GY

中华人民共和国广播电影电视行业标准

GY/T 281-2014

音频扩展文件格式 MBWF/RF64

An extended file format for audio MBWF/RF64

2014-12-03 发布　　2014-12-03 实施

国家新闻出版广电总局　发布

目　　次

前　言

本标准按照GB/T 1.1-2009给出的规则起草。

本标准依据EBU-TECH 3306：2009《MBWF/RF64：音频扩展文件格式》起草。

本标准由全国广播电影电视标准化技术委员会（SAC/TC 239）归口。

本标准起草单位：国家新闻出版广电总局广播电视规划院。

本标准主要起草人：张建东、邓向冬、肖辉。

音频扩展文件格式 MBWF/RF64

1 范围

本标准规定了专业广播领域内的一种超过4G字节大小的数字音频文件格式要求。

本标准适用于数字音频文件的录制、归档和交换。

2 术语、定义和缩略语

2.1 术语和定义

下列术语和定义适用于本标准。

2.1.1

环绕声数字音频编码 audio coding generation 3；AC-3

在ETSI TS 102 366标准中定义的数字音频压缩编解码标准。

2.1.2

广播音频扩展 broadcast audio extension；bext

一种音频数据块，用于定义音频资料文件交换所需的基本参数，如创作者姓名、创作日期等。

2.1.3

MPEG-2 AAC

在ISO/IEC 13818-7标准中定义的数字音频压缩编解码标准。

2.1.4

数字影院系统 digital theater systems；DTS

一种数字音频压缩编解码系统。

2.1.5

Dolby E

一种数字音频压缩编解码系统，可用于音频信号的馈送和分配环节。

2.1.6

RF64

一种基于64位的资源交换文件格式。

2.2 缩略语

下列缩略语适用于本标准。

BWF 广播波形格式（Broadcast Wave Format）
GUID 全球唯一标识符（Globally Unique Identifier）
MBWF 多声道广播波形格式（Multichannel Broadcast Wave Format）
MPEG 运动图像专家组（Moving Picture Experts Group）
PCM 脉冲编码调制（Pulse Code Modulation）
RIFF 资源交换文件格式（Resource Interchange File Format）
WAVE 波形音频文件格式（Waveform audio file format）

3 概述

RF64 文件格式能够满足多声道声音节目在广播和归档应用中的长期需求。

RF64 文件格式在 RIFF/WAVE 文件基本规范的基础上进行了以下扩展：

- 文件大小超过 4G 字节；
- 扩展了 Wave Format Extensible 中多声道参数的定义，可承载最多至 18 个环绕声道、立体声下混声道和非 PCM 编码数据比特流信号。

RF64 文件格式作为 RIFF/WAVE 格式和 BWF 及其补充数据块的一种可兼容扩展格式，扩展了 RIFF/WAVE 和 BWF 的最大文件容量，以满足多声道声音节目广播和音频归档的应用需求。

RF64 可应用于从声音采集、编辑到播出的整个节目生产过程，以及多声道文件的短期或长期归档。

本标准中将包含bext块的RF64文件定义为MBWF（Multichannel BWF)文件，术语“RF64”与“MBWF”可视为同义。

4 基本用户需求

音频扩展文件格式应满足如下的基本用户需求：

- 文件格式应开放且基于已发布的规范；
- 保持与 BWF 和 RIFF/WAVE 的兼容；
- 支持线性 PCM 信号的存储；
- 支持超过 4G 字节的文件；
- 至少支持 8 声道信号存储（5.1+双声道立体声）；
- 适用于包含在单个文件中的 5.1 声道和双声道立体声信号的同步播出；
- 支持音频码流的存储；
- 适用于音频信号编辑；
- 适用于从文件中直接导出浏览版本（文件格式元数据）；
- 包含播放所需的技术元数据（如杜比元数据）；
- 适用于经济易用的软件播放器；
- 易于软件开发者和制造商实现。

此外，根据实际系统的运行经验，节目制作和归档中通常需要在单个文件中存储和传送PCM和/或非PCM音频数据，因此在RF64格式中增加了容纳非PCM音频流（如：AC-3和DTS）的机制。

5 RF64 文件格式定义

5.1 Wave Format Extensible 声道模板

Wave Format Extensible 的声道模板包含的前 18 个“#define”定义，用于指定不同的扬声器位置（声道分配），还有一个“#define”设置为“SPEAKER_ALL”，表示开启所有扬声器（声道）。Wave Format Extensible 声道模板的扬声器位置/声道分配定义及标识见表 1。

表1 Wave Format Extensible 声道模板

扬声器位置/声道分配定义	标识
#define SPEAKER_FRONT_LEFT	0x00000001
#define SPEAKER_FRONT_RIGHT	0x00000002
#define SPEAKER_FRONT_CENTER	0x00000004
#define SPEAKER_LOW_FREQUENCY	0x00000008
#define SPEAKER_BACK_LEFT	0x00000010
#define SPEAKER_BACK_RIGHT	0x00000020
#define SPEAKER_FRONT_LEFT_OF_CENTER	0x00000040
#define SPEAKER_FRONT_RIGHT_OF_CENTER	0x00000080
#define SPEAKER_BACK_CENTER	0x00000100
#define SPEAKER_SIDE_LEFT	0x00000200
#define SPEAKER_SIDE_RIGHT	0x00000400
#define SPEAKER_TOP_CENTER	0x00000800
#define SPEAKER_TOP_FRONT_LEFT	0x00001000
#define SPEAKER_TOP_FRONT_CENTER	0x00002000
#define SPEAKER_TOP_FRONT_RIGHT	0x00004000
#define SPEAKER_TOP_BACK_LEFT	0x00008000
#define SPEAKER_TOP_BACK_CENTER	0x00010000
#define SPEAKER_TOP_BACK_RIGHT	0x00020000
#define SPEAKER_ALL	0x80000000

为满足第 4 章列出的用户需求，RF64 需要对基本 Wave Format Extensible 的声道模板进行扩展。声道模板的标识使用一个 32 位变量存储，因此，还可定义另外 13 个“#define”来满足新增的用户需求。

5.2 PCM 双声道立体声下混定义扩展

Wave Format Extensible 中不包含 PCM 双声道立体声信号的定义，因此在声道模板中增加以下定义：

#define SPEAKER_STEREO_LEFT 0x20000000

#define SPEAKER_STEREO_RIGHT 0x40000000

使用此定义，可满足在单一文件中包含一个多声道“X.1”信号和一个双声道立体声下混信号。

5.3 控制数据定义扩展

增加了两个“#define”用于定义控制数据：

#define SPEAKER_CONTROLSAMPLE_1 0x08000000

#define SPEAKER_CONTROLSAMPLE_2 0x10000000

控制数据可以存储在文件中，技术或内容元数据定位的方法预留定义。

5.4 非 PCM 比特流数据定义扩展

符合 IEC 61937-1、IEC 61937-3、IEC 61937-5、IEC 61937-6、SMPTE 337M、SMPTE 338M、SMPTE 339M 和 SMPTE 340M 标准的比特流信号包含以各种方式编码的非 PCM 多声道音频数据。

通过文件格式中的比特流存储方式，AC-3、Dolby E、DTS、MPEG-1 和 MPEG-2（包括所有三个层）、MPEG-2 AAC 编码的音频数据，与线性 PCM 信号的存储方式类似，以数据帧的方式存储于文件中。文件格式中比特流音频信号的嵌入结构，类似于线性 PCM RIFF/WAVE 或 BWF 文件中两个交织立体声音频声道的结构。

RIFF 文件中仅有一个格式块，用于定义音频数据块中所有交织声道的通用参数。如果同一个文件中还存在其他 PCM 音频，则比特流声道格式必须遵从文件中其他多声道 PCM 或双声道立体声 PCM 的声道格式。

在文件的接收端，只有通过 AES3 或 SPDIF 接口解码比特流时，方可获知非 PCM 音频编码的类型，因此将来有必要增加一个新的块定义来指示文件中包含的比特流格式。

对 Wave Format Extensible 声道模版新增以下 4 个“#define”定义，用于承载两种非 PCM 格式：

```
#define SPEAKER_BITSTREAM_1_LEFT                0x00800000
#define SPEAKER_BITSTREAM_1_RIGHT               0x01000000
#define SPEAKER_BITSTREAM_2_LEFT                0x02000000
#define SPEAKER_BITSTREAM_2_RIGHT               0x04000000
```

5.5 支持超过 4G 字节的文件

4G 字节文件大小限制源于 RIFF/WAVE 和 BWF 格式中的 32 位寻址方式，使用 32 位的最大寻址范围为 4294967296 字节（即 4G 字节），为支持超过 4G 字节的文件，本标准使用 64 位寻址方式。

如果仅将 BWF 中每个字段的长度改为 64 位，显然将生成一个与标准 RIFF/WAVE 格式不兼容的文件。标准 RIFF/WAVE 文件格式见图 1，其中除 fmt 块数据和音频数据外，其余字段长度均为 32 位。

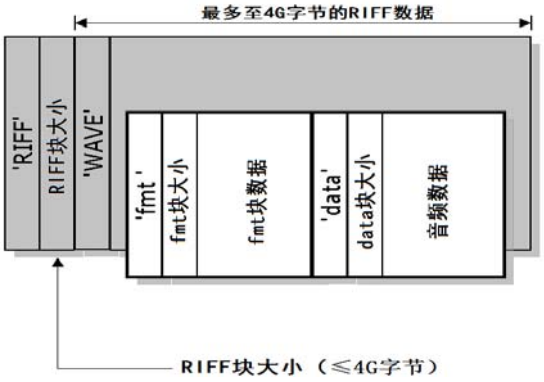

图1 标准 RIFF/WAVE 格式

因此，本标准定义了一种新的基于 64 位的资源交换文件格式(Resource Interchange File Format)，称为 RF64。RF64 与原有的 RIFF/WAVE 格式基本相同，不同之处如下所述：

- 文件起始的四个字节，以“RF64”标识替代“RIFF”；
- 文件中定义了一个必须包含的“ds64”（data size 64）块，该块须紧接在“RF64” 块之后。

“ds64”块包括三个必选的 64 位整型值，替代 RIFF/WAVE 格式中原有的三个 32 位字段：

- RIFF 块大小；
- data 块大小；

- fact 块中的样本计数值。

对 RIFF/WAVE 格式中上述三个 32 位字段应用以下替代规则：

如果字段的 32 位值不是“-1”（即十六进制的 0xFFFFFFFF），则使用该 32 位值；如果值为“-1”，则使用“ds64”块中的 64 位值。

一种支持 64 位块大小的可选 RF64 块结构体（struct[1)]）定义见附录 A 中的 A.1 和 A.2。

RF64/WAVE 文件格式的完整结构示例如图 2 所示。图 2 中除预留 64 位数据、fmt 块数据和音频数据字段外，其余字段均为 32 位。

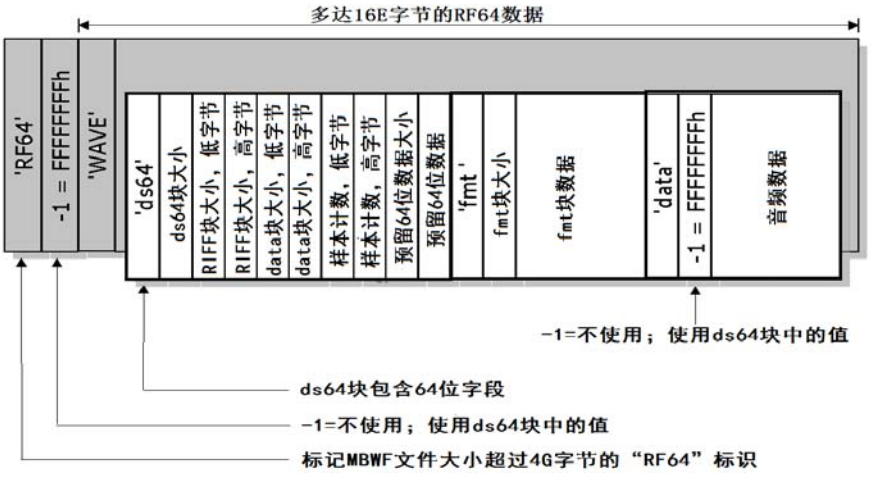

图2　支持多达 16E[2)]字节文件大小的 RF64/WAVE 文件格式的完整结构示例

5.6　BWF 和 RF64 间归档的兼容性

即使在节目录制中采用更高采样率和多声道音频，也不是所有音频文件都会超过 4G 字节，因此，对小于 4G 字节的音频文件仍应使用 BWF 格式存储。

由于录制程序无法预知在录音结束时文件大小是否会超过 4G 字节（即是否需要使用 RF64），因此录制程序需具备在所写入数据达到 4G 字节限制处将录制文件实时从 BWF 格式转换到 RF64 格式的功能，同时不影响录制的继续进行。

该项需求可以通过在 BWF 中插入与“ds64”块同样大小的“JUNK”[3)]块从而预留额外的空间来实现。预留的空间对 BWF 格式无实际意义，但当需要转换为 RF64 格式时，使用该空间承载“ds64”块。

RIFF/WAVE 文件格式的向上兼容性结构见图 3。图 3 中除 JUNK 块数据、fmt 块数据和音频数据字段外，其余字段均为 32 位。

1)　“struct”是 C/C++中用于定义结构体类型和/或结构体变量的关键字。

2)　E 字节即 exa byte（1E 字节=2^{60}字节）。

3)　“JUNK”块是 RIFF/WAVE 基本规范的一部分，作用为占位，音频应用程序将忽略该块。

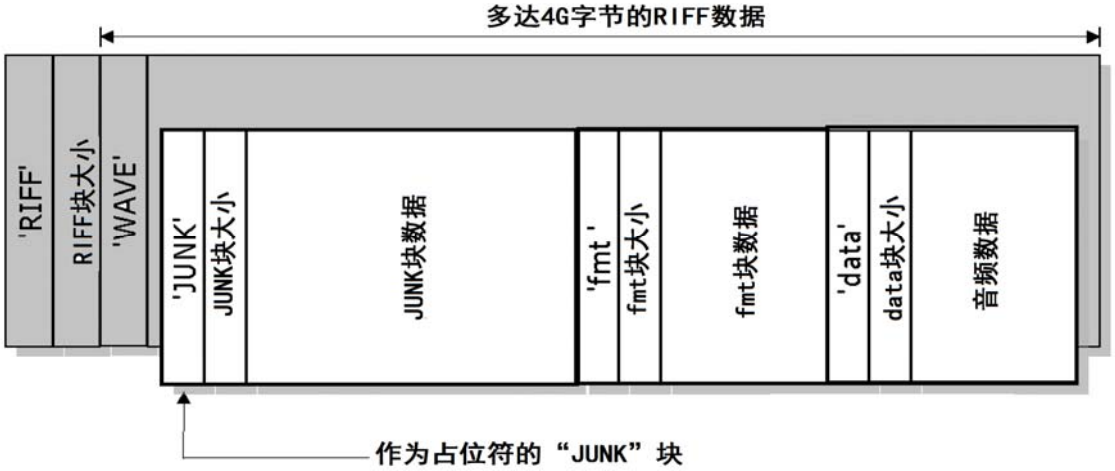

图3　RIFF/WAVE 文件格式的向上兼容性结构

支持 RF64 格式的应用程序在开始录制时，首先创建第一个块为“JUNK”块的标准 RIFF/WAVE 或 BWF 格式文件，录制中，应用程序检查 RIFF 和数据大小，当发现超过 4G 字节时，将执行以下操作：

- 以“ds64”替代“JUNK”块标识（将原来的 JUNK 块转换为 ds64 块）；
- 在“ds64”块中，按块定义格式插入 RIFF 块大小，data 块大小和样本计数值；
- 将原 32 位的 RIFF 块大小，data 块大小和样本计数值设为-1（0xFFFFFFFF）；
- 以“RF64”标识替代文件的最初 4 个字节 “RIFF”；
- 继续录制。

5.7　标记块的定义

实际应用中，发现 RIFF/WAVE 文件格式关于 cue 块的定义存在如下一些问题：

- 现行的 cue 块采用 32 位寻址方式，因此仅对 RF64 文件的前 4G 字节音频数据有效；
- cue 块的定义描述不明确，导致一些开发者在其应用程序中以不当的方式实现标记功能；
- 针对数据净荷是线性音频还是压缩数据这两种情况，软件开发者必须以不同的方式对标记进行处理，这将影响软件开发代码的简单性和准确性；
- 标记中的标签（labels）并未跟其一起存储于“cue”块，而是存储在“labl”块中，使对标记的处理更加复杂。

因此，本标准定义了一个新的 RF64 标记块（见附录 A.3 和 A.4），该块包含了标记的位置及其标签。

RF64 文件很大（通常超过 4G 字节），如果删除标记时无需对整个文件进行重新处理，将使 RF64 文件更易于使用。通过在 FLAG 字段中设置/重置一个比特位，从而将某个标记置为有效或无效来实现。

在开始写入 data 块前预留一些存放标记的空间，在录音和将音频数据写入 data 块的过程中，将标记也写入文件。

根据统计显示，大多数标记的标签只占用几个字符的长度，因此可使用一个固定长度的字段来承载标签，相对于动辄超过 4G 字节的 RF64 音频文件而言，由此引入的空间开销很少。例如，3000 个典型的标记占用的空间不超过 1M 字节，在一个 4G 字节的 RF64 文件中，10000 个标记占用的空间不超过文件大小的 0.1%。

最后，可以使用产品和/或供应商专用标记记录一些特定的信息，如颜色等。由于此信息只与特定供应商相关，可使用 GUID（全球唯一标识符）标识来区分，确保只有该供应商的应用程序使用此信息，而其他所有软件忽略该数据。此外，由于该信息仅对特定供应商应用程序有用，无需与其他供应商共享，各供应商可按自己的方式使用专用标记中的数据。

注：由于RIFF/WAVE或RF64文件有可能同时包含cue块和RF64标记块，因此强制规定应用程序首先在文件中查找RF64标记块，如找到RF64标记块，则仅使用RF64标记块中的标记信息（cue点），如未找到RF64标记块，则查找并使用“cue”块。

附 录 A
（规范性附录）
RIFF/WAVE 和 RF64/WAVE 结构的形式化描述

A.1 RIFF/WAVE（BWF）格式中的块和结构体[4)]

```
 struct RiffChunk                        // 声明 RiffChunk 结构体
{
     char              chunkId[4];       //  ‘RIFF’
     unsigned int32    chunkSize;        // 传统 RIFF/WAVE 文件的大小，4 字节
     char              riffType[4];      //  ‘WAVE’
};
 struct JunkChunk                        // 声明 JunkChunk 结构体
{
     char              chunkId[4];       //  ‘JUNK’
     unsigned int32    chunkSize;        // ‘JUNK’ 块大小，4 字节。如果意欲作为‘ds64’
                                         // 块的占位符，本成员变量的值至少为 28。
     char              chunkData[ ]5);   // 填充字节
};
 struct FormatChunk                      //声明 FormatChunk 结构体
{
     char              chunkId[4];       //  ‘fmt ’
     unsigned int32    chunkSize;        //  ‘fmt ’ 块大小，4 字节
     unsigned int16    formatType;       // WAVE_FORMAT_PCM = 0x0001，等等
     unsigned int16    channelCount;     // 1 = mono, 2 = stereo，等等
     unsigned int32    sampleRate;       // 32000, 44100, 48000，等等
     unsigned int32    bytesPerSecond;   // 仅对压缩数据格式重要
     unsigned int16    blockAlignment;   // 一组样本的容器大小（以字节为单位）
     unsigned int16    bitsPerSample;    // 每样本的有效比特 16, 20 或 24
     unsigned int16    cbSize            // 需存储的额外信息大小(cbSize 之后)
     char              extraData[22]     // WAVE_FORMAT_EXTENSIBLE 的额外数据
};
 struct DataChunk                        // 声明 DataChunk 结构体
{
     char              chunkId[4];       //  ‘data’
     unsigned int32    chunkSize;        //  ‘data’块大小，4 字节
     char              waveata[ ]        // 音频样本
};
```

A.2 RF64/WAVE（MBWF）格式中的新块和结构体

4) 在 RIFF/WAVE 文件中有效的其他块在 RF64/WAVE 文件中同样有效，例如，RIFF/WAVE 的 BWF 扩展项也可在 RF64/WAVE 文件中不予改变地使用。

5) 空括号“[ ]”并不符合 C/C++的标准句法规则。这里表示数组包括可变数量的元素（甚至也可能是零个元素）。

```
struct RF64Chunk                        // 声明 RF64Chunk 结构体
{
    char              chunkId[4];       //  'RF64'
    unsigned int32    chunkSize;        // -1 = 0xFFFFFFFF 意味着不使用此数据, 代而使用
                                        // 'ds64' 块riffSizeHigh 和 riffSizeLow成员变量值
    char              rf64Type[4];      //  'WAVE'
};
struct ChunkSize64                      // 声明 ChunkSize64 结构体
{
    char              chunkId[4];       // chunk ID (例如,"big1")
    unsigned int32    chunkSizeLow;     // chunk 大小, 低 4 字节
    unsigned int32    chunkSizeHigh;    // chunk 大小, 高 4 字节
};
struct DataSize64Chunk                  // 声明 DataSize64Chunk 结构体
{
    char              chunkId[4];       //  'ds64'
    unsigned int32    chunkSize;        //  'ds64' 块大小, 4 字节
    unsigned int32    riffSizeLow;      // RF64 块大小, 低 4 字节
    unsigned int32    riffSizeHigh;     // RF64 块大小, 高 4 字节
    unsigned int32    dataSizeLow;      // data 块大小, 低 4 字节
    unsigned int32    dataSizeHigh;     // data 块大小, 高 4 字节
    unsigned int32    sampleCountLow;   // fact 块中样本计数, 低 4 字节
    unsigned int32    sampleCountHigh;  // fact 块中样本计数, 高 4 字节
    unsigned int32    tableLength;      // 成员数组"table"中有效条目的数量
    chunkSize64       table[ ];
};
struct Guid
{
    unsigned int32    data1;
    unsigned int16    data2;
    unsigned int16    data3;
    unsigned int32    data4;
    unsigned int32    data5;
};
struct FormatExtensibleChunk            // 声明用于 WAVE_FORMAT_EXTENSIBLE 的
                                        // FormatExtensibleChunk 结构体
{
    char              chunkId[4];       //  'fmt '
    unsigned int32    chunkSize;        //  'fmt ' 块大小, 4 字节
    unsigned int16    formatType;       // WAVE_FORMAT_EXTENSIBLE = 0xFFFE
    unsigned int16    channelCount;     // 1 = mono,2 = stereo, 等等
    unsigned int32    sampleRate;       // 32000, 44100, 48000, 等等
    unsigned int32    bytesPerSecond;   // 仅对压缩数据格式重要
```

```
    unsigned int16   blockAlignment;     // 一组样本的容器大小（以字节为单位）
    unsigned int16   bitsPerSample;      // 以每样本容器大小×8表示的每样本的比特数，如 8,
                                         //16, 24
    unsigned int16   cbSize              // 需存储的额外信息大小(cbSize之后)
    unsigned int16   validBitsPerSample  // 每样本的有效比特数，如 8, 16, 20, 24
    unsigned int32   channelMask         // 用于声道配置的声道模板
    Guid             subFormat           // KSDATAFORMAT_SUBTYPE_PCM
                                         // data1 = 0x00000001
                                         // data2 = 0x0000
                                         // data3 = 0x0010
                                         // data4 = 0xAA000080
                                         // data5 = 0x719B3800
    };
```

ds64 块中作为可选成员变量的 ChunkSize64 用于存储除 data 块外其他需要 64 位寻址的块的长度。目前，即使在非常大的音频文件中，除 data 块外，也没有其他标准块类型有可能超过 4G 字节大小（如，仅当“data”块大小达到约 512G 字节时，BWF 中“levl”块通常会超过 4G 字节）。

A.3 现有Cue Point块

```
struct CuePoint                      // 声明 CuePoint 结构体
{
  unsigned int32    identifier;      // 标记点的惟一标识符
  unsigned int32    position;        // 按播放顺序的标记点位置
  char              dataChunkId[4];  // 通常为‘data’
  unsigned int32    chunkStart;      // 用于存在 wave lists 的情况
  unsigned int32    blockStart;      // 包含标记点的压缩数据块的起点(不用于 PCM)
  unsigned int32    sampleOffset;    // 标记点的样本偏移量(对 PCM，为绝对值；对压缩数据，
                                     //为相对于压缩数据块起点的相对值)
};
struct CueChunk                      // 声明 CueChunk 结构体
{
    char            chunkId[4];      // ‘cue’
    unsigned int32  chunkSize;       // ‘cue’块大小，4 字节
    unsigned int32  cuePointCount;   // 标记点数
    CuePoint        cuePoints[];     // 标记点
};
struct ListChunk                     // 声明 ListChunk 结构体
{
    char            chunkId[4];      // ‘list’
    unsigned int32  chunkSize;       // ‘list’块大小，4 字节
    char            typeId[4];       // ‘adtl’（associated data list）
};
```

```
struct LabelChunk                       // 声明 LabelChunk 结构体
{
    char            chunkId[4];         // ‘labl’
    unsigned int32  chunkSize;          // ‘labl’块大小, 4 字节
    unsigned int32  identifier;         // 标记点的惟一标识符
    char            text[ ];            // 标签文本：NULL 结尾的字符串(ANSI)
};
```

A.4 RF64/WAVE（MBWF）格式中新Marker块和结构体

```
struct MarkerEntry                          // 声明 MarkerEntry 结构体
{
  unsigned int32    flags;                  // 标志字段
  unsigned int32    sampleOffsetLow;        // 标记点在 data 块中以样本为单位的偏移量，低 4
                                            //字节
  unsigned int32    sampleOffsetHigh;       // 标记点在 data 块中以样本为单位的偏移量，高 4
                                            //字节
  unsigned int32    byteOffsetLow;          // 时间线上先于标记点且离该点最近的压缩样本帧
                                            //起始位置的偏移量，低 4 字节
  unsigned int32    byteOffsetHigh;         //时间线上先于标记点且离标记点最近的压缩样本帧
                                            //起始位置的偏移量，高 4 字节
  unsigned int32    intraSmplOffsetHigh;    // 以样本为单位的相对于样本帧起始点的标记点位
                                            //置偏移量，高 4 字节
  unsigned int32    intraSmplOffsetLow;     //以样本为单位的相对于样本帧起始点的标记点位置
                                            //偏移量，低 4 字节
  char              labelText[256][6];      // 以 NULL 结尾的标签字符串
  unsigned int32    lablChunkIdentifier;    // 指向‘list’块中‘labl’子块的链接标识符
  Guid              vendorAndProduct;       // 识别特定的供应商应用程序的 GUID
  unsigned int32    userData1;              //特定于应用程序的用户数据，4 字节
  unsigned int32    userData2;              //特定于应用程序的用户数据，4 字节
  unsigned int32    userData3;              //特定于应用程序的用户数据，4 字节
  unsigned int32    userData4;              //特定于应用程序的用户数据，4 字节
};

struct MarkerChunk                          // 声明 MarkerChunk 结构体
{
    char            chunkId[4];             //  ‘r64m’
    unsigned int32  chunkSize               //  ‘r64m’块大小，4 字节
    MarkerEntry     markers[];              // 标记条目
};
```

flags字段定义

flags字段定义不同的块特性和结构体中成员变量的有效性。

比特 0	0	MarkerEntry无效，跳过
	1	MarkerEntry有效
比特 1	0	byteOffset成员变量无效，不使用
	1	byteOffset成员变量有效
比特 2	0	intraSmplOffset成员变量无效，不使用
	1	intraSmplOffset成员变量有效

6) 字符串以 ANSI 或 UTF-8 编码，取决于标志字段。

比特 3	0	标记的标签字符串存储于labelText成员变量（如果标签字符串为空，则标记无标签）
	1	标记的标签存储于labl块；使用lablChunkIdentifier值去检索
比特 4	0	labelText字符串为ANSI编码
	1	labelText字符串为UTF-8编码

参 考 文 献

[1] GY/T 168-2001 广播音频数据文件格式规范——广播波形格式（BWF）

[2] IEC 61937-1 Digital audio -Interface for non-linear PCM encoded audio bitstreams applying IEC 60958-Part 1: General

[3] IEC 61937-3 Digital audio -Interface for non-linear PCM encoded audio bitstreams applying IEC 60958 -Part 3: Non-linear PCM bitstreams according to the AC-3 and enhanced AC-3 formats

[4] IEC 61937-5 Digital audio -Interface for non-linear PCM encoded audio bitstreams applying IEC 60958 - Part 5: Non-linear PCM bitstreams according to the DTS (Digital Theater Systems) format(s)

[5] IEC 61937-6 Digital audio - Interface for non-linear PCM encoded audio bitstreams applying IEC 60958 - Part 6: Non-linear PCM bitstreams according to the MPEG-2 AAC and MPEG-4 AAC audio formats

[6] SMPTE 337M Format for Non-PCM Audio and Data in an AES3 Serial Digital Audio Interface

[7] SMPTE 338M Format for Non-PCM Audio and Data in AES3 —Data Types

[8] SMPTE 339M Format for Non-PCM Audio and Data in AES3 —Generic Data Types

[9] SMPTE 340M Format for Non-PCM Audio and Data in AES3 —ATSC A/52 (AC-3) Data Type

中 华 人 民 共 和 国
广播电影电视行业标准
音频扩展文件格式 MBWF/RF64
GY/T 281－2014
*
国家广播电影电视总局广播电视规划院出版发行
责任编辑：王佳梅
查询网址：www.abp.gov.cn
北京复兴门外大街二号
联系电话：（010）86093424　　86092923
邮政编码：100866