value in the field is not “−1” (= FFFFFFFF hex) then this 32-bit value is used.

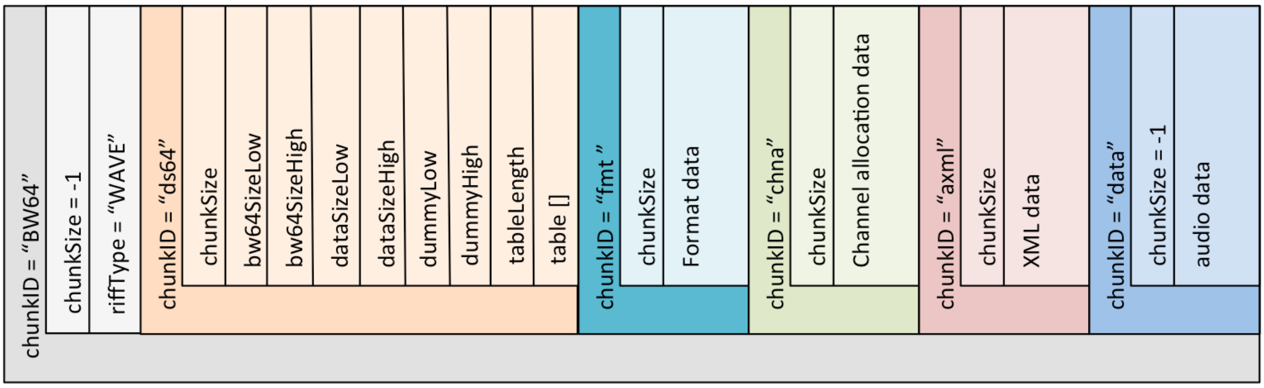
If the 32-bit value in the field is “−1” the 64-bit value in the ‘ds64’ chunk is used instead.

- One optional array of structs (see Annex A) with additional 64-bit chunk sizes is possible

The complete structure of the BW64 file format is illustrated in Fig. 2, where the chunkSize values for the <BW64> and <data> chunks are set to −1, to allow them to use 64-bit size values from the <ds64> chunk.

FIGURE 2

BW64 file structure



2.5 Achieving compatibility between RIFF/WAVE and BW64

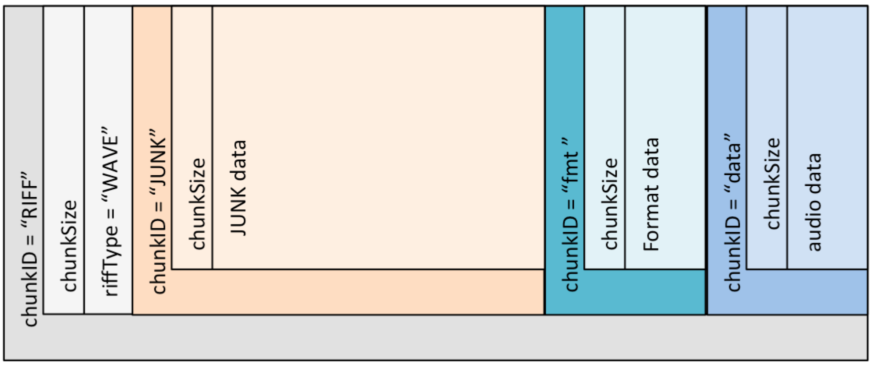
In spite of higher sampling frequencies and multi-channel audio, some production audio files will inevitably be smaller than 4 Gbyte and they should therefore stay in the short-form RIFF/WAVE format (as described in Annex 2). The problem arises that a recording application cannot know in advance whether the recorded audio it is compiling will exceed 4 Gbyte or not at end of recording (i.e. whether it needs to use BW64 or not).

The solution is to enable the recording application to switch from RIFF/WAVE to BW64 on the fly at the 4 Gbyte size limit, while the recording is still going on.

This is achieved by reserving additional space in the RIFF/WAVE by inserting a <JUNK> chunk that is of the same size as a <ds64> chunk. This reserved space has no meaning for short-form WAVE, but will become the <ds64> chunk, if a transition to BW64 is necessary. The diagram in Fig. 3 shows the <JUNK> placeholder chunk placed before the <fmt > chunk.

FIGURE 3

File structure with JUNK chunk



At the beginning of a recording, a BW64-aware application will create a standard RIFF/WAVE with a 'JUNK' chunk as the first chunk. While recording, it will check the RIFF and data sizes. If they exceed 4 Gbyte, the application will:

- Replace the chunkID <JUNK> with <ds64> chunk. (This transforms the previous <JUNK> chunk into a <ds64> chunk).
- Insert the RIFF size, 'data' chunk size and sample count in the <ds64> chunk
- Set RIFF size, 'data' chunk size and sample count in the 32 bit fields to −1 = FFFFFFFF hex
- Replaces the ID 'RIFF' with 'BW64' in the first four bytes of the file
- Continue with the recording.

2.6 Existing Chunks and Structs in the RIFF/WAVE format

The chunks that exist in the RIF/WAVE format as shown below:

```
struct RiffChunk            // declare RiffChunk structure
{
    CHAR   chunkId[4];      // 'RIFF'
    DWORD  chunkSize;       // 4 byte size of the traditional RIFF/WAVE file
    CHAR   riffType[4];     // 'WAVE'
};

struct FormatChunk          // declare FormatChunk structure
{
    CHAR   chunkId[4];      // 'fmt '
    DWORD  chunkSize;       // 4 byte size of the 'fmt ' chunk
    WORD   formatTag;       // WAVE_FORMAT_PCM = 0x0001, etc.
    WORD   channelCount;    // 1 = mono, 2 = stereo, etc.
    DWORD  sampleRate;      // 32000, 44100, 48000, etc.
    DWORD  bytesPerSecond;  // only important for compressed formats
    WORD   blockAlignment;  // container size (in bytes) of one set of samples
    WORD   bitsPerSample;   // valid bits per sample 16, 20 or 24
    WORD   cbSize;          // extra information (after cbSize) to store
                            // should be set to zero as extraData is not used
    CHAR   extraData[22];   // extra data of WAVE_FORMAT_EXTENSIBLE when necessary,
                            // should not be used as cbSize will be zero.
};

struct DataChunk            // declare DataChunk structure
{
    CHAR   chunkId[4];      // 'data'
    DWORD  chunkSize;       // 4 byte size of the 'data' chunk
    CHAR   waveData[ ];     // audio samples
};
```

The empty array brackets indicate a variable number of elements can be used (including zero).

2.6.1 Elements of the <RIFF> chunk

The <RIFF> chunk is the top level for the file.

Field	Description
chunkId	This is the 4 character array {‘R’, ‘I’, ‘F’, ‘F’} used for chunk identification.
chunkSize	4 byte value of the size of the file.
riffType	This is the 4 character array {‘W’, ‘A’, ‘V’, ‘E’} indicates that the file is a WAVE-type audio file.

2.6.2 Elements of the <fmt > chunk

The <fmt > chunk contains information about the audio sample formats stored in the <data> chunk.

Field	Description
chunkId	This is the 4 character array {‘f’, ‘m’,‘t’, ‘ ’} used for chunk identification.
chunkSize	4 byte value of the size of the chunk.
formatTag	This is a 2 byte value that represents the format of the audio samples. The value of 0x0001 means the format is PCM, 0x0000 for unknown formats.
channelCount	This is a 2 byte value indicating the number of audio tracks in the file.
sampleRate	This is a 4 byte value indicating the sample rate of the audio in Hz.
bytesPerSecond	The average number of bytes per second at which the waveform data should be transferred. Playback software can estimate the buffer size using this value.
blockAlignment	The block alignment (in bytes) of the waveform data. Playback software needs to process a multiple of **blockAlignment** bytes of data at a time, so the value of **blockAlignment** can be used for buffer alignment.
bitsPerSample	This is the number of bits per sample per channel. Each channel is assumed to have the same sample resolution. If this field is not needed, then it should be set to zero.
cbSize	The size in bytes of the extraData structure.
extraData	Extra data used to store the WAVE_FORMAT_EXTENSIBLE information. Not to be used in BW64.

The FormatChunk is already the specialised format chunk for PCM audio data.

The extraData array in FormatChunk is used when the formatTag is set to 0XFFFE (WAVE_FORMAT_EXTENSIBLE). As multichannel audio should be described using ADM metadata, the use of this formatTag should be avoided. However, it should be possible that implementations are able to deal with reading a file containing this formatTag and handling it in a sensible manner.

To ensure the FormatChunk does not contradict with the <chna> and <axml> chunk information, it is recommended to set formatTag of 0x0001 for PCM audio, and 0x0000 (formatTag = unknown) for all other non-PCM audio.

2.6.3 Elements of the <data> chunk

The <data> chunk is for storing the audio samples.

Field	Description
chunkId	This is the 4 character array {‘d’, ‘a’, ‘t’, ‘a’} used for chunk identification.
chunkSize	4 byte value of the size of the chunk.
waveData	This is where the audio samples are stored. The samples are stored in little-endian byte order. Multiple tracks are stored by interleaving on a sample-by-sample basis. For example, for 16-bit 2-track audio:

Byte	Sample	Track
0	0 – LSB	1
1	0 – MSB	1
2	0 – LSB	2
3	0 – MSB	2
4	1 – LSB	1
5	1 – MSB	1
6	1 – LSB	2
7	1 – MSB	2

3 BW64 top level chunk

3.1 Definition

The <BW64> top level chunk is used instead of the <RIFF> chunk used in 32-bit sized files. By reading this chunk it means a <ds64> chunk should exist for reading the 64-bit sizes. The <BW64> chunk is shown below:

```
struct BW64Chunk            // declare BW64Chunk structure
{
    CHAR chunkId[4];        // 'BW64'
    DWORD chunkSize;        // -1 = 0xFFFFFFFF means don't use this data, use
                            // riffSizeHigh and riffSizeLow in 'ds64' chunk instead
    CHAR BW64Type[4];       // 'WAVE'
};
```

3.2 Elements of the <BW64> chunk

Field	Description
chunkId	This is the 4 character array {‘b’, ‘w’, ‘6’, ‘4’} used for chunk identification.

chunkSize 4 byte value that should be set to -1 (0xFFFFFFFF) to indicate that this size value is not used and the <ds64> chunk should be used for determining sizes.

BW64Type This is the 4 character array {‘W’, ‘A’, ‘V’, ‘E’} indicates that the file is a WAVE-type audio file.

4 DS64 and JUNK chunks

4.1 Definitions

The <ds64> chunk carries 64-bit size values for the file size, <data> chunk and an array of 64-bit size values of other definable chunks. The structure for the <ds64> chunk is shown below, followed by the structure for **ChunkSize64** table that carries the sizes for definable chunks (other than <data>). The empty array brackets indicate a variable number of elements can be used (including zero).

```
struct DataSize64Chunk      // declare DataSize64Chunk structure
{
    CHAR chunkId[4];        // ‘ds64’, FOURCC chunk identifier
    DWORD chunkSize;        // 4 byte size of the <ds64> chunk
    DWORD bw64SizeLow;      // low 4 byte size of <BW64> block
    DWORD bw64SizeHigh;     // high 4 byte size of <BW64> block
    DWORD dataSizeLow;      // low 4 byte size of <data> chunk
    DWORD dataSizeHigh;     // high 4 byte size of <data> chunk
    DWORD dummyLow;         // dummy value for cross compatibility
    DWORD dummyHigh;        // dummy value for cross compatibility
    DWORD tableLength;      // number of valid entries in array “table”
    ChunkSize64 table[ ];   // array of chunk sizes for chunks exceeding 4 Gbytes
};

struct ChunkSize64          // declare ChunkSize64 structure
{
    CHAR chunkId[4];        // chunkID of chunk which needs 64bit addressing;
                            // e.g. ‘axml’ is used when <axml> chunk exceeds 4 Gbytes
    DWORD chunkSizeLow;     // low 4 byte chunk size
    DWORD chunkSizeHigh;    // high 4 byte chunk size
};
```

The <JUNK> chunk is a placeholder for the <ds64> chunk that is used if a 32-bit sized audio file is being generated that may need converting on-the-fly into a 64-bit sized file later. The size of <JUNK> must match the size of the potential <ds64> chunk that will replace it. The structure of the chunk is shown here:

```
struct JunkChunk             // declare JunkChunk structure
{
    CHAR   chunkId[4];       // ‘JUNK’
```

```
    DWORD  chunkSize;       // 4 byte size of the 'JUNK' chunk. This must be at
                            // least 28 if the chunk is intended as a place-holder
                            // for a 'ds64' chunk.
    CHAR   chunkData[4];    // dummy bytes
};
```

4.2 Elements of the <ds64> chunk

Field	Description
chunkId	This is the 4 character array {'d', 's', '6', '4'} used for chunk identification.
chunkSize	4 byte size of the <ds64> chunk.
bw64SizeLow	This is the low 4 byte size of the <BW64> block. The 64-bit data size is expressed as 0xHHHHLLLL if <bw64SizeLow> and <bw64SizeHigh> are 0xLLLL and 0xHHHH, respectively. The 32-bit unsigned quantity is in little-endian format.
bw64SizeHigh	This is the high 4 byte size of the <BW64> block. The 32-bit unsigned quantity is in little-endian format.
dataSizeLow	This is the low 4 byte size of the <data> chunk. The 64-bit data size is expressed as 0xHHHHLLLL if <dataSizeLow> and <dataSizeHigh> are 0xLLLL and 0xHHHH, respectively. The 32-bit unsigned quantity is in little-endian format.
dataSizeHigh	This is the high 4 byte size of the <data> chunk. The 32-bit unsigned quantity is in little-endian format.
dummyLow	This is a 4 byte dummy value that should be ignored when read, and set to zero when writing. It exists to ensure compatibility with the EBU Tech 3306 RF64 specification, which uses this value to carry size information about the <fact> chunk that does not exist in the BW64 format.
dummyHigh	This is a 4 byte dummy value that should be ignored when read, and set to zero when writing. Its purpose is the same as <dummyLow>.
tableLength	This is the number of valid entries in the array "ChunkSize64 table"
ChunkSize64 table	This is the array of chunk sizes for chunks exceeding 4 Gbytes.

The **ChunkSize64** table is specified as follows. An array of **ChunkSize64** structs is used to store the length of any chunk other than <data> in the optional part of the <ds64> chunk. Currently, the only chunk type other than <data> is likely to exceed a size of 4 Gbytes would be the <axml> chunk (possible in extremely large object-based audio files).

Field	Description
chunkId	This 4 character array is used to refer to <chunkID> of the chunk which needs 64-bit addressing. For example, the 4 character array {'a', 'x', 'm', 'l'} is used for the <axml> chunk.
chunkSizeLow	This is the low 4 byte size of the chunk referring to <chunkID>. The 32-bit unsigned quantity is in little-endian format.

chunkSizeHigh	This is the high 4 byte size of the chunk referring to <chunkID>. The 32-bit unsigned quantity is in little-endian format.

4.3 Elements of the <JUNK> chunk

Field	Description
chunkId	This is the 4 character array {‘J’, ‘U’, ‘N’, ‘K’} used for chunk identification.
chunkSize	4 byte size of the <JUNK> chunk. Must be at least 28 to be a placeholder the for <ds64> chunk.
chunkData	Dummy data to be ignored.

5 AXML chunk

5.1 Definition

The <axml> chunk may contain any data compliant with the XML 1.0 format or later, a widespread format for data exchange [1]. Note that the <axml> chunk may contain XML fragments from more than one Schema. It may occur in any order with the other RIFF chunks within the same file.

The <axml> chunk consists of a header followed by data compliant with the XML format. The overall length of the chunk is not fixed.

See § 8 for an example on how the <axml> chunk in BW64 can be used to carry broadcast metadata, including the parameters in the former <bext> and <ubxt> chunks.

```
struct axml_chunk
{
    CHAR    ckID[4];        // {'a','x','m','l'}
    DWORD   ckSize;         // size of the <axml> chunk in bytes
    CHAR    xmlData[ ];     // text data in XML
};
```

As the XML may take up more than 4Gbytes it might be necessary to use the <ds64> chunk to allow a 64-bit size field for the <axml> chunk. Below is some pseudo-code to illustrate how this can be achieved using the table array in the <ds64> chunk.

```
DataSize64Chunk.tableLength = 1;   // number of valid entries in array "table"
DataSize64Chunk.table[0] = {
    ChunkSize64.chunkId = {`a`, `x`, `m`, `l`};   // chunk ID for the <axml> chunk
     chunkSizeLow = xxxx     // low 4 byte chunk size
     chunkSizeHigh = xxxx    // high 4 byte chunk size
}
```

5.2 Elements of the <axml> chunk

ckID	This is the 4 character array {‘a’, ‘x‘, ‘m‘, ‘l‘} used for chunk identification.

ckSize This is the size of the data section of the chunk in bytes. (It does not include the 8 bytes used by ckID and ckSize.)

xmlData This field contains the text information in XML.

The XML data structure is hierarchical and data are stored in text strings according to XML 1.0 format or later.

If the receiving device cannot interpret the content of the <axml> chunk in accordance with the specification stated in the XML, the entire chunk shall be ignored.

6 CHNA chunk

6.1 Definition

The <chna> chunk is a chunk that is specifically defined for the use with ADM as defined in Recommendation ITU-R BS.2076. The <chna> chunk consists of a header followed by the number of tracks and number of track UIDs used. This is followed by an array of ID structures that each contains IDs corresponding to ADM element IDs.

The size of the chunk depends upon the number of track UIDs to be defined. The number of ID structures must be equal to or greater than the number of track UIDs used. By allowing the number of ID structures to exceed the number of UIDs, it can facilitate updating and adding new IDs to the chunk without having to change the size of the chunk. For example, it may not be clear how many UIDs will be generated at the beginning, so if the number of ID structures in the chunk is set to 64 (as this is considered by the implementer to be more than enough for their task); the software then generates 55 UIDs (an example number of initial UIDs) which fill up the first 55 ID structures, so the remaining 9 ID structures are set to zero values.

The ADM IDs within the chunk can either refer to ADM metadata carried in the <axml> chunk, or in an external common definition file. If the last four hexadecimal digits of the IDs are value of 0x0FFF and below then they are defined as common definitions in Recommendation ITU-R BS.2094-0 ‐ *Common Definitions for the Audio Definition Model* (for example channel definitions for ‘FrontLeft’ and ‘FrontRight’). Any IDs with values of 0x1 000 and above are defined as custom definitions, so will be contained in the <axml> chunk within the file.

The audioID structure contains an index to the track used in the <data> chunk (which contains the audio samples), starting with the value of 1 for the first track. It contains a UID for the track, which the ADM metadata will contain. The audio elements of a track may be differently in the course of a file; in this case, there will be a different UID for each definition. Therefore it is possible to have multiple UIDs for each track. The other two values in the structure are references to the IDs of the ADM’s audioTrackFormat and audioPackFormat elements.

```
struct chna_chunk
{
    CHAR     ckID[4];         // {'c','h','n','a'}
    DWORD    ckSize;          // size of the <chna> chunk
    WORD     numTracks;       // number of tracks used
    WORD     numUIDs;         // number of track UIDs used
    audioID  ID[N];           // IDs for each track  (where N >= numUIDs)
};
```

```
struct audioID
{
    WORD    trackIndex;     // index of track in file
    CHAR    UID[12];        // audioTrackUID value
    CHAR    trackRef[14];   // audioTrackFormatID reference
    CHAR    packRef[11];    // audioPackFormatID reference
    CHAR    pad;            // padding byte to ensure even number of bytes
}
```

6.2 Elements of the <chna> chunk

ckID This is the 4 character array {'c','h','n','a'}[1] for chunk identification.

ckSize This is the size of the data section of the chunk in bytes. (It does not include the 8 bytes used by ckID and ckSize.)

numTracks The number of tracks used in the file. Even if a track contains more than one set of IDs, it is still just one track.

numUIDs The number of UIDs used in the file. As it is possible to give a single track multiple UIDs (covering different time periods), this could be a greater value than **numTracks**. This value should match the number of defined IDs in **ID**.

ID The structure containing the set of audio reference IDs for the track. This array contains N IDs, where N >= numUIDs. When numUIDs is less than N the contents of the unused track IDs are set to zero. When reading the chunk the value of N can be derived from ckSize, as ckSize = 4 + (N * 40), so N = (ckSize – 4) / 40.

trackIndex The index of the track in the file, starting at 1. This corresponds directly to the order of the tracks interlaced in the <data> chunk.

UID The audioTrackUID value of the track. The character array has the format ATU_xxxxxxxx where x is a hexadecimal digit.

trackRef The audioTrackFormatID reference of the track. The character array has the format AT_xxxxxxxx_xx where x is a hexadecimal digit.

packRef The audioPackFormatID reference of the track. The character array has the format AP_xxxxxxxx where x is a hexadecimal digit. When audioPackFormatID is not required (when audioStreamFormat is referring to an audioPackFormat rather than an audioChannelFormat) this field should be filled with null values.

pad A single byte to ensure the audioID structure has an even number of bytes.

When an **ID** is not being used the **trackIndex** should be given the value of zero and the other fields should be given null strings that are the same length as the usual ID string used. So the null string for packRef would consist of 11 null characters (ASCII value zero) and trackRef would consist of 14 null characters.

6.3 Informative Examples

To help illustrate the operation of the <chna> chunk some simple examples are described here. The pseudo-code in each example uses the string-like notation for the IDs (e.g. "AT_00010001_01"),

---

[1] **Remark:** The definition DWORD ckID = "chna" would not be unique. Different architectures produce different orders of the characters. Therefore we define char ckID[4] = {'c','h','n','a'} instead.

where in practice an array of characters should be used to ensure correct ordering of the characters (so it would be actually done this way: {'A','T','_','0','0','0','1','0','0','0','1','_','0','1'}).

6.3.1 Simple stereo file

The majority of audio files in existence are still 2-channel stereo files, with the first track containing the left channel, and the second track containing the right channel. The ADM has a definition of a left channel with an ID of AT_00010001_01, and the right channel with an ID of AT_00010002_01. The stereo pack definition has the ID of AP_00010002.

The pseudo-code is shown below:

```
ckID = {'c','h','n','a'};
ckSize = 84;
numTracks = 2;
numUIDs = 2;
ID[0]={ trackIndex=1; UID="ATU_00000001"; trackRef="AT_00010001_01"; packRef="AP_00010002"; pad='\0`; };
ID[1]={ trackIndex=2; UID="ATU_00000002"; trackRef="AT_00010002_01"; packRef="AP_00010002"; pad='\0`; };
```

The number of ID structures is 2, so there are no unused ID structures in this example.

6.3.2 Simple object-based example

Audio objects may only cover a sub-section of time in the audio file. To save space, non-overlapping objects may share the same track. This is where multiple UIDs in the same track would occur. This example also uses more ID structures (32 in this case) than numUIDs to show how unused ID structures are set to zero.

```
ckID = {'c','h','n','a'};
ckSize = 1284;
numTracks = 2;
numUIDs = 4;
ID[0]={ trackIndex=1; UID="ATU_00000001"; trackRef="AT_00031001_01"; packRef="AP_00031001"; pad='\0`; };
ID[1]={ trackIndex=1; UID="ATU_00000002"; trackRef="AT_00031003_01"; packRef="AP_00031002"; pad='\0`; };
ID[2]={ trackIndex=1; UID="ATU_00000003"; trackRef="AT_00031004_01"; packRef="AP_00031003"; pad='\0`; };
ID[3]={ trackIndex=2; UID="ATU_00000004"; trackRef="AT_00031002_01"; packRef="AP_00031001"; pad='\0`; };
ID[4]={ trackIndex=0; UID=['\0']*12;      trackRef=['\0']*14;        packRef=['\0']*11;     pad='\0`; };
   :
ID[31]={ trackIndex=0; UID=['\0']*12;      trackRef=['\0']*14;        packRef=['\0']*11;     pad='\0`; };
```

The first track contains 3 UIDs, so will contain 3 different objects (with the track IDs of AT_00031001_01, AT_00031003_01 and AT_00031004_01) at different time locations within the file. The second track contains one UID, so contains one object. This object has the same pack ID (AP_00031001) as the first object in track 1. This suggests the first object contains two channels carried in both track 1 and track 2. The ADM metadata carried in the <axml> would be used to clarify the allocation of channels and tracks.

6.3.3 Multi-content example

The BW64 file could contain multiple content in a single file, such as a main 5.1 mix on the first 6 tracks, with a foreign language stereo mix on the next 2 tracks. Recommendation ITU-R BS.1738 contains several configurations, and the example will show how Production Scenario 5 from that Recommendation can be dealt with in the <chna> chunk. This scenario contains 8 tracks, the first 6 contain a 5.1 complete mix, and the second 2 tracks contain a stereo international mix. The resulting <chna> is shown below:

```
ckID = {'c','h','n','a'};
ckSize = 324;
numTracks = 8;
numUIDs = 8;
ID[0]={ trackIndex=1; UID="ATU_00000001"; trackRef="AT_00010001_01"; packRef="AP_00010003"; pad='\0`; };
ID[1]={ trackIndex=2; UID="ATU_00000002"; trackRef="AT_00010002_01"; packRef="AP_00010003"; pad='\0`; };
ID[1]={ trackIndex=3; UID="ATU_00000003"; trackRef="AT_00010003_01"; packRef="AP_00010003"; pad='\0`; };
ID[1]={ trackIndex=4; UID="ATU_00000004"; trackRef="AT_00010004_01"; packRef="AP_00010003"; pad='\0`; };
ID[1]={ trackIndex=5; UID="ATU_00000005"; trackRef="AT_00010005_01"; packRef="AP_00010003"; pad='\0`; };
ID[1]={ trackIndex=6; UID="ATU_00000006"; trackRef="AT_00010006_01"; packRef="AP_00010003"; pad='\0`; };
ID[1]={ trackIndex=7; UID="ATU_00000007"; trackRef="AT_00010001_01"; packRef="AP_00010002"; pad='\0`; };
ID[1]={ trackIndex=8; UID="ATU_00000008"; trackRef="AT_00010002_01"; packRef="AP_00010002"; pad='\0`; };
```

The ADM metadata in the <axml> chunk will contain information on how the two mixes are split.

7 Compatibility with Recommendation ITU-R BS.1352

As the BWF format (Recommendation ITU-R BS.1352) is the short-form RIFF/WAVE file format (as described in Annex 2) with extra chunks, most notably the <bext> chunk, there is a need to understand the compatibility between BWF and BW64.

BWF chunks	BW64 chunks	How to handle
<fmt>	<fmt>	Use conventionally
<data>	<data>	Use conventionally
<fact>	<fact>	Use conventionally [, though it is probably redundant so could be omitted].
-	<ds64>	See § 2.4
-	<JUNK>	See § 2.4
-	<chna>	See § 4.4
-	<axml>	See § 4.4 for channel allocations. Use for broadcast metadata that would exists in <bext> chunk.
<bext>	-	If reading a <bext> chunk, convert to corresponding <axml> chunk data to carry ADM and any other broadcast related XML metadata. See § 8 for more details.

8 Generating XML Broadcast Metadata

Recommendation ITU-R BS.1352 carries broadcast metadata in the <bext> and <ubxt> chunks. These chunks have fixed length fields and are limited to the specified fields, thus preventing any other broadcast related metadata being carried. The <axml> chunk in BW64 can carry any XML metadata, so can be used to carry broadcast metadata, including the parameters in the <bext> and <ubxt> chunks.

To carry <bext>/<ubxt> parameters in the <axml> chunk the following XML structure should be used, where the comments prefixed by 'BEXT' indicate the <bext>/<ubxt> chunk parameters.

```
<?xml version="1.0" encoding="UTF-8"?>
<ebuCoreMain                      xmlns="urn:ebu:metadata-schema:ebuCore_2015"
xmlns:dc="http://purl.org/dc/elements/1.1/"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <coreMetadata>
    <creator>
      <contactDetails>
        <name>
          <!--BEXT: bextOriginator -->
        </name>
      </contactDetails>
      <organisationDetails>
        <organisationName>
          <!--BEXT: bextOriginatorReference -->
        </organisationName>
      </organisationDetails>
    </creator>

    <description typeDefinition="bextDescription">
      <dc:description>
        <!--BEXT: bextDescription -->
      </dc:description>
    </description>
    <date>
      <!--BEXT: bextOriginationDate and bextOriginationTime below-->
      <created startDate="2000-10-10" startTime="12:00:00"/>
    </date>

    <format>
      <audioFormatExtended>
        <!--BEXT: bextTimeReference below-->
        <audioProgramme audioProgrammeID="..." start="00:00:00:00">
          <!--Other audioProgramme metadata here -->
        </audioProgramme>
        <!--Other ITU-R BS.2076 ADM metadata here -->
      </audioFormatExtended>
      <technicalAttributeString typeDefinition="CodingHistory">
        <!--BEXT: bextCodingHistory -->
      </technicalAttributeString>
    </format>

    <identifier                                          formatLabel="UMID"
formatLink="http://www.ebu.ch/metadata/cs/ebu_IdentifierTypeCodeCS.xml#1.1">
      <dc:identifier>
        <!--BEXT: bextUMID-->
      </dc:identifier>
    </identifier>
  </coreMetadata>
</ebuCoreMain>
```

The XML is based on the EBUCore [2] and AESCore [3] metadata schemas, which are compatible with Recommendation ITU-R BS.2076.

When reading an ITU-R BS.1352 BWF file with the intention of converting it to a BW64 file, the <bext>/<ubxt> chunks should be converted to the XML described here for inclusion within the <axml> chunk.

9 File Extension of the BW64 format file

The file extension of the files conforming to the BW64 format is defined as “.wav”. This allows legacy software to be able to read the chunks in the file that it understands (primarily <fmt > and <data>), so that at least the audio samples can be accessed.

While it is not recommended to use any alternative file extensions when generating BW64 files, it could be anticipated that a “.bw64” extension may be inappropriately used. Therefore software that reads BW64 file should be tolerant to this alternative file extension.

10 Bibliography

[1] Extensible Markup Language (XML) 1.0 W3C Recommendation 26-November-2008 http://www.w3.org/TR/2008/REC-xml-20081126

[2] EBU Tech 3293, “EBU Core Metadata Set v.1.6”.

[3] AES 60-2011, “AES standard for audio metadata – Core audio metadata”.

Annex 2 (informative)

RIFF WAVE (.WAV) file format

The information in this Annex is taken from the specification documents of the RIFF file format. It is included for information only. It is included due to the lack of a reliable external source for referencing.

1 Waveform audio file format (WAVE)

The WAVE format is defined as follows. Programs must expect (and ignore) any unknown chunks encountered, as with all RIFF forms. However, <fmt-ck> must always occur before <wave-data>, and both of these chunks are mandatory in a WAVE file.

```
<WAVE-form> ->
    RIFF('WAVE'
        <fmt-ck>    // Format chunk
        [<fact-ck>]   // Fact chunk
        [<other-ck>]  // Other optional chunks
        <wave-data>)  // Sound data
```

The WAVE chunks are described in the following sections:

1.1 WAVE format chunk

The WAVE format chunk <fmt-ck> specifies the format of the <wave-data>. The <fmt-ck> is defined as follows:

```
<fmt-ck> ->fmt(<common-fields>
               <format-specific-fields>)
```

```
<common-fields> ->
    Struct {
        WORD  wFormatTag;           // Format category
        WORD  nChannels;            // Number of channels
        DWORD nSamplesPerSec;       // Sampling rate
        DWORD nAvgBytesPerSec;      // For buffer estimation
        WORD  nBlockAlign;          // Data block size
    }
```

The fields in the <common-fields> portion of the chunk are as follows:

Field	Description
wFormatTag	A number indicating the WAVE format category of the file. The content of the <format-specific-fields> portion of the <fmt-ck> and the interpretation of the waveform data, depend on this value.
nchannels	The number of channels represented in the waveform data, such as 1 for mono or 2 for stereo.
nSamplesPerSec	The sampling rate (in samples per second) at which each channel should be reproduced.
nAvgBytesPerSec	The average number of bytes per second at which the waveform data should be transferred. Playback software can estimate the buffer size using this value.
nBlockAlign	The block alignment (in bytes) of the waveform data. Playback software needs to process a multiple of <nBlockAlign> bytes of data at a time, so the value of <nBlockAlign> can be used for buffer alignment.

The <format-specific-fields> consists of zero or more bytes of parameters. Which parameters occur depends on the WAVE format category – see the following sections for details. Playback software should be written to allow for (and ignore) any unknown <format-specific-fields> parameters that occur at the end of this field.

1.2 WAVE format categories

The format category of a WAVE file is specified by the value of the <wFormatTag> field of the <fmt> chunk. The representation of data in <wave-data>, and the content of the <format-specific-fields> of the <fmt> chunk, depend on the format category.

Among the currently defined open non-proprietary WAVE format categories are as follows:

wFormatTag	Value	Format Category
WAVE_FORMAT_UNKNOWN	0x0000	unknown
WAVE_FORMAT_PCM	0x0001	PCM format
WAVE_FORMAT_IEEE_FLOAT	0x0003	IEEE float
WAVE_FORMAT_EXTENSIBLE	0xFFFE	Wave Format Extensible – determined by SubFormat

NOTE – Only the WAVE_FORMAT_PCM and WAVE_FORMAT_UNKNOWN formats are used at present with the BW64. Details of the PCM WAVE format are given in the following § 2. General information on other WAVE formats is given in § 3. Other WAVE formats may be defined in future.

In the past WAVE_FORMAT_EXTENSIBLE would have been used for multichannel files, but that should be avoided in the future.

1.3 Fact chunk

The <fact-ck> stores file dependent information about the contents of non-PCM WAVE files. Therefore, this chunk is not used in this version of the BW64 format. This chunk is defined as follows:

```
<fact-ck> ->  fact( <dwSampleLength:DWORD> )
```

<dwSampleLength> represents the length of the data in samples. The <nSamplesPerSec> field from the wave format header is used in conjunction with the <dwSampleLength> field to determine the length of the data in seconds.

The fact chunk is required for all new non-PCM WAVE formats. The chunk is not required for the standard WAVE_FORMAT_PCM files.

The fact chunk will be expanded to include any other information required by future WAVE formats. Added fields will appear following the <dwSampleLength> field. Applications can use the chunk size field to determine which fields are present.

1.4 Other optional chunks

A number of other chunks are specified for use in the WAVE format. Details of these chunks are given in the specifications of the WAVE format and in any updates to be issued later.

NOTE – The WAVE format can support other optional chunks that can be included in WAVE files to carry specific information. These are considered to be private chunks and will be ignored by applications that cannot interpret them.

2 PCM format

If the <wFormatTag> field of the <fmt-ck> is set to WAVE_FORMAT_PCM, then the waveform data consists of samples represented in PCM format. For PCM waveform data, the <format-specific-fields> is defined as follows:

```
<PCM-format-specific> ->
    struct {
        WORD  nBitsPerSample;    // Sample size
    }
```

The <nBitsPerSample> field specifies the number of bits of data used to represent each sample of each channel. If there are multiple channels, the sample size is the same for each channel.

The <nBlockAlign> field should be equal to the following formula, rounded to the next whole number

nChannels × BytesPerSample

The value of BytesPerSample should be calculated by rounding up nBitsPerSample to the next whole byte. Where the audio sample word is less than an integer number of bytes, the most significant bits of the audio sample are placed in the most significant bits of the data word, the unused data bits adjacent to the least significant bit should be set to zero

For PCM data, the <nAvgBytesPerSec> field of the <fmt> chunk should be equal to the following formula.

nSamplesPerSec × nBblockAlign

NOTE 1 – The original WAVE specification permits, for example 20-bit samples from two channels to be packed into 5 bytes-sharing a single byte for the least significant bits of the two channels. This Recommendation specifies a whole number of bytes per audio sample in order to reduce ambiguity in implementations and to achieve maximum interchange compatibility.

2.1 Data packing for PCM WAVE files

In a single-channel WAVE file, samples are stored consecutively. For stereo WAVE files, channel 0 represents the left-hand channel, and channel 1 represents the right-hand channel. In multiple-channel WAVE files, samples are interleaved.

The following diagrams show the data packing for 8-bit mono and stereo WAVE files:

Data packing for 8-bit mono PCM

Sample 1	Sample 2	Sample 3	Sample 4
Channel 0	Channel 0	Channel 0	Channel 0

Data packing for 8-bit stereo PCM

Sample 1		Sample 2	
Channel 0 (left)	Channel 1 (right)	Channel 0 (left)	Channel 1 (right)

The following diagrams show the data packing for 16-bit mono and stereo WAVE files:

Data packing for 16-bit mono PCM

Sample 1		Sample 2	
Channel 0 low-order byte	Channel 0 high-order byte	Channel 0 low-order byte	Channel 0 high-order byte

Data packing for 16-bit stereo PCM

Sample 1			
Channel 0 (left)	Channel 0 (left)	Channel 1 (right)	Channel 1 (right)
low-order byte	high-order byte	low-order byte	high-order byte

2.2 Data format of the samples

Each sample is contained in an integer i. The size of i is the smallest number of bytes required to contain the specified sample size. The least significant byte is stored first. The bits that represent the sample amplitude are stored in the most significant bits of i, and the remaining bits are set to zero.

For example, if the sample size (recorded in <nBitsPerSample>) is 12 bits, then each sample is stored in a two-byte integer. The least significant four bits of the first (least significant) byte are set to zero. The data format and maximum and minimum values for PCM waveform samples of various sizes are as follows:

Sample size	Data format	Maximum value	Minimum value
One to eight bits	Unsigned integer	255 (0xFF)	0
Nine or more bits	Signed integer i	Largest positive value of i	Most negative value of i

For example, the maximum, minimum, and midpoint values for 8-bit and 16-bit PCM waveform data are as follows:

Format	Maximum value	Minimum value	Midpoint value
8-bit PCM	255 (0xFF)	0	128 (0x80)
16-bit PCM	32767 (0x7FFF)	–32768 (–0x8000)	0

2.3 Examples of PCM WAVE files

Example of a PCM WAVE file with 11.025 kHz sampling rate, mono, 8 bits per sample:

```
RIFF('WAVE' fmt(1, 1, 11025, 11025, 1, 8)
          data(<wave-data>) )
```

Example of a PCM WAVE file with 22.05 kHz sampling rate, stereo, 8 bits per sample:

```
RIFF('WAVE' fmt(1, 2, 22050, 44100, 2, 8)
          data(<wave-data>) )
```

2.4 Storage of WAVE data

The <**wave-data**> contains the waveform data. It is defined as follows:

```
<wave-data> ->  { <data-ck> }
    <data-ck> -> data( <wave-data> )
```

2.5 Fact chunk

The <fact-ck> fact chunk stores important information about the contents of the WAVE file. This chunk is defined as follows:

```
<fact-ck> ->   fact(<dwFileSize:DWORD>)   // Number of samples
```

The chunk is not required for PCM files.

The fact chunk will be expanded to include any other information required by future WAVE formats. Added fields will appear following the <dwFileSize> field. Applications can use the chunk size field to determine which fields are present.

2.6 Other optional chunks

A number of other chunks are specified for use in the WAVE format. Details of these chunks are given in the specification of the WAVE format and any updates issued later.

NOTE 1 – The WAVE format can support other optional chunks that can be included in WAVE files to carry specific information. These are considered to be private chunks and will be ignored by applications that cannot interpret them.

3 WAVE format extension

The extended wave format structure added to the <fmt-ck> is used to define all non-PCM format wave data, and is described as follows. The general extended waveform format structure is used for all non PCM formats.

```
typedef struct waveformat_extended_tag {
    WORD  wFormatTag;      // format type
    WORD  nChannels;       // number of channels (i.e. mono, stereo...)
    DWORD nSamplesPerSec;  // sample rate
    DWORD nAvgBytesPerSec; // for buffer estimation
    WORD  nBlockAlign;     // block size of data
    WORD  wBitsPerSample;  // number of bits per sample of mono data
    WORD  cbSize;          // the count in bytes of the extra size
} WAVEFORMATEX;
```

Field	Description
wFormatTag	Defines the type of WAVE file.
nChannels	Number of channels in the wave, 1 for mono, 2 for stereo.
nSamplesPerSec	Frequency of the sample rate of the wave file. This should be 48000 or 44100 etc. This rate is also used by the sample size entry in the fact chunk to determine the duration of the data.
nAvgBytesPerSec	Average data rate. Playback software can estimate the buffer size using the <nAvgBytesPerSec> value.
nBlockAlign	The block alignment (in bytes) of the data in <data-ck>. Playback software needs to process a multiple of <nBlockAlign> bytes of data at a time, so that the value of <nBlockAlign> can be used for buffer alignment
wBitsPerSample	This is the number of bits per sample per channel. Each channel is assumed to have the same sample resolution. If this field is not needed, then it should be set to zero.
cbSize	The size in bytes of the extra information in the WAVE format header not including the size of the WAVEFORMATEX structure.

NOTE – The fields following the <cbSize> field contain specific information needed for the WAVE format defined in the field <wFormatTag>.

Annex 3 (normative)

Definitions of primitive data types

The following are atomic labels, which are labels that refer to primitive data types. The equivalent C data type is also listed.

Label	Meaning	C Type
<CHAR>	8-bit signed integer	signed char
<BYTE>	8-bit unsigned integer	unsigned char
<INT>	16-bit signed integer in little-endian format	signed int
<WORD>	16-bit unsigned quantity in little-endian format	unsigned int
<LONG>	32-bit signed integer in little-endian format	signed long
<DWORD>	32-bit unsigned quantity in little-endian format	unsigned long
<FLOAT>	32-bit IEEE floating point number	Float
<DOUBLE>	64-bit IEEE floating point number	Double
<STR>	String (a sequence of characters)	
<ZSTR>	NULL-terminated string	
<BSTR>	String with byte (8-bit) size prefix	
<WSTR>	String with word (16-bit) size prefix	
<BZSTR>	NULL-terminated string with byte size prefix	