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.

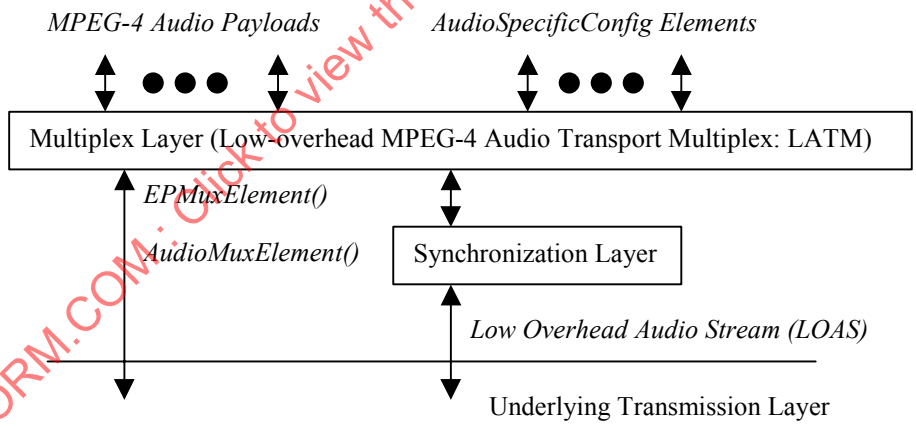

Figure 1.1 – Concept of MPEG-4 Audio Transport

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.

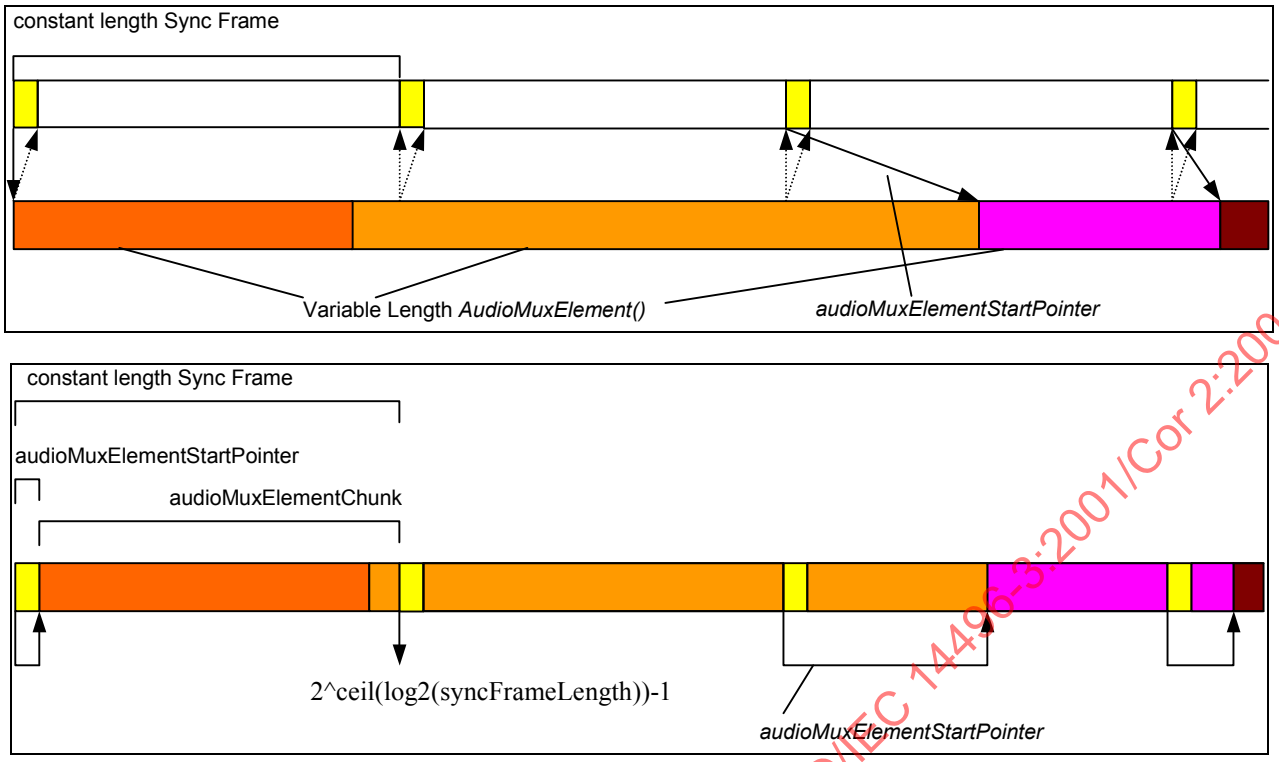

Figure 1.2 – Synchronization in AudioPointerStream()

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.

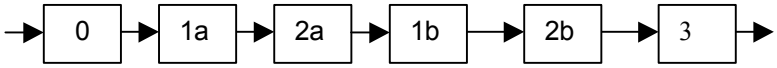

Figure 1.3 – Instance order in EPAudioMuxElement()

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().

allStreamsSameTimeFraming A data element indicating whether all payloads, which are multiplexed in PayloadMux(), share a common time base.

numSubFrames A data element indicating how many PayloadMux() frames are multiplexed (numSubFrames+1). If more than one PayloadMux() frame is multiplexed, all PayloadMux() share a common StreamMuxConfig().The minimum value is 0 indicating 1 subframe.

numProgram A data element indicating how many programs are multiplexed (numProgram+1). The minimum value is 0 indicating 1 program.

numLayer A data element indicating how many scalable layers are multiplexed (numLayer+1). The minimum value is 0 indicating 1 layer.

useSameConfig A data element indicating whether AudioSpecificConfig() for the payload in the previous layer or program is applied for the payload in the current layer or program.

useSameConfig	Description
0	AudioSpecificConfig() is present.
1	AudioSpecificConfig() is not present. AudioSpecificConfig() in the previous layer or program should be applied.

ascLen[prog][lay] A helper variable indicating the length in bits of the subsequent AudioSpecificConfig() data function including possible fill bits.

fillBits Fill bits.

frameLengthType A data element indicating the frame length type of the payload. For CELP and HVXC objects, the frame length (bits/frame) is stored in tables and only the indices to point out the frame length of the current payload is transmitted instead of sending the frame length value directly.

frameLengthType	Description
0	Payload with variable frame length. The payload length in bytes is directly specified with 8-bit codes in PayloadLengthInfo().
1	Payload with fixed frame length. The payload length in bits is specified with frameLength in StreamMuxConfig().
2	Reserved
3	Payload for a CELP object with one of 2 kinds of frame length. The payload length is specified by two table-indices, namely CELPframeLengthTableIndex and MuxSlotLengthCoded.
4	Payload for a CELP or ER_CELP object with fixed frame length. CELPframeLengthTableIndex specifies the payload length.
5	Payload for an ER_CELP object with one of 4 kinds of frame length. The payload length is specified by two table-indices, namely CELPframeLengthTableIndex and MuxSlotLengthCoded.
6	Payload for a HVXC or ER_HVXC object with fixed frame length. HVXCframeLengthTableIndex specifies the payload length.
7	Payload for an HVXC or ER_HVXC object with one of 4 kinds of frame length. The payload length is specified by two table-indices, namely HVXCframeLengthTableIndex and MuxSlotLengthCoded.

latmBufferFullness[streamID[prog][lay]] data element indicating the state of the bit reservoir in the course of encoding the first access unit of a particular program and layer in an AudioMuxElement(). It is transmitted as the number of available bits in the bit reservoir divided by the NCC divided by 32 and truncated to an integer value. A value of hexadecimal FF signals that the particular program and layer is of variable rate. In this case, 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).
In the case of (audioMuxVersion == 0), bits spend for data other than any payload (e.g. multiplex status information or other data) are considered in the first occuring latmBufferFullness in an AudioMuxElement(). For AAC, the limitations given by the minimum decoder input buffer apply (see subpart 4, subclause 4.5.3.1). In the case of (allStreamsSameTimeFraming==1), and if only one program and one layer is present, this leads to an LATM configuration similar to ADTS.
In the case of (audioMUxVersion == 1), bits spend for data other than any payload are considered by taraBufferFullness.

coreFrameOffset identifies the first CELP frame of the current super-frame. It is defined only in case of scalable configurations with CELP core and AAC enhancement layer(s) and transmitted with the first AAC enhancement layer. The value 0 identifies the first CELP frame following StreamMuxConfig() as the first CELP frame of the current super-frame. A value > 0 signals the number of CELP frames that the first CELP frame of the current super-frame is transmitted earlier in the bitstream.

frameLength A data element indicating the frame length of the payload with frameLengthType of 1. The payload length in bits is specified as 8 * (frameLength + 20).

CELPframeLengthTableIndex A data element indicating one of two indices for pointing out the frame length for a CELP or ER_CELP object. (Table 1.25 and Table 1.26)

HVXCframeLengthTableIndex A data element indicating one of two indices for pointing out the frame length for a HVXC or ER_HVXC object. (Table 1.24)

otherDataPresent A flag indicating the presence of the other data than audio payloads.

otherDataPresent	Description
0	The other data than audio payload otherData is not multiplexed.
1	The other data than audio payload otherData is multiplexed.

otherDataLenBits A helper variable indicating the length in bits of the other data.

crcCheckPresent A data element indicating the presence of CRC check bits for the StreamMuxConfig() data functions.

crcCheckPresent	Description
0	CRC check bits are not present.
1	CRC check bits are present.

crcCheckSum A data element indicating the CRC check bits.

1.7.3.2.4 LatmGetValue()

bytesForValue A data element indicating the number of occurencies of the data element valueTmp.

valueTmp A data element used to calculate the helper variable value.

value A helper variable representing a value returned by the data function LatmGetValue().

1.7.3.2.5 PayloadLengthInfo()

tmp A data element indicating the payload length of the payload with frameLengthType of 0. The value 255 is used as an escape value and indicates that at least one more **tmp** value is following. The overall length of the transmitted payload is calculated by summing up the partial values.

MuxSlotLengthCoded A data element indicating one of two indices for pointing out the payload length for CELP, HVXC, ER_CELP, and ER_HVXC objects.

numChunk A data element indicating the number of payload chunks (numChunk+1). Each chunk may belong to an access unit with a different time base; only used if allStreamsSameTimeFraming is set to zero. The minimum value is 0 indicating 1 chunk.

streamIndx A data element indicating the stream. Used if payloads are splitted into chunks.

chunkCnt Helper variable to count number of chunks.

progSIndx,laySIndx Helper variables to identify program and layer number from **streamIndx.**

progCIndx,layCIndx Helper variables to identify program and layer number from **chunkCnt.**

AuEndFlag A flag indicating whether the payload is the last fragment, in the case that an access unit is transmitted in pieces.

AuEndFlag	Description
0	The fragmented piece is not the last one.
1	The fragmented piece is the last one.

1.7.3.2.6 PayloadMux()

payload The actual audio payload by means of either an access unit (allStreamsSameTimeFraming == 1) or a part of a concatenation of subsequent access units (allStreamsSameTimeFraming == 0).

1.7.3.3 Tables

Table 1.24 – Frame length of HVXC [bits]

		MuxSlotLengthCoded			
frameLengthType[]	HVXCframeLengthTableIndex[]	00	01	10	11
6	0	40			
6	1	80			
7	0	40	28	2	0
7	1	80	40	25	3

Table 1.25 – Frame Length of CELP Layer 0 [bits]

CELPframeLenghTable Index	Fixed-Rate frameLengthType[] =4	1-of-4 Rates (Silence Compression) frameLengthType[]=5				1-of-2 Rates (FRC) frameLengthType[]=3	
		MuxSlotLengthCoded				MuxSlotLengthCoded	
		00	01	10	11	00	01
0	154	156	23	8	2	156	134
1	170	172	23	8	2	172	150
2	186	188	23	8	2	188	166
3	147	149	23	8	2	149	127
4	156	158	23	8	2	158	136
5	165	167	23	8	2	167	145
6	114	116	23	8	2	116	94
7	120	122	23	8	2	122	100
8	126	128	23	8	2	128	106
9	132	134	23	8	2	134	112
10	138	140	23	8	2	140	118
11	142	144	23	8	2	144	122
12	146	148	23	8	2	148	126
13	154	156	23	8	2	156	134
14	166	168	23	8	2	168	146
15	174	176	23	8	2	176	154
16	182	184	23	8	2	184	162
17	190	192	23	8	2	192	170
18	198	200	23	8	2	200	178
19	206	208	23	8	2	208	186
20	210	212	23	8	2	212	190
21	214	216	23	8	2	216	194
22	110	112	23	8	2	112	90
23	114	116	23	8	2	116	94
24	118	120	23	8	2	120	98
25	120	122	23	8	2	122	100
26	122	124	23	8	2	124	102
27	186	188	23	8	2	188	166
28	218	220	40	8	2	220	174
29	230	232	40	8	2	232	186
30	242	244	40	8	2	244	198
31	254	256	40	8	2	256	210
32	266	268	40	8	2	268	222
33	278	280	40	8	2	280	234
34	286	288	40	8	2	288	242
35	294	296	40	8	2	296	250
36	318	320	40	8	2	320	276
37	342	344	40	8	2	344	298
38	358	360	40	8	2	360	314
39	374	376	40	8	2	376	330
40	390	392	40	8	2	392	346
41	406	408	40	8	2	408	362
42	422	424	40	8	2	424	378
43	136	138	40	8	2	138	92
44	142	144	40	8	2	144	98
45	148	150	40	8	2	150	104
46	154	156	40	8	2	156	110
47	160	162	40	8	2	162	116
48	166	168	40	8	2	168	122
49	170	172	40	8	2	172	126
50	174	176	40	8	2	176	130
51	186	188	40	8	2	188	142

52	198	200	40	8	2	200	154
53	206	208	40	8	2	208	162
54	214	216	40	8	2	216	170
55	222	224	40	8	2	224	178
56	230	232	40	8	2	232	186
57	238	240	40	8	2	240	194
58	216	218	40	8	2	218	172
59	160	162	40	8	2	162	116
60	280	282	40	8	2	282	238
61	338	340	40	8	2	340	296
62-63	reserved						

Table 1.26 – Frame Length of CELP Layer 1-5 [bits]

CELPframeLenghTableIndex	Fixed-Rate frameLengthType []=4	1-of-4 Rates (Silence Compression) frameLengthType[]=5 MuxSlotLengthCoded			
		00	01	10	11
0	80	80	0	0	0
1	60	60	0	0	0
2	40	40	0	0	0
3	20	20	0	0	0
4	368	368	21	0	0
5	416	416	21	0	0
6	464	464	21	0	0
7	496	496	21	0	0
8	284	284	21	0	0
9	320	320	21	0	0
10	356	356	21	0	0
11	380	380	21	0	0
12	200	200	21	0	0
13	224	224	21	0	0
14	248	248	21	0	0
15	264	264	21	0	0
16	116	116	21	0	0
17	128	128	21	0	0
18	140	140	21	0	0
19	148	148	21	0	0
20-63	reserved				

In subclause 1.A.3.2.1 (Definitions:Bitstream elements for ADTS) replace:

channel_configuration: Indicates the channel configuration used. If channel_configuration is greater than 0, the channel configuration is given by Table 6.3, see subclause 6.3.4. If channel_configuration equals 0, the channel configuration is not specified in the header and must be given by a program_config_element following as first bitstream element in the first raw_data_block after the header, or by the implicit configuration (see subpart 4) or must be known in the application.

with:

channel_configuration: Indicates the channel configuration used. In the case of (channel_configuration > 0), the channel configuration is given in Table 1.11. In the case of (channel_configuration == 0), the channel configuration is not specified in the header, but as follows:

MPEG-2/4 ADTS: A single program_config_element() following as first syntactic element in the first raw_data_block() after the header specifies the channel configuration. Note that the

program_config_element() might not be present in each frame. An MPEG-4 ADTS decoder should not generate any output until it received a program_config_element(), while an MPEG-2 ADTS decoder may assume an implicit channel configuration.

MPEG-2 ADTS: Beside the usage of a program_config_element(), the channel configuration may be assumed to be given implicitly (see ISO/IEC13818-7) or may be known in the application.

In subclause 1.A.3.2.2 (Description) remove:

The ADTS only supports a raw_data_stream() with only one program. The program may have up to 7 channels plus an independently switched coupling channel.

Replace the content of subclause 1.C (Patent statements) with:

The International Organization for Standardization and the International Electrotechnical Commission (IEC) draw attention to the fact that it is claimed that compliance with this part of ISO/IEC 14496 may involve the use of patents.

ISO and IEC take no position concerning the evidence, validity and scope of these patent rights.

The holders of these patent rights have assured the ISO and IEC that they are willing to negotiate licences under reasonable and non-discriminatory terms and conditions with applicants throughout the world. In this respect, the statements of the holders of these patents right are registered with ISO and IEC. Information may be obtained from the companies listed in the Table below.

Attention is drawn to the possibility that some of the elements of this part of ISO/IEC 14496 may be the subject of patent rights other than those identified in this annex. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Alcatel
AT&T
BBC
Bosch
British Telecommunications
Canon
CCETT
Coding Technologies
Columbia Innovation Enterprise
Columbia University
Creative
CSELT
DemoGraFX
DirecTV
Dolby
EPFL
ETRI
France Telecom
Fraunhofer
Fujitsu
GC Technology Corporation
Hitachi
Hyundai
IBM
Institut fuer Rundfunktechnik

JVC
KPN
Matsushita Electric Industrial Co., Ltd.
Microsoft
Mitsubishi
NEC
NHK
Nokia
NTT
NTT Mobile Communication Networks
OKI
Philips
Rockwell
Samsung
Sarnoff
Scientific Atlanta
Sharp
Siemens
Sony
Telenor
Thomson

In subclause 3.B.5.3 (Encoding process) replace:

$$acf[k] = \sum_{n=0}^{frame\_size-k-1} sw[n] \cdot sw[n+k], 0 \le k \le lpc\_order$$

with:

$$acf[k] = \sum_{n=0}^{window\_size-k-1} sw[n] \cdot sw[n+k], 0 \le k \le lpc\_order$$

In 4.4.2.7 (Subsidary payloads), Table 4.49 (Syntax of ltp_data()) replace:

```
if(window_sequence==EIGHT_SHORT_SEQUENCE) {
    for (w=0; w<num_windows; w++ )  {
        ltp_short_used[w]                          1        uimsbf
        if (ltp_short_used [w]) {
            ltp_short_lag_present[w]               1        uimsbf
            if (ltp_short_lag_present[w]) {
                ltp_short_lag[w]                   4        uimsbf
            }
        }
    }
} else {
    for ( sfb=0; sfb< min(max_sfb,
        MAX_LTP_LONG_SFB; sfb++ ) {
        ltp_long_used[sfb]                         1        uimsbf
    }
}
```

with:

```
        if(window_sequence!=EIGHT_SHORT_SEQUENCE) {
            for ( sfb=0; sfb< min(max_sfb,
                MAX_LTP_LONG_SFB; sfb++ ) {
                ltp_long_used[sfb]                    1          uimsbf
            }
        }
```

In subclause 4.5.1.1 (GASpecificConfig()), remove:

However, there is one exception to this rule, which is described in subclause 4.6.14.1 for Table 4.112.

In subclause 4.5.1.1 (GASpecificConfig()), replace:

DependsOnCoreCoder Set to 1 if a coder at a different sampling rate is used as a core coder in a scalable bitstream.

CoreCoderDelay The delay in samples that has to be applied to the up-sampled core decoder output, before the MDCT calculation. To save memory it is also possible to delay the bitstream by the appropriate number of core frames instead. In that case it could be necessary to decode a core frame more than once.

with:

dependsOnCoreCoder Signals that a core coder has been used in an underlying base layer of a scalable AAC configuration.

coreCoderDelay The delay in samples that has to be applied to the up-sampled (if necessary) core decoder output, before the MDCT calculation.

In subclause 4.5.1 (Decoding of the GA specific configuration), replace:

An MPEG-4 Audio decoder is only required to follow the Program Config Element in GASpecificConfig(). The decoder shall ignore any Program Config Elements that may occur in raw data blocks. PCEs transmitted in raw data blocks cannot be used to convey decoder configuration information.

4.5.1.2 Program Config Element (PCE)

with:

4.5.1.2 program_config_element() (PCE)

A program_config_element() may occur outside the AAC payload e. g. as part of the GASpecificConfig() or the adif_header(), but also inside the AAC payload as syntactic element in a raw_data_block().

Note that the channel configuration given in a program_config_element() inside the AAC payload is evaluated only, if no channel configuration is given outside the AAC payload. In the context of ISO/IEC 14496-3 this is only the case for MPEG-4 ADTS with channel_configation==0.

In any case only one program may be configured at a certain time.

In subclause 4.5.1.2 (Program Config Element), replace:

4.5.1.2.1 Implicit and defined channel configurations

The AAC audio syntax provides two ways to convey the mapping of channels within a set of syntactic elements to physical locations of speakers. The first way is a default mapping based on the specific set of syntactic elements received and the order in which they are received. The most common mappings are further defined in subpart 1, Table 1.11. If MPEG-4 Audio is used together with the MPEG-4 Systems audio compositor only these mappings shall be used. If a mapping shown in subpart 1, Table 1.11 is not used, the following methods describe the default determination of channel mapping:

1) Any number of SCEs may appear (as long as permitted by other constraints, for example profile, level). If this number of SCEs is odd, then the first SCE represents the front center channel, and the other SCEs represent L/R pairs of channels, proceeding from center front outwards and back to center rear.

If the number of SCEs is even, then the SCEs are assigned as pairs as center-front L/R, in pairs proceeding out and back from center front toward center back.

2) Any number of CPEs or *PAIRS* of SCEs may appear. Each CPE or pair of SCEs represents one L/R pair, proceeding from where the first sets of SCEs left off, pairwise until reaching either center back pair.

3) Any number of SCEs may appear. If this number is even, allocating pairs of SCEs Left/Right, from 2), back to center back. If this number is odd, allocated as L/R pairs, except for the final SCE, which is assigned to center back.

4) Any number of LFEs may appear. No speaker mapping is defined in case of multiple LFEs.

In case of this default (or implicit) mapping the number and order of SCEs, CPEs and LFEs and the resulting configuration may not change within the bitstream without sending a program_config_element, i.e. an implicit reconfiguration is not allowed.

Other audio syntactic elements that do not imply additional output speakers, such as coupling channel_element(), may follow the listed set of syntactic elements. Obviously non-audio syntactic elements may be received in addition and in any order relative to the listed syntactic elements.

If reliable mapping of channel set to speaker geometry is a concern, then it is recommended that an implicit mapping from subpart 1, Table 1.11, or a Program Config Element be used.

For more complicated configurations a **Program Config Element** (PCE) is defined. The same restrictions apply with respect to the PCE as defined in ISO/IEC 13818-7. However, an MPEG-4 decoder is only required to parse PCEs in raw_data_blocks(), without interpreting them. Only the PCE provided within GASpecificConfig() describes the decoder configuration for the elementary stream under consideration. This implies that only one program can be configured at a certain time.

with:

4.5.1.2.1 Channel configuration

The AAC audio syntax provides three ways to convey the mapping of channels within a set of syntactic elements to physical locations of speakers. However in in the context of ISO/IEC 14496-3only two of them are permitted. However, in the context of ISO/IEC 14496-3 only two of them are permitted.

4.5.1.2.1.1 Explicit channel mapping using default channel settings

The most common mappings are defined in subpart 1, Table 1.11. If MPEG-4 Audio is used together with the MPEG-4 Systems audio compositor only these mappings shall be used.

4.5.1.2.1.2 Explicit channel mapping using a program_config_element()

Any possible channel configuration can be specified using a program_config_element(). The same specifications and restrictions as defined in ISO/IEC 13818-7 apply with respect to the PCE when used in the context of ISO/IEC 14496-3.

An MPEG-4 decoder is always required to parse any program_config_element() inside the AAC payload. However, the decoder is only required to evaluate it, if no channel configuration is given outside the AAC payload.

4.5.1.2.1.3 Implicit channel mapping

This kind of channel mapping as specified in ISO/IEC13818-7 is not permitted in the context of ISO/IEC 14496-3.

In subclause 4.5.2.1.4.1 (General), replace:

- The index of the highest non-zero spectral coefficient present in the element is 12

with:

- Only the lowest 12 spectral coefficients of any LFE may be non-zero

In subclause 4.5.2.2.1.1 (Help elements), replace:

last_max_sfb	max_sfb of the previous coding layer. Set to '4* no_of_dc_groups', if the previous layer is running at a different sampling rate, or is a non GA coder.

with:

last_max_sfb	max_sfb of the previous coding layer. If the previous layer is running at a different sampling rate or is a non GA coder, last_max_sfb is set to '4* no_of_dc_groups-1' if the window type is not SHORT_WINDOW, otherwise it is set to the lowest sfb covering all diff_short_lines.

In subclause 4.5.2.2.3 (Valid combinations of AAC with either TwinVQ or CELP), replace:

Three major classes of scalable configurations with AAC exist, depending on the coder types used:

1. AAC layers only
2. Narrow-band CELP base layer plus AAC
3. TwinVQ base layer plus AAC

with:

Three major classes of scalable configurations with AAC exist, depending on the coder types used:

1. AAC layers only
2. Narrow-band CELP base layer plus AAC
3. TwinVQ base layer plus AAC

In any configuration, the transmitted bandwidth (by means of max_sfb in the case of AAC and TwinVQ and by means of no_of_dc_groups or diff_short_lines in the case of CELP) of a certain layer must not be smaller than that of the preceding layer.

In subclause 4.5.2.2.5.2 (Frame length adaptation / super-frames), Table 4.61 (AAC frame length for 960 samples per frame and super-frame length of AAC/CELP combinations), remove the frame lengths given in brackets in all cells with the entry "AAC / CELP frames per super-frame".

In subclause 4.5.2.2.5.3 (CELP core coder with AAC running at 88.2 kHz, 44.1 kHz, or 22.05 kHz sampling rate), Table 4.62 (Super-frame parameters of AAC/CELP combinations at AAC sampling rates of 88.2 kHz, 44.1 kHz and 22.05 kHz), remove the frame lengths given in brackets in all cells with the entry "AAC / CELP frames per super-frame".

In subclause 4.5.2.3.1 (Definitions), replace:

ics_reserved_bit bit reserved for future use

with:

ics_reserved_bit flag reserved for future use. Shall be '0'.

In subclause 4.5.2.4.1 (Error sensitivity category assignment), Table 4.64 (Error sensitivity category assignement) replace:

CPE

with:

CPE / stereo layer

In subclause 4.5.2.4.2 (Category instances and its dependency structure), replace:

Note: Channel dependent information consists of individual_channel_stream() (ICS). As an exception, ltp_data() is treated as channel dependent information even if it is not part of ICS.

with:

Note: Channel dependent information consists of individual_channel_stream() (ICS). Exceptions:

- tp_dlata_present and ltp_data() are treated as channel dependent information even if they are not part of ICS.
- for the ER AAC Scalable object type, tns_data_present, tns_data(), diff_control_data() and diff_control_data_lr() are treated as channel dependent information even if they are not part of ICS.

In subclause 4.5.2.7.2 (Decoding process) replace:

The transport of the DRC information does not involve the MPEG-4 System layer but is handled completely within the GA bitstream elements instead. Furthermore, the dynamic range control processing is carried out within the GA decoder. No DRC related information is passed on to a subsequent audio compositor.

with:

The transport of the DRC information does not involve the MPEG-4 System layer but is handled completely within the GA data elements instead. Furthermore, the evaluation of potentially available dynamic range control information in the GA decoder is optional. No DRC related information is passed on to a subsequent audio compositor.

In subclause 4.5.2.7.2 (Decoding process) replace:

The following ordering principles are used to assign the exclude_mask to channel outputs:

- If a PCE is present (explicit speaker mapping), …
- For the case of an implicit speaker mapping (no PCE present), …

with:

The following ordering principles are used to assign the exclude_mask to channel outputs:

- If a PCE is present, …
- If no PCE is present, …

In subclause 4.5.2.7.2 (Decoding process) replace the pseudo code as follows:

```
#define FRAME_SIZE 1024 /* Change to 960 for 960-framing*/
bottom = 0;
drc_num_bands = 1;
if (drc_bands_present)
    drc_num_bands += drc_band_incr;
else
    drc_band_top[0] = FRAME_SIZE/4 - 1;
for (bd = 0; bd < drc_num_bands; bd++)  {
    top = 4 * (drc_band_top[bd] + 1);

    /* Decode DRC gain factor */
    if (dyn_rng_sgn[bd])
        factor = 2^(-ctrl1*dyn_rng_ctl[bd]/24);   /* compress */
    else
        factor = 2^(ctrl2*dyn_rng_ctl[bd]/24);    /* boost */

    /* If program reference normalization is done in the digital domain, modify
     * factor to perform normalization.
     * prog_ref_level can alternatively be passed to the system for modification
     * of the level in the analog domain. Analog level modification avoids problems
     * with reduced DAC SNR (if signal is attenuated) or clipping (if signal is boosted)
     */
    factor *= 0.5^((target_level-prog_ref_level)/24);

    /* Apply gain factor */
    for (i = bottom; i < top; i++)
        spec[i] *= factor;

    bottom = top;
}
```

In subclause 4.5.4 (Tables), replace the corresponding entries of Table 4.82 (scalefactor bands for a window length of 2048 and 1920 (values for 1920 in brackets) for LONG_WINDOW, LONG_START_WINDOW, LONG_STOP_WINDOW at 64 kHz) with:

fs [kHz]	64
num_swb_long_window	47 (46)
swb	swb_offset_long_window
46	984 (960)
	1024 (-)

In subclause 4.5.4 (Tables), Table 4.92 (AAC error sensitivity category assignment for main payload), remove:

1	1	1	ltp_short_lag	ltp_data()
1	1	1	ltp_short_lag_present	ltp_data()
1	1	1	ltp_short_used	ltp_data()

In subclause 4.5.4 (Tables), Table 4.92 (AAC error sensitivity category assignment for main payload), replace:

1	-	0	tns_channel_mono_layer	aac_scalable_main_header()

with:

-	-	0	tns_channel_mono_layer	aac_scalable_main_header()

In subclause 4.6.2.3.2 (Decoding of scalefactors), replace:

Note that scalefactors, sf[g][sfb], must be within the range of zero to 256, both inclusive.

with:

Note that scalefactors, sf[g][sfb], must be within the range of zero to 255, both inclusive.

In subclause 4.6.2.3.2 (Decoding of scalefactors), replace:

A decoded value of ±7 is used as ESC_FLAG. It signals that an escape value exists, that has to be added to +7 or subtracted from -7 in order to find the actual scalefactor value. This escape value is Huffman encoded.

with:

In the case of sf_escapes_present==1, a decoded value of ±7 is used as ESC_FLAG. It signals that an escape value exists, that has to be added to +7 or subtracted from -7 in order to find the actual scalefactor value. This escape value is Huffman encoded.

Replace the content of subclause 4.6.6 (Frequency domain prediction) with:

See ISO/IEC13818-7:2004, subclause 13.3.2 "Predictor Processing".

Notes:

The use of the prediction tool is object type / profile dependent. See subpart 1 for detailed information on the MPEG-4 Audio object types and profiles.

The frequency domain prediction tool can be used only for AudioObjectType 1 (AAC Main).

In 4.6.7.2 (Definitions), remove:

ltp_short_used	1 bit indicating whether LTP is used for each short window (1) or not (0)
ltp_short_lag_present	1 bit indicating whether **ltp_short_lag** is actually transmitted (1), or omitted (0) from the bitstream, which means that the value of **ltp_short_lag** is 0
ltp_short_lag	4-bit number specifying the relative delay for each short window to **ltp_lag** from -8 to 7

MAX_LTP_SHORT_SFB = 8 (for short frames)

In 4.6.7.2 (Definitions), replace:

ltp_coef	3-bit index indicating the LTP coefficient in the table below. For all short windows in the current frame, the same coefficient is always used.

with:

ltp_coef	3-bit index indicating the LTP coefficient in the table below.

In 4.6.7.3 (Decoding Process), remove:

In case of short windows the second control step defines which of the short windows in the coding block LTP is applied to.

The decoding process is different for long and short windows.

For each short window, the bit **ltp_short_used** is read from the bitstream. If the **ltp_short_used** is not set, the quantized value of spectral component is reconstructed directly from the transmitted data and the time domain signal can be reconstructed for this particular subframe. If the **ltp_short_used** is set, the **ltp_short_lag**_present is read. If **ltp_short_lag_present** is set then **ltp_short_lag** is read. If **ltp_short_lag_present** is not set, the value of **ltp_short_lag** is set to 0. The value of **ltp_short_lag** is combined with **ltp_lag** and **ltp_coef** to calculate the predicted time domain signal for this particular subframe. Using the MDCT for short windows, the predicted spectral components are calculated and the spectral components in the first eight scalefactor bands are added to the quantized prediction error reconstructed from the transmitted data.

In 4.6.7.3 (Decoding Process), remove:

```
else {
    for (w=0; w<num_windows; w++) {
        if(ltp_data_present && ltp_short_used[w] {
            x_est = predict();
            X_est = MDCT(x_est)
            for ( sfb=0; sfb<8; sfb++)
                X_rec = X_est + Y_rec;
            }
        else
            X_rec = Y_rec;
    }
}
```

In subclause 4.6.7.4 (Integration of LTP with other GA tools), add the following subclause:

4.6.7.4.3 LTP with dependently switched coupling

No dependently switched coupling and hence no dependently switched CCE is permitted in any audio object type that utilizes LTP.

In subclause 4.6.8.1 (M/S stereo), add the following subclause:

4.6.8.1.4 Integration of the M/S stereo tool for the audio object type AAC scalable

The same MS mask is applied to all layers. If subsequent layers specify an increasing max_sfb, ms_mask_present and ms_used[][] are transmitted for the additional scale factor bands and groups only (see Table 4.54 - Syntax of ms_data()).

In 4.6.8.2.3 (Decoding Process), replace:

The use of intensity stereo coding is signaled by the use of the pseudo codebooks INTENSITY_HCB and INTENSITY_HCB2 (15 and 14) in the right channel (use of these codebooks in a left channel of a channel pair element is illegal).

with:

The use of intensity stereo coding is signaled by the use of the pseudo codebooks INTENSITY_HCB and INTENSITY_HCB2 (15 and 14) only in the right channel of a channel_pair_element() having a common ics_info() (**common_window** == 1).

In subclause 4.6.13.3 (Decoding process), replace:

```
nrg = global_gain - NOISE_OFFSET - 256;
for (g=0; g<num_window_groups; g++)  {

    /* Decode noise energies for this group */
    for (sfb=0; sfb<max_sfb; sfb++)
        if (is_noise(g,sfb))
            noise_nrg[g][sfb] = nrg += dpcm_noise_nrg[g][sfb];

    /* Do perceptual noise substitution decoding */
    for (b=0; b<window_group_length[g]; b++)  {
```