INTERNATIONAL STANDARD ISO/IEC 14496-3:2001
TECHNICAL CORRIGENDUM 2

Published 2004-06-15

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION • МЕЖДУНАРОДНАЯ ОРГАНИЗАЦИЯ ПО СТАНДАРТИЗАЦИИ • ORGANISATION INTERNATIONALE DE NORMALISATION
INTERNATIONAL ELECTROTECHNICAL COMMISSION • МЕЖДУНАРОДНАЯ ЭЛЕКТРОТЕХНИЧЕСКАЯ КОМИССИЯ • COMMISSION ÉLECTROTECHNIQUE INTERNATIONALE

Information technology — Coding of audio-visual objects —

Part 3:
Audio

TECHNICAL CORRIGENDUM 2

Technologies de l'information — Codage des objets audiovisuels —
Partie 3: Codage audio

RECTIFICATIF TECHNIQUE 2

Technical Corrigendum 2 to ISO/IEC 14496-3:2001 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

In subclause 1.5.2.2 (Complexity units), Table 1.3 (Complexity of Audio Object Types and SR conversion), replace:

Sampling Rate	rf = 2, 3, 4, 6	2	0.5	

with:

Sampling Rate	rf = 2, 3, 4, 6, 8, 12	2	0.5	

ICS 35.040 **Ref. No. ISO/IEC 14496-3:2001/Cor.2:2004(E)**

Published in Switzerland

Replace subclause 1.7 (MPEG-4 Audio transport stream) with:

1.7 *MPEG-4 Audio transport stream*

1.7.1 Overview

This subclause defines a mechanism to transport ISO/IEC 14496-3 (MPEG-4 Audio) streams without using ISO/IEC 14496-1 (MPEG-4 Systems) for audio-only applications. Figure 1.1 shows the concept of MPEG-4 Audio transport. The transport mechanism uses a two-layer approach, namely a multiplex layer and a synchronization layer. The multiplex layer (Low-overhead MPEG-4 Audio Transport Multiplex: LATM) manages multiplexing of several MPEG-4 Audio payloads and their AudioSpecificConfig() elements. The synchronization layer specifies a self-synchronized syntax of the MPEG-4 Audio transport stream which is called Low Overhead Audio Stream (LOAS). The interface format to a transmission layer depends on the conditions of the underlying transmission layer as follows:

- LOAS shall be used for the transmission over channels where no frame synchronization is available.
- LOAS may be used for the transmission over channels with fixed frame synchronization.
- A multiplexed element (AudioMuxElement() / EPMuxElement()) without synchronization shall be used only for transmission channels where an underlying transport layer already provides frame synchronization that can handle arbitrary frame size.

The details of the LOAS and the LATM formats are described in subclauses 1.7.2 and 1.7.3, respectively.

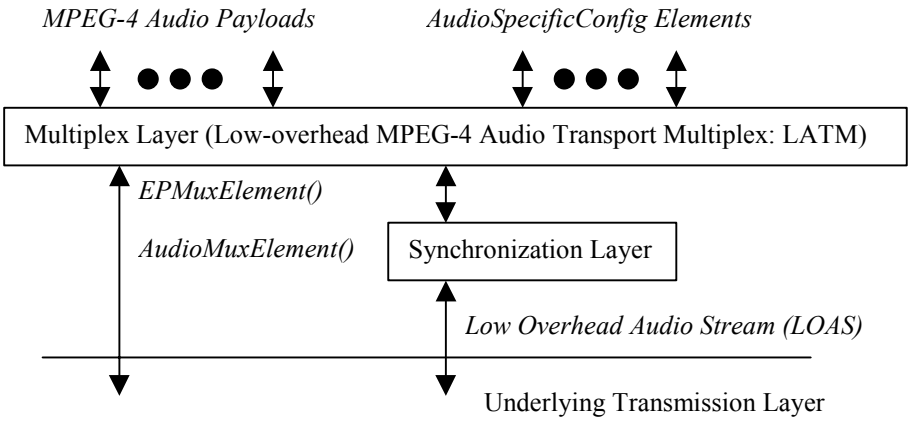

Figure 1.1 – Concept of MPEG-4 Audio Transport

The mechanism defined in this subclause should not be used for transmission of TTSI objects (12), Main Synthetic objects (13), Wavetable Synthesis objects (14), General MIDI objects (15) and Algorithmic Synthesis and Audio FX objects (16). It should further not be used for transmission of any object in conjunction with (epConfig==1). For those objects, other multiplex and transport mechanisms might be used, e.g. those defined in MPEG-4 Systems.

1.7.2 Synchronization Layer

The synchronization layer provides the multiplexed element with a self-synchronized mechanism to generate LOAS. The LOAS has three different types of format, namely AudioSyncStream(), EPAudioSyncStream() and AudioPointerStream(). The choice for one of the three formats is dependent on the underlying transmission layer.

- AudioSyncStream()

AudioSyncStream() consists of a syncword, the multiplexed element with byte alignment, and its length information. The maximum byte-distance between two syncwords is 8192 bytes. This self-synchronized stream shall be used for the case that the underlying transmission layer comes without any frame synchronization.

- EPAudioSyncStream()

For error prone channels, an alternative version to AudioSyncStream() is provided. This format has the same basic functionality as the previously described AudioSyncStream(). However, it additionally provides a longer syncword and a frame counter to detect lost frames. The length information and the frame counter are additionally protected by a FEC code.

- AudioPointerStream()

AudioPointerStream() shall be used for applications using an underlying transmission layer with fixed frame synchronization, where transmission framing cannot be synchronized with the variable length multiplexed element. Figure 1.2 shows synchronization in AudioPointerStream(). This format utilizes a pointer indicating the start of the next multiplex element in order to synchronize the variable length payload with the constant transmission frame.

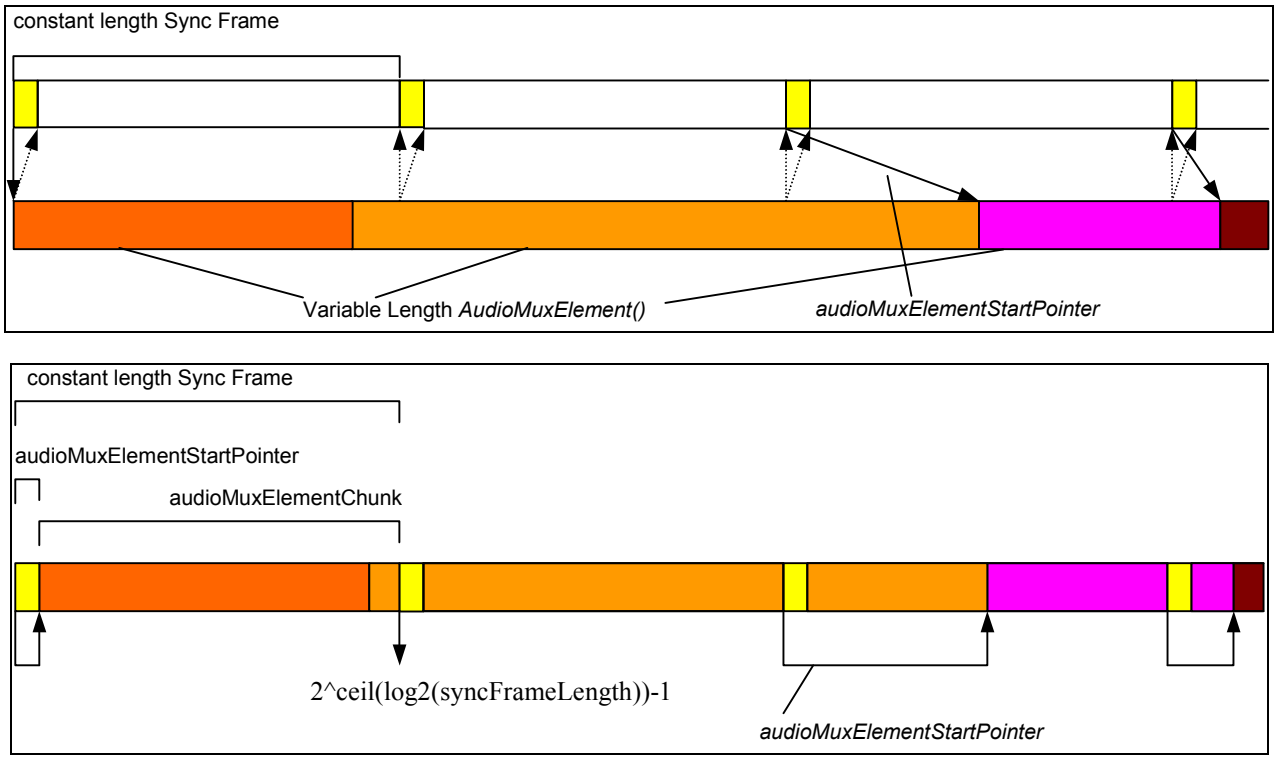

Figure 1.2 – Synchronization in AudioPointerStream()

1.7.2.1 Syntax

Table 1.16 – Syntax of AudioSyncStream()

Syntax	No. of bits	Mnemonic
AudioSyncStream()		
{		
while (nextbits() == **0x2B7**) { /* syncword */	**11**	**bslbf**
audioMuxLengthBytes;	**13**	**uimsbf**
AudioMuxElement(1);		
}		
}		

Table 1.17 – Syntax of EPAudioSyncStream()

Syntax	No. of bits	Mnemonic
EPAudioSyncStream()		
{		
while (nextbits() == **0x4de1**) { /* syncword */	**16**	**bslbf**
futureUse;	**4**	**uimsbf**
audioMuxLengthBytes;	**13**	**uimsbf**
frameCounter;	**5**	**uimsbf**
headerParity;	**18**	**bslbf**
EPMuxElement(1, 1);		
}		
}		

Table 1.18 – Syntax of AudioPointerStream()

Syntax	No. of bits	Mnemonic
AudioPointerStream (syncFrameLength)		
{		
while (! EndOfStream) {		
AudioPointerStreamFrame (syncFrameLength);		
}		
}		

Table 1.18a – Syntax of AudioPointerStreamFrame()

Syntax	No. of bits	Mnemonic
AudioPointerStreamFrame(length)		
{		
audioMuxElementStartPointer;	**ceil(log2(length))**	**uimsbf**
audioMuxElementChunk;	**length – ceil(log2(length))**	**bslbf**
}		

1.7.2.2 Semantics

1.7.2.2.1 AudioSyncStream()

audioMuxLengthBytes A 13-bit data element indicating the byte length of the subsequent AudioMuxElement() with byte alignment (AudioSyncStream) or the subsequent EPMuxElement() (EPAudioSyncStream).

AudioMuxElement() A multiplexed element as specified in subclause 1.7.3.2.2.

1.7.2.2.2 EPAudioSyncStream()

futureUse A 4-bit data element for future use, which shall be set to ‘0000’.

audioMuxLengthBytes see subclause 1.7.2.2.1.

frameCounter A 5-bit data element indicating a sequential number which is used to detect lost frames. The number is continuously incremented for each multiplexed element as a modulo counter.

headerParity A 18-bit data element which contains a BCH (36,18) code shortened from BCH (63,45) code for the elements **audioMuxLengthBytes** and **frameCounter**. The generator polynomial is $x^{18}+x^{17}+x^{16}+x^{15}+x^{9}+x^{7}+x^{6}+x^{3}+x^{2}+x+1$. The value is calculated with this generator polynomial as described in subclause 1.8.4.3.

EPMuxElement() An error resilient multiplexed element as specified in subclause 1.7.3.2.1.

1.7.2.2.3 AudioPointerStream()

AudioPointerStreamFrame() A sync frame of fixed length provided by an underlying transmission layer.

audioMuxElementStartPointer A data element indicating the starting point of the first AudioMuxElement() within the current AudioPointerStreamFrame(). The number of bits required for this data element is calculated as ceil(log2(syncFrameLength)). The transmission frame length has to be provided from the underlying transmission layer. The maximum possible value of this data element is reserved to signal that there is no start of an AudioMuxElement() in this sync frame.

audioMuxElementChunk A part of a concatenation of subsequent AudioMuxElement()'s (see Figure 1.2).

1.7.3 Multiplex Layer

The LATM layer multiplexes several MPEG-4 Audio payloads and AudioSpecificConfig() syntax elements into one multiplexed element. The multiplexed element format is selected between AudioMuxElement() and EPMuxElement() depending on whether error resilience is required in the multiplexed element itself, or not. EPMuxElement() is an error resilient version of AudioMuxElement() and may be used for error prone channels.

The multiplexed elements can be directly conveyed on transmission layers with frame synchronization. In this case, the first bit of the multiplexed element shall be located at the first bit of a transmission payload in the underlying transmission layer. If the transmission payload allows only byte-aligned payload, padding bits for byte alignment shall follow the multiplexed element. The number of the padding bits should be less than 8. These padding bits should be removed when the multiplexed element is de-multiplexed into the MPEG-4 Audio payloads. Then, the MPEG-4 Audio payloads are forwarded to the corresponding MPEG-4 Audio decoder tool.

Usage of LATM in case of scalable configurations with CELP core and AAC enhancement layer(s):

- *Instances of the AudioMuxElement() are transmitted in equidistant manner.*
- *The represented timeframe of one AudioMuxElement() is similar to a multiple of a super-frame timeframe.*
- *The relative number of bits for a certain layer within any AudioMuxElement() compared to the total number of bits within this AudioMuxElement() is equal to the relative bitrate of that layer compared to the bitrate of all layers.*
- *In case of coreFrameOffset = 0 and latmBufferFullness = 0, all core coder frames and all AAC frames of a certain super-frame are stored within the same instance of AudioMuxElement().*
- *In case of coreFrameOffset > 0, several or all core coder frames are stored within previous instances of AudioMuxElement().*
- *Any core layer related configuration information refers to the core frames transmitted within the current instance of the AudioMuxElement(), independent of the value of coreFrameOffset.*
- *A specified latmBufferFullness is related to the first AAC frame of the first super-frame stored within the current AudioMuxElement().*
- *The value of latmBufferFullness can be used to determine the location of the first bit of the first AAC frame of the current layer of the first super-frame stored within the current AudioMuxElement() by means of a backpointer:*
- $backPointer = -meanFrameLength + latmBufferFullness + currentFrameLength$

  The backpointer value specifies the location as a negative offset from the current AudioMuxElement(), i. e. it points backwards to the beginning of an AAC frame located in already received data. Any data not belonging to the payload of the current AAC layer is not taken into account. If (latmBufferFullness == '0'), then the AAC frame starts after the current AudioMuxElement().

Note that the possible LATM configurations are restricted due to limited signalling capabilities of certain data elements as follows:

- Number of layers: 8 (numLayer has 3 bit)
- Number of streams: 16 (streamIndx has 4 bit)
- Number of chunks: 16 (numChunk has 4 bit)

1.7.3.1 Syntax

Table 1.19 – Syntax of EPMuxElement()

Syntax	No. of bits	Mnemonic
EPMuxElement(epDataPresent, muxConfigPresent)		
{		
if (epDataPresent) {		
epUsePreviousMuxConfig;	**1**	**bslbf**
epUsePreviousMuxConfigParity;	**2**	**bslbf**
if (!epUsePreviousMuxConfig) {		
epSpecificConfigLength;	**10**	**bslbf**
epSpecificConfigLengthParity;	**11**	**bslbf**
ErrorProtectionSpecificConfig();		
ErrorProtectionSpecificConfigParity();		
}		
ByteAlign();		
EPAudioMuxElement(muxConfigPresent);		
}		
else {		
AudioMuxElement(muxConfigPresent);		
}		
}		

Table 1.20 – Syntax of AudioMuxElement()

Syntax	No. of bits	Mnemonic
AudioMuxElement(muxConfigPresent)		
{		
if (muxConfigPresent) {		
useSameStreamMux;	**1**	**bslbf**
if (!useSameStreamMux)		
StreamMuxConfig();		
}		
if (audioMuxVersionA == 0) {		
for (i = 0; i <= numSubFrames; i++) {		
PayloadLengthInfo();		
PayloadMux();		
}		
if (otherDataPresent) {		
for(i = 0; i < otherDataLenBits; I++) {		
otherDataBit;	**1**	**bslbf**
}		
}		
}		
else {		
/* tbd */		
}		
ByteAlign();		
}		

Table 1.21 – Syntax of StreamMuxConfig()

Syntax	No. of bits	Mnemonic
StreamMuxConfig()		
{		
audioMuxVersion;	**1**	**bslbf**
if (audioMuxVersion == 1) {		
audioMuxVersionA;	**1**	**bslbf**
else {		
audioMuxVersionA = 0;		
}		
if (audioMuxVersionA == 0) {		
if (audioMuxVersion == 1) {		
taraBufferFullness = LatmGetValue();		
}		
streamCnt = 0;		
allStreamsSameTimeFraming;	**1**	**uimsbf**
numSubFrames;	**6**	**uimsbf**
numProgram;	**4**	**uimsbf**
for (prog = 0; prog <= numProgram; prog++) {		
numLayer;	**3**	**uimsbf**
for (lay = 0; lay <= numLayer; lay++) {		
progSIndx[streamCnt] = prog; laySIndx[streamCnt] = lay;		
streamID [prog][lay] = streamCnt++;		
if (prog == 0 && lay == 0) {		
useSameConfig = 0;		
} else {		
useSameConfig;	**1**	**uimsbf**
}		
if (! useSameConfig)		
if (audioMuxVersion == 1) {		
ascLen = LatmGetValue();		
}		
ascLen -= AudioSpecificConfig();		Note 1
fillBits;	**ascLen**	**bslbf**
}		
frameLengthType[streamID[prog][lay]];	**3**	**uimsbf**
if (frameLengthType[streamID[prog][lay] == 0) {		
latmBufferFullness[streamID[prog][lay]];	**8**	**uimsbf**
if (! allStreamsSameTimeFraming) {		
if ((AudioObjectType[lay] == 6 ||		
AudioObjectType[lay] == 20) &&		
(AudioObjectType[lay-1] == 8 ||		
AudioObjectType[lay-1] == 24)) {		
coreFrameOffset;	**6**	**uimsbf**
}		
}		
} else if (frameLengthType[streamID[prog][lay]] == 1) {		
frameLength[streamID[prog][lay]];	**9**	**uimsbf**
} else if (frameLengthType[streamID[prog][lay]] == 4		
frameLengthType[streamID[prog][lay]] == 5		
frameLengthType[streamID[prog][lay]] == 3) {		
CELPframeLengthTableIndex[streamID[prog][lay]];	**6**	**uimsbf**
} else if (frameLengthType[streamID[prog][lay]] == 6		
frameLengthType[streamID[prog][lay]] == 7) {		
HVXCframeLengthTableIndex[streamID[prog][lay]];	**1**	**uimsbf**
}		
}		
}		

```
        otherDataPresent;                                          1     uimsbf
        if (otherDataPresent) {
            if ( audioMuxVersion == 1 ) {
                otherDataLenBits = LatmGetValue();
            }
            else {
                otherDataLenBits = 0; /* helper variable 32bit */
                do {
                    otherDataLenBits *= 2^8;
                    otherDataLenEsc;                               1     uimsbf
                    otherDataLenTmp;                               8     uimsbf
                    otherDataLenBits += otherDataLenTmp;
                } while (otherDataLenEsc);
            }
        }
        crcCheckPresent;                                           1     uimsbf
        if (crcCheckPresent) crcCheckSum;                          8     uimsbf
    }
    else {
        /* tbd */
    }
}
```

Note 1: AudioSpecificConfig() returns the number of bits read.

Table 1.COR2-1 – Syntax of LatmGetValue()


```
Syntax                                                     No. of bits  Mnemonic
LatmGetValue()
    bytesForValue;                                             2        uimsbf
    value = 0; /* helper variable 32bit */
    for ( i = 0; i <= bytesForValue; i++ ) {
        value *= 2^8;
        valueTmp;                                              8        uimsbf
        value += valueTmp;
    }
    return value;
}
```

Table 1.22 – Syntax of PayloadLengthInfo()

```
Syntax                                                     No. of bits  Mnemonic
PayloadLengthInfo()
{
    if (allStreamsSameTimeFraming) {
        for (prog = 0; prog <= numProgram; prog++) {
            for (lay = 0; lay <= numLayer; lay++) {
                if (frameLengthType[streamID[prog][ lay]] == 0) {
                    MuxSlotLengthBytes[streamID[prog][ lay]] = 0;
                    do {  /* always one complete access unit */
                        tmp;                                       8        uimsbf
                        MuxSlotLengthBytes[streamID[prog][ lay]] += tmp;
                    } while(tmp == 255);
                } else {
                    if (  frameLengthType[streamID[prog][ lay]] == 5 ||
                          frameLengthType[streamID[prog][ lay]] == 7 ||
```

```
                    frameLengthType[streamID[prog][ lay]] == 3 ) {
                    MuxSlotLengthCoded[streamID[prog][ lay]];          2          uimsbf
                }
            }
        }
    }
} else {
    numChunk;                                                          4          uimsbf
    for (chunkCnt = 0; chunkCnt <= numChunk; chunkCnt++) {
        streamIndx;                                                    4          uimsbf
        prog = progCIndx[chunkCnt] = progSIndx[streamIndx];
        lay   = layCIndx[chunkCnt]   = laySIndx   [streamIndx];
        if (frameLengthType[streamID[prog][lay]] == 0) {
            MuxSlotLengthBytes[streamID[prog][ lay]] = 0;
            do {  /* not necessarily a complete access unit */
                tmp;                                                   8          uimsbf
                MuxSlotLengthBytes[streamID[prog][lay]] += tmp;
            } while (tmp == 255);
            AuEndFlag[streamID[prog][lay]];                            1          bslbf
        } else  {
            if (frameLengthType[streamID[prog][lay]] == 5 ||
                frameLengthType[streamID[prog][lay]] == 7 ||
                frameLengthType[streamID[prog][lay]] == 3) {
                MuxSlotLengthCoded[streamID[prog][lay]];               2          uimsbf
            }
        }
    }
}
}
```


Table 1.23 – Syntax of PayloadMux()

```
Syntax                                                          No. of bits  Mnemonic
PayloadMux()
{
    if (allStreamsSameTimeFraming) {
        for (prog = 0; prog <= numProgram; prog++) {
            for (lay = 0; lay <= numLayer; lay++) {
                payload [streamID[prog][ lay]];
            }
        }
    } else {
        for (chunkCnt = 0; chunkCnt <= numChunk; chunkCnt++) {
            prog = progCIndx[chunkCnt];
            lay = layCIndx  [chunkCnt];
            payload [streamID[prog][ lay]];
        }
    }
}
```

1.7.3.2 Semantics

1.7.3.2.1 EPMuxElement()

For parsing of EPMuxElement(), an epDataPresent flag shall be additionally set at the underlying layer. If epDataPresent is set to 1, this indicates EPMuxElement() has error resiliency. If not, the format of EPMuxElement() is identical to AudioMuxElement(). The default for both flags is 1.

epDataPresent	Description
0	EPMuxElement() is identical to AudioMuxElement()
1	EPMuxElement() has error resiliency

epUsePreviousMuxConfig A flag indicating whether the configuration for the MPEG-4 Audio EP tool in the previous frame is applied in the current frame.

epUsePreviousMuxConfig	Description
0	The configuration for the MPEG-4 Audio EP tool is present.
1	The configuration for the MPEG-4 Audio EP tool is not present. The previous configuration should be applied.

epUsePreviousMuxConfigParity A 2-bits element which contains the parity for **epUsePreviousMuxConfig.** Each bit is a repetition of **epUsePreviousMuxConfig.** Majority decides.

epSpecificConfigLength A 10-bit data element to indicate the size of ErrorProtectionSpecificConfig()

epSpecificConfigLengthParity An 11-bit data element for epHeaderLength, calculated as described in subclause 1.8.4.3 with "1) Basic set of FEC codes".
Note: This means shortened Golay(23,12) is used

ErrorProtectionSpecificConfig() A data function covering configuration information for the EP tool which is applied to AudioMuxElement() as defined in subclause 1.8.2.1.

ErrorProtectionSpecificConfigParity() A data function covering the parity bits for **ErrorProtectionSpecificConfig()**, calculated as described in subclause 1.8.4.3, Table 1.45.

EPAudioMuxElement() A data function covering **e**rror resilient multiplexed element that is generated by applying the EP tool to AudioMuxElement() as specified by ErrorProtectionSpecificConfig(). Therefore data elements in AudioMuxElement() are subdivided into different categories depending on their error sensitivity and collected in instances of these categories. Following sensitivity categories are defined:

elements	error sensitivity category
useSameStreamMux + StreamMuxConfig()	0
PayloadLengthInfo()	1
PayloadMux()	2
otherDataBits	3

Note 1: There might be more than one instance of error sensitivity category 1 and 2 depending on the value of the variable **numSubFrames** defined in **StreamMuxConfig()**. Figure 1.3 shows an example for the order of the instances assuming numSubFrames is one (1).

Note 2: EPAudioMuxElement() has to be byte aligned, therefore **bit_stuffing** in ErrorProtectionSpecificConfig() should be always on.

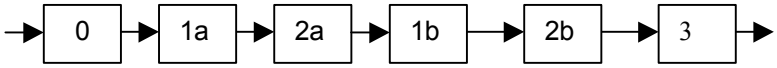

Figure 1.3 – Instance order in EPAudioMuxElement()

1.7.3.2.2 AudioMuxElement()

In order to parse an AudioMuxElement(), a muxConfigPresent flag shall be set at the underlying layer. If muxConfigPresent is set to 1, this indicates multiplexing configuration (StreamMuxConfig()) is multiplexed into AudioMuxElement(), i.e. in-band transmission. If not, StreamMuxConfig() should be conveyed through out-band means, such as session announcement/description/control protocols.

muxConfigPresent	Description
0	out-band transmission of StreamMuxConfig()
1	in-band transmission of StreamMuxConfig()

useSameStreamMux A flag indicating whether the multiplexing configuration in the previous frame is applied in the current frame.

useSameStreamMux	Description
0	The multiplexing configuration is present.
1	The multiplexing configuration is not present. The previous configuration should be applied.

otherDataBit A 1-bit data element indicating the other data information.

1.7.3.2.3 StreamMuxConfig()

AudioSpecificConfig() is specified in subclause 1.6.2.1. In this case it constitutes a standalone element in itself (i.e. it does not extend the class BaseDescriptor as in the case of subclause 1.6).

audioMuxVersion A data element to signal the used multiplex syntax.
Note: In addition to (audioMuxVersion == 0), (audioMuxVersion == 1) supports the transmission of a taraBufferFullness and the transmission of the lengths of individual AudioSpecificConfig() data functions.

audioMuxVersionA A data element to signal the bitstream syntax version. Possible values: 0 (default), 1 (reserved for future extensions).

taraBufferFullness A helper variable indicating the state of the bit reservoir in the course of encoding the LATM status information. It is transmitted as the number of available bits in the tara bit reservoir divided by 32 and truncated to an integer value. The maximum value that can be signaled using any setting of bytesForValue signals that the particular program and layer is of variable rate. This might be the value of hexadecimal FF (bytesForValue == 0), FFFF (bytesForValue == 1), FFFFFF (bytesForValue == 2) or FFFFFFFF (bytesForValue == 3). In these cases, buffer fullness is not applicable. The state of the bit reservoir is derived according to what is stated in subpart 4, subclause 4.5.3.2 (Bit reservoir). The LATM status information considered by the taraBufferFullness comprises any data of the AudioMuxElement() except of PayloadMux().

