INTERNATIONAL STANDARD

ISO/IEC 14496-3

First edition
1999-12-15

AMENDMENT 1
2000-09-15

Information technology — Coding of audio-visual objects —

Part 3: **Audio**

AMENDMENT 1: Audio extensions

Technologies de l'information — Codage des objets audiovisuels —

Partie 3: Codage audio

AMENDEMENT 1: Extensions audio

Reference number
ISO/IEC 14496-3:1999/Amd.1:2000(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.ch
Web www.iso.ch

Printed in Switzerland

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 3.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this Amendment may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Amendment 1 to International Standard ISO/IEC 14496-3:2000 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

Annexes A, B, C, D, E and F of this Amendment are for information only.

Introduction

MPEG-4 version 2 is an amendment to MPEG-4 version 1. This document contains the description of bitstream and decoder extensions related to new tools defined within MPEG-4 version 2. As long as nothing else is mentioned, the description made in MPEG-4 version 1 is not changed but only extended.

Overview

ISO/IEC 14496-3 (MPEG-4 Audio) is a new kind of audio standard that integrates many different types of audio coding: natural sound with synthetic sound, low bitrate delivery with high-quality delivery, speech with music, complex soundtracks with simple ones, and traditional content with interactive and virtual-reality content. By standardizing individually sophisticated coding tools as well as a novel, flexible framework for audio synchronization, mixing, and downloaded post-production, the developers of the MPEG-4 Audio standard have created new technology for a new, interactive world of digital audio.

MPEG-4, unlike previous audio standards created by ISO/IEC and other groups, does not target a single application such as real-time telephony or high-quality audio compression. Rather, MPEG-4 Audio is a standard that applies to *every* application requiring the use of advanced sound compression, synthesis, manipulation, or playback. The subparts that follow specify the state-of-the-art coding tools in several domains; however, MPEG-4 Audio is more than just the sum of its parts. As the tools described here are integrated with the rest of the MPEG-4 standard, exciting new possibilities for object-based audio coding, interactive presentation, dynamic soundtracks, and other sorts of new media, are enabled.

Since a single set of tools is used to cover the needs of a broad range of applications, *interoperability* is a natural feature of systems that depend on the MPEG-4 Audio standard. A system that uses a particular coder—for example, a real-time voice communication system making use of the MPEG-4 speech coding toolset—can easily share data and development tools with other systems, even in different domains, that use the same tool—for example, a voicemail indexing and retrieval system making use of MPEG-4 speech coding.

The following subclauses give a more detailed overview of the capabilities and functionalities provided with MPEG-4 Audio version 2.

New concepts

With this extension, new tools are added to the MPEG-4 standard, while none of the existing tools of version 1 is replaced. Version 2 is therefore fully backward compatible to version 1.

In the area of Audio, new tools are added in MPEG-4 version 2 to provide the following new functionalities:

- Error Robustness

The error robustness tools provide improved performance on error-prone transmission channels. They can be distinguished into codec specific error resilience tools and an common error protection tool.

Improved error robustness for AAC is provided by a set of error resilience tools. These tools reduce the perceived deterioration of the decoded audio signal that is caused by corrupted bits in the bitstream. The following tools are provided to improve the error robustness for several parts of an AAC frame:

- Virtual CodeBook tool (VCB11)
- Reversible Variable Length Coding tool (RVLC)
- Huffman Codeword Reordering tool (HCR)

Improved error robustness capabilities for all coding tools are provided through the error resilient bitstream payload syntax. It allows advanced channel coding techniques, which can be adapted to the special needs of the different coding tools. This error resilient bitstream payload syntax is mandatory for all version 2 object types.

The error protection tool (EP tool) provides unequal error protection (UEP) for MPEG-4 Audio in conjunction with the error resilient bitstream payload. UEP is an efficient method to improve the error robustness of source coding schemes. It is used by various speech and audio coding systems operating over error-prone channels such as mobile telephone networks or Digital Audio Broadcasting (DAB). The bits of the coded signal representation are first grouped into different classes according to their error sensitivity. Then error protection is individually applied to the different classes, giving better protection to more sensitive bits.

- Low-Delay Audio Coding

The MPEG-4 General Audio Coder provides very efficient coding of general audio signals at low bitrates. However it has an algorithmic delay of up to several 100ms and is thus not well suited for applications requiring low coding delay, such as real-time bi-directional communication. As an example, for the General Audio Coder operating at 24 kHz sampling rate and 24 kbit/s this results in an algorithmic coding delay of about 110 ms plus up to additional 210 ms for the bit reservoir. To enable coding of general audio signals with an algorithmic delay not exceeding 20 ms, MPEG-4 version 2 specifies a Low-Delay Audio Coder which is derived from MPEG-2/4 Advanced Audio Coding (AAC). It operates at up to 48 kHz sampling rate and uses a frame length of 512 or 480 samples, compared to the 1024 or 960 samples used in standard MPEG-2/4 AAC. Also the size of the window used in the analysis and synthesis filterbank is reduced by a factor of 2. No block switching is used to avoid the "look-ahead'" delay due to the block switching decision. To reduce pre-echo artefacts in case of transient signals, window shape switching is provided instead. For non-transient parts of the signal a sine window is used, while a so-called low overlap window is used in case of transient signals. Use of the bit reservoir is minimized in the encoder in order to reach the desired target delay. As one extreme case, no bit reservoir is used at all.

- Fine Grain Scalability

Bitrate scalability, also known as embedded coding, is a very desirable functionality. The General Audio Coder of version 1 supports large step scalability where a base layer bitstream can be combined with one or more enhancement layer bitstreams to utilize a higher bitrate and thus obtain a better audio quality. In a typical configuration, a 24 kbit/s base layer and two 16 kbit/s enhancement layers could be used, permitting decoding at a total bitrate of 24 kbit/s (mono), 40 kbit/s (stereo), and 56 kbit/s (stereo). Due to the side information carried in each layer, small bitrate enhancement layers are not efficiently supported in version 1. To address this problem and to provide efficient small step scalability for the General Audio Coder, the Bit-Sliced Arithmetic Coding (BSAC) tool is available in version 2. This tool is used in combination with the AAC coding tools and replaces the noiseless coding of the quantized spectral data and the scalefactors. BSAC provides scalability in steps of 1 kbit/s per audio channel, i.e. 2 kbit/s steps for a stereo signal. One base layer bitstream and many small enhancement layer bitstreams are used. The base layer contains the general side information, specific side information for the first layer and the audio data of the first layer. The enhancement streams contain only the specific side information and audio data for the corresponding layer. To obtain fine step scalability, a bit-slicing scheme is applied to the quantized spectral data. First the quantized spectral values are grouped into frequency bands. Each of these groups contains the quantized spectral values in their binary representation. Then the bits of a group are processed in slices according to their significance. Thus first all most significant bits (MSB) of the quantized values in a group are processed, etc. These bit-slices are then encoded using an arithmetic coding scheme to obtain entropy coding with minimal redundancy. Various arithmetic coding models are provided to cover the different statistics of the bit-slices. The scheme used to assign the bit-slices of the different frequency bands to the enhancement layer is constructed in a special way. This ensures that, with an increasing number of enhancement layers utilized by the decoder, quantized spectral data is refined by providing more of the less significant bits. But also the bandwidth is increased by providing bit-slices of the spectral data in higher frequency bands.

- Parametric Audio Coding

The Parametric Audio Coding tools combine very low bitrate coding of general audio signals with the possibility of modifying the playback speed or pitch during decoding without the need for an effects processing unit. In combination with the speech and audio coding tools of version 1, improved overall coding efficiency is expected for applications of object based coding allowing selection and/or switching between different coding techniques.

Parametric Audio Coding uses the Harmonic and Individual Lines plus Noise (HILN) technique to code general audio signals at bitrates of 4 kbit/s and above using a parametric representation of the audio signal. The basic idea of this technique is to decompose the input signal into audio objects which are described by appropriate source models and represented by model parameters. Object models for sinusoids, harmonic tones, and noise are utilized in the HILN coder.

This approach allows to introduce a more advanced source model than just assuming a stationary signal for the duration of a frame, which motivates the spectral decomposition used e.g. in the MPEG-4 General Audio Coder. As known from speech coding, where specialized source models based on the speech generation process in the human vocal tract are applied, advanced source models can be advantageous in particular for very low bitrate coding schemes.

Due to the very low target bitrates, only the parameters for a small number of objects can be transmitted. Therefore a perception model is employed to select those objects that are most important for the perceptual quality of the signal.

In HILN, the frequency and amplitude parameters are quantized according to the “just noticeable differences” known from psychoacoustics. The spectral envelope of the noise and the harmonic tone is described using LPC modeling as known from speech coding. Correlation between the parameters of one frame and between consecutive frames is exploited by parameter prediction. The quantized parameters are finally entropy coded and multiplexed to form a bitstream.

A very interesting property of this parametric coding scheme arises from the fact that the signal is described in terms of frequency and amplitude parameters. This signal representation permits speed and pitch change functionality by simple parameter modification in the decoder. The HILN Parametric Audio Coder can be combined with MPEG-4 Parametric Speech Coder (HVXC) to form an integrated parametric coder covering a wider range of signals and bitrates. This integrated coder supports speed and pitch change. Using a speech/music classification tool in the encoder, it is possible to automatically select the HVXC for speech signals and the HILN for music signals. Such automatic HVXC/HILN switching was successfully demonstrated and the classification tool is described in the informative Annex of the version 2 standard.

- CELP Silence Compression

The silence compression tool reduces the average bitrate thanks to compression at a lower-bitrate for silence. In the encoder, a voice activity detector is used to distinguish between regions with normal speech activity and those with silence or background noise. During normal speech activity, the CELP coding as in version 1 is used. Otherwise a Silence Insertion Descriptor (SID) is transmitted at a lower bitrate. This SID enables a Comfort Noise Generator (CNG) in the decoder. The amplitude and the spectral shape of this comfort noise are specified by energy and LPC parameters in similar methods to those in a normal CELP frame. These parameters are optionally re-transmitted in the SID and thus can be updated as required.

- Extended HVXC

In the Version 1 HVXC, variable bitrate mode of 2.0 kbit/s maximum is supported as well as 2.0 and 4.0 kbit/s fixed bitrate modes. In the Version 2 Error Resilient (ER) HVXC, the variable bitrate mode of 4.0 kbit/s maximum is additionally supported. The ER HVXC therefore provides fixed bitrate modes (2.0-4.0kbit/s) and variable bitrate mode(<2.0kbit/s, <4.0kbit/s) both in a scalable and non-scalable scheme. In the variable bitrate modes, non-speech parts are detected in unvoiced signals, and a smaller number of bits is used for these non-speech parts to reduce the average bitrate. ER HVXC provides communications-quality to near-toll-quality speech in the 100-3800 Hz band at 8kHz sampling rate. When the variable bitrate mode is allowed, operation at lower average bitrate is possible. Coded speech with variable bitrate mode at typical bitrate of 1.5kbit/s average, and at typical bitrate of 3.0kbit/s average has essentially the same quality as 2.0 kbit/s fixed rate and 4.0 kbit/s fixed rate respectively. The functionality of pitch and speed change during decoding is supported for all modes. ER HVXC has the syntax with the error sensitivity classes to be used with the EP-Tool, and the error concealment functionality is supported for the use for error-prone channel like mobile communication channels. The ER HVXC speech coder targets applications from mobile and satellite communications, to Internet telephony, to packaged media and speech databases.

Capabilities

Overview of capabilities

MPEG-4 Audio version 2 provides the following new capabilities:

- error robustness (including error resilience as well as error protection)
- low delay audio coding
- backchannel
- fine granule scalability
- parametric audio
- silence compression in CELP
- extended HVXC

Those new capabilities are discussed in more detail below.

Error robustness

Error resilience tools for AAC

Several tools are provided to increase the error resilience for AAC. These tools improve the perceptual audio quality of the decoded audio signal in case of corrupted bitstreams, which may occur e. g. in the presence of noisy transmission channels.

The Virtual CodeBooks tool (VCB11) extends the sectioning information of an AAC bitstream. This permits to detect serious errors within the spectral data of an MPEG-4 AAC bitstream. Virtual codebooks are used to limit the largest absolute value possible within a certain scalefactor band where escape values are. While referring to the same codes as codebook 11, the sixteen virtual codebooks introduced by this tool provide sixteen different limitations of the spectral values belonging to the corresponding section. Due to this, errors within spectral data resulting in spectral values exceeding the indicated limit can be located and appropriately concealed.

The Reversible Variable Length Coding tool (RVLC) replaces the Huffman and DPCM coding of the scalefactors in an AAC bitstream. The RVLC uses symmetric codewords to enable both forward and backward decoding of the scalefactor data. In order to have a starting point for backward decoding, the total number of bits of the RVLC part of the bitstream is transmitted. Because of the DPCM coding of the scalefactors, also the value of the last scalefactor is transmitted to enable backward DPCM decoding. Since not all nodes of the RVLC code tree are used as codewords, some error detection is also possible.

The Huffman codeword reordering (HCR) algorithm for AAC spectral data is based on the fact that some of the codewords can be placed at known positions so that these codewords can be decoded independent of any error within other codewords. Therefore, this algorithm avoids error propagation to those codewords, the so-called priority codewords (PCW). To achieve this, segments of known length are defined and those codewords are placed at the beginning of these segments. The remaining codewords (non-priority codewords, non-PCW) are filled into the gaps left by the PCWs using a special algorithm that minimizes error propagation to the non-PCWs codewords. This reordering algorithm does not increase the size of spectral data. Before applying the reordering algorithm itself, a pre-sorting process is applied to the codewords. It sorts all codewords depending on their importance, i. e. it determines the PCWs.

Error protection

The EP tool provides unequal error protection. It receives several classes of bits from the audio coding tools, and then applies forward error correction codes (FEC) and/or cyclic redundancy codes (CRC) for each class, according to its error sensitivity.

The error protection tool (EP tool) provides the unequal error protection (UEP) capability to the ISO/IEC 14496-3 codecs. Main features of this tool are:

- providing a set of error correcting/detecting codes with wide and small-step scalability, in performance and in redundancy
- providing a generic and bandwidth-efficient error protection framework, which covers both fixed-length frame bitstreams and variable-length frame bitstreams
- providing a UEP configuration control with low overhead

Error resilient bitstream reordering

Error resilient bitstream reordering allows the effective use of advanced channel coding techniques like unequal error protection (UEP), that can be perfectly adapted to the needs of the different coding tools. The basic idea is to rearrange the audio frame content depending on its error sensitivity in one or more instances belonging to different error sensitivity categories (ESC). This rearrangement works either data element-wise or even bit-wise. An error resilient bitstream frame is build by concatenating these instances.

Low delay

The low delay coding functionality provides the ability to extend the usage of generic low bitrate audio coding to applications requiring a very low delay of the encoding / decoding chain (e.g. full-duplex real-time communications). In contrast to traditional low delay coders based on speech coding technology, the concept of this low delay coder is based on general perceptual audio coding and is thus suitable for a wide range of audio signals. Specifically, it is derived closely from the proven architecture of MPEG-2/4 Advanced Audio Coding (AAC). Furthermore, all capabilities for coding of 2 (stereo) or more sound channels (multi-channel) are available within the low delay coder as inherited from Advanced Audio Coding.

Upstream

To allow for user on a remote side to dynamically control the streaming of the server, backchannel streams carrying user interaction information are defined.

Fine granule scalability in audio

BSAC provides fine grain scalability in steps of 1 kbit/s per audio channel, i.e. 2 kbit/s steps for a stereo signal. One base layer bitstream and many small enhancement layer bitstreams are used. In order to implement the fine grain scalability efficiently in MPEG-4 system, the fine grain audio data can be divided into the large-step layers and the large-step layers are concatenated from the several sub-frames. And the configuration of the payload transmitted over Elementary Stream (ES) can be changed dynamically depending on the environment such as the network traffic or the user interaction. So, BSAC can allow for real-time adjustments to the quality of service.

In addition to fine grain scalablity, it can improve the quality of the audio signal which is decoded from the bitstreams transmitted over error-prone channels such as mobile communication networks or Digital Audio Broadcasting (DAB)

HILN: harmonic and individual lines plus noise (parametric audio coding)

MPEG-4 parametric audio coding uses the HILN technique (Harmonic and Individual Lines plus Noise) to code non-speech signals like music at bitrates of 4 kbit/s and higher using a parametric representation of the audio signal. HILN allows independent change of speed and pitch during decoding. Furthermore HILN can be combined with MPEG-4 parametric speech coding (HVXC) to form an integrated parametric coder covering a wider range of signals and bitrates.

Silence compression for CELP

The silence compression tool comprises a Voice Activity Detector (VAD), a Discontinuous Transmission (DTX) unit and a Comfort Noise Generator (CNG) module. The tool encodes/decodes the input signal at a lower bitrates during the non-active-voice (silent) frames. During the active-voice (speech) frames, MPEG-4 CELP encoding and decoding are used.

Extension of HVXC

The operation of maximum of 4.0 kbit/s variable bitrate mode of the MPEG-4 parametric speech coder HVXC is provided in addition to the Version1 HVXC functionalities including 2.0-4.0 kbit/s fixed bitrate mode and 2.0 kbit/s maximum variable bitrate mode. Here extended operation of the variable bitrate mode with 4.0 kbit/s maximum is provided which allows higher quality variable rate coding.

Information technology — Coding of audio-visual objects —

Part 3:
Audio

Amendment 1: Audio extensions

1 Scope

ISO/IEC 14496-3 (MPEG-4 Audio) is a new kind of audio standard that integrates many different types of audio coding: natural sound with synthetic sound, low bitrate delivery with high-quality delivery, speech with music, complex soundtracks with simple ones, and traditional content with interactive and virtual-reality content. By standardizing individually sophisticated coding tools as well as a novel, flexible framework for audio synchronization, mixing, and downloaded post-production, the developers of the MPEG-4 Audio standard have created new technology for a new, interactive world of digital audio.

MPEG-4, unlike previous audio standards created by ISO/IEC and other groups, does not target a single application such as real-time telephony or high-quality audio compression. Rather, MPEG-4 Audio is a standard that applies to *every* application requiring the use of advanced sound compression, synthesis, manipulation, or playback. The subparts that follow specify the state-of-the-art coding tools in several domains; however, MPEG-4 Audio is more than just the sum of its parts. As the tools described here are integrated with the rest of the MPEG-4 standard, exciting new possibilities for object-based audio coding, interactive presentation, dynamic soundtracks, and other sorts of new media, are enabled.

Since a single set of tools is used to cover the needs of a broad range of applications, *interoperability* is a natural feature of systems that depend on the MPEG-4 Audio standard. A system that uses a particular coder—for example, a real-time voice communication system making use of the MPEG-4 speech coding toolset—can easily share data and development tools with other systems, even in different domains, that use the same tool—for example, a voicemail indexing and retrieval system making use of MPEG-4 speech coding.

2 Normative references

The following normative documents contain provisions which, through reference in this text, constitute provisions of this Amendment. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this Amendment are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.

ISO/IEC 11172-3:1993, *Information technology - Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s - Part 3: Audio.*

ITU-T Rec.H.222.0(1995) | ISO/IEC 13818-1:1996, *Information technology – Generic coding of moving pictures and associated audio information: Systems.*

ISO/IEC 13818-3:1998, *Information technology – Generic coding of moving pictures and associated audio information - Part 3: Audio.*

ISO/IEC 13818-7:1997, *Information technology – Generic coding of moving pictures and associated audio information - Part 7: Advanced Audio Coding (AAC).*

ITU-T Recommendation H.223 (1998), Annex C, Multiplexing protocol for low bit-rate multimedia communication over highly error-prone channels.

3 Terms and definitions

3.1 virtual codebook – If several codebook values refer to one and the same physical codebook, these values are called virtual codebooks.

See also ISO/IEC 14496-3 Subpart 1 Main.

4 Symbols and abbreviations

RVLC – Reversible Variable Length Coding

VCB11 – Virtual Codebooks for codebook 11

EP – Error Protection

ER – Error Resilience or Error Resilient (as appropriate)

HCR – Huffman Codeword Reordering

HILN – Harmonic and Individual Lines plus Noise

BSAC – Bit Sliced Arithmetic Coding

See also ISO/IEC 14496-3 Subpart 1 Main.

5 Technical overview

5.1 Extended MPEG-4 audio object types

5.1.1 Audio object type definition

Table 1: Audio Object Type definition

Tools / Audio Object Type	13818-7 main	13818-7 LC	13818-7 SSR	PNS	LTP	TLSS	Twin VQ	CELP	HVXC	TTSI	SA tools	SASBF	MIDI	BSAC	HILN	Low Delay AAC	HVXC 4kbs VR	Silence Compression	Error Robust	Hierarchy	Object Type ID
Null																					0
AAC main	X			X																contains AAC LC	1
AAC LC		X		X																	2
AAC SSR			X	X																	3
AAC LTP		X		X	X															contains AAC LC	4

(Reserved)																				5
AAC Scalable		X		X	X	X														6
TwinVQ					X		X													7
CELP								X												8
HVXC									X											9
(Reserved)																				10
(Reserved)																				11
TTSI										X										12
Main synthetic											X	X	X						contains W/T & Algor. synthesis	13
Wavetable synthesis												X	X						contains General MIDI	14
General MIDI													X							15
Algorithmic Synthesis and Audio FX											X									16
ER AAC LC		X		X														X		17
(Reserved)																				18
ER AAC LTP		X		X	X													X	contains ER AAC LC	19
ER AAC scalable		X		X		X												X		20
ER TwinVQ							X											X		21
ER BSAC				X										X				X		22
ER AAC LD				X	X											X		X		23
ER CELP								X									X	X		24
ER HVXC									X							X		X		25
ER HILN															X			X		26
ER Parametric									X						X		X		X		27
(Reserved)																				28
(Reserved)																				29
(Reserved)																				30
(Reserved)																				31

Note: The error-robust tool is composed of error resilience and error protection. With respect ot the decoder it is mandatory to equip the bit parsing function for error protection. However, it is optional to have the error detection and correction function.

5.1.2 Description

5.1.2.1 NULL object type

See ISO/IEC 14496-3 Subpart 1 Main.

5.1.2.2 AAC Main object type

See ISO/IEC 14496-3 Subpart 1 Main.

5.1.2.3 AAC Low Complexity (LC) Object type

See ISO/IEC 14496-3 Subpart 1 Main.

5.1.2.4 AAC Scalable Sampling Rate (SSR) object type

See ISO/IEC 14496-3 Subpart 1 Main.

5.1.2.5 AAC Long Term Predictor (LTP) object type

See ISO/IEC 14496-3 Subpart 1 Main.

5.1.2.6 AAC Scalable object type

See ISO/IEC 14496-3 Subpart 1 Main.

5.1.2.7 TwinVQ object type

See ISO/IEC 14496-3 Subpart 1 Main.

5.1.2.8 CELP object type

See ISO/IEC 14496-3 Subpart 1 Main.

5.1.2.9 HVXC object type

See ISO/IEC 14496-3 Subpart 1 Main.

5.1.2.10 TTSI object type

See ISO/IEC 14496-3 Subpart 1 Main.

5.1.2.11 Main Synthetic object type

See ISO/IEC 14496-3 Subpart 1 Main.

5.1.2.12 Wavetable Synthesis object type

See ISO/IEC 14496-3 Subpart 1 Main.

5.1.2.13 General MIDI object type

See ISO/IEC 14496-3 Subpart 1 Main.

5.1.2.14 Algorithmic Synthesis and Audio FX object type

See ISO/IEC 14496-3 Subpart 1 Main.

5.1.2.15 Error Resilient (ER) AAC Low Complexity (LC) object type

The Error Resilient (ER) MPEG-4 AAC Low Complexity object type is the counterpart to the MPEG-4 AAC Low Complexity object, with additional error resilient functionality.

5.1.2.16 Error Resilient (ER) AAC Long Term Predictor (LTP) object type

The Error Resilient (ER) MPEG-4 AAC LTP object type is the counterpart to the MPEG-4 AAC LTP object, with additional error resilient functionality.

5.1.2.17 Error Resilient (ER) AAC scalable object type

The Error Resilient (ER) MPEG-4 AAC scalable object type is the counterpart to the MPEG-4 AAC scalable object, with additional error resilient functionality.

5.1.2.18 Error Resilient (ER) TwinVQ object type

The Error Resilient (ER) TwinVQ object type is the counterpart to the MPEG-4 TwinVQ object, with additional error resilient functionality.

5.1.2.19 Error Resilient (ER) BSAC object type

The ER BSAC object is supported by the fine grain scalablility tool (BSAC: Bit-Sliced Arithmetic Coding). It provides error resilience as well as fine step scalability in the MPEG-4 General Audio (GA) coder. It is used in combination with the AAC coding tools and replaces the noiseless coding and the bitstream formatting of MPEG-4 version 1 GA coder A large number of scalable layers are available, providing 1 kbit/s/ch enhancement layer, i.e. 2 kbit/s steps for a stereo signal.

5.1.2.20 Error Resilient (ER) AAC LD object type

The AAC LD object is supported by the low delay AAC coding tool. It also permits combinations with the PNS tool and the LTP tool. AAC LD object provides the ability to extend the usage of generic low bitrate audio coding to applications requiring a very low delay of the encoding / decoding chain (e.g. full-duplex real-time communications).

5.1.2.21 Error Resilient (ER) CELP object type

The ER CELP object is supported by silence compression and ER tools. It provides the ability to reduce the average bitrate thanks to a lower-bitrate compression for silence, with additional error resilient functionality.

5.1.2.22 Error Resilient (ER) HVXC object type

The ER HVXC object is supported by the parametric speech coding (HVXC) tools, which provide fixed bitrate modes (2.0-4.0 kbit/s) and variable bitrate modes (< 2.0 kbit/s and < 4.0 kbit/s) both in a scalable and a non-scalable scheme, and the functionality of pitch and speed change. The syntax to be used with the EP-Tool, and the error concealment functionality are supported for the use for error-prone channels. Only 8 kHz sampling rate and mono audio channel are supported.

5.1.2.23 Error Resilient (ER) HILN object type

The ER HILN object is supported by the parametric audio coding tools (HILN: Harmonic and Individual Lines plus Noise) which provide coding of general audio signals at very low bitrates ranging from below 4 kbit/s to above 16 kbit/s. Bitrate scalability and the functionality of speed and pitch change are available. The ER HILN object supports mono audio objects at a wide range of sampling rates.

5.1.2.24 Error Resilient (ER) Parametric object type

The ER Parametric object is supported by the parametric audio coding and speech coding tools HILN and HVXC. This integrated parametric coder combines the functionalities of the ER HILN and the ER HVXC objects. Only 8 kHz sampling rate and mono audio channel are supported.

5.2 Audio profiles and levels

5.2.1 Profiles

In ISO/IEC 14496-3 Amd 1, four new profiles are defined.

The **High Quality Audio Profile** contains the CELP speech coder and the Low Complexity AAC coder including Long Term Prediction. Scalable coding coding can be performed by the AAC Scalable object type. Optionally, the new error resilient (ER) bitstream syntax may be used. The **Low Delay Audio Profile** contains the HVXC and CELP speech coders (optionally using the ER bitstream syntax), the low-delay AAC coder and the Text-to-Speech interface TTSI. The **Natural Audio Profile** contains all natural audio coding tools available in MPEG-4. The **Mobile Audio Internetworking Profile** contains the low-delay and scalable AAC object types including TwinVQ and BSAC. This profile is intended to extend communication applications using non-MPEG speech coding algorithms with high quality audio coding capabilities.

Table 2: Audio Profiles definition

Audio Object Type	High Quality Audio Profile	Low Delay Audio Profile	Natural Audio Profile	Mobile Audio Internetworking Profile	Object Type ID
Null					0
AAC main			X		1
AAC LC	X		X		2
AAC SSR			X		3
AAC LTP	X		X		4
(reserved)					5
AAC Scalable	X		X		6
TwinVQ			X		7
CELP	X	X	X		8
HVXC		X	X		9
(reserved)					10
(reserved)					11
TTSI		X	X		12
Main synthetic					13
Wavetable synthesis					14
General MIDI					15
Algorithmic Synthesis and Audio FX					16
ER AAC LC	X		X	X	17
(reserved)					18
ER AAC LTP	X		X		19
ER AAC Scalable	X		X	X	20
ER TwinVQ			X	X	21
ER BSAC			X	X	22
ER AAC LD		X	X	X	23
ER CELP	X	X	X		24
ER HVXC		X	X		25
ER HILN			X		26
ER Parametric			X		27
(reserved)					28
(reserved)					29
(reserved)					30
(reserved)					31

5.2.2 Complexity units

Complexity units are defined to give an approximation of the decoder complexity in terms of processing power and RAM usage required for processing MPEG-4 Audio bitstreams in dependence of specific parameters.

The approximated processing power is given in “Processor Complexity Units” (PCU), specified in integer numbers of MOPS. The approximated RAM usage is given in “RAM Complexity Units” (RCU), specified in mostly integer numbers of kWords (1000 words). The RCU numbers do not include working buffers that can be shared between different objects and/or channels.

The following Table 3 gives complexity estimates for the different object types:

Table 3: Complexity of Audio Object Types

Object Type	Parameters	PCU (MOPS)	RCU (kWords)	Remarks
ER AAC LC	fs=48kHz	3	3	3)
ER AAC SSR	fs=48kHz	4	3	3)
ER AAC LTP	fs=48kHz	4	4	3)
ER AAC Scalable	fs=48kHz	5	4	3), 4)
ER TwinVQ	fs=24kHz	2	3	3)
ER BSAC	fs=48kHz (input buffer size=26000bits)	4	4	3)
	fs=48kHz (input buffer size=106000bits)	4	8	
ER AAC LD	fs=48kHz	3	2	3)
ER CELP	fs=8kHz	2	1	
	fs=16kHz	3	1	
ER HVXC	fs=8kHz	2	1	
ER HILN	fs=16kHz, ns=93	15	2	2)
	fs=16kHz, ns=47	8	2	
ER Parametric	fs=8kHz, ns=47	4	2	1), 2)

Definitions:

- fs = sampling frequency
- ns = max. number of sinusoids being synthesized, see below

Notes:

1) Parametric coder in HILN mode, for HVXC mode see ER HVXC.

2) PCU depends on fs and ns, see below

3) PCU proportional to sampling frequency

4) Includes core decoder

PCU for HILN:

The computational complexity of HILN depends on the sampling frequency fs and the maximum number of sinusoids ns to be synthesized simultaneously. The value of ns for a frame is the total number of harmonic and individual lines synthesized in that frame, i.e. the number of starting plus continued plus ending lines. For fs in kHz, the PCU in MOPS is calculated as follows:

PCU = (1 + 0.15*ns) * fs/16

The typical maximum values of ns are 47 for 6 kbit/s HILN and 93 for 16 kbit/s HILN bitstreams.

5.2.3 Levels within the profiles

A number of 0 stages of interleaving for the EP-tool indicates that the EP is not used in that particular level. The notation used to specify the number of audio channels indicates the number of full bandwidth channels and the number of low-frequency enhancement channels. For example, “5.1” indicates 5 full bandwidth channels and one low-frequency enhancement channel.

Table 4: Levels for the High Quality Audio Profile

Level	Max. channels/object	Max. sampling rate [kHz]	Max PCU *2	Max RCU *2	EP-Tool: Max. redundancy by class FEC *1	EP-Tool: Max. # stages of interleaving per object
1	2	22.05	5	8	0 %	0
2	2	48	10	8	0 %	0
3	5.1	48	25	12 *3	0 %	0
4	5.1	48	100	42 *3	0 %	0
5	2	22.05	5	8	20%	9
6	2	48	10	8	20%	9
7	5.1	48	25	12 *3	20%	22
8	5.1	48	100	42 *3	20%	22

*1: This number does not cover FEC for the EP header, i. e. FEC for the EP header is always permitted. In case of several audio objects the limit is valid independently for each audio object. This value is the maximum redundancy for the Audio object, which has the longest frame length, in each profile & level.

*2: Level 5 to 8 do not include RAM and computational complexity for the EP tool.

*3: Sharing of work buffers between multiple objects or channel pair elements is assumed.

Table 5: Levels for the Low Delay Audio Profile

Level	Max. channels/object	Max. sampling rate [kHz]	Max PCU *2	Max RCU *2	EP-Tool: Max. redundancy by class FEC *1	EP-Tool: Max. # stages of interleaving per object
1	1	8	2	1	0 %	0
2	1	16	3	1	0 %	0
3	1	48	3	2	0 %	0
4	2	48	24	12 *3	0 %	0
5	1	8	2	1	100%	5
6	1	16	3	1	100%	5
7	1	48	3	2	20%	5
8	2	48	24	12 *3	20%	9

*1: This number does not cover FEC for the EP header, i. e. FEC for the EP header is always permitted. In case of several audio objects the limit is valid independently for each audio object. This value is the maximum redundancy for the Audio object, which has the longest frame length, in each profile & level.

*2: Level 5 to 8 do not include RAM and computational complexity for the EP tool.

*3: Sharing of work buffers between multiple objects or channel pair elements is assumed.

Table 6: Levels for the Natural Audio Profile

Level	Max. sampling rate [kHz]	Max PCU *2	EP-Tool: Max. redundancy by class FEC *1	EP-Tool: Max. # stages of interleaving per object
1	48	20	0 %	0
2	96	100	0 %	0
3	48	20	20%	9
4	96	100	20%	22

*1: This number does not cover FEC for the EP header, i. e. FEC for the EP header is always permitted. In case of several audio objects the limit is valid independently for each audio object. This value is the maximum redundancy for the Audio object, which has the longest frame length, in each profile & level.

*2: Level 3 and 4 do not include computational complexity for the EP tool.

No RCU limitations are specified for this profile.

Table 7: Levels for the Mobile Audio Internetworking Profile

Level	Max. channels/object	Max. sampling rate [kHz]	Max PCU *3	Max RCU *2 *3	Max. # audio objects	EP-Tool: Max. redundancy by class FEC *1	EP-Tool: Max. # stages of interleaving per object
1	1	24	2.5	4	1	0 %	0
2	2	48	10	8	2	0 %	0
3	5.1	48	25	12 *4	-	0 %	0
4	1	24	2.5	4	1	20%	5
5	2	48	10	8	2	20%	9
6	5.1	48	25	12 *4	-	20%	22

*1: This number does not cover FEC for the EP header, i. e. FEC for the EP header is always permitted. In case of several audio objects the limit is valid independently for each audio object. This value is the maximum redundancy for the Audio object, which has the longest frame length, in each profile & level.

*2: The maximum RCU for one channnel in any object in this profile is 4. For the ER BSAC, this limits the input buffer size. The maximum possible input buffer size in bits for this case is given in PCU/RCU Table 3.

*3: Level 4 to 6 do not include RAM and computational complexity for the EP tool.

*4: Sharing of work buffers between multiple objects or channel pair elements are assumed.

6 Extension to interface to ISO/IEC 14496-1 (MPEG-4 Systems)

6.1 Introduction

The header streams are transported via MPEG-4 systems. These streams contain configuration information, which is necessary for the decoding process and parsing of the raw data streams. However, an update is only necessary if there are changes in the configuration.

The payloads contain all information varying on a frame to frame basis and therefore carry the actual audio information.

6.2 Extension to syntax

6.2.1 AudioSpecificInfo

Table 8: Syntax of AudioSpecificInfo()

Syntax	No. of bits	Mnemonic
AudioSpecificInfo ()		
{		
audioObjectType;	**5**	**bslbf**
samplingFrequencyIndex;	**4**	**bslbf**
if (samplingFrequencyIndex==0xf)		
samplingFrequency;	**24**	**uimsbf**
channelConfiguration;	**4**	**bslbf**
if (audioObjectType == 1 || audioObjectType == 2 ||		
audioObjectType == 3 || audioObjectType == 4 ||		
audioObjectType == 6 || audioObjectType == 7)		
GASpecificConfig();		
if (audioObjectType == 8)		
CelpSpecificConfig();		
if (audioObjectType == 9)		
HvxcSpecificConfig();		
if (audioObjectType == 12)		
TTSSpecificConfig();		
if (audioObjectType == 13 || audioObjectType == 14 ||		
audioObjectType == 15 || audioObjectType==16)		
StructuredAudioSpecificConfig();		
/* the following Objects are Amendment 1 Objects */		
if (audioObjectType == 17 || audioObjectType == 19 ||		
audioObjectType == 20 || audioObjectType == 21 ||		
audioObjectType == 22 || audioObjectType == 23)		
GASpecificConfig();		
if (audioObjectType == 24)		
ErrorResilientCelpSpecificConfig();		
if (audioObjectType == 25)		
ErrorResilientHvxcSpecificConfig();		
if (audioObjectType == 26 || audioObjectType == 27)		
ParametricSpecificConfig();		
if (audioObjectType == 17 || audioObjectType == 19 ||		
audioObjectType == 20 || audioObjectType == 21 ||		
audioObjectType == 22 || audioObjectType == 23 ||		
audioObjectType == 24 || audioObjectType == 25 ||		
audioObjectType == 26 || audioObjectType == 27) {		
epConfig;	**2**	**bslbf**
if (epConfig == 2)		
ErrorProtectionSpecificConfig();		
}		
}		

6.2.1.1 HvxcSpecificConfig

Defined in ISO/IEC 14496-3 subpart 2.

6.2.1.2 CelpSpecificConfig

Defined in ISO/IEC 14496-3 subpart 3.

6.2.1.3 GASpecificConfig

Defined in subclause 8.2.1

6.2.1.4 StructuredAudioSpecificConfig

Defined in ISO/IEC 14496-3 subpart 5.

6.2.1.5 TTSSpecificConfig

Defined in ISO/IEC 14496-3 subpart 6.

6.2.1.6 ParametricSpecificConfig

Defined in subclause 7.3.1.

6.2.1.7 ErrorProtectionSpecificConfig

Defined in subclause 9.2.1.

6.2.1.8 ErrorResilientCelpSpecificConfig

Defined in subclause 11.4.

6.2.1.9 ErrorResilientHvxcSpecificConfig

Defined in subclause 12.3.1.

6.2.2 Payloads

For the NULL object the payload shall be 16 bit signed integer in the range from -32768 to +32767. The payloads for all other audio object types are defined in the corresponding parts. These are the basic entities to be carried by the systems transport layer. Note that for all natural audio coding schemes the output is scaled for a maximum of 32767/-32768. However, the MPEG-4 System compositor expects a scaling.

The following table shows an overview about where the Elementary Stream payloads for the error resilient Audio Object Types can be found and where the detailed syntax is defined.

Table 9: ER Audio Object Types

ER Audio Object Type	definition of elementary stream payloads	detailed syntax definition
ER AAC LC	subclause 10.5	ISO/IEC 14496-3 subpart 4 and subclause 8.5
ER AAC LTP	subclause 10.5	ISO/IEC 14496-3 subpart 4 and subclause 8.5
ER AAC scalable	subclause 10.5	ISO/IEC 14496-3 subpart 4 and subclause 8.5
ER Twin VQ	subclause 10.4	ISO/IEC 14496-3 subpart 4
ER BSAC	subclause 10.7	subclause 8.3.3.1
ER AAC LD	subclause 10.5	ISO/IEC 14496-3 subpart 4 and subclause 8.4
ER CELP	subclause 10.2	ISO/IEC 14496-3 subpart 3 and subclause 11.4
ER HVXC	subclause 10.3	ISO/IEC 14496-3 subpart 2 and subclause 12.3.2
ER HILN	subclause 10.6	subclause 7.3.2
ER Parametric	subclause 10.6	ISO/IEC 14496-3 subpart 2 and subclause 7.3.2

6.3 Semantics

6.3.1 audioObjectType

A five bit field indicating the audio object type. This is the master switch which selects the actual bitstream syntax of the audio data. In general, different object type use a different bitstream syntax. The interpretation of this field is given in Table 1

6.3.2 samplingFrequency

See ISO/IEC 14496-3 Subpart 1 Main.

6.3.3 samplingFrequencyIndex

See ISO/IEC 14496-3 Subpart 1 Main.

6.3.4 channelConfiguration

See ISO/IEC 14496-3 Subpart 1 Main.

6.3.5 epConfig

This variable signals what kind of error robust configuration is used, i. e. how instances of error sensitivity categories are obtained on decoder site.

Table 10: epConfig

epConfig	Description
0	All instances of all sensitivity categories belonging to one frame are stored within one access unit.
1	Each instance of each sensitivity category belonging to one frame is stored separately within a single access unit, i.e. there are as many elementary streams existent as instances defined within a frame.
2	The error protection decoder has to be applied. Its input is an error protected access unit and its output are several error protection class instances. Each instance of each sensitivity category belonging to one frame corresponds to one of these error protection class instances.
3	Reserved

6.4 Upstream

6.4.1 Introduction

Upstreams are defined to allow for user on a remote side to dynamically control the streaming of the server.

The need for an up-stream channel is signaled to the client terminal by supplying an appropriate elementary stream descriptor declaring the parameters for that stream. The client terminal opens this up-stream channel in a similar manner as it opens the downstream channels. The entities (e.g. media encoders & decoders) that are connected through an up-stream channel are known from the parameters in its elementary stream descriptor and from the association of the elementary stream descriptor to a specific object descriptor.

An up-stream can be associated to a single downstream or a group of down streams. The stream type of the downstream to which the up-stream is associated defines the scope of the up-stream. When the up-stream is associated to a single downstream it carries messages about the downstream it is associated to. The syntax and semantics of messages for MPEG-4 Audio are defined in the next subclause.

6.4.2 Syntax

Table 11: Syntax of AudioUpstreamPayload()

Syntax	No. of bits	Mnemonic
AudioUpstreamPayload()		
{		
upStreamType;	**4**	**uimsbf**
switch (upStreamType) {		
case 0: /* scalablity control */		
numOfLayer;	**6**	**uimsbf**
for (layer = 0; layer < numOfLayer; layer++) {		
avgBitrate[layer];	**24**	**uimsbf**
}		
break;		
case 1: /* BSAC frame interleaving */		
numOfSubFrame;	**5**	**uimsbf**
break;		
case 2: /* quality feedback */		
multiLayOrSynEle;	**1**	**uimsbf**
if (multiLayOrSynEle) {		
layOrSynEle;	**6**	**uimsbf**
}		
else {		
layOrSynEle = 1;		
}		
numFrameExp[layOrSynEle];	**4**	**uimsbf**
lostFrames[layOrSynEle];	**numFrameExp [layOrSynEle]**	**uimsbf**
break;		
case 3: /* bitrate control */		
avgBitrate;	**24**	**uimsbf**
break;		
default: /* reserved for future use */		
break;		
}		
}		

6.4.3 Definitions

upStreamType A 4-bit unsigned integer value representing the type of the up-stream as defined in the followingTable 12

Table 12: Definition of upStreamType

UpStreamType	Type of Audio up-stream
0	scalability control
1	BSAC frame interleaving
2	quality feedback
3	bitrate control
4 – 15	reserved for future use

avgBitrate[layer] The average bitrate in bits per second of a large step layer, which the client requests to be transmitted from the server.

numOfSubFrame A 5-bit unsigned integer value representing the number of the frames which are grouped and transmitted in order to reduce the transmission overhead. The transmission overhead is decreased but the delay is increased as numOfSubFrame is increased.

multiLayOrSynEle This bit signals, whether or not a multi-channel or multi-layer configuration is used. Only in that case a layer number or a syntactic element number needs is transmitted.

layOrSynEle A 6-bit unsigned integer value representing the numer of the syntactic element (in case of multi-channel setup) or the number of the layer (in case of multi-layer setup), to which the following quality feedback information belongs. This number refers to one of the layers or one of the syntactic elements contained within the associated Audio object. If the Audio object does neither support scalability nor multi-channel capabilities, this value is implicitly set to 1.

numFrameExp[layOrSynEle] This value indicates the number of last recently passed frames $\left(2^{numFrameExp}-1\right)$ considered in the following lostFrames value.

lostFrames[layOrSynEle] This field contains the number of lost frames with respect to the indicated layer or syntactic element within the last recently passed frames signalled by numFrameExp.

avgBitrate The average bitrate in bits per second of the whole Audio object, which the client requests to be transmitted from the server.

6.4.4 Decoding process

First, **upStreamType** is parsed which represents the type of the up-stream. The remaining decoding process depends upon the type of the up-stream.

6.4.4.1 Decoding of scalability control

Next is the value **numOfLayer**. It represents the number of the data elements **avgBitrate** to be read. **avgBitrate** follows.

6.4.4.2 Decoding of BSAC frame interleaving

The data element to be read is **numOfSubFrame**. It represents the number of the sub-frames to be interleaved in BSAC tool. BSAC can allow for runtime adjustments to the quality of service. When the content of upstream is transmitted from the client to the server to implement a stream dynamically and interactively. BSAC data are split and interleaved in the server. The detailed process for implementing an AU payload in the server is described in the clause ‘informative Annex: Encoder’ of 14496-3 Amd 1.

6.4.4.3 Decoding of quality feedback

The real frame loss rate in percent can be derived using the following formula:

$$frameLossRate[layOrSynEle]=\frac{lostFrames[layOrSynEle]}{2^{numFrameExp[layOrSynEle]}-1}*100\%$$

6.4.4.4 Decoding of bitrate control

avgBitrate is parsed**.**

6.5 MPEG-4 Audio transport stream

6.5.1 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 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 AudioSpecificInfo() 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 in the LOAS format and the AudioMuxElement() format are described in subclauses 6.5.2 and 6.5.3, respectively.

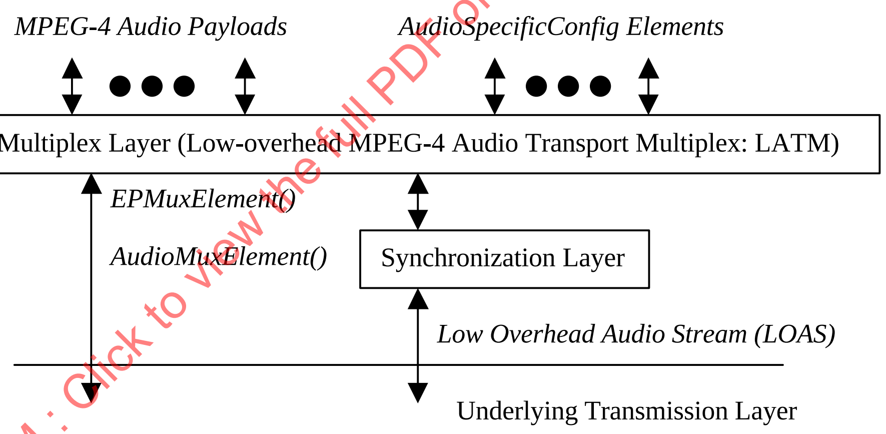


Figure 1: Concept of MPEG-4 Audio Transport

6.5.2 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 a underlying transmission layer with fixed frame synchronization, where transmission framing can not be synchronized with the variable length multiplexed element. Figure 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.

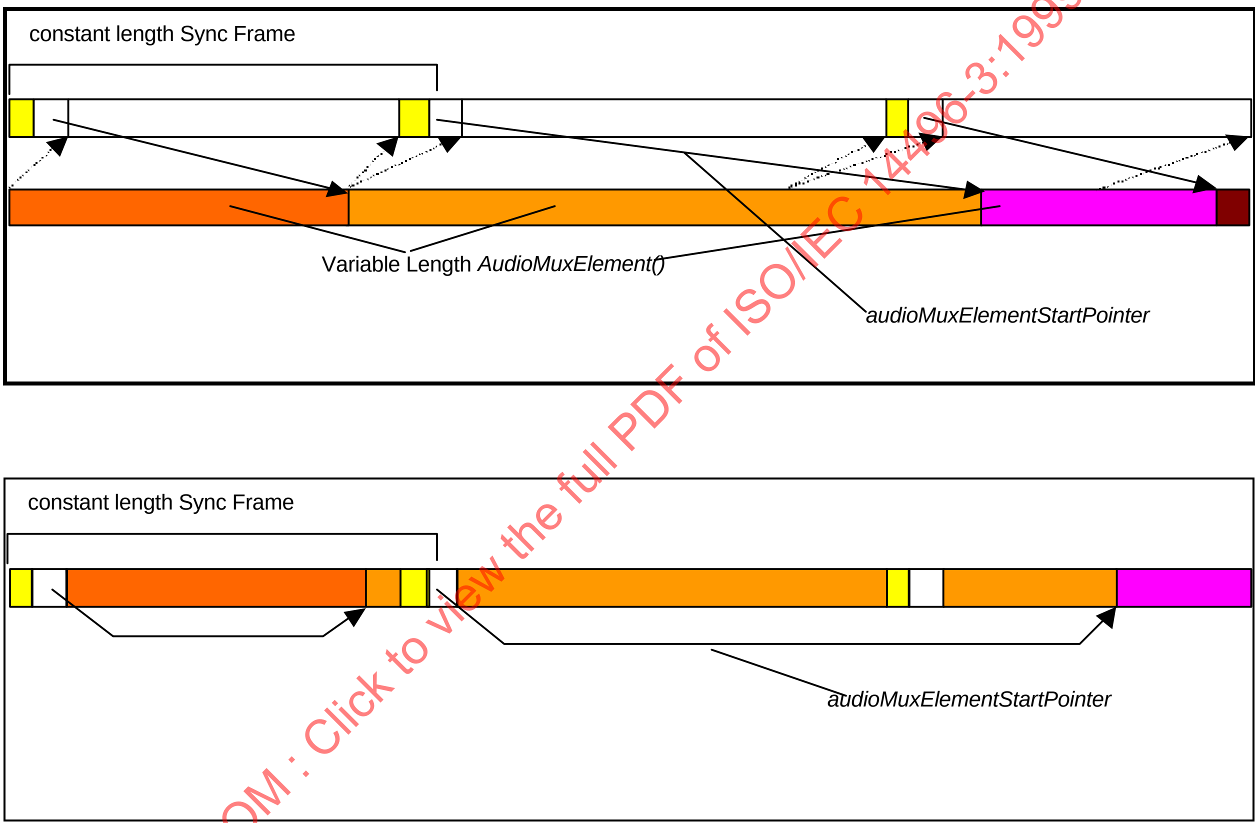


Figure 2: Synchronization in AudioPointerStream()

6.5.2.1 Syntax

Table 13: Syntax of AudioSyncStream()

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

Table 14: 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 15: Syntax of AudioPointerStream()

Syntax	No. of bits	Mnemonic
AudioPointerStream(length)		
{		
ByteAlign();		
audioMuxElementStartPointer;	ceil(ld(length))	**uimsbf**
AudioMuxElement(1);		
}		

6.5.2.2 Semantics

audioMuxLengthBytesLast A 13-bit field indicating the byte length of the multiplexed element with byte alignment.

futureUse A 4-bit field for future use.

audioMuxLengthBytes A 13-bit field indicating the byte length of the multiplexed element.

frameCounter A 5-bit field 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 field 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 generation polynomial as described in subclause 9.4.3.

audioMuxElementStartPointer A field indicating the start point of the multiplexed element. The number of bits required for this field is calculated as nbits = ceil(log2(the transmission frame length)). The transmission frame length should be provided from the transmission layer.

AudioMuxElement() A multiplexed element as specified in subclause 6.5.3.

EPMuxElement() An error resilient multiplexed element as specified in subclause 6.5.3.

6.5.3 Multiplex Layer

The LATM layer multiplexes several MPEG-4 Audio payloads and AudioSpecificInfo() 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, zero-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.

6.5.3.1 Syntax

Table 16: 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);		
ByteAlign();		
}		
}		

Table 17: Syntax of AudioMuxElement()

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

Table 18: Syntax of StreamMuxConfig()

Syntax	No. of bits	Mnemonic
StreamMuxConfig()		
{		
streamCnt = 0;		
allStreamsSameTimeFraming;	**1**	**uimsbf**
numSubFrames;	**3**	**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) {		
AudioSpecificInfo();		
}else {		
useSameConfig;	**1**	**uimsbf**
if (!useSameConfig)		
AudioSpecificInfo();		
}		
frameLengthType[streamID[prog][lay]];	**3**	**uimsbf**
if (frameLengthType[streamID[prog][lay] == 0) {		
blockDelay[streamID[prog][lay]];	**5**	**uimsbf**
fractionalDelayPresent[streamID[prog][lay]];	**1**	**uimsbf**
if (fractionalDelayPresent[streamID[prog][lay]])		
fractionalDelay[streamID[prog][lay]];	**8**	**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) **otherDataLenBits;**	**8**	**uimsbf**
crcCheckPresent;	**1**	
if(crcCheckPresent) **crcCheckSum;**	**8**	**uimsbf**
}		

Table 19: 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) {		
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 (cnt=0; cnt < numChunk; cnt++) {		
streamIndx;	**4**	**uimsbf**
prog = progCIndx[chunkCnt] = progSIndx[streamIndx]**;**		
lay = layCIndx[chunkCnt] = laySIndx [streamIndx];		
if(frameLengthType[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 20: 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]];
        }
    }
}		

6.5.3.2 Semantics

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. 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.

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

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.

epHeaderLength A 10-bit field to indicate the size of ErrorProtectionSpecificConfig()

epHeaderLengthParity: A 11-bit field for epHeaderLength, calculated as described in subclause 9.4.3 with “1)Basic set of FEC codes”.

Note: This means shortened Golay(23,12) is used

ErrorProtectionSpecificConfig() Configuration information for the EP tool which is applied to AudioMuxElement() as defined in subclause 9.2.1.

ErrorProtectionSpecificConfigParity() The parity bits for **ErrorProtectionSpecificConfig()**, calculated as described in subclause 9.4.3 with “1) Basic Set of FEC codes”.

EPAudioMuxElement() Error 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 3 shows an example for the order of the instances assuming numSubFrames is two (2).

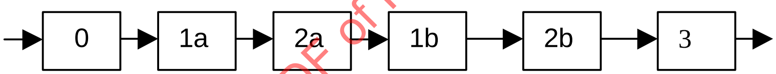


Figure 3: Instance order in EPAudioMuxElement()

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

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 field indicating the other data information.

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

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

numProgram A field indicating how many programs are multiplexed. The minimum value is 1.

numLayer A field indicating how many scalable layers are multiplexed. The minimum value is 1.

useSameConfig A flag indicating whether AudioSpecificInfo() for the payload in the previous layer or program is applied for the payload in the current layer or program.

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

frameLengthType A field 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 indexes 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-indexes, 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-indexes, 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-indexes, namely HVXCframeLengthTableIndex and MuxSlotLengthCoded.

blockDelay A field indicating the time base of the payload with frameLengthType of 0. The time base is specified by a two-layer scheme. blockDelay provides a coarse time base in multiples of tb1 = frame_length_samples / sampling_rate. If enhanced time resolution is desired, fractionalDelay may specify an additional fractional value.

fractionalDelayPresent A flag indicating the presence of fractionalDelay in the current payload.

fractionalDelayPresent	Description
0	fractionalDelay is not present.
1	fractionalDelay is present.

fractionalDelay A field indicating the enhanced time resolution for the time base of the payload with frameLengthType of 0.

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

CELPframeLengthTableIndex A field indicating one of two indexes for pointing out the frame length for a CELP or ER_CELP object. (Table 22and Table 23)

HVXCframeLengthTableIndex A field indicating one of two indexes for pointing out the frame length for a HVXC or ER_HVXC object. (Table 21)

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 field indicating the length of the other data.

crcCheckPresent A flag indicating the presence of CRC check bits for StreamMuxConfig() elements.

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

crcCheckSum A field indicating the CRC check bits.

tmp A field 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 field indicating one of two indexes for pointing out the payload length for CELP, HVXC, ER_CELP, and ER_HVXC objects.

numChunk A field indicating the number of payload chunks. 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 1.

streamIndx A field 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.

Table 21: 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 22: Frame Length of CELP Layer 0 [bits]

	Fixed-Rate frameLengthType[] =4	1-of-4 Rates (Silence Compression) frameLengthType[]=5				1-of-2 Rates (FRC) frameLengthType[]=3	
		MuxSlotLengthCoded				MuxSlotLengthCoded	
CELPframeLenghTable Index		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 23: 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				

7 Parametric audio coding (HILN)

7.1 Overview of the tools

MPEG-4 parametric audio coding uses the HILN technique (Harmonic and Individual Lines plus Noise) to code audio signals like music at bitrates of 4 kbit/s and higher using a scalable parametric representation of the audio signal. HILN allows independent change of speed and pitch during decoding. Furthermore HILN can be combined with MPEG-4 parametric speech coding (HVXC) to form an integrated parametric coder covering a wider range of signals and bitrates.

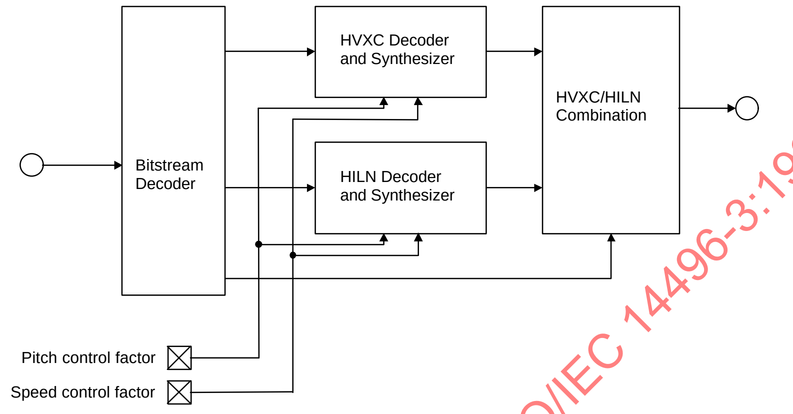


Figure 4: Block diagram of the integrated parametric decoder

The integrated parametric coder can operate in the following modes:

Table 24: Parametric coder operation modes

PARAmode	Description
0	HVXC only
1	HILN only
2	switched HVXC / HILN
3	mixed HVXC / HILN

PARAmodes 0 and 1 represent the fixed HVXC and HILN modes. PARAmode 2 permits automatic switching between HVXC and HILN depending on the current input signal type. In PARAmode 3 the HVXC and HILN decoders can be used simultaneously and their output signals are added (mixed) in the parametric decoder.

In “switched HVXC / HILN” and “mixed HVXC / HILN” modes both HVXC and HILN decoder tools are operated alternatively or simultaneously according to the PARAswitchMode or PARAmixMode of the current frame. To obtain proper time alignment of both HVXC and HILN decoder output signals before they are added, a FIFO buffer compensates for the time difference between HVXC and HILN decoder delay.

To avoid hard transitions at frame boundaries when the HVXC or HILN decoders are switched on or off, the respective decoder output signals fade in and out smoothly. For the HVXC decoder a 20 ms linear fade is applied when it is switched on or off. The HILN decoder requires no additional fading because of the smooth synthesis windows utilized in the HILN synthesizer. It is only necessary to reset the HILN decoder (numLine = 0) if the current bitstream frame contains no HILNframe().

7.2 Terms and definitions

For the purposes of Subclause 7 the following definitions apply:

HVXC: Harmonic Vector Excitation Coding (parametric speech coding).

HILN: Harmonic and Individual Lines plus Noise (parametric audio coding).

individual line: A spectral component described by frequency, amplitude and phase.

harmonic lines: A set of spectral components having a common fundamental frequency.

noise component: A signal component modeled as noise.

pi: The constant $\pi = 3.14159...$

A general glossary and list of symbols and abbreviations is located in clause 0.

7.3 Bitstream syntax

7.3.1 Decoder configuration (ParametricSpecificConfig)

The decoder configuration information for parametric coding is transmitted in the DecoderConfigDescriptor() of the base layer and the optional enhancement layer Elementary Stream (see Subclause 6.2.1).

Parametric Base Layer -- Configuration
For the parametric coder in unscalable mode or for the base layer in HILN scalable mode the following ParametricSpecificConfig() is required:

```
ParametricSpecificConfig() {
    PARAconfig();
}
```

Parametric HILN Enhancement / Extension Layer -- Configuration
To use HILN as core in an "T/F scalable with core" mode, in addition to the HILN base layer an HILN enhancement layer is required. In HILN bitrate scalable operation, in addition to the HILN base layer one or more HILN extension layers are permitted. Both the enhancement layer and the extension layer have the following ParametricSpecificConfig():

```
ParametricSpecificConfig() {
    HILNenexConfig();
}
```

An MPEG-4 Natural Audio Object using Parametric Coding is transmitted in one or more Elementary Streams: The base layer stream, an optional enhancement layer stream, and one or more optional extension layer streams.

The bitstream syntax is described in pseudo-C code.

The mnemonics LARH1, LARH2, LARH3, LARN1, LARN2, DIA, DIF, DHF, DFS indicate that a "vlclbf" codeword is used. The corresponding codebooks are given in Subclause 7.3.2.3.

The mnemonic SDC indicates that a "vlclbf" codeword is used which is decoded by the HILN SubDivisionCode described in Subclause 7.3.2.4 using the parameters for SDCdecode() as given in the bitstream syntax description.

7.3.1.1 Parametric Audio decoder configuration

Table 25: Syntax of PARAconfig()

Syntax	No. of bits	Mnemonic
PARAconfig()		
{		
PARAmode;	**2**	**uimsbf**
if (PARAmode != 1) {		
ErHVXCconfig();		
}		
if (PARAmode != 0) {		
HILNconfig();		
}		
extensionFlag;	**1**	**uimsbf**
if (extensionFlag) {		
/* to be defined in MPEG-4 Phase 3 */		
}		
}		

Table 26: PARAmode

PARAmode	frameLength	Description
0	20 ms (N = 160 samples)	HVXC only
1	see Subclause 7.3.1.2 and Subclause 7.5.1.4.3.3	HILN only
2	40 ms (N = 320 samples)	HVXC/HILN switching
3	40 ms (N = 320 samples)	HVXC/HILN mixing

7.3.1.2 HILN decoder configuration

Table 27: Syntax of HILNconfig()

Syntax	No. of bits	Mnemonic
HILNconfig()		
{		
HILNquantMode;	**1**	**uimsbf**
HILNmaxNumLine;	**8**	**uimsbf**
HILNsampleRateCode;	**4**	**uimsbf**
HILNframeLength;	**12**	**uimsbf**
HILNcontMode;	**2**	**uimsbf**
}		

Table 28: Syntax of HILNenexConfig()

Syntax	No. of bits	Mnemonic
HILNenexConfig()		
{		
HILNenhaLayer;	**1**	**uimsbf**
if (HILNenhaLayer) {		
HILNenhaQuantMode;	**2**	**uimsbf**
}		
}		

Table 29: HILNsampleRateCode

HILNsampleRateCode	sampleRate	maxFIndex
0	96000	890
1	88200	876
2	64000	825
3	48000	779
4	44100	765
5	32000	714
6	24000	668
7	22050	654
8	16000	603
9	12000	557
10	11025	544
11	8000	492
12	7350	479
13	reserved	reserved
14	reserved	reserved
15	reserved	reserved

Table 30: linebits

HILNmaxNumLine	0	1	2..3	4..7	8..15	16..31	32..63	64..127	128..255
linebits	0	1	2	3	4	5	6	7	8

Table 31: HILNcontMode

HILNcontMode	additional decoder line continuation (see subclause 7.5.1.4.3.1)
0	harmonic lines <-> individual lines and harmonic lines <-> harmonic lines
1	mode 0 plus individual lines <-> individual lines
2	no additional decoder line continuation
3	(reserved)

The number of frequency enhancement bits (fEnhbits[i]) in HILNenhaFrame() (see Subclause 7.3.2.2) is calculated as follows:

- individual line, see INDIenhaPara():

```
fEnhbits[i] = max(0,fEnhbitsBase[ILFreqIndex[i]] + fEnhbitsMode[HILNenhaQuantMode])
```

- harmonic line, see HARMenhaPara():

```
fEnhbits[i] = max(0,fEnhbitsBase[harmFreqIndex] + fEnhbitsMode[HILNenhaQuantMode] +
                    fEnhbitsHarm[i])
```

Table 32: fEnhbitsBase

lLFreqIndex	harmFreqIndex	fEnhbitsBase
0..159	0..1243	0
160..269	1244..1511	1
270..380	1512..1779	2
381..491	1780..2047	3
492..602		4
603..713		5
714..890		6

Table 33: fEnhbitsMode

HILNenhaQuantMode	0	1	2	3
fEnhbitsMode	-3	-2	-1	0

Table 34: fEnhbitsHarm

i	0	1	2..3	4..7	8..9
fEnhbitsHarm[i]	0	1	2	3	4

Table 35: HILN constants

tmbits	4
atkbits	4
decbits	4
tmEnhbits	3
atkEnhbits	2
decEnhbits	2
phasebits	5

7.3.2 Bitstream frame (alPduPayload)

The dynamic data for parametric coding is transmitted as AL-PDU payload in the base layer and the optional enhancement or extension layer Elementary Stream.

Parametric Base Layer -- Access Unit payload
For the parametric coder in unscalable mode or for the base layer in HILN scalable mode the following bitstream frame payload is defined:

```
alPduPayload {
    PARAframe();
}
```

Parametric HILN Enhancement / Extension Layer -- Access Unit payload
To parse and decode the HILN enhancement layer, information decoded from the HILN base layer is required.

To parse and decode an HILN extension layer, information decoded from the HILN base layer and possible lower HILN extension layers is required. The bitstream syntax of the HILN extension layers is described in a way which requires the HILN base and extension bitstream frames being parsed in proper order:

1. HILNbasicFrame() base bitstream frame
2. HILNextFrame(1) 1st extension bitstream frame (if base bitstream frame available)
3. HILNextFrame(2) 2nd extension bitstream frame (if base and 1st extension bitstream frame available)

4. etc.

For the enhancement layer and the extension layer in HILN scalable mode the following bitstream frame payload is defined:

```
alPduPayload {
    HILNenexFrame();
}
```

7.3.2.1 Parametric Audio bitstream frame

Table 36: Syntax of PARAframe()

Syntax	No. of bits	Mnemonic
PARAframe()		
{		
if (PARAmode == 0) {		
ErHVXCframe(HVXCrate);		
}		
else if (PARAmode == 1) {		
HILNframe();		
}		
else if (PARAmode == 2) {		
switchFrame();		
}		
else if (PARAmode == 3) {		
mixFrame();		
}		
}		

Table 37: Syntax of switchFrame()

Syntax	No. of bits	Mnemonic
switchFrame()		
{		
PARAswitchMode;	**1**	**uimsbf**
if (PARAswitchMode == 0) {		
ErHVXCdoubleframe(HVXCrate);		
}		
else {		
HILNframe();		
}		
}		

One of the following PARAswitchModes is selected in each frame:

Table 38: PARAswitchMode

PARAswitchMode	Description
0	HVXC only
1	HILN only

Table 39: Syntax of mixFrame()

Syntax	No. of bits	Mnemonic
mixFrame()		
{		
PARAmixMode;	**2**	**uimsbf**
if (PARAmixMode == 0) {		
ErHVXCdoubleframe(HVXCrate);		
}		
else if (PARAmixMode == 1) {		
HILNframe();		
ErHVXCdoubleframe(2000);		
}		
else if (PARAmixMode == 2) {		
HILNframe();		
ErHVXCdoubleframe(4000);		
}		
else if (PARAmixMode == 3) {		
HILNframe();		
}		
}		

One of the following PARAmixModes is selected in each frame:

Table 40: PARAmixMode

PARAmixMode	Description
0	HVXC only
1	HVXC 2 kbit/s & HILN
2	HVXC 4 kbit/s & HILN
3	HILN only

Table 41: Syntax of HVXCdoubleframe()

Syntax	No. of bits	Mnemonic
ErHVXCdoubleframe(rate)		
{		
if (rate >= 3000) {		
ErHVXCfixframe(4000);		
ErHVXCfixframe(rate);		
}		
else {		
ErHVXCfixframe(2000);		
ErHVXCfixframe(rate);		
}		
}		

7.3.2.2 HILN bitstream frame

Table 42: Syntax of HILNframe()

Syntax	No. of bits	Mnemonic
HILNframe()		
{		
numLayer = 0;		
HILNbasicFrameESC0();		
HILNbasicFrameESC1();		
HILNbasicFrameESC2();		
HILNbasicFrameESC3();		
HILNbasicFrameESC4();		
layNumLine[0] = numLine;		
layPrevNumLine[0] = prevNumLine;		
for (k=0; k<prevNumLine; k++) {		
layPrevLineContFlag[0][k] = prevLineContFlag[k];		
}		
}		

Table 43: Syntax of HILNbasicFrameESC0()

Syntax	No. of bits	Mnemonic
HILNbasicFrameESC0()		
{		
prevNumLine = numLine;		
/* prevNumLine is set to the number of lines */		
/* in the previous frame */		
/* prevNumLine = 0 for the first bitstream frame */		
numLine;	**linebits**	**uimsbf**
harmFlag;	**1**	**uimsbf**
noiseFlag;	**1**	**uimsbf**
envFlag;	**1**	**uimsbf**
phaseFlag;	**1**	**uimsbf**
maxAmplIndexCoded;	**4**	**uimsbf**
maxAmplIndex = 4*maxAmplIndexCoded;		
if (harmFlag) {		
HARMbasicParaESC0();		
}		
if (noiseFlag) {		
NOISEbasicParaESC0();		
}		
}		

Table 44: Syntax of HILNbasicFrameESC1()

Syntax	No. of bits	Mnemonic
HILNbasicFrameESC1()		
{		
if (harmFlag) {		
HARMbasicParaESC1();		
}		
if (noiseFlag) {		
NOISEbasicParaESC1();		
}		
INDIbasicParaESC1();		
}		

Table 45: Syntax of HILNbasicFrameESC2()

Syntax	No. of bits	Mnemonic
HILNbasicFrameESC2()		
{		
INDIbasicParaESC2 ();		
}		

Table 46: Syntax of HILNbasicFrameESC3()

Syntax	No. of bits	Mnemonic
HILNbasicFrameESC3()		
{		
if (envFlag) {		
envTmax;	**tmbits**	**uimsbf**
envRatk;	**atkbits**	**uimsbf**
envRdec;	**decbits**	**uimsbf**
}		
if (harmFlag) {		
HARMbasicParaESC3();		
}		
if (noiseFlag) {		
NOISEbasicParaESC3();		
}		
INDIbasicParaESC3();		
if (harmFlag) {		
harmFreqStretch;	**1..7**	**HFS**
}		
}		

Table 47: Syntax of HILNbasicFrameESC4()

Syntax	No. of bits	Mnemonic
HILNbasicFrameESC4()		
{		
if (harmFlag) {		
HARMbasicParaESC4();		
}		
if (noiseFlag) {		
NOISEbasicParaESC4();		
}		
INDIbasicParaESC4();		
}		

Table 48: Syntax of HARMbasicParaESC0()

Syntax	No. of bits	Mnemonic
HARMbasicParaESC0()		
{		
prevHarmAmplIndex = harmAmplIndex;		
prevHarmFreqIndex = harmFreqIndex;		
harmContFlag;	**1**	**uimsbf**
harmEnvFlag;	**1**	**uimsbf**
if (!harmContFlag) {		
harmAmplRel;	**6**	**uimsbf**
harmAmplIndex = maxAmplIndex + harmAmplRel;		
harmFreqIndex;	**11**	**uimsbf**
}		
}		

Table 49: Syntax of HARMbasicParaESC1()

Syntax	No. of bits	Mnemonic
HARMbasicParaESC1()		
{		
numHarmParaIndex;	**4**	**uimsbf**
numHarmPara = numHarmParaTable[numHarmParaIndex];		
numHarmLineIndex;	**5**	**uimsbf**
numHarmLine = numHarmLineTable[numHarmLineIndex];		
if (harmContFlag) {		
contHarmAmpl	**3..8**	**DIA**
harmAmplIndex = prevHarmAmplIndex + contHarmAmpl;		
contHarmFreq	**2..9**	**DHF**
harmFreqIndex = prevHarmFreqIndex + contHarmFreq;		
}		
for (i=0; i<2; i++) {		
harmLAR[i];	**4..19**	**LARH1**
}		
}		

Table 50: Syntax of HARMbasicParaESC3()

Syntax	No. of bits	Mnemonic
HARMbasicParaESC3()		
{		
for (i=2; i<min(7,numHarmPara); i++) {		
harmLAR[i];	**3..18**	**LARH2**
}		
for (i=7; i<numHarmPara; i++) {		
harmLAR[i];	**2..17**	**LARH3**
}		
}		

Table 51: Syntax of HARMbasicParaESC4()

Syntax	No. of bits	Mnemonic
HARMbasicParaESC4()		
{		
if (phaseFlag && ! harmContFlag) {		
numHarmPhase;	**6**	**uimsbf**
}		
else {		
numHarmPhase = 0;		
}		
for (i=0; i<numHarmPhase; i++) {		
harmPhase[i];	**phasebits**	**uimsbf**
harmPhaseAvail[i] = 1;		
}		
for (i=numHarmPhase; i<numHarmLine; i++) {		
harmPhaseAvail[i] = 0;		
}		
}		

Table 52: numHarmParaTable

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
numHarmParaTable[i]	2	3	4	5	6	7	8	9	11	13	15	17	19	21	23	25

Table 53: numHarmLineTable

i	0	1	2	3	4	5	6	7
numHarmLineTable[i]	3	4	5	6	7	8	9	10
i	8	9	10	11	12	13	14	15
numHarmLineTable[i]	12	14	16	19	22	25	29	33
i	16	17	18	19	20	21	22	23
numHarmLineTable[i]	38	43	49	56	64	73	83	94
i	24	25	26	27	28	29	30	31
numHarmLineTable[i]	107	121	137	155	175	197	222	250

Table 54: Syntax of NOISEbasicParaESC0()

Syntax	No. of bits	Mnemonic
NOISEbasicParaESC0()		
{		
prevNoiseAmplIndex = noiseAmplIndex;		
noiseContFlag;	**1**	**uimsbf**
noiseEnvFlag;	**1**	**uimsbf**
if (!noiseContFlag) {		
noiseAmplRel;	**6**	**uimsbf**
noiseAmplIndex = maxAmplIndex + noiseAmplRel;		
}		
}		

Table 55: Syntax of NOISEbasicParaESC1()

Syntax	No. of bits	Mnemonic
NOISEbasicParaESC1()		
{		
if (noiseContFlag) {		
contNoiseAmpl;	**3..8**	**DIA**
noiseAmplIndex = prevNoiseAmplIndex + contNoiseAmpl;		
}		
numNoiseParaIndex;	**4**	**uimsbf**
numNoisePara = numNoiseParaTable[numNoiseIndex];		
for (i=0; i<min(2,numNoisePara); i++) {		
noiseLAR[i];	**2..17**	**LARN1**
}		
}		

Table 56: Syntax of NOISEbasicParaESC3()

Syntax	No. of bits	Mnemonic
NOISEbasicParaESC3()		
{		
for (i=2; i<numNoisePara; i++) {		
noiseLAR[i];	**1..18**	**LARN2**
}		
}		

Table 57: Syntax of NOISEbasicParaESC4()

Syntax	No. of bits	Mnemonic
NOISEbasicParaESC4()		
{		
if (noiseEnvFlag) {		
noiseEnvTmax;	**tmbits**	**uimsbf**
noiseEnvRatk;	**atkbits**	**uimsbf**
noiseEnvRdec;	**decbits**	**uimsbf**
}		
}		

Table 58: numNoiseParaTable

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
numNoiseParaTable[i]	1	2	3	4	5	6	7	9	11	13	15	17	19	21	23	25

Table 59: Syntax of INDIbasicParaESC1()

Syntax	No. of bits	Mnemonic
INDIbasicParaESC1()		
{		
for (k=0; k<prevNumLine; k++) {		
prevLineContFlag[k];	**1**	**uimsbf**
}		
i = 0;		
for (k=0; k<prevNumLine; k++) {		
if (prevLineContFlag[k]) {		
linePred[i] = k;		
lineContFlag[i++] = 1;		
}		
}		
while (i<numLine) {		
lineContFlag[i++] = 0;		
}		
}		

Table 60: Syntax of INDIbasicParaESC2()

Syntax	No. of bits	Mnemonic
INDIbasicParaESC2()		
{		
lastNLFreq = 0;		
for (i=0; i<prevNumLine; i++) {		
prevILFreqIndex[i] = ILFreqIndex[i];		
prevILAmplIndex[i] = ILAmplIndex[i];		
}		
for (i=0; i<numLine; i++) {		
if (envFlag) {		
lineEnvFlag[i];	**1**	**uimsbf**
}		
}		
for (i=0; i<numLine; i++) {		
if (!lineContFlag[i]) {		
if (numLine-1-i < 7) {		
ILFreqInc[i];	**0..14**	**SDC**
/* SDCdecode (maxFindex-lastNLFreq, */		
/* sdcILFTable[numLine-1-i]) */		
}		
else {		
ILFreqInc[i];	**0..14**	**SDC**
/* SDCdecode (maxFindex-lastNLFreq, */		
/* sdcILFTable[7]) */		
}		
ILFreqIndex[i] = lastNLFreq + ILFreqInc[i];		
lastNLFreq = ILFreqIndex[i];		
if (HILNquantMode) {		
ILAmplRel[i];	**4..10**	**SDC**
/* SDCdecode (50, sdcILATable) */		
ILAmplIndex[i] = maxAmplIndex + ILAmplRel[i];		
}		
else {		
ILAmplRel[i];	**3..9**	**SDC**
/* SDCdecode (25, sdcILATable) */		
ILAmplIndex[i] = maxAmplIndex + 2*ILAmplRel[i];		
}		
}		
}		
}		

Table 61: Syntax of INDIbasicParaESC3()

Syntax	No. of bits	Mnemonic
INDIbasicParaESC3()		
{		
for (i=0; i<numLine; i++) {		
if (lineContFlag[i]) {		
DILFreq[i];	**2..10**	**DIF**
ILFreqIndex[i] = prevILFreqIndex[linePred[i]] + DILFreq[i];		
DILAmpl[i];	**3..8**	**DIA**
ILAmplIndex[i] = prevILAmplIndex[linePred[i]] + DILAmpl[i];		
}		
}		
}		

Table 62: Syntax of INDIbasicParaESC4()

Syntax	No. of bits	Mnemonic
INDIbasicParaESC4()		
{		
if (phaseFlag) {		
numLinePhase;	**linebits**	**uimsbf**
}		
else {		
numLinePhase = 0;		
}		
j = 0;		
for (i=0; i<numLine; i++) {		
if (! linePred[i] && j<numLinePhase) {		
linePhase[i];	**phasebits**	**uimsbf**
linePhaseAvail[i] = 1;		
j++;		
}		
else {		
linePhaseAvail[i] = 0;		
}		
}		
}		

Table 63: Syntax of HILNenexFrame()

Syntax	No. of bits	Mnemonic
HILNenexFrame()		
{		
/* HILNenhaLayer value in ParametricSpecificConfig() of */		
/* this Elementary Stream must be used here! */		
if (HILNenhaLayer) {		
HILNenhaFrame();		
}		
else {		
numLayer++;		
HILNextFrame(numLayer);		
}		
}		

Table 64: Syntax of HILNenhaFrame()

Syntax	No. of bits	Mnemonic
HILNenhaFrame()		
{		
if (envFlag) {		
envTmaxEnha;	**tmEnhbits**	**uimsbf**
envRatkEnha;	**atkEnhbits**	**uimsbf**
envRdecEnha;	**decEnhbits**	**uimsbf**
}		
if (harmFlag) {		
HARMenhaPara();		
}		
INDIenhaPara();		
}		

Table 65: Syntax of HARMenhaPara()

Syntax	No. of bits	Mnemonic
HARMenhaPara()		
{		
for (i=0; i<min(numHarmLine,10); i++) {		
harmFreqEnha[i];	**fEnhbits[i]**	**uimsbf**
harmPhase[i];	**phasebits**	**uimsbf**
}		
}		

Table 66: Syntax of INDIenhaPara()

Syntax	No. of bits	Mnemonic
INDIenhaPara()		
{		
for (i=0; i<numLine; i++) {		
lineFreqEnha[i];	**fEnhbits[i]**	**uimsbf**
linePhase[i];	**phasebits**	**uimsbf**
}		
}		

Table 67: Syntax of HILNextFrame()

Syntax	No. of bits	Mnemonic
HILNextFrame(numLayer)		
{		
layPrevNumLine[numLayer] = layNumLine[numLayer];		
/* layPrevNumLine[numLayer] = 0 for the */		
/* first bitstream frame */		
addNumLine[numLayer];	**linebits**	**uimsbf**
if (phaseFlag) {		
layNumLinePhase[numLayer];	**linebits**	**uimsbf**
}		
layNumLine[numLayer] = layNumLine[numLayer-1] + addNumLine[numLayer];		
for (k=0; k<layPrevNumLine[numLayer-1]; k++) {		
if (layPrevLineContFlag[numLayer-1][k]) {		
layPrevLineContFlag[numLayer][k] = 1;		
}		
else {		
layPrevLineContFlag[numLayer][k];	**1**	**uimsbf**
}		
}		
for (k=layPrevNumLine[numLayer-1]; k<layPrevNumLine[numLayer]; k++) {		
layPrevLineContFlag[numLayer][k];	**1**	**uimsbf**
}		
i = layNumLine[numLayer-1];		
for (k=0; k<layPrevNumLine[numLayer-1]; k++) {		
if (!layPrevLineContFlag[numLayer-1][k] && layPrevLineContFlag[numLayer][k]) {		
linePred[i] = k;		
lineContFlag[i++] = 1		
}		
}		
for (k=layPrevNumLine[numLayer-1]; k<layPrevNumLine[numLayer]; k++) {		
if (layPrevLineContFlag[numLayer][k]) {		
linePred[i] = k;		
lineContFlag[i++] = 1;		
}		
}		
while (i<layNumLine[numLayer]) {		
lineContFlag[i++] = 0;		
}		
INDIextPara(numLayer);		
if (phaseFlag) {		
INDIextPhasePara(numLayer);		
}		
}		

Table 68: Syntax of INDIextPara()

Syntax	No. of bits	Mnemonic
INDIextPara(numLayer)		
{		
lastNLFreq = 0;		
for (i=layPrevNumLine[numLayer-1];		
i<layPrevNumLine[numLayer]; i++) {		
prevILFreqIndex[i] = ILFreqIndex[i];		
prevILAmplIndex[i] = ILAmplIndex[i];		
}		
for (i=layNumLine[numLayer-1];		
i<layNumLine[numLayer]; i++) {		
if (envFlag) {		
lineEnvFlag[i];	**1**	**uimsbf**
}		
if (lineContFlag[i]) {		
DILFreq[i];	**2..10**	**DIF**
ILFreqIndex[i] = prevILFreqIndex[linePred[i]] + DILFreq[i];		
DILAmpl[i];	**3..8**	**DIA**
ILAmplIndex[i] = prevILAmplIndex[linePred[i]] + DILAmpl[i];		
}		
else {		
if (layNumLine[numLayer]-1-i < 7) {		
ILFreqInc[i];	**0..14**	**SDC**
/* SDCdecode (maxFindex-lastNLFreq, */		
/* sdcILFTable[layNumLine[numLayer]-1-i]) */		
}		
else {		
ILFreqInc[i];	**0..14**	**SDC**
/* SDCdecode (maxFindex-lastNLFreq, */		
/* sdcILFTable[7]) */		
}		
ILFreqIndex[i] = lastNLFreq + ILFreqInc[i];		
lastNLFreq = ILFreqIndex[i];		
if (HILNquantMode) {		
ILAmplRel[i];	**4..10**	**SDC**
/* SDCdecode (50, sdcILATable) */		
ILAmplIndex[i] = maxAmplIndex + ILAmplRel[i];		
}		
else {		
ILAmplRel[i];	**3..9**	**SDC**
/* SDCdecode (25, sdcILATable) */		
ILAmplIndex[i] = maxAmplIndex + 2*ILAmplRel[i];		
}		
}		
}		
}		

Table 69: Syntax of INDIextPhasePara()

Syntax	No. of bits	Mnemonic
INDIextPhasePara(numLayer)		
{		
j = 0;		
for (i=layNumLine[numLayer-1]; i<layNumLine[numLayer];		
i++) {		
if (! linePred[i] && j<layNumLinePhase[numLayer]) {		
linePhase[i];	**phasebits**	**uimsbf**
linePhaseAvail[i] = 1;		
j++;		
}		
else {		
linePhaseAvail[i] = 0;		
}		
}		
}		

7.3.2.3 HILN codebooks

Table 70: LARH1 code (harmLAR[0..1])

codeword	harmLAR[i]	codeword	harmLAR[i]
1000000000000000100	-6.350	0100	0.050
1000000000000000101	-6.250	0101	0.150
1000000000000000110	-6.150	0110	0.250
1000000000000000111	-6.050	0111	0.350
100000000000000100	-5.950	00100	0.450
100000000000000101	-5.850	00101	0.550
100000000000000110	-5.750	00110	0.650
100000000000000111	-5.650	00111	0.750
10000000000000100	-5.550	000100	0.850
10000000000000101	-5.450	000101	0.950
10000000000000110	-5.350	000110	1.050
10000000000000111	-5.250	000111	1.150
1000000000000100	-5.150	0000100	1.250
1000000000000101	-5.050	0000101	1.350
1000000000000110	-4.950	0000110	1.450
1000000000000111	-4.850	0000111	1.550
100000000000100	-4.750	00000100	1.650
100000000000101	-4.650	00000101	1.750
100000000000110	-4.550	00000110	1.850
100000000000111	-4.450	00000111	1.950
10000000000100	-4.350	000000100	2.050
10000000000101	-4.250	000000101	2.150
10000000000110	-4.150	000000110	2.250
10000000000111	-4.050	000000111	2.350
1000000000100	-3.950	0000000100	2.450
1000000000101	-3.850	0000000101	2.550
1000000000110	-3.750	0000000110	2.650
1000000000111	-3.650	0000000111	2.750
100000000100	-3.550	00000000100	2.850
100000000101	-3.450	00000000101	2.950
100000000110	-3.350	00000000110	3.050
100000000111	-3.250	00000000111	3.150

10000000100	-3.150	000000000100	3.250
10000000101	-3.050	000000000101	3.350
10000000110	-2.950	000000000110	3.450
10000000111	-2.850	000000000111	3.550
1000000100	-2.750	0000000000100	3.650
1000000101	-2.650	0000000000101	3.750
1000000110	-2.550	0000000000110	3.850
1000000111	-2.450	0000000000111	3.950
100000100	-2.350	00000000000100	4.050
100000101	-2.250	00000000000101	4.150
100000110	-2.150	00000000000110	4.250
100000111	-2.050	00000000000111	4.350
10000100	-1.950	000000000000100	4.450
10000101	-1.850	000000000000101	4.550
10000110	-1.750	000000000000110	4.650
10000111	-1.650	000000000000111	4.750
1000100	-1.550	0000000000000100	4.850
1000101	-1.450	0000000000000101	4.950
1000110	-1.350	0000000000000110	5.050
1000111	-1.250	0000000000000111	5.150
100100	-1.150	00000000000000100	5.250
100101	-1.050	00000000000000101	5.350
100110	-0.950	00000000000000110	5.450
100111	-0.850	00000000000000111	5.550
10100	-0.750	000000000000000100	5.650
10101	-0.650	000000000000000101	5.750
10110	-0.550	000000000000000110	5.850
10111	-0.450	000000000000000111	5.950
1100	-0.350	0000000000000000100	6.050
1101	-0.250	0000000000000000101	6.150
1110	-0.150	0000000000000000110	6.250
1111	-0.050	0000000000000000111	6.350

Table 71: LARH2 code (harmLAR[2..6])

codeword	harmLAR[.]	codeword	harmLAR[.]
100000000000000010	-4.725	010	0.075
100000000000000011	-4.575	011	0.225
10000000000000010	-4.425	0010	0.375
10000000000000011	-4.275	0011	0.525
1000000000000010	-4.125	00010	0.675
1000000000000011	-3.975	00011	0.825
100000000000010	-3.825	000010	0.975
100000000000011	-3.675	000011	1.125
10000000000010	-3.525	0000010	1.275
10000000000011	-3.375	0000011	1.425
1000000000010	-3.225	00000010	1.575
1000000000011	-3.075	00000011	1.725
100000000010	-2.925	000000010	1.875
100000000011	-2.775	000000011	2.025
10000000010	-2.625	0000000010	2.175
10000000011	-2.475	0000000011	2.325
1000000010	-2.325	00000000010	2.475
1000000011	-2.175	00000000011	2.625
100000010	-2.025	000000000010	2.775
100000011	-1.875	000000000011	2.925
10000010	-1.725	0000000000010	3.075
10000011	-1.575	0000000000011	3.225
1000010	-1.425	00000000000010	3.375
1000011	-1.275	00000000000011	3.525
100010	-1.125	000000000000010	3.675
100011	-0.975	000000000000011	3.825
10010	-0.825	0000000000000010	3.975
10011	-0.675	0000000000000011	4.125
1010	-0.525	00000000000000010	4.275
1011	-0.375	00000000000000011	4.425
110	-0.225	000000000000000010	4.575
111	-0.075	000000000000000011	4.725

Table 72: LARH3 code (harmLAR[7..25])

codeword	harmLAR[.]	codeword	harmLAR[.]
10000000000000001	-2.325	01	0.075
1000000000000001	-2.175	001	0.225
100000000000001	-2.025	0001	0.375
10000000000001	-1.875	00001	0.525
1000000000001	-1.725	000001	0.675
100000000001	-1.575	0000001	0.825
10000000001	-1.425	00000001	0.975
1000000001	-1.275	000000001	1.125
100000001	-1.125	0000000001	1.275
10000001	-0.975	00000000001	1.425
1000001	-0.825	000000000001	1.575
100001	-0.675	0000000000001	1.725
10001	-0.525	00000000000001	1.875
1001	-0.375	000000000000001	2.025
101	-0.225	0000000000000001	2.175
11	-0.075	00000000000000001	2.325

Table 73: LARN1 code (noiseLAR[0,1])

codeword	noiseLAR[.]	codeword	noiseLAR[.]
10000000000000001	-4.65	01	0.15
1000000000000001	-4.35	001	0.45
100000000000001	-4.05	0001	0.75
10000000000001	-3.75	00001	1.05
1000000000001	-3.45	000001	1.35
100000000001	-3.15	0000001	1.65
10000000001	-2.85	00000001	1.95
1000000001	-2.55	000000001	2.25
100000001	-2.25	0000000001	2.55
10000001	-1.95	00000000001	2.85
1000001	-1.65	000000000001	3.15
100001	-1.35	0000000000001	3.45
10001	-1.05	00000000000001	3.75
1001	-0.75	000000000000001	4.05
101	-0.45	0000000000000001	4.35
11	-0.15	00000000000000001	4.65

Table 74: LARN2 code (noiseLAR[2..25])

codeword	noiseLAR[.]	codeword	noiseLAR[.]
110000000000000001	-6.35	101	0.35
11000000000000001	-5.95	1001	0.75
1100000000000001	-5.55	10001	1.15
110000000000001	-5.15	100001	1.55
11000000000001	-4.75	1000001	1.95
1100000000001	-4.35	10000001	2.35
110000000001	-3.95	100000001	2.75
11000000001	-3.55	1000000001	3.15
1100000001	-3.15	10000000001	3.55
110000001	-2.75	100000000001	3.95
11000001	-2.35	1000000000001	4.35
1100001	-1.95	10000000000001	4.75
110001	-1.55	100000000000001	5.15
11001	-1.15	1000000000000001	5.55
1101	-0.75	10000000000000001	5.95
111	-0.35	100000000000000001	6.35
0	0.00		

Table 75: DIA code

codeword	value	codeword	value
111 1 1111	-25	001	1
111 1 1110	-24	011 0	2
111 1 1101	-23	100 0	3
111 1 xxxx	-y	101 0 0	4
111 1 0001	-11	101 0 1	5
111 1 0000	-10	110 0 00	6
110 1 11	-9	110 0 01	7
110 1 10	-8	110 0 10	8
110 1 01	-7	110 0 11	9
110 1 00	-6	111 0 0000	10
101 1 1	-5	111 0 0001	11
101 1 0	-4	111 0 xxxx	y
100 1	-3	111 0 1101	23
011 1	-2	111 0 1110	24
010	-1	111 0 1111	25
000	0		

Table 76: DIF code

codeword	value	codeword	value
11 11 1 11111	-42	01 0	1
11 11 1 11110	-41	10 0 0	2
11 11 1 11101	-40	10 0 1	3
11 11 1 xxxxx	-y	11 00 0	4
11 11 1 00001	-12	11 01 0 0	5
11 11 1 00000	-11	11 01 0 1	6
11 10 1 11	-10	11 10 0 00	7
11 10 1 10	-9	11 10 0 01	8
11 10 1 01	-8	11 10 0 10	9
11 10 1 00	-7	11 10 0 11	10
11 01 1 1	-6	11 11 0 00000	11
11 01 1 0	-5	11 11 0 00001	12
11 00 1	-4	11 11 0 xxxxx	y
10 1 1	-3	11 11 0 11101	40
10 1 0	-2	11 11 0 11110	41
01 1	-1	11 11 0 11111	42
00	0		

Table 77: DHF code

codeword	value	codeword	value
11 1 111111	-69	01 0	1
11 1 111110	-68	10 0 00	2
11 1 111101	-67	10 0 01	3
11 1 xxxxxx	-y	10 0 10	4
11 1 000001	-7	10 0 11	5
11 1 000000	-6	11 0 000000	6
10 1 11	-5	11 0 000001	7
10 1 10	-4	11 0 xxxxxx	y
10 1 01	-3	11 0 111101	67
10 1 00	-2	11 0 111110	68
01 1	-1	11 0 111111	69
00	0		

Table 78: HFS code

codeword	value	codeword	value
1 1 1 1111	-17	1 0 0	1
1 1 1 1110	-16	1 0 1 0000	2
1 1 1 1101	-15	1 0 1 0001	3
1 1 1 xxxx	-y	1 0 1 xxxx	y
1 1 1 0001	-3	1 0 1 1101	15
1 1 1 0000	-2	1 0 1 1110	16
1 1 0	-1	1 0 1 1111	17
0	0		

Notes on Table 75 to Table 78: The grouping of bits within a codeword (e.g. “1 1 1 1111”) is provided for easier readability only. Codewords not explicitly listed in the codebooks (e.g. “1 1 1 xxxx”) are defined by incrementing the implicit part of the codeword “xxxx” (uimsbf) and the magnitude “y” of the corresponding value. In all cases, the codewords and values for the two smallest and the three largest magnitudes are listed explicitly.

7.3.2.4 HILN SubDivisionCode (SDC)

The SubDivisionCode (SDC) is an algorithmically generated variable length code, based on a given table and a given number of different codewords. The decoding process is defined below.

The idea behind this coding scheme is the subdivision of the probability density function into two parts which represent an equal probability. One bit is transmitted that determines the part the value to be coded is located. This subdivision is repeated until the width of the part is one and then its position is equal to the value being coded. The positions of the boundaries are taken out of a table of 32 quantized, fixed point values. Besides this table (parameter “tab”) the number of different codewords (parameter “k”) is needed too.

The following C function SDCDecode(k, tab) together with the 9 tables sdcILATable[32] and sdcILFTable[8][32] describe the decoding. The function GetBit() returns the next bit in the stream.

```
int sdcILATable[32] = {
    0,  13,  27,  41,  54,  68,  82,  96, 110, 124, 138, 152, 166, 180, 195, 210,
  225, 240, 255, 271, 288, 305, 323, 342, 361, 383, 406, 431, 460, 494, 538, 602
};
```

```
int sdcILFTable[8][32] = {
{   0,  53,  87, 118, 150, 181, 212, 243, 275, 306, 337, 368, 399, 431, 462, 493,
  524, 555, 587, 618, 649, 680, 711, 743, 774, 805, 836, 867, 899, 930, 961, 992 },
{   0,  34,  53,  71,  89, 106, 123, 141, 159, 177, 195, 214, 234, 254, 274, 296,
  317, 340, 363, 387, 412, 438, 465, 494, 524, 556, 591, 629, 670, 718, 774, 847 },
{   0,  26,  41,  54,  66,  78,  91, 103, 116, 128, 142, 155, 169, 184, 199, 214,
  231, 247, 265, 284, 303, 324, 346, 369, 394, 422, 452, 485, 524, 570, 627, 709 },
{   0,  23,  35,  45,  55,  65,  75,  85,  96, 106, 117, 128, 139, 151, 164, 177,
  190, 204, 219, 235, 252, 270, 290, 311, 334, 360, 389, 422, 461, 508, 571, 665 },
{   0,  20,  30,  39,  48,  56,  64,  73,  81,  90,  99, 108, 118, 127, 138, 149,
  160, 172, 185, 198, 213, 228, 245, 263, 284, 306, 332, 362, 398, 444, 507, 608 },
{   0,  18,  27,  35,  43,  50,  57,  65,  72,  79,  87,  95, 104, 112, 121, 131,
  141, 151, 162, 174, 187, 201, 216, 233, 251, 272, 296, 324, 357, 401, 460, 558 },
{   0,  16,  24,  31,  38,  45,  51,  57,  64,  70,  77,  84,  91,  99, 107, 115,
  123, 132, 142, 152, 163, 175, 188, 203, 219, 237, 257, 282, 311, 349, 403, 493 },
{   0,  12,  19,  25,  30,  35,  41,  46,  51,  56,  62,  67,  73,  79,  85,  92,
   99, 106, 114, 122, 132, 142, 153, 165, 179, 195, 213, 236, 264, 301, 355, 452 }
};

int SDCDecode (int k, int *tab)
{
   int *pp;
   int g,dp,min,max;

   min=0;
   max=k-1;
   pp=tab+16;
   dp=16;

   while ( min!=max )
   {
      if ( dp ) g=(k*(*pp))>>10; else g=(max+min)>>1;
      dp>>=1;
      if ( GetBit()==0 ) { pp-=dp; max=g; } else { pp+=dp; min=g+1; }
   }
   return max;
}
```

7.4 Bitstream semantics

7.4.1 Decoder configuration (ParametricSpecificConfig)

7.4.1.1 Parametric Audio decoder configuration

Bitstream elements:

PARAmode A 2 bit field indicating parametric coder operation mode.

extensionFlag A flag indicating the presence of MPEG-4 version 3 data (for future use).

7.4.1.2 HILN decoder configuration

Bitstream elements:

HILNquantMode A 1 bit field indicating the individual line quantizer mode.

HILNmaxNumLine An 8 bit field indicating the maximum number of individual lines in a bitstream frame. It also determines linebits field size (see Table 30).

HILNsampleRateCode A 4 bit field indicating the sampling rate used for HILN parameter decoding (see Table 29).

HILNframeLength	A 12 bit field indicating the HILN frame length in samples at the sampling rate indicated by HILNsampleRateCode.
HILNcontMode	A 2 bit field indicating the additional decoder line continuation mode (see Table 31).
HILNenhaLayer	A flag indicating whether this Elementary Stream contains an enhancement layer or an extension layer.
HILNenhaQuantMode	A 2 bit field indicating frequency enhancement quantizer mode (see Table 33).

7.4.2 Bitstream frame (alPduPayload)

7.4.2.1 Parametric Audio bitstream frame

Bitstream elements:

PARAswitchMode	A flag indicating which coding tool is used in the current frame of a HVXC/HILN switching bitstream (see Table 38).
PARAmixMode	A 2 bit field indicating which coding tools are used in the current frame of a HVXC/HILN mixing bitstream (see Table 40).

7.4.2.2 HILN bitstream frame

Bitstream elements:

numLine	A field indicating the number of individual lines in the current frame.
harmFlag	A flag indicating the presence of harmonic line data in the current frame.
noiseFlag	A flag indicating the presence of noise component data in the current frame.
phaseFlag	A flag indicating the presence of line start phase data in the current frame.
numLinePhase	A field indicating the number of individual lines with start phase in the current frame.
maxAmplIndexCoded	A field indicating the maximum amplitude of a new signal component in the current frame.
envFlag	A flag indicating the presence of envelope data in the current frame.
envTmax	Coded envelope parameter: time of maximum.
envRatk	Coded envelope parameter: attack rate.
envRdec	Coded envelope parameter: decay rate.
prevLineContFlag[k]	A flag indicating that the k-th individual line of the previous frame is continued in the current frame.
numHarmParaIndex	A field indicating the number of harmonic line LPC parameters in the current frame (see Table 52).
numHarmLineIndex	A field indicating the number of harmonic lines in the current frame (see Table 53).
harmContFlag	A flag indicating that the harmonic lines are continued from the previous frame.

numHarmPhase	A field indicating the number of harmonic lines with start phase in the current frame.
harmEnvFlag	A flag indicating that the amplitude envelope is applied to the harmonic lines.
contHarmAmpl	Coded amplitude change of the harmonic lines (see Table 75).
contHarmFreq	Coded fundamental frequency change of the harmonic lines (see Table 77).
harmAmplRel	Coded relative amplitude of the harmonic lines.
harmFreqIndex	Coded fundamental frequency of the harmonic lines.
harmFreqStretch	Coded frequency stretching parameter of the harmonic lines (see Table 78).
harmLAR[i]	Coded LAR LPC parameters of the harmonic lines (see Table 70, Table 71, Table 72).
harmPhase[i]	Coded phase of i-th harmonic line.
numNoiseParaIndex	A field indicating the number of noise LPC parameters in the current frame (see Table 58).
noiseContFlag	A flag indicating that the noise is continued from the previous frame.
noiseEnvFlag	A flag indicating that noise envelope data is present in the current frame.
contNoiseAmpl	Coded amplitude change of the noise (see Table 75).
noiseAmplRel	Coded relative amplitude of the noise.
noiseEnvTmax	Coded noise envelope parameter: time of maximum
noiseEnvRatk	Coded noise envelope parameter: attack rate.
noiseEnvRdec	Coded noise envelope parameter: decay rate.
noiseLAR[i]	Coded LAR LPC parameters of the noise (see Table 73, Table 74).
lineEnvFlag[i]	A flag indicating that the amplitude envelope is applied to the i-th individual line.
DILFreq[i]	Coded frequency change of i-th individual line (see Table 76).
DILAmpl[i]	Coded amplitude change of i-th individual line (see Table 75).
ILFreqInc[i]	Coded frequency increment of i-th individual line (see Subclause 7.3.2.4).
ILAmplRel[i]	Coded relative amplitude of i-th individual line (see Subclause 7.3.2.4).
linePhase[i]	Coded phase of i-th individual line.
envTmaxEnha	Coded envelope enhancement parameter: time of maximum.
envRatkEnha	Coded envelope enhancement parameter: attack rate.
envRdecEnha	Coded envelope enhancement parameter: decay rate.
harmFreqEnha[i]	Coded frequency enhancement of i-th harmonic line.
lineFreqEnha[i]	Coded frequency enhancement of i-th individual line.

addNumLine[i]	A field indicating the number of individual lines in extension layer i of the current frame.
layNumLinePhase[i]	A field indicating the number of individual lines with start phase in extension layer i of the current frame.
layPrevLineContFlag[i][k]	A flag indicating that the k-th individual line of the previous frame is continued in extension layer i of the current frame.

Help elements:

numLayer	The number of extension layers available (0 if only base layer available).
layNumLine[i]	The total number of individual lines in the current frame as conveyed in the base layer and the first i extension layers.
prevNumLine	The number of individual lines in the previous frame.
layPrevNumLine[i]	The total number of individual lines in the previous frame as conveyed in the base layer and the first i extension layers.
maxAmplIndex	The maximum amplitude of a new signal component in the current frame.
linePred[i]	Index of the predecessor in the previous frame of the i-th individual line in the current frame.
lineContFlag[i]	A flag indicating that line i in the current frame is continued from the previous frame.
numHarmPara	The number of harmonic line LPC parameters in the current frame.
numHarmLine	The number of harmonic lines in the current frame.
harmAmplIndex	Amplitude index of the harmonic lines in the current frame.
harmFreqIndex	Fundamental frequency index of the harmonic lines in the current frame.
prevHarmAmplIndex	Amplitude index of the harmonic lines in the previous frame.
prevHarmFreqIndex	Fundamental frequency index of the harmonic lines in the previous frame.
harmPhaseAvail[i]	A flag indicating that start phase information is available for the i-th harmonic line.
numNoisePara	The number of noise LPC parameters in the current frame.
noiseAmplIndex	Amplitude index of the noise in the current frame.
prevNoiseAmplIndex	Amplitude index of the noise in the previous frame.
lastNLFreq	Individual line frequency increment accumulator.
ILFreqIndex[i]	Frequency index of the i-th individual line in the current frame.
ILAmplIndex[i]	Amplitude index of the i-th individual line in the current frame.
prevILFreqIndex[i]	Frequency index of the i-th individual line in the previous frame.
prevILAmplIndex[i]	Amplitude index of the i-th individual line in the previous frame.
linePhaseAvail[i]	A flag indicating that start phase information is available for the i-th individual line.

linebits Number of bits for numLine.

tmbits Number of bits for envTmax.

atkbits Number of bits for encRatk.

decbits Number of bits for envRdec.

tmEnhbits Number of bits for envTmaxEnha.

atkEnhbits Number of bits for encRatkEnha.

decEnhbits Number of bits for envRdecEnha.

fEnhbits[i] Number of bits for lineFreqEnha[i] and harmFreqEnha[i] (see Subclause 7.3.1.2).

phasebits Number of bits for linePhase and harmPhase.

7.5 Parametric decoder tools

7.5.1 HILN decoder tools

The Harmonic and Individual Lines plus Noise (HILN) decoder utilizes a set of parameters which are encoded in the bitstream to describe the audio signal. Three different signal models are supported:

Table 79: HILN signal models

signal model	description	essential parameters
harmonic lines	group of sinusoidal signals with common fundamental frequency	fundamental frequency and amplitudes of the spectral lines
individual lines	sinusoidal signals	frequency and amplitude of the individual spectral lines
noise	spectrally shaped noise signal	spectral shape and power of the noise

The HILN decoder first reconstructs these parameters from the bitstream with a set of decoding tools and then synthesizes the audio signal based on these parameters using a set of synthesizer tools:

- harmonic line decoder
- individual line decoder
- noise decoder
- harmonic and individual line synthesizer
- noise synthesizer

The HILN decoder tools reconstruct the parameters of the harmonic and individual lines (frequency, amplitude) and the noise (spectral shape) as well as possible envelope parameters from the bitstream.

The HILN synthesizer tools reconstruct one frame of the audio signal based on the parameters decoded by the HILN decoder tools for the current bitstream frame.

The samples of the decoded audio signal have a full scale range of [-32768, 32767] and eventual outliers must be limited (“clipped”) to these values.

The HILN decoder supports a wide range of frame lengths and sampling frequencies. By scaling the synthesizer frame length with an arbitrary factor, speed change functionality is available at the decoder. By scaling the line frequencies and resampling the noise signal with an arbitrary factor, pitch change functionality is available at the decoder.

The HILN decoder can operate in two different modes, as basic decoder and as enhanced decoder. The basic decoder which is used for normal operation only evaluates the information available in the bitstream elements HILNbasicFrame() to reconstruct the audio signal. To allow large step scalability in combination with other coder tools (e.g. GA scalable) the additional bitstream elements HILNenhaFrame() need to be transmitted and the HILN decoder must operate in the enhanced mode which exploits the information of both HILNbasicFrame() and HILNenhaFrame(). This mode reconstructs an audio signal with well defined phase relationships which can be combined with a residual signal coded at higher bitrates using an enhancement coder (e.g. GA scalable). If the HILN decoder is used in this way as a core for a scalable coder no noise signal must be synthesized for the signal which is given to the enhancement decoder.

Due to the parametric signal representation utilized by the HILN parametric coder, it is well suited for applications requiring bitrate scalable coding. HILN bitrate scalable coding is accomplished by supplementing the data encoded in an HILNbasicFrame() of the basic bitstream by data encoded in one or more HILNextFrame() of one or more extension bitstreams transmitted as additional Elementary Streams. It should be noted that the coding efficiency of a combined bitstream consisting of a basic and one or more extension bitstreams is slightly lower than the coding efficiency of an non-scalable basic bitstream having the same total bitrate.

7.5.1.1 Harmonic line decoder

7.5.1.1.1 Tool description

This tool decodes the parameters of the harmonic lines transmitted in the bitstream.

7.5.1.1.2 Definitions

prevNumHarmPara	The number of harmonic line LPC parameters in the previous frame.
harmLPCPara[i]	Harmonic line LPC parameter i in the current frame (LARs for harmonic tone spectrum).
prevHarmLPCPara[i]	Harmonic line LPC parameter i in the previous frame (LARs for harmonic tone spectrum).
hFreq	Fundamental frequency of the harmonic lines.
hStretch	Frequency stretching of the harmonic lines.
harmAmpl	Harmonic tone amplitude.
harmPwr	Harmonic tone power.
hLineAmpl[i]	Amplitude of i-th harmonic line.
hLineFreq[i]	Frequency of i-th harmonic line (in Hz).
hLineAmplEnh[i]	Enhanced amplitude of i-th harmonic line.
hLineFreqEnh[i]	Enhanced frequency of i-th harmonic line (in Hz).
hLinePhaseEnh[i]	Phase of i-th harmonic line (in rad).
ha[i]	Unscaled amplitude of i-th harmonic line.
r[i]	LPC reflection coefficients.

h[i] LPC impulse response.

H(z) LPC system function.

7.5.1.1.3 Decoding process

If the harmFlag is set and thus HARMbasicPara() data and in enhancement mode HARMenhaPara() data is available in the current frame, the parameters of the harmonic lines are decoded and dequantized as follows:

7.5.1.1.3.1 Basic decoder

A harmonic tone is represented by its fundamental frequency, its power and a set of LPC-Parameters.

First the harmNumPara LAR parameters are reconstructed. Prediction from the previous frame is used when harmContFlag is set.

```
float harmLPCMean[25] = { 5.0, -1.5, 0.0, 0.0, 0.0, ... , 0.0 };
float harmPredCoeff[25] = { 0.75, 0.75, 0.5, 0.5, 0.5, ... , 0.5 };

for (i=0; i<numHarmPara; i++) {
    if ( i<prevNumHarmPara && harmContFlag )
        pred = harmLPCMean[i] +
            (prevHarmLPCPara[i]-harmLPCMean[i])*harmPredCoeff[i];
    else
        pred = harmLPCMean[i];
    harmLPCPara[i] = pred + harmLAR[i];
}
```

Parameters needed in the following frame are stored in the frame-to-frame memory:

```
prevNumHarmPara = numHarmPara;
for (i=0; i<numHarmPara; i++)
    prevHarmLPCPara[i] = harmLPCPara[i];
```

The fundamental frequency and stretching of the harmonic lines are dequantized:

```
hFreq = 20 * exp( log(4000./20.) * (harmFreqIndex+0.5) / 2048.0 );
hStretch = harmFreqStretch / 16000.0;
```

The amplitude and power of the harmonic tone is dequantized as follows:

```
harmAmpl = 32768 * pow(10, -1.5*(harmAmplIndex+0.5)/20 );
harmPwr = harmAmpl*harmAmpl;
```

The harmEnvFlag and harmContFlag flags require no further dequantization; they are directly passed on to the synthesizer tool.

The LPC-Parameters are transmitted in the form of “Logarithmic Area Ratios” (LAR) as described above. After decoding the parameters the frequencies and amplitudes of the harmNumLine partials of the harmonic tone are calculated as follows:

The frequencies of the harmonic lines are calculated:

```
for (i=0; i<numHarmLine; i++)
    hLineFreq[i] = hFreq * (i+1) * (1 + hStretch*(i+1));
```

The LPC-Parameters represent an IIR-Filter. The amplitudes of the sinusoids are obtained by calculating the absolute value of this filter's system function H(z) at the corresponding frequencies.

```
for (i=0; i<numHarmLine; i++)
    ha[i] = abs( H( exp( j * pi * (i+1)/(numHarmLine+1) ) );
```

with j = sqrt(-1) and

```
   H(z) = 1 / ( 1 - h[0]*pow(z,-1) - h[1]*pow(z,-2) - ... - h[numHarmPara-1]*pow(z,-
numHarmPara) )
```

The impulse response h[i] is calculated from the LARs by the following algorithm:

In a first step the LARs are converted to reflection coefficients:

```
   for (i=0; i<numHarmPara; i++)
      r[i] = ( exp(harmLPCPara[i]) - 1 ) / ( exp(harmLPCPara[i]) + 1 );
```

After this the reflection coefficients are converted to the time response. The C function given below does this conversion in place (call with x[i]=r[i] and N=numHarmPara; returns with x[i]=h[i]):

```
   void Convert_k_to_h (float *x, int N)
   {
      int i,j;
      float  a,b,c;

      for (i=1; i<N; i++)
      {
         c = x[i];

         for (j=0; j<i-j-1; j++)
         {
            a = x[ j ];
            b = x[i-j-1];
            x[j] = a-c*b;
            x[i-j-1] = b-c*a;
         }
         if ( j==i-j-1 )
            x[j] -= c*x[j];
      }
   }
```

After calculating the amplitudes ha[i] they must be normalized and multiplied with harmAmpl to find the harmonic line amplitudes meeting the condition:

```
   sum ( hLineAmpl[i]*hLineAmpl[i] ) = power of harmonic tone
```

This is realized as follows:

```
   p = 0.0;
   for (i=0; i<numHarmLine; i++)
      p += ha[i]*ha[i];
   s = sqrt( harmPwr / p );
   for (i=0; i<numHarmLine; i++)
      hLineAmpl[i] = ha[i] * s;
```

The optional phase information is decoded as follows:

```
   for (i=0; i<numHarmLine; i++) {
      if (harmPhaseAvail[i]) {
         hStartPhase[i] = 2*pi*(harmPhase[i]+0.5)/(1<<phasebits)-pi;
         hStartPhaseAvail[i] = 1;
      }
      else
         hStartPhaseAvail[i] = 0;
   }
```

7.5.1.1.3.2 Enhanced decoder

In this mode, the harmonic line parameters decoded by the basic decoder are refined and also line phases are decoded using the information contained in HARMenhaPara() as follows:

For the first maximum 10 harmonic lines i

```
i = 0 .. min(numHarmLine,10)-1
```

the enhanced harmonic line parameters are calculated using the basic harmonic line parameters and the data in the enhancement bitstream:

```
hLineAmplEnh[i] = hLineAmpl[i];
hLineFreqEnh[i] = hLineFreq[i] *
       (1+((harmFreqEnh[i]+0.5)/(1<<fEnhbits[i])-0.5)*(hFreqRelStep-1));
```

where hFreqRelStep is the ratio of two neighboring fundamental frequency quantizer steps:

```
hFreqRelStep = exp(log(4000/20)/2048));
```

For both line types the phase is decoded from the enhancement bitstream:

```
hLinePhaseEnh[i] = 2*pi*(harmPhase[i]+0.5)/(1<<phasebits)-pi;
```

7.5.1.2 Individual line decoder

7.5.1.2.1 Tool description

The individual line basic bitstream decoder reconstructs the line parameters frequency, amplitude, and envelope from the bitstream. The enhanced bitstream decoder reconstructs the line parameters frequency, amplitude, and envelope with finer quantization and additionally reconstructs the line parameters phase.

7.5.1.2.2 Definitions

t_max	Envelope parameter: time of maximum.
r_atk	Envelope parameter: attack rate.
r_dec	Envelope parameter: decay rate.
ampl[i]	Amplitude of i-th individual line.
freq[i]	Frequency of i-th individual line (in Hz).
startPhase[i]	Start phase of i-th individual line.
startPhaseAvail[i]	A flag indicating that start phase information is available for the i-th individual line.
t_maxEnh	Enhanced envelope parameter: time of maximum.
r_atkEnh	Enhanced envelope parameter: attack rate.
r_decEnh	Enhanced envelope parameter: decay rate.
amplEnh[i]	Enhanced amplitude of i-th individual line.
freqEnh[i]	Enhanced frequency of i-th individual line (in Hz).
phaseEnh[i]	Phase of i-th individual line (in rad).

7.5.1.2.3 Decoding process

7.5.1.2.3.1 Basic decoder

The basic decoder reconstructs the line parameters from the data contained in HILNbasicFrame() and INDIbasicPara() in the following way:

For each frame, first the number of individual lines encoded in this frame is read from HILNbasicFrame():

```
numLine
```

Then the frame envelope flag is read from HILNbasicFrame():

```
envFlag
```

If envFlag = 1 then the 3 envelope parameters t_max, r_atk, and r_dec are decoded from HILNbasicFrame():

```
t_max = (envTmax+0.5)/(1<<tmbits);
r_atk = tan(pi/2*max(0,envRatk-0.5)/((1<<atkbits)-1))/0.2;
r_dec = tan(pi/2*max(0,envRdec-0.5)/((1<<decbits)-1))/0.2;
```

These envelope parameters are valid for the harmonic lines as well as for the individual lines. Thus the envelope parameters envTmax, envRatk, envRdec must be dequantized if present, even if numLine == 0.

For each line k of the previous frame

```
k = 0 .. prevNumLine-1
```

the previous line continuation flag is read from HILNbasicFrame():

```
prevLineContFlag[k]
```

If prevLineContFlag[k] == 1 then line k of the previous frame is continued in the current frame. If prevLineContFlag[k] == 0 then line k of the previous frame is not continued.

In the current frame, first the parameters of all continued lines are encoded followed by the parameters of the new lines. Therefore, the line continuation flag and the line predecessor are determined before decoding the line parameters:

```
i = 0;
for (k=0;k<prevNumLine;k++)
    if (prevLineContFlag[k]) {
        linePred[i] = k;
        lineContFlag[i++] = 1;
    }
while (i<numLine)
    lineContFlag[i++] = 0;
```

The parameters of new lines are encoded with increasing frequency index, using a differential encoding scheme. Therefore the following initialization is required once for each frame:

```
lastNLFreq = 0;
```

For each line i of the current frame

```
i = 0 .. numLine-1
```

the line parameters are decoded from INDIbasicPara() now.

If envFlag == 1 then the line envelope flag is read from INDIbasicPara():

```
lineEnvFlag[i]
```

If lineContFlag[i] == 1 then the parameters of a continued line are decoded from INDIbasicPara() based on the amplitude and frequency index of its predecessor in the previous frame:

```
ILFreqIndex[i] = prevILFreqIndex[linePred[i]] + DILFreq[i];
ILAmplIndex[i] = prevILAmplIndex[linePred[i]] + DILAmpl[i];
```

If lineContFlag[i] == 0 then the parameters of a new line are decoded from INDIbasicPara():

```
if (numLine-1-i < 7)
    ILFreqInc[i] = SDCdecode (maxFindex-lastNLFreq, sdcILFTable[numLine-1-i]);
else
    ILFreqInc[i] = SDCdecode (maxFindex-lastNLFreq, dcILFTable[7]);
ILFreqIndex[i] = lastNLFreq + ILFreqInc[i];
lastNLFreq = ILFreqIndex[i];
if (HILNquantMode) {
    ILAmplRel[i] = SDCdecode (50, sdcILATable);
    ILAmplIndex[i] = maxAmplIndex + ILAmplRel[i];
}
else {
    ILAmplRel[i] = SDCdecode (25, sdcILATable);
    ILAmplIndex[i] = maxAmplIndex + 2*ILAmplRel[i];
}
```

The line parameter indices are stored for decoding of the line parameters of the next frame:

```
prevNumLine = numLine;
for (i=0; i<prevNumLine; i++) {
    prevILFreqIndex[i] = ILFreqIndex[i];
    prevILAmplIndex[i] = ILAmplIndex[i];
}
```

The basic decoder also handles combinations of a basic layer and one or more extension layers. If data from a total of numLayer extension layers is available to the basic decoder, the values of layNumLine[numLayer] and layPrevNumLine[numLayer] are to be used instead of numLine and prevNumLine respectively. The values of ILAmplIndex[i], ILFreqIndex[i], lineContFlag[i], and linePred[i] as determined by the bitstream syntax description are to be used.

The amplitudes and frequencies of the individual lines are now dequantized from the indices:

```
for (i=0; i<numLine; i++) {
    ampl[i] = 32768 * pow(10, -1.5*(ILAmplIndex+0.5)/20 );
    if ( ILFreqIndex<160 )
        freq[i] = (ILFreqIndex+0.5) * 3.125;
    else
        freq[i] = 500 * exp( 0.00625 * (ILFreqIndex+0.5-160) );
}
```

The optional start phase information is decoded as follows:

```
for (i=0; i<numLine; i++) {
    if (linePhaseAvail[i]) {
        startPhase[i] = 2*pi*(linePhase[i]+0.5)/(1<<phasebits)-pi;
        startPhaseAvail[i] = 1;
    }
    else
        startPhaseAvail[i] = 0;
}
```

If the decoding process starts with an arbitrary frame of a bitstream, all individual lines which are marked in the bitstream as to be continued from previous frames which have not been decoded are to be muted.

7.5.1.2.3.2 Enhanced decoder

The enhanced decoder refines the line parameters obtained from the basic decoder and also decodes the line phases. The additional information is contained in bitstream element INDIenhaPara() and evaluated in the following way:

First, all operations of the basic decoder have to be carried out in order to allow correct decoding of parameters for continued lines.

If envFlag == 1 then the enhanced parameters t_maxEnh, r_atkEnh, and r_decEnh are decoded using the envelope data conveyed in HILNbasicFrame() and HILNenhaFrame():

```
t_maxEnh = (envTmax+(envTmaxEnh+0.5)/(1<<tmEnhbits))/(1<<tmbits);
if (envRatk==0)
    r_atkEnh = 0;
else
    r_atkEnh = tan(pi/2*(envRatk-1+(envRatkEnh+0.5)/(1<<atkEnhbits))/
                    ((1<<atkbits)-1))/0.2;
if (envRdec==0)
    r_decEnh = 0;
else
    r_decEnh = tan(pi/2*(envRdec-1+(envRdecEnh+0.5)/(1<<decEnhbits))/
                    ((1<<decbits)-1))/0.2;
```

For each line i of the current frame

```
i = 0 .. numLine-1
```

the enhanced line parameters are obtained by refining the parameters from the basic decoder with the data in INDIenhaPara():

```
amplEnh[i] = ampl[i];
if (fEnhbits[i]!=0) {
    if ( ILFreqIndex<160 )
        freqEnh[i] = (ILFreqIndex+0.5 + ((lineFreqEnh[i]+0.5)/(1<<fEnhbits[i])-0.5)) *
3.125;
    else
        freqEnh[i] = 500 * exp( 0.00625 * (ILFreqIndex+0.5-160 +
                        ((lineFreqEnh[i]+0.5)/(1<<fEnhbits[i])-0.5)) );
}
else
    freqEnh[i] = freq[i];
```

For both line types the phase is decoded from the enhancement bitstream:

```
phaseEnh[i] = 2*pi*(linePhase[i]+0.5)/(1<<phasebits)-pi;
```

7.5.1.3 Noise decoder

7.5.1.3.1 Tool description

This tool decodes the noise parameters transmitted in the bitstream.

7.5.1.3.2 Definitions

prevNumNoisePara The number of noise LPC parameters in the previous frame.

noiseLPCPara[i] Noise LPC parameter i in the current frame (LARs for noise spectrum).

prevNoiseLPCPara[i] Noise LPC parameter i in the previous frame (LARs for noise spectrum).

noiseAmpl Noise amplitude.

noisePwr Noise power.

noiseT_max Noise envelope parameter: time of maximum.

noiseR_atk Noise envelope parameter: attack rate.

noiseR_dec Noise envelope parameter: decay rate.

7.5.1.3.3 Decoding process

7.5.1.3.3.1 Basic decoder

If the noiseFlag is set and thus NOISEbasicPara() data is available in the current frame, the parameters of the noise signal component are decoded and dequantized as follows:

The noise is represented by its power and a set of LPC-Parameters.

First the noiseNumPara LAR parameters are reconstructed. Prediction from the previous frame is used when noiseContFlag is set.

```
float noiseLPCMean[25] = { 2.0, -0.75, 0.0, 0.0, 0.0, ... , 0.0 };

for (i=0; i<numNoisePara; i++) {
    if ( i<prevNumNoisePara && noiseContFlag )
        pred = noiseLPCMean[i] + (prevNoiseLPCPara[i]-noiseLPCMean[i])*0.75;
    else
        pred = noiseLPCMean[i];
    noiseLPCPara[i] = pred + noiseLAR[i];
}
```

Parameters needed in the following frame are stored in the frame-to-frame memory:

```
prevNumNoisePara = numNoisePara;
for (i=0; i<numNoisePara; i++) {
    prevNoiseLPCPara[i] = noiseLPCPara[i];
```

The amplitude and power of the noise is dequantized as follows:

```
noiseAmpl = 32768 * pow(10, -1.5*(noiseAmplIndex+0.5)/20 );
noisePwr = noiseAmpl*noiseAmpl;
```

If noiseEnvFlag == 1 then the noise envelope parameters noiseEnvTmax, noiseEnvRatk, and noiseEnvRdec are dequantized into noiseT_max, noiseR_atk, and noiseR_dec in the same way as described for the individual line decoder (see subclause 7.5.1.2.3.1).

7.5.1.3.3.2 Enhanced decoder

Since there is no enhancement data for noise components, there is no specific enhanced decoding mode for noise parameters. If noise is to be synthesized with enhancement data present for the other components, the basic noise parameter decoder can be used. However it has to be noted that if the HILN decoder is used as a core in a scalable coder no noise signal must be synthesized for the signal which is given to the enhancement decoder.

7.5.1.4 Harmonic and individual line synthesizer

7.5.1.4.1 Tool description

This tool synthesizes the audio signal according to the harmonic and individual line parameters decoded by the corresponding decoder tools. It includes the combination of the harmonic and individual lines, the basic synthesizer and the enhanced synthesizer. To obtain the complete decoded audio signal, the output signal of this tool is added to the output signal of the noise synthesizer as described in Subclause 0.

7.5.1.4.2 Definitions

totalNumLine	Total number of lines in the current frame to be synthesized (individual plus harmonic).
sampleRate	Sampling rate in Hz as indicated by HILNsampleRateCode (see Table 29).
synthSampleRate	Sampling rate in Hz of synthesized output signal x[n].
speedFactor	Synthesis speed change factor (>1 for faster than original playback speed).
pitchFactor	Synthesis pitch change factor (>1 for higher than original playback pitch).
T	Synthesis frame length in seconds.
N	Synthesis frame length in samples.
env(t)	Amplitude envelope function in the current frame.
a(t)	Instantaneous amplitude of the line being synthesized.
p(t)	Instantaneous phase of the line being synthesized.
x(t)	Synthesized output signal.
x[n]	Sampled synthesized output signal.
startPhi[i]	Start phase of the i-th line in the current frame (in rad).
phi[i]	End phase of the i-th line in the current frame (in rad).
previousEnvFlag	Envelope flag in the previous frame.
previousT_max	Envelope parameter t_max in the previous frame.
previousR_atk	Envelope parameter r_atk in the previous frame.
previousR_dec	Envelope parameter r_dec in the previous frame.
previousEnv(t)	Amplitude envelope function in the previous frame.
previousTotalNumLine	Total number of lines in the previous frame.
previousAmpl[k]	Amplitude of the k-th line in the previous frame.
previousFreq[k]	Frequency of the k-th line in the previous frame (in Hz).
previousPhi[k]	End phase of the k-th line in the previous frame (in rad).
previousLineEnvFlag[k]	A flag indicating that the previous amplitude envelope is applied to the k-th line in the previous frame.
previousT_maxEnh	Enhanced envelope parameter t_max in the previous frame.
previousR_atkEnh	Enhanced envelope parameter r_atk in the previous frame.
previousR_decEnh	Enhanced envelope parameter r_dec in the previous frame.
previousAmplEnh[k]	Enhanced amplitude of the k-th line in the previous frame.
previousFreqEnh[k]	Enhanced frequency of the k-th line in the previous frame (in Hz).

previousPhaseEnh[k] Phase of the k-th line in the previous frame (in rad).

7.5.1.4.3 Synthesis process

7.5.1.4.3.1 Combination of harmonic and individual lines

For the synthesis of the harmonic lines the same synthesis technique as for the individual lines is used.

If no harmonic component is decoded for the following steps numHarmLine has to be set to zero.

Otherwise the parameters of the harmonic lines are appended to the list of individual line parameters as decoded by the individual line decoder:

```
for (i=0; i<numHarmLine; i++) {
    freq[numLine+i] = hLineFreq[i];
    ampl[numLine+i] = hLineAmpl[i];
    lineContFlag[numLine+i] = harmContFlag;
    linePred[numLine+i] = prevNumLine+i;
    lineEnvFlag[numLine+i] = harmEnvFlag;
    startPhase[numLine+i] = hStartPhase[i];
    startPhaseAvail[numLine+i] = hStartPhaseAvail[i];
}
```

Thus the total number of line parameters passed to the harmonic and individual line synthesizer is:

```
totalNumLine = numLine + numHarmLine;
```

Depending on the value of HILNcontMode it is possible to connect lines in adjacent frames in order to avoid phase discontinuities in the case of transitions to and from harmonic lines (HILNcontMode == 0) or additionally from individual lines to individual lines for which the continue bit lineContFlag in the bitstream was not set by the encoder (HILNcontMode == 1). This additional line continuation as described below can also be completely disabled (HILNcontMode == 2).

For each line i = 0 .. totalNumLine-1 of the current frame that has no predecessor (i.e. lineContFlag[i] == 0), the best-fitting line j of the previous frame having no successor and with the combination meeting the requirements specified by HILNcontMode as described above is determined by maximizing the following measure q:

```
df = freq[i] / previousFreq[j];
df = max(df, 1/df);
da = ampl[i] / previousAmpl[j];
da = max(da, 1/da);
q = (1 - (df-1)/(dfCont-1)) * (1 - (da-1)/(daCont-1));
```

where dfCont = 1.05 and daCont = 4 are the maximum relative frequency and amplitude changes permitted. For additional line continuations determined in this way, the line predecessor information is updated:

```
lineContFlag[i] = 1;
linePred[i] = j;
```

If there is not at least one possible predecessor with df < dfCont and da < daCont, lineContFlag[i] and linePred[i] remain unchanged.

For the enhanced synthesizer, the enhanced harmonic (up to maximum of 10) and individual line parameters are combined as follows:

```
for (i=0; i<min(10,numHarmLine); i++) {
    freqEnh[numLine+i] = hLineFreqEnh[i];
    amplEnh[numLine+i] = hLineAmplEnh[i];
    lineContFlag[numLine+i] = harmContFlag;
    linePred[numLine+i] = prevNumLine+i;
    lineEnvFlag[numLine+i] = harmEnvFlag;
    phaseEnh[numLine+i] = hLinePhaseEnh[i];
 }
```

Thus the total number of line parameters passed to the enhanced harmonic and individual line synthesizer, if the HILN decoder is used as a core in a scalable coder, is:

```
totalNumLine = numLine + min(10,numHarmLine);
```

Since phase information is available for all of these lines , no line continuation is introduced for the enhanced synthesizer.

7.5.1.4.3.2 Speed and pitch change

Due to the parametric signal representation utilized by the HILN coder and the phase continuation provided by the basic line synthesizer, the playback speed and pitch can be easily modified during the signal synthesis in the decoder. If playback at the original speed and pitch is desired, the corresponding control factors are set to their default values:

```
speedFactor = 1.0;
pitchFactor = 1.0;
```

When the enhanced synthesizer is used instead of the basic synthesiser, speedFactor and pitchFactor should always be set to their default value 1.0.

Speed change is realized by modifying the synthesis frame length according to the desired speedFactor as described in Subclause 7.5.1.4.3.3.

Pitch change is realized by modifying the frequency parameters of the harmonic and individual lines as follows:

```
for (i=0; i<totalNumLine; i++) {
    freq[i] *= pitchFactor;
}
```

The noise synthesizer described in Subclause 7.5.1.4.3.3 also supports speed and pitch change as detailed there.

7.5.1.4.3.3 Synthesis framing

The harmonic and individual line synthesizer reconstructs one frame of the audio signal. Since the line parameters encoded in a bitstream frame are valid for the center of the corresponding frame of the audio signal, the synthesizer generates the one-frame long section of the audio signal x(t) that starts at the center of the previous frame (t = 0) and that ends at the center of the current frame (t = T).

As default, the HILN synthesizer operates at the sampling frequency synthSampleRate as indicated by the samplingFrequency conveyed in the AudioSpecificInfo() (see Table 8):

```
synthSampleRate = samplingFrequency
```

The synthesis frame contains N samples:

```
N = (int)(HILNframeLength * synthSampleRate / sampleRate / speedFactor + 0.5);
```

Thus the duration T of the synthesis frame is:

```
T = N / synthSampleRate;
```

In the following, the calculation of the synthesized output signal

```
x(t)
```

for 0 <= t < T is described. The time discrete version (i.e. the actual frame of output samples) is defined as

```
x[n] = x(t) with t = (n+0.5)*(T/N)
```

for 0 <= n < N.

The noise synthesizer described in Subclause 0 uses the same synthesis framing as the harmonic and individual line synthesizer described here.

7.5.1.4.3.4 Basic synthesizer

Some parameters of the previous frame (names starting with "previous") are taken out of the frame-to-frame memory which has to be reset before decoding the first frame of a bitstream.

First the envelope functions previousEnv(t) and env(t) of the previous and current frame are calculated according to the following rules:

If envFlag == 1 then the envelope function env(t) is derived from the envelope parameters t_max, r_atk, and r_dec. With T being the frame length, env(t) is calculated for -T/2 <= t < 3/2*T:

```
if (-1/2 <= t/T && t/T < t_max)
    env(t) = max(0,1-(t_max-t/T)*r_atk);
if (t_max <= t/T && t/T < 3/2)
    env(t) = max(0,1-(t/T-t_max)*r_dec);
```

If envFlag == 0 then a constant envelope function env(t) is used:

```
env(t) = 1;
```

Accordingly previousEnv(t) is calculated from the parameters previousT_max, previousR_atk, previousR_dec and previousEnvFlag.

The envelope parameters transmitted in case of envFlag == 1 are valid for the harmonic lines as well as for the individual lines. Thus the envelope functions always must be generated, even if all lineEnvFlag[i] == 0.

Before the synthesis is performed, the accumulator x(t) for the synthesized audio signal is cleared for 0 <= t < T:

```
x(t) = 0;
```

The lines i continuing from the previous frame to the current frame

```
all i=0 .. totalNumLine-1 that have lineContFlag[i] == 1
```

are synthesized as follows for 0 <= t < T:

```
k = linePred[i];
ap(t) = previousAmpl[k];
if (previousLineEnvFlag[k] == 1)
    ap(t) *= previousEnv(t+T/2);
ac(t) = ampl[k];
if (lineEnvFlag[i] == 1)
    ac(t) *= env(t-T/2);
short_x_fade = (previousLineEnvFlag[k] && !(previousR_atk < 5 &&
                (previousT_max > 0.5 || previousR_dec < 5))) ||
                (lineEnvFlag[i] && !(r_dec < 5 && (t_max < 0.5 || r_atk < 5)));
if (short_x_fade == 1) {
    if (0 <=t && t < 7/16*T)
        a(t) = ap(t);
    if (7/16*T <=t && t < 9/16*T)
        a(t) = ap(t) + (ac(t)-ap(t))*(t/T-7/16)*8;
    if (9/16*T <= t&& t < T)
        a(t) = ac(t);
}
else
    a(t) = ap(t) + (ac(t)-ap(t))*t/T;
p(t) = previousPhi[k]+2*pi*previousFreq[k]*t+
```

```
                  2*pi*(freq[i]-previousFreq[k])/(2*T)*t*t;
x(t) += a(t)*sin(p(t));
phi[i] = p(T)
```

The lines i starting in the current frame

```
all i=0 .. totalNumLine-1 that have lineContFlag[i] == 0
```

are synthesized as follows for 0 <= t <T:

```
if (lineEnvFlag[i] && !(r_dec < 5 && (t_max < 0.5 || r_atk < 5))) {
   if (0 <= t&& t < 7/16*T)
      fade_in(t) = 0;
   if (7/16*T <= t&& t < 9/16*T)
      fade_in(t) = 0.5 - 0.5*cos((8*t/T-7/2)*pi);
   if (9/16*T <= t&& t < T)
      fade_in(t) = 1;
}
else
   fade_in(t) = 0.5-0.5*cos(t/T*pi);
a(t) = fade_in(t)*ampl[i];
if (lineEnvFlag[i] == 1)
   a(t) *= env(t-T/2);
if (startPhaseAvail[i])
   startPhi[i] = startPhase[i];
else
   startPhi[i] = random(2*pi);
p(t) = startPhi[i] + 2*pi*freq[i]*(t-T);
x(t) += a(t)*sin(p(t));
phi[i] = p(T)
```

random(x) is a function returning a random number with uniform distribution in the interval

```
0 <= random(x) < x
```

The lines k ending in the previous frame

```
all k=0 .. previousTotalNumLine-1 that have prevLineContFlag[k] == 0
```

are synthesized as follows for 0 <= t < T:

```
if (previousLineEnvFlag[k] && !(previousR_atk < 5 &&
      (previousT_max > 0.5 || previousR_dec < 5))) {
   if (0 <= t&& t < 7/16*T)
      fade_out(t) = 1;
   if (7/16*T <= t&& t < 9/16*T)
      fade_out(t) = 0.5 + 0.5*cos((8*t/T-7/2)*pi);
   if (9/16*T <= t&& t < T)
      fade_out(t) = 0;
}
else
   fade_out(t) = 0.5+0.5*cos(t/T*pi);
a(t) = fade_out(t)*previousAmpl[k];
if (previousLineEnvFlag[k] == 1)
   a(t) *= previousEnv(t+T/2);
p(t) = previousPhi[k]+2*pi*previousFreq[k]*t;
x(t) += a(t)*sin(p(t));
```

In order to avoid aliasing distortion, synthesized lines are muted (i.e. a(t) = 0) while their instantaneous frequency is above or equal half the sampling frequency, i.e.

```
d phi(t) / dt >= pi*N/T
```

Parameters needed in the following frame are stored in the frame-to-frame memory:

```
previousEnvFlag = envFlag;
previousT_max = t_max;
previousR_atk = r_atk;
previousR_dec = r_dec;
previousTotalNumLine = totalNumLine;
for (i=0; i<totalNumLine; i++) {
    previousFreq[i] = freq[i];
    previousAmpl[i] = ampl[i];
    previousPhi[i] = fmod(p[i],2*pi);
    previousLineEnvFlag[i] = lineEnvFlag[i];
}
```

fmod(x,2*pi) is a function returning the 2*pi modulus of x.

7.5.1.4.3.5 Enhanced synthesizer

The enhanced synthesizer is based on the basic synthesizer but evaluates also the line phases for reconstructing one frame of the audio signal. Since the line parameters encoded in a bitstream frame and the corresponding enhancement frame are valid for the middle of the corresponding frame of the audio signal, the harmonic and individual line synthesizer generates the one frame long section of the audio signal that starts in the middle of the previous frame and ends in the middle of the current frame.

Some parameters of the previous frame (names starting with "previous") are taken out of the frame-to-frame memory which has to be reset before decoding the first frame of a bitstream.

First the envelope functions previousEnv(t) and env(t) of the previous and current frame are calculated according to the following rules:

If envFlag == 1 then the envelope function env(t) is derived from the envelope parameters t_maxEnh, r_atkEnh, and r_decEnh. With T being the frame length, env(t) is calculated for -T/2 <= t < 3/2*T:

```
if (-1/2 <= t/T && t/T < t_maxEnh)
    env(t) = max(0,1-(t_maxEnh-t/T)*r_atkEnh);
if (t_maxEnh <= t/T && t/T < 3/2)
    env(t) = max(0,1-(t/T-t_maxEnh)*r_decEnh);
```

If envFlag == 0 then a constant envelope function env(t) is used:

```
env(t) = 1;
```

Accordingly previousEnv(t) is calculated from the parameters previousT_maxEnh, previousR_atkEnh, previousR_decEnh and previousEnvFlag.

The envelope parameters transmitted in case of envFlag == 1 are valid for the harmonic lines as well as for the individual lines. Thus the envelope functions always must be generated, even if all lineEnvFlag[i] == 0.

Before the synthesis is performed, the accumulator x(t) for the synthesized audio signal is cleared for 0 <= t < T:

```
x(t) = 0
```

All lines i in the in the current frame

```
all i =0 .. totalNumLine-1
```

are synthesized as follows for 0 <= t < T:

```
if (envFlag && !(r_decEnh < 5 && (t_maxEnh < 0.5 || r_atkEnh < 5))) {
    if (0 <= t&& t < 7/16*T)
        fade_in(t) = 0;
    if (7/16*T <= t&& t < 9/16*T)
        fade_in(t) = 0.5 - 0.5*cos((8*t/T-7/2)*pi);
    if (9/16*T <= t&& t < T)
        fade_in(t) = 1;
```

```
}
else
   fade_in(t) = 0.5-0.5*cos(t/T*pi);
a(t) = fade_in(t)*amplEnh[i];
if (envFlag[i] == 1)
   a(t) *= env(t-T/2);
phi(t) = 2*pi*freqEnh[i]*(t-T)+phaseEnh[i];
x(t) += a(t)*sin(phi(t));
```

The lines k in the previous frame

```
all k=0 .. previousTotalNumLine-1
```

are synthesized as follows for 0 <= t < T:

```
if (previousEnvFlag && !(previousR_atkEnh < 5 &&
      (previousT_maxEnh > 0.5 || previousR_decEnh < 5))) {
   if (0 <= t&& t < 7/16*T)
      fade_out(t) = 1;
   if (7/16*T <= t&& t < 9/16*T)
      fade_out(t) = 0.5 + 0.5*cos((8*t/T-7/2)*pi);
   if (9/16*T <= t&& t < T)
      fade_out(t) = 0;
}
else
   fade_out(t) = 0.5+0.5*cos(t/T*pi);
a(t) = fade_out(t)*previousAmplEnh[k];
if (previousEnvFlag[k] == 1)
   a(t) *= previousEnv(t+T/2);
phi(t) = 2*pi*previousFreqEnh[k]*t+previousPhaseEnh[i];
x(t) += a(t)*sin(phi(t));
```

If the instantaneous frequency of a line is above or equal half the sampling frequency, i.e.

```
d phi(t) / dt >= pi*N/T
```

it is not synthesized to avoid aliasing distortion.

Parameters needed in the following frame are stored in the frame-to-frame memory:

```
previousEnvFlag = envFlag;
previousT_maxEnh = t_maxEnh;
previousR_atkEnh = r_atkEnh;
previousR_decEnh = r_decEnh;
previousTotalNumLine = totalNumLine;
for (i=0; i<totalNumLine; i++) {
   previousFreqEnh[i] = freqEnh[i];
   previousAmplEnh [i] = amplEnh[i];
   previousPhaseEnh [i] = phaseEnh [i];
}
```

7.5.1.5 Noise synthesizer

7.5.1.5.1 Tool description

This tool synthesizes the noise part of the output signal based on the noise parameters decoded by the noise decoder. Finally the noise signal is added to the output signal of the harmonic and individual line synthesizer (Subclause 7.5.1.4) to obtain the complete decoded audio signal

7.5.1.5.2 Definitions

noiseWin[n] Window for noise overlap-add.

noiseEnv[n] Envelope for noise component in the current frame.

M	Frame length in samples before resampling.
w[m]	White noise with power pw.
xf[m]	Filtered noise signal in the current frame.
xn[n]	Synthesized noise signal in the current frame.
previousXn[n]	Synthesized noise signal in the previous frame.
previousNoiseWin[n]	Window and envelope for noise component in the previous frame.
r[i]	LPC reflection coefficients.
h[i]	LPC impulse response.
hlp[i]	Low-pass resampling filter impulse response.

7.5.1.5.3 Synthesis process

7.5.1.5.3.1 Basic synthesizer

If noise parameters are transmitted for the current frame, a noise signal with a spectral shape as described by the noise parameters decoded from the bitstream is synthesized and added to the audio signal generated by the harmonic and individual line synthesizer. The synthesis framing is as described in Subclause 7.5.1.4.3.3.

The noise is represented by its power and a set of LPC-Parameters. As described in the harmonic tone decoder (Subclause 7.5.1.1.3.1), the noise LPC parameters are converted to the reflection coefficients r[i] and to the time response h[i]:

```
for ( i = 0; i < numNoisePara; i++ )
    r[i] = ( exp(noiseLPCPara[i]) - 1 ) / ( exp(noiseLPCPara[i]) + 1 );
```

After this the reflection coefficients r[i] are converted to the time response h[i] using the C function

```
void Convert_k_to_h (float *x, int N)
```

given in Subclause 7.5.1.1.3.1 (call with x[i]=r[i] and N=numNoisePara; returns with x[i]=h[i]).

Now the filtered noise signal xf[m] is generated by applying the LPC synthesis IIR filter to a white noise represented by random numbers w[m]. The power of this zero-mean white noise is denoted pw. For a noise with uniform distribution in [-1,1] the power is

```
pw = 1/3
```

To achieve the required noise signal power, the following scaling factor s is required:

```
ss = 1.0;
for (i=0; i<numNoisePara; i++)
    ss *= 1-r[i]*r[i];
s = noiseAmpl * sqrt( ss/pw );
```

Then the white noise w[m] is IIR-filtered to obtain the synthesized noise signal xf[m]

```
for (m=startup; m<2*M; m++) {
    xf[m] = s * w[m];
    for (i=0; i<min(m-startup,numNoisePara); i++)
        xf[m] += h[i]*xf[m-i-1];
}
```

To ensure that the IIR filter can reach a sufficiently steady state, a startup phase is used:

```
startup = -numNoisePara
```

If decoded with a pitch change (i.e. pitchPactor != 1.0) or with a different sample rate than the encoder (i.e. synthSampleRate != sampleRate), a resample operation must be applied to the signal xf[m] using the resampling factor

```
resampleFactor = ( sampleRate * pitchFactor ) / synthSampleRate;
```

where e.g. pitchFactor of 2 indicates that this signal is synthesized at twice its original pitch. Otherwise the resampleFactor is set to 1.0. Based on the resampleFactor, the framle length M before resampling is defined:

```
M = N * resampleFactor;
```

The resampling can be realized by applying two lowpass FIR filter operations to the signal xf[m] and linearly interpolating between these two values to obtain the final noise signal xn[n].

```
if ( resampleFactor<1 )
    fc = 1;
else
    fc = 1/resampleFactor;
```

The following function calculates the time response hlp[0..31] of an appropriate lowpass FIR filter with 16 taps and an oversampling factor of 4. The cutoff frequency is fc.

```
void GenLPFilter (float *hlp, double fc)
{
    double x,f;
    int i;

    hlp[0] = (float) fc;
    for (i=1; i<32; i++)
    {
        x = i*pi/4.0;
        hlp[i] = (float) ((0.54+0.46*cos(0.125*x))*sin(fc*x)/x);
    }
}
```

To perform the FIR filter operation the following C function can be used. The parameters are the signal, the time response (as returned by the function above) and the position of the sampling point. The position is given as the difference between the nearest sample position prior to the desired sample position (x[7]) and the desired sample position. Therefore 0 <= pos < 1. The interpolation is done between x[7] and x[8], the returned value represents a sample position of 7+pos.

```
float LPInterpolate (float *x, float *hlp, double pos)
{
    long   j;
    double s,t;

    pos *= 4.0;
    j = (long) pos;
    pos -= (double) j;

    s = t = 0.0;
    j = 32-j;

    if ( j==32 )
    {
        t = h[31]*(*x);
        x++;
        j -= 4;
    }
    while (j>0)
    {
        s += h[j ]*(*x);
        t += h[j-1]*(*x);
```

```
            x++;
            j -= 4;
        }
        j = -j;
        while (j<32-1)
        {
            s += h[j ]*(*x);
            t += h[j+1]*(*x);
            x++;
            j += 4;
        }
        if (j<32)
            s += h[j]*(*x);

        return (float) (s+pos*(t-s));
    }
```

Using the functions GenLPFilter() and LPInterpolate(), the resampling is done as described below. xf[m] is set to 0.0 for m < startup and m >= 2*M.

```
    GenLPFilter(hlp, fc);
    for (n=0; n<2*N; n++)
        xn[n] = LPInterpolate(xf+((int)n*resampleFactor)-7, hlp, frac(n*resampleFactor));
```

If resampleFactor == 1.0, xf[m] is simply copied to xn[n] without resampling:

```
    for (n=0; n<2*N; n++)
        xn[n] = xf[n];
```

For smooth cross-fade of the noise signal at the boundary between two adjacent frames, the following window is used for this overlap-add operation:

```
    for (n=0; n<N; n++) {
        if (n < N*3/8)
            noiseWin[n] = 0;
        if (N*3/8 <= n && n < N*5/8)
            noiseWin[n] = sin(pi/2 * (n-N*3/8+0.5)/(N*2/8));
        if (N*5/8 <= n)
            noiseWin[n] = 1;
        noiseWin[2*N-1-n] = noiseWin[n];
    }
```

Now the envelope function noiseEnv[n] is calculated. If noiseEnvFlag == 1 then the envelope function

```
    noiseEnv[n] = noiseEnv(t) with t = (n+0.5)*(T/N)-0.5
```

is derived from the envelope parameters noiseT_max, noiseR_atk, and noiseR_dec for -T/2 <= t < 3/2*T (i.e. 0 <= n < 2*N):

```
    if (-1/2 <= t/T && t/T < noiseT_max)
        noiseEnv(t) = max(0,1-(noiseT_max-t/T)*noiseR_atk);
    if (noiseT_max <= t/T && t/T < 3/2)
        noiseEnv(t) = max(0,1-(t/T-noiseT_max)*noiseR_dec);
```

If noiseEnvFlag == 0 then a constant envelope function noiseEnv(t) is used:

```
    noiseEnv[n] = 1;
```

The noise signal xn[n] is windowed for overlap-add and multiplied with the envelope noiseEnv[n]. Then this signal and the noise from the previous frame previousXn[n] are added to the signal x[n] from the harmonic and individual line synthesizer to construct the complete synthesized signalx[n]:

```
    for (n=0; n<N; n++)
        x[n] += xn[n]*noiseWin[n]*noiseEnv[n] + previousXn[n];
```

The second half of the generated noise signalxn[n] is stored in the frame-to-frame memory previousXn[n] for overlap-add:

```
for (n=0; n<N; n++)
    previousXn[n] = xn[N+n]*noiseWin[N+n]*noiseEnv[N+n];
```

The previousXn[n] memory has to be reset to 0.0 before decoding of the first frame.

7.5.1.5.3.2 Enhanced synthesizer

Since there is no enhancement data for noise components, there is no specific enhanced synthesizer mode for noise components. If noise is to be synthesized with enhancement data present for the other components, the basic noise synthesizer decoder can be used. However it has to be noted that if the HILN decoder is used as a core in a scalable coder no noise signal must be synthesized for the signal which is given to the enhancement decoder.

7.5.2 Integrated parametric coder

The integrated parametric coder can operate in four different modes as shown in Table 24. PARAmodes 0 and 1 represent the fixed HVXC and HILN modes. PARAmode 2 permits automatic switching between HVXC and HILN depending on the current input signal type. In PARAmode 3 the HVXC and HILN coders can be used simultaneously and their output signals are added (mixed) in the decoder.

The integrated parametric coder uses a frame length of 40 ms and a sampling rate of 8 kHz and can operate at 2025 bit/s or any higher bitrate. Operation at 4 kbit/s or higher is suggested.

7.5.2.1 Integrated parametric decoder

For the “HVXC only” and “HILN only” modes the parametric decoder is not modified.

In “switched HVXC / HILN” and “mixed HVXC / HILN” modes both HVXC and HILN decoder tools are operated alternatively or simultaneously according to the PARAswitchMode or PARAmixMode of the current frame. To obtain proper time alignment of both HVXC and HILN decoder output signals before they are added, the difference between HVXC and HILN decoder delay has to be compensated with a FIFO buffer:

- If HVXC is used in the low delay decoder mode, its output must be delayed for 100 samples (i.e. 12.5 ms).
- If HVXC is used in the normal delay decoder mode, its output must be delayed for 80 samples (i.e. 10 ms).

To avoid hard transitions at frame boundaries when the HVXC or HILN decoders are switched on or off, the respective decoder output signals are faded in and out smoothly. For the HVXC decoder a 20 ms linear fade is applied when it is switched on or off. The HILN decoder requires no additional fading because of the smooth synthesis windows utilized in the HILN synthesizer. It is only necessary to operate the HILN decoder with no new components for the current frame (i.e. force numLine = 0, harmFlag = 0, noiseFlag = 0) if the current bitstream frame contains no “HILNframe()”.

8 Extension to General Audio Coding

8.1 Encoder and decoder block diagrams

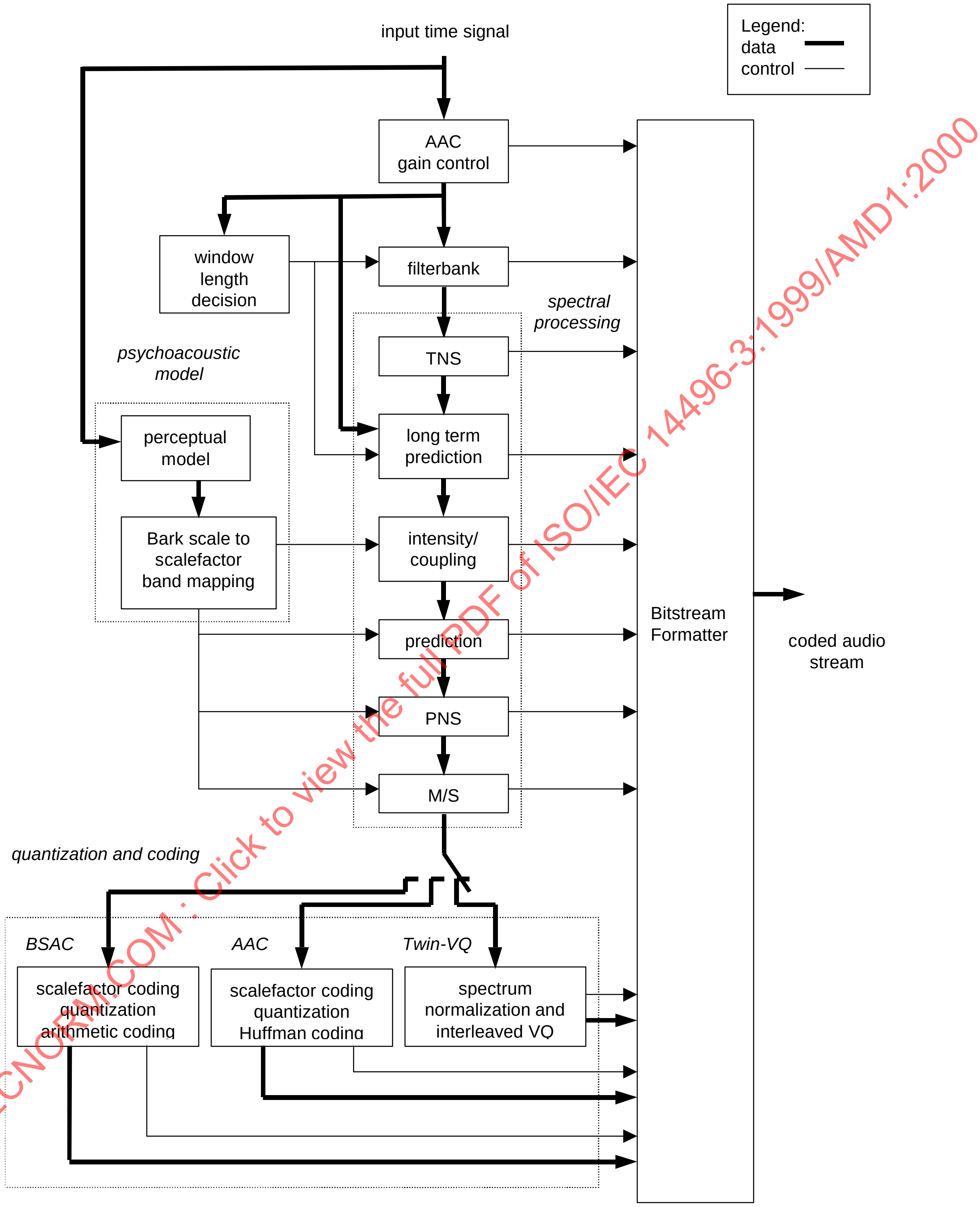


Figure 5: Block diagram of GA non scalable encoder

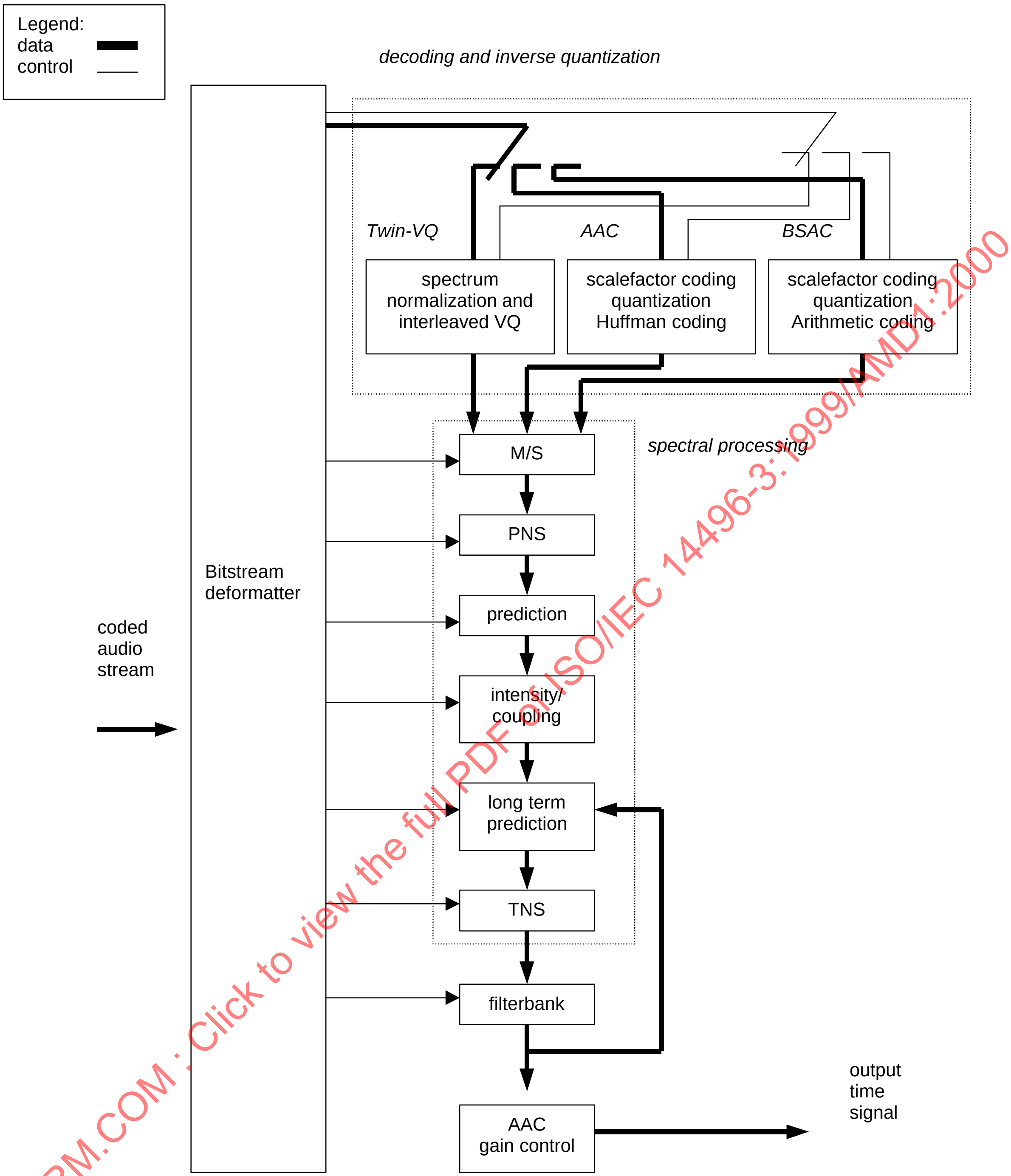


Figure 6: Block diagram of GA non scalable decoder

8.2 Decoder configuration (GASpecificConfig)

8.2.1 Syntax

Table 80: Syntax of GASpecificConfig()

Syntax	No. of bits	Mnemonic
GASpecificConfig (samplingFrequencyIndex, channelConfiguration, audioObjectType)		
{		
frameLengthFlag;	**1**	**bslbf**
dependsOnCoreCoder;	**1**	**bslbf**
if (dependsOnCoreCoder) {		
coreCoderDelay;	**14**	**uimsbf**
}		
extensionFlag;	**1**	**bslbf**
if (! channelConfiguration) {		
program_config_element ();		
}		
if (extensionFlag) {		
if (audioObjectType == 22) {		
numOfSubFrame;	**5**	**bslbf**
layer_length;	**11**	**bslbf**
}		
if (audioObjectType == 17 || audioObjectType == 18 || audioObjectType == 19 || audioObjectType == 20 || audioObjectType == 21 || audioObjectType == 23) {		
aacSectionDataResilienceFlag;	**1**	**bslbf**
aacScalefactorDataResilienceFlag;	**1**	**bslbf**
aacSpectralDataResilienceFlag;	**1**	**bslbf**
}		
extensionFlag3;	**1**	**bslbf**
if (extensionFlag3) {		
/* tbd in version 3 */		
}		
}		
}		

8.2.2 Semantics

Within ISO/IEC 14496-3, subpart 4 (GA) chapter 5 (General Information), section 5.1 (Decoding of GA specific configuration), sub-section 5.1.1 GA SpecificConfig has to be applied. In addition, the following data elements have to be considered:

numOfSubFrame A 5-bit unsigned integer value representing the number of the sub-frames which are grouped and transmitted in a super-frame.

layer_length An 11-bit unsigned integer value representing the average length of the large-step layers in bytes.

aacSectionDataResilienceFlag This flag signals a different coding scheme of AAC section data. If codebook 11 is used, this scheme transmits additional information about the maximum absolute value for spectral lines. This allows error detection of spectral lines that are larger than this value.

aacScalefactorDataResilienceFlag This flag signals a different coding scheme of the AAC scalefactor data, that is more resilient against errors as the original one.

aacSpectralDataResilienceFlag This flag signals a different coding scheme (HCR) of the AAC spectral data, that is more resilient against errors as the original one

8.3 Fine granule audio (BSAC)

8.3.1 Overview of tools

BSAC stands for bit sliced arithmetic coding and is the name of a noiseless coding kernel that provides a fine grain scalability and the error resilience in the MPEG-4 General Audio(GA) coder. The BSAC noiseless coding module is an alternative to the AAC coding module, with all other modules of the AAC-based coder remaining unchanged. The BSAC noiseless coding is used to make the bitstream scalable and error-resilient and further reduce the redundancy of the scalefactors and the quantized spectrum

The inputs to the BSAC decoding tool are:

- The noiselessly coded bit-sliced data
- The target layer information to be decoded

The outputs from the BSAC decoding tool are:

- The decoded integer representation of the scalefactors
- The quantized value for the spectra

8.3.2 Bitstream syntax

8.3.2.1 Bitstream payload

Table 81: Syntax of top level payload for audio object type *ER BSAC* (bsac_payload())

Syntax	No. of bits	Mnemonic
bsac_payload(lay)		
{		
for (frm=0; frm<numOfSubFrame; frm++) {		
bsac_lstep_element(frm, lay);		
}		
/*		
bsac_lstep_element(frm, lay) should be mapped to the fine grain audio data, bsac_raw_data_block(), for the actual decoding. See subclause “Decoding of payload for audio object type *ER BSAC*” for more detailed description.*/		
}		

Table 82: Syntax of bsac_lstep_element()

Syntax	No. of bits	Mnemonic
bsac_lstep_element(frm, lay)		
{		
offset=LayerStartByte[frm][lay];		
for(i=0;i<LayerLength[frm][lay];i++)		
bsac_stream_byte[frm][offset+i];	**8**	**uimsbf**
/*		
bsac_stream_byte should be mapped to the fine grain audio data, bsac_raw_data_block(), for the actual decoding. See subclause “Decoding of payload for audio object type *ER BSAC*” for more detailed description.		
*/		
}		

Table 83: bsac_raw_data_block()

Syntax	No. of bits	Mnemonic
bsac_raw_data_block()		
{		
bsac_base_element();		
layer=slayer_size;		
while(data_available() && layer<(top_layer+slayer_size)) {		
bsac_layer_element(nch, layer);		
layer++;		
}		
byte_alignment();		
}		

Table 84: Syntax of bsac_base_element()

Syntax	No. of bits	Mnemonic
bsac_base_element()		
{		
frame_length;	**11**	**uimbf**
bsac_header();		
general_header();		
byte_alignment();		
for (slayer = 0; slayer < slayer_size; slayer++)		
bsac_layer_element(slayer);		
}		

Table 85: Syntax of bsac_header()

Syntax	No. of bits	Mnemonic
bsac_ header()		
{		
header_length;	**4**	**uimbf**
sba_mode;	**1**	**uimbf**
top_layer;	**6**	**uimbf**
base_snf_thr;	**2**	**uimbf**
for(ch=0;ch<nch;ch++)		
max_scalefactor[ch];	**8**	**uimbf**
base_band;	**5**	**uimbf**
for(ch=0;ch<nch;ch++) {		
cband_si_type[ch];	**5**	**uimbf**
bsae_scf_model[ch];	**3**	**uimbf**
enh_scf_model[ch];	**3**	**uimbf**
max_sfb_si_len[ch];	**4**	**uimbf**
}		
}		

Table 86: Syntax of general_header()

Syntax	No. of bits	Mnemonic
general_header ()		
{		
reserved_bit;	**1**	**bslbf**
window_sequence;	**2**	**uimsbf**
window_shape;	**1**	**uimsbf**
if(window_sequence == EIGHT_SHORT_SEQUENCE) {		
max_sfb;	**4**	**uimsbf**
scale_factor_grouping;	**7**	**uimsbf**
} else {		
max_sfb;	**6**	**uimsbf**
}		
pns_data_present;	**1**	**uimbf**
if (pns_data_present**)**		
pns_start_sfb;	**6**	**uimbf**
if(nch == 2)		
ms_mask_present;	**2**	**bslbf**
for(ch=0 ch< nch; ch++) {		
tns_data_present[ch];	**1**	**bslbf**
if(tns_data_present[ch])		
tns_data();		
ltp_data_present[ch];	**1**	**bslbf**
if(ltp_data_present[ch])		
ltp_data(last_max_sfb, max_sfb);		
}		
}		

Table 87: Syntax of bsac_layer_element()

Syntax	No. of bits	Mnemonic
bsac_layer_element(layer)		
{		
layer_cband_si(layer);		
layer_sfb_si(layer);		
bsac_layer_spectra (layer);		
if (!sba_mode) {		
bsac_lower_spectra (layer)		
}		
else if (terminal_layer[layer]) {		
bsac_lower_spectra (layer);		
bsac_higher_spectra (layer);		
}		
}		

Table 88: Syntax of layer_cband_si()

Syntax	No. of bits	Mnemonic
layer_cband_si (layer)		
{		
g = layer_group[layer];		
for (ch=0; ch<nch; ch++) {		
for(cband=layer_start_cband[g][layer];		
cband<layer_end_cband[g][layer]; cband++) {		
acode_cband_si[ch][g][cband];	**1..14**	**bslbf**
}		
}		
}		

Table 89: Syntax of layer_sfb_si()

Syntax	No. of bits	Mnemonic
layer_sfb_si (layer)		
{		
g = layer_group[layer];		
for (ch=0; ch<nch; ch++)		
for(sfb=layer_start_sfb[layer];sfb<layer_end_sfb[layer];sfb++) {		
if (nch==1) {		
if(pns_data_present && sfb >= pns_start_sfb) {		
acode_noise_flag[g][sfb];	**1**	**bslbf**
}		
} else if (stereo_side_info_coded[g][sfb]==0) {		
if (ms_mask_present !=2) {		
if (ms_mask_present==1) {		
acode_ms_used[g][sfb];	**1**	**bslbf**
pns_data_present = 0;		
} else if (ms_mask_present==3) {		
acode_stereo_info[g][sfb];	**0..4**	**bslbf**
}		
if(pns_data_present && sfb>=pns_start_sfb) {		
acode_noise_flag_l[g][sfb];	**1**	**bslbf**
acode_noise_flag_r[g][sfb];	**1**	**bslbf**
if(ms_mask_present==3 && stereo_info==3) {		
if(noise_flag_l && noise_flag_r){		
acode_noise_mode[g][sfb];	**2**	**bslbf**
}		
}		
}		
}		
stereo_side_info_coded[g][sfb] = 1;		
}		
if (noise_flag[ch][g][sfb]) {		
if (noise_pcm_flag[ch]==1) {		
acode_max_noise_energy[ch];	**9**	**bslbf**
noise_pcm_flag[ch] = 0;		
}		
acode_dpcm_noise_energy_index[ch][g][sfb];	**0..14**	**bslbf**
} else if (stereo_info[g][sfb]>=2 && ch==1) {		
acode_is_position_index[g][sfb];	**0..14**	**bslbf**
} else {		
acode_scf_index[ch][g][sfb];	**1..14**	**bslbf**
}		
}		
}		

Table 90: Syntax of bsac_layer_spectra()

Syntax	No. of bits	Mnemonic
bsac_layer_spectra(layer)		
{		
g = layer_group[layer];		
start_index[g] = layer_start_ index[layer];		
end_index[g] = layer_end_ index[layer];		
if (layer < slayer_size)		
thr_snf = base_snf_thr;		
else		
thr_snf = 0;		
bsac_spectral_data (g, g+1, thr_snf, cur_snf);		
}		

Table 91: Syntax of bsac_lower_spectra()

Syntax	No. of bits	Mnemonic
bsac_lower_spectra(layer)		
{		
for (g = 0; g < num_window_groups; g++) {		
start_index[g] = 0;		
end_index[g] = 0;		
}		
for (play = 0; play < layer; play++) {		
end_index[-layer_group[play]] = layer_end_index[play];		
}		
bsac_spectral_data (0, num_window_groups, 0, unc_snf);		
}		

Table 92: Syntax of bsac_higher_spectra()

Syntax	No. of bits	Mnemonic
bsac_higher_spectra(layer)		
{		
for (nlay=layer+1;nlay < top_layer+slayer_size;nlay++) {		
g = layer_group[nlay];		
start_index[g] = layer_start_index[nlay];		
end_index[g] = layer_end_index[nlay];		
bsac_spectral_data (g, g+1, 0, unc_snf);		
}		
}		

Table 93: Syntax of bsac_spectral_data ()

Syntax	No. of bits	Mnemonic
bsac_spectral_data(start_g, end_g, thr_snf, cur_snf)		
{		
if (layer_data_available()) return;		
for (snf=maxsnf; snf>thr_snf; snf--)		
for (g = start_g; g < end_g; g++)		
for (i=start_index[g];i<end_index[g]; i++)		
for(ch=0;ch<nch;ch++) {		
if (cur_snf[ch][g][i]<snf) continue;		
if (!sample[ch][g][i] || sign_is_coded[ch] [g][i])		
acod_sliced_bit[ch][g][i][snf];	**0..6**	**bslbf**
if (sample[ch][g][i] && !sign_is_coded[ch] [g][i]) {		
if (layer_data_available()) return;		
acod_sign[ch][g][i];	**1**	**bslbf**
sign_is_coded[ch][g][i] = 1;		
}		
cur_snf[ch][g][i]--;		
if (layer_data_available()) return;		
}		
}		

8.3.3 General information

8.3.3.1 Decoding of payload for audio object type ER BSAC (bsac_ payload())

Fine grain scalability would create large overhead if one would try to transmit fine grain layers over multiple elementary streams (ES). So, in order to reduce overhead and implement the fine grain scalability efficiently in current MPEG-4 system, the server can organize the fine grain audio data into the payload by dividing the fine grain audio data into the large-step layers and concatenating the large step layers of the several sub-frames. Then the payload is transmitted over ES.

So, the payload transmitted over ES requires the rearrangement process for the actual decoding.

8.3.3.1.1 Definitions

bsac_payload(lay) Sequnece of bsac_lstep_element()s. Syntactic element of the payload transmitted over layth layer ES. A bsac_payload(lay) basically consists of several layth layer bitstream, bsac_lstep_element() of serveral sub-frames.

bsac_lstep_element(frm, lay) Syntactic element for the layth large-step layer bitstream of frmth sub-frame.

bsac_stream_byte[frm][offset+i] (offset+i)-th byte which is extracted from the payload. After bsac stream bytes are extracted from all the payloads that have been transmitted to the receiver, these data are concatenated and saved in the array *bsac_stream_byte[frm][]* which is the bitstream of frmth sub-frame. Then, we proceed to decode the concatenated stream,*bsac_stream_byte[frm][]* using the syntax of the BSAC fine grain scalability.

Help elements:

data_available() function that returns “1” as long as data is available, otherwise “0”

LayerStartByte[frm][lay] Start position of layth large-step layer in bytes which is located on frmth sub-frame’s bitstream. See sub-clause 8.3.3.1.2 for the calculation process of this value.

LayerLength[frm][lay]	Length of the large-step layer in bytes which is located on the payload of lay^{th} layer ES and concatenated to frm^{th} sub-frame's bitstream. See sub-clause 8.3.3.1.2 for the calculation process of this value.
LayerOffset[frm][lay]	Start position of the large-step layer of frm^{th} frame in bytes which is located on the payload of lay^{th} layer ES. See sub-clause 8.3.3.1.2 for the calculation process of this value.
frm	index of frame in which bsac stream bytes are saved.
lay	index of the large-step layer over which the fine granule audio data is transmitted.
numOfSubFrame	Number of the sub-frames which are grouped and transmitted in a super-frame in order to reduce the transmission overhead.
layer_length	Average length of the large-step layers in bytes which are assembled in a payload.
numOfLayer	number of the large-step layers which the fine grain audio data is divided into.

8.3.3.1.2 Decoding process

On the sync layer (SL) of MPEG-4 system, an elementary stream is packetized into access units or parts thereof. Such a packet is called SL packet. Access Unit(AU)s are the only semantic entities at the sync layer (SL) of MPEG-4 system that need to be preserved from end to end. AUs are used as the basic unit for synchronization which are made up of one or more SL packets.

The dynamic data for the BSAC is transmitted as SL_Packet payload in the base layer Elementary Stream(ES) and the enhancement layer ESs. The dynamic data is made up of the large-step layers of one or more subsequent sub-frames.

When the SL packets of an AU arrives in the receiver, a sequence of packet is mapped into a payload which is split into the large step layers, bsac_lstep_layer(frm, lay) for the subsequent sub-frames. And the split layers should be concatenated with the large-step layers which are transmitted over the other ES.

In the receiver, BSAC data is reconstructed from the payloads as shown in Figure 7.

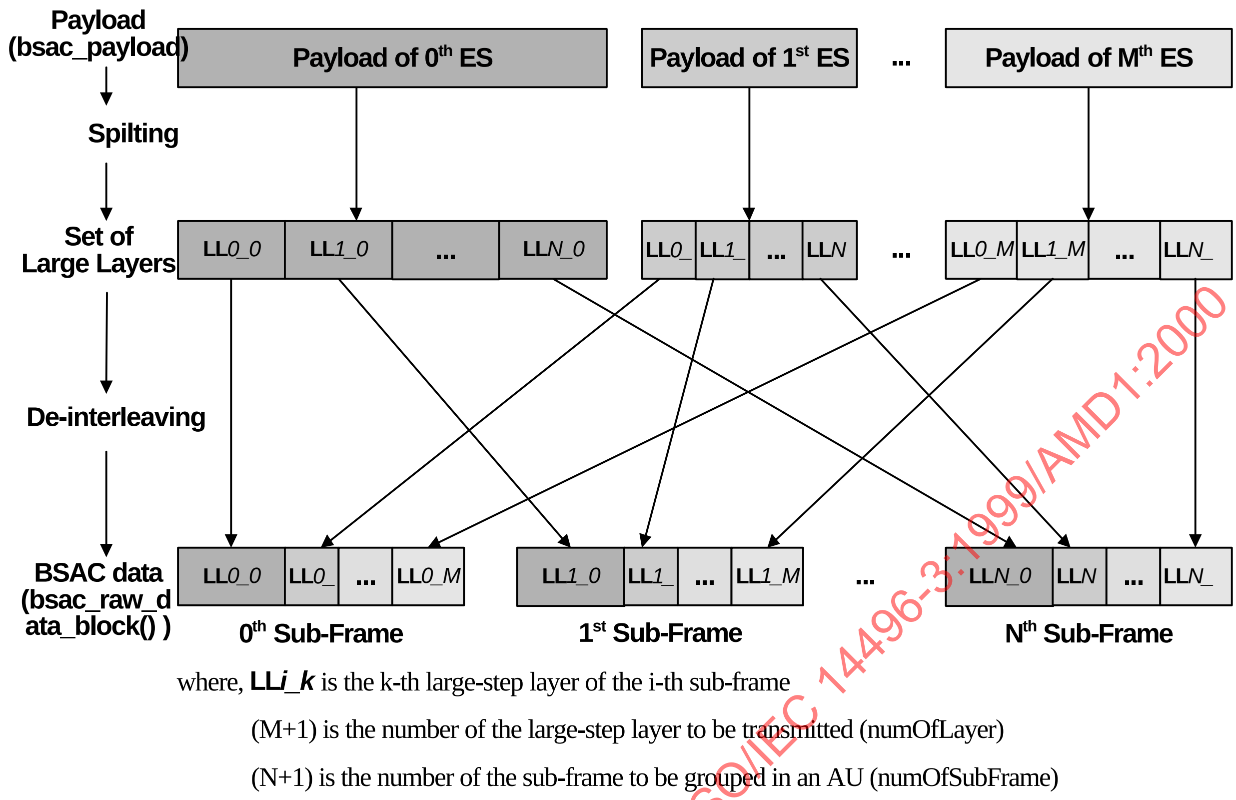


Figure 7: Reconstruction of BSAC data

The large-step layers are split from a payload of k^{th} layer ES that is organized as shown in Figure 8.

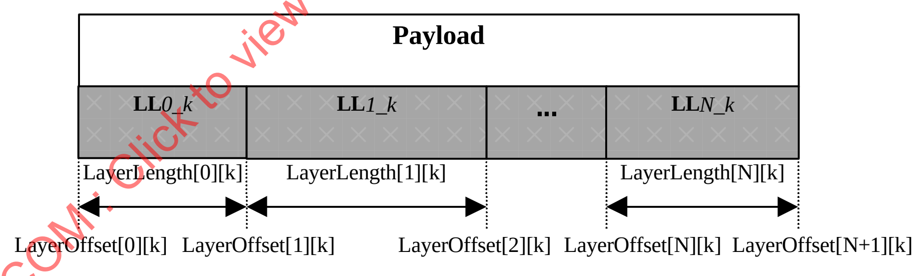


Figure 8: Structure of the payload

The split large-step layers are deinterleaved and concatenated to map the entire fine grain BSAC data. And then, decode the concatenated bitstreams using the syntax (bsac_raw_data_block()) for fine grain scalability to make the reconstructed signal.

Some help variables and arrays are needed to describe the re-arranging process of the payload transmitted over ES. These help variables depend on layer, numOfLayer, **numOfSubFrame**, **layer_length** and **frame_length** and must be built up for mapping bsac_raw_data_block() of each sub-frame from the payloads. The pseudo code shown below describes

- how to calculate *LayerLength[i][k]*, the length of the large-step layer which is located on the fine granule audio data, bsac_raw_data_block() of i^{th} sub-frame.

- how to caluculate *LayerOffset[i][k]* which indicates the start position of the large-step layer of i^{th} frame which is located on the payload of the k^{th} ES (bsac_payload())

- how to calculate *LayerStartByte[i][k]* which indicates the start position of the large-step layer which is located on the fine granule audio data, bsac_raw_data_block() of i^{th} sub-frame

```
for (k = 0; k < numOfLayer; k ++) {
    LayerStartByte[0][k] = 0;
    for (i = 0; i < numOfSubFrame; i++) {
       if (k == (numOfLayer-1) ) {
          LayerEndByte[i][k] = frame_length[i];
       } else {
          LayerEndByte[i][k]=LayerStartByte[i][k] + layer_length[k];
          if (frame_length[i] < LayerEndByte[i][k])
             LayerEndByte [i][k] = frame_length[i];
       }
       LayerStartByte[i+1][k] = LayerEndByte[i][k];
       LayerLength[i][k] = LayerEndByte[i][k] - LayerStartByte[i][k];
    }
}
for (k = 0; k < numOfLayer; k ++) {
    LayerOffset[0][k] = 0;
    for (i = 0; i < numOfSubFrame; i++) {
       LayerOffset[i+1][k] = LayerOffset[i][k] + LayerLength[i][k];
    }
}
```

Where, *frame_length[i]* is the length of i^{th} frame's bitstream which is obtained from the syntax element **frame_length** and layer_length[i] is the average length of the large-step layers in the payload of i^{th} layer ES and is obtained from Audio DecoderSpecificInfo.

8.3.3.2 Decoding of a bsac_raw_data_block()

8.3.3.2.1 Definitions

Bitstream elements:

bsac_raw_data_block()	block of raw data that contains coded audio data, related information and other data. A bsac_raw_data_block() basically consists of bsac_base_element() and several bsac_layer_element().
bsac_base_element()	Syntactic element of the base layer bitstream containing coded audio data, related information and other data.
frame_length	the length of the frame including headers in bytes.
bsac_header()	contains general information used for BSAC.
header_length	the length of the headers including frame_length, bsac_header() and general_header() in bytes. The actual length is (header_length+7) bytes. However if header_length is 0, it represents that the actual length is smaller than or equal to 7 bytes. And if header_length is 15, it represents that the actual length is larger than or equal to (15+7) bytes and should be calculated through the decoding of the headers .
sba_mode	indicates that the segmented binary arithmetic coding (SBA) scheme is used if this element is 1. Otherwise the general binary arithmetic coding scheme is used.
top_layer	top scalability layer index

base_snf_thr	significance threshold used for coding the bit-sliced data of the base layer.
base_band	indicates the maximum spectral line of the base layer. If the window_sequence is SHORT_WINDOW, 4*base_band is the maximum spectral line. Otherwise, 32*base_band is the maximum spectral line.
max_scalefactor[ch]	the maximum value of the scalefactors
cband_si_type[ch]	the type of the coding band side-information(si). Using this element, the largest value of cband_si's and the arithmetic model for decoding cband_si can be set as shown in Table 94.
base_scf_model[ch]	the arithmetic model for decoding the scalefactors in the base layer.
enh_scf_model[ch]	the arithmetic model used for decoding the scalefactors in the other enhancement layers.
max_sfb_si_len[ch]	maximum length which can be used per channel for coding the scalefactor-band side information including scalefactor and stereo-related information within a scalefactor band. This value has a offset(5). The actual maximum length is (max_sfb_si_len+5). This value is used for determining the bitstream size of each layer.
general_header()	contains header data for the General Audio Coding
reserved_bit	bit reserved for future use
window_sequence	indicates the sequence of windows. See ISO/IEC 14496-3 General Audio Coding.
window_shape	A 1 bit field that determines what window is used for the trailing part of this analysis window
max_sfb	number of scalefactor bands transmitted per group
scale_factor_grouping	A bit field that contains information about grouping of short spectral data
pns_data_present	the flag indicating whether the perceptual noise substitution(pns) will be used (1) or not (0).
pns_start_sfb	the scalefactor band from which the pcns tool is started.
ms_mask_present	this two bit field (see) indicates that the stereo mask is 00 Independent 01 1 bit mask of ms_used is located in the layer sfb side information part (layer_sfb_si()). 10 All ms_used are ones 11 2 bit mask of stereo_info is located in the layer sfb side information part layer_sfb_si()).
layer_cband_si()	contains the coding band side information necessary for Arithmetic encoding/decoding of the bit-sliced data within a coding band.
layer_sfb_si()	contains the side information of a scalefactor band such as the stereo-, the pns and the scalefactor information.bsac_layer_element() Syntactic element of the enhancement layer bitstream containing coded audio data for a time period of 1024(960) samples, related information and other data.
bsac_layer_spectra()	contains the arithmetic coded audio data of the quantized spectral coefficients which are newly added to each layer. See subclause 8.3.3.2.5 for the new spectral coefficients.

bsac_lower_spectra()	contains the arithmetic coded audio data of the quantized spectral coeffients which are lower than the spectra added to each layer.
bsac_higher_spectra()	contains the arithmetic coded audio data of the quantized spectral coeffients which are higher than the spectra added to each layer.
bsac_spectral_data()	contains the arithmetic coded audio data of the quantized spectral coeffients.

Help elements:

data_available()	function that returns "1" as long as bitstream is available, otherwise "0"
nch	a bitstream element that identifies the number of the channel.
scalefactor window band	term for scalefactor bands within a window. See ISO/IEC 14496-3 General Audio Coding.
scalefactor band	term for scalefactor band within a group. In case of EIGHT_SHORT_SEQUENCE and grouping a scalefactor band may contain several scalefactor window bands of corresponding frequency. For all other window_sequences scalefactor bands and scalefactor window bands are identical.
g	group index
win	window index within group
sfb	scalefactor band index within group
swb	scalefactor window band index within window
num_window_groups	number of groups of windows which share one set of scalefactors. See subclause 8.3.3.2.4
window_group_length[g]	number of windows in each group. See subclause 8.3.3.2.4
bit_set(bit_field,bit_num)	function that returns the value of bit number bit_num of a bit_field (most right bit is bit 0)
num_windows	number of windows of the actual window sequence. See subclause 8.3.3.2.4
num_swb_long_window	number of scalefactor bands for long windows. This number has to be selected depending on the sampling frequency. See ISO/IEC 14496-3 General Audio Coding.
num_swb_short_window	number of scalefactor window bands for short windows. This number has to be selected depending on the sampling frequency. See ISO/IEC 14496-3 General Audio Coding.
num_swb	number of scalefactor window bands for shortwindows in case of EIGHT_SHORT_SEQUENCE, number of scalefactor window bands for long windows otherwise. See subclause 8.3.3.2.4
swb_offset_long_window[swb]	table containing the index of the lowest spectral coefficient of scalefactor band sfb for long windows. This table has to be selected depending on the sampling frequency. See ISO/IEC 14496-3 General Audio Coding.
swb_offset_short_window[swb]	table containing the index of the lowest spectral coefficient of scalefactor band sfb for short windows. This table has to be selected depending on the sampling frequency. See ISO/IEC 14496-3 General Audio Coding.

swb_offset[g][swb] table containing the index of the lowest spectral coefficient of scalefactor band sfb for short windows in case of EIGHT_SHORT_SEQUENCE, otherwise for long windows. See subclause 8.3.3.2.4

layer_group[layer] indicates the group index of the spectral data to be added newly in the scalability layer

layer_start_sfb[layer] indicates the index of the lowest scalefactor band index to be added newly in the scalability layer

layer_end_sfb[layer] indicates the highest scalefactor band index to be added newly in the scalability layer

layer_start_cband[layer] indicates the lowest coding band index to be added newly in the scalability layer

layer_end_cband[layer] indicates the highest coding band index to be added newly in the scalability layer

layer_start_index[layer] indicates the index of the lowest spectral component to be added newly in the scalability layer

layer_end_index[layer] indicates the index of the highest spectral component to be added newly in the scalability layer

start_index[g] indicates the index of the lowest spectral component to be coded in the group *g*

end_index[g] indicates the index of the highest spectral component to be coded in the group *g*

layer_data_available() function that returns "1" as long as each layer's bitstream is available, otherwise "0". In other words, this function indicates whether the remaing bitstream of each layer is available or not.

terminal_layer[layer] indicates whether a layer is the terminal layer of a segment which is made up of one or more scalability layers. If the segmented binary arithmetic coding is not activated, all these values are always set to 0 except that of the top layer. Otherwise, these values are defined as described in subclause 8.3.4.5.3.

8.3.3.2.2 Decoding process

8.3.3.2.2.1 zero-stuffing

In order to do the arithmetic decoding perfectly, 32-bit zero value should be concatenated to the bitstream. In case of the SBA mode, the bitstream of a frame is split into the several segments. So, zero value should be concatenated to all segments. For the detailed description, see subclause 8.3.4.5.3. However, in case of the non-SBA mode, one zero stuffing is good enough since the bitstream of a frame is not split.

8.3.3.2.2.2 bsac_raw_data_block

A total BSAC stream, bsac_raw_data_block has the layered structure. First, bsac_base_element is parsed and decoded which is the bitstream for base scalability layer. Then, bsac_layer_element for the next enhancement layer is parsed and decoded. bsac_layer_element decoding routine is repeated while the decoded bitstream data is available and layer is smaller than or equal to the top layer, **top_layer.**

8.3.3.2.2.3 bsac_base_element

A bsac_base_element is made up of **frame_length**, bsac_header, general_header and bsac_layer_element().

First, **frame_length** is parsed from syntax. It represents the length of the frame including headers in bytes.

The syntax elements for the base layer are parsed which are composed of a bsac_header(), a general_header(), a layer_cband_si(), layer_sfb_si() and bsac_layer_element. bsac_base_element has several bsac_layer_element because the base layer is split into the several sub-layers for the error resilience of the base layer. The number of the sub-layers, *slayer_size* is calculated using the group index and the coding band as shown in subclause 8.3.3.2.5.

8.3.3.2.2.4 Recovering a bsac_header

BSAC provides a 1-kits/sec/ch fine grain scalability which has the layered structure, one base layer and several enhancement layers. Base layer contains the general side information for all the layers, the specific side information for the base layer and the audio data. The general side information is transmitted in the syntax of bsac_header() and general_header().

bsac_header consists of **top_layer**, **header_length**, **sba_mode, base_band**, **max_scalefactor**, **cband_si_type**, **base_scf_model** and **enh_scf_model**. All the bitstream elements are included in the form of the unsigned integer.

First, 4 bit **header_length** is parsed which represents the length of the headers including frame_length, bsac_header and general_header in bytes. The length of the headers is (header_length+7)*8. Next, 1bit **sba_mode** is parsed which resprents whether the segmented binary arithmetic coding(SBA) is used or the binary arithmetic coding is used.

Next, 6 bit **top_layer** is parsed which represents the top scalability layer index to be encoded. Next, 2 bit **base_snf_thr** is parsed which represents the significance threshold used for coding the bit-sliced data of the base layer.

Next, 8 bit **max_scalefactor** is parsed which represents the maximum value of the scalefactors. If the number of the channel is not 1, this value is parsed one more.

Next, 5-bit **base_band** is parsed which represents minimum spectral line which is coded in the base layer. If the window sequence is SHORT_WINDOW, 4*base_band indicates the minimum spectral line. Otherwise 32*base_band indicates the minimum spectral line.

And, 5 bit **cband_si_type** is parsed which represents the arithmetic model of cband_si and the largest cband_si which can be decoded as shown in Table 94. 3 bit **base_scf_model** and **enh_scf_model** are parsed which represent the arithmetic model table for the scalefactors of the base layer and the other enhancement layers, respectively. Next, 4 bit **max_sfb_si_len** is pared which represents the maximum length of the scalefactor band side information to be able to used in each layer. The maximum length is (max_sfb_si_len+5).

8.3.3.2.2.5 Recovering a general_header

The order for decoding the syntax of a bsac_header is:

- get reserved_bit
- get window_sequence
- get window_shape
- get max_sfb
- get scale_factor_grouping if the window_sequence is EIGHT_SHORT_SEQUENCE
- get pns_present
- get pns_start_sfb if present
- get ms_mask_present flag if the number of the channel is 2
- get tns_data_present

- get TNS data if present
- get ltp_data_present
- get ltp data if present

If the number of the channel is not 1, the decoding of another channel is done as follows:

- get tns_data_present
- get TNS data if present
- get ltp_data_present
- get ltp data if present

The process of recovering tns_data and ltp_data is described in ISO/IEC 14496-3 General Audio Coding.

8.3.3.2.2.6 bsac_layer_element

A bsac_layer_element is an enhancement layer bitstream and composed of layer_cband_si(), layer_sfb_si(), bsac_layer_spectra(), bsac_lower_spectra() and bsac_higher_spectra(). Decoding process of bsac_layer_element is as follows:

Decode layer_cband_si

Decode layer_sfb_si

Decode bsac_layer_spectra

Decode bsac_lower_spectra

Decode bsac_higher_spectra

8.3.3.2.2.7 Decoding of coding band side information (layer_cband_si)

The spectral coefficients are divided into coding bands which contain 32 quantized spectral coefficients for the noiseless coding. Coding bands(abbreviation ‘cband’) are the basic units used for the noiseless coding.

cband_si represents the MSB plane and the probability table of the sliced bits within a coding band as shown in Table 96. Using this cband_si, the bit-sliced data of each coding band are arithmetic-coded.

cband_si is arithmetic_coded with the model which is given in the syntax element **cband_si_type** as shown in Table 94.

An overview of how to decode cband_si will be given in subclause 8.3.4.4.

8.3.3.2.2.8 Decoding of the scalefactor band side information (layer_sfb_si)

An overview of how to decode layer_sfb_si will be given here. layer_sfb_si is made up of as follows:

Decoding of stereo_info, ms_used or noise_flag.

Decoding of scalefactors

8.3.3.2.2.9 Decoding of stereo_info, noise_flag or ms_used

Decoding process of stereo_info, noise_flag or ms_used is depended on pns_data_present, number of channel, ms_mask_present.

If pns data is not present, decoding process is as follows:

If ms_mask_present is 0, arithmetic decoding of stereo_info or ms_used is not needed.

If ms_mask_present is 2, all ms_used values are ones in this case. So, M/S stereo processing of AAC is done at all scalefactor band.

If ms_mask_present is 1, 1 bit mask of max_sfb bands of ms_used is conveyed in this case. So, ms_used is arithmetic decoded. M/S stereo processing of AAC is done according to the decoded ms_used.

If ms_mask_present is 3, stereo_info is arithmetic decoded. stereo_info is two-bit flag per scalefactor band indicating the M/S coding or Intensity coding mode. If stereo_info is not 0, M/S stereo or intensity stereo of AAC is done with these decoded data.

If pns data is present and the number of channel is 1, decoding process is as follows:

If the number of channel is 1 and pns data is present, noise flag of the scalefactor bands between **pns_start_sfb** to **max_sfb** is arithmetic decoded. Perceptual noise substitution is done according to the decoded noise flag.

If pns data is present and the number of channel is 2, decoding process is as follows:

If ms_mask_present is 0, noise flag for pns is arithmetic decoded. Perceptual noise substitution of independent mode is done according to the decoded noise flag.

If ms_mask_present is 2, all ms_used values are ones in this case. So, M/S stereo processing of AAC is done at all scalefactor band. However, there is no pns processing regardless of pns_data_present flag

If ms_mask_present is 1, 1 bit mask of max_sfb bands of ms_used is conveyed in this case. So, ms_used is arithmetic decoded. M/S stereo processing of AAC is done according to the decoded ms_used. However, there is no pns processing regardless of pns_data_present flag

If ms_mask_present is 3, stereo_info is arithmetic decoded. If stereo_info is 1 or 2, M/S stereo or intensity stereo processing of AAC is done with these decoded data and there is no pns processing. If stereo_info is 3 and scalefactor band is smaller than pns_start_sfb, out_of_phase intensity stereo processing is done. If stereo_info is 3 and scalefactor band is larger than or equal to pns_start_sfb, noise flag for pns is arithmetic decoded. And then if the both noise flags of two channel are 1, noise substitution mode is arithmetic decoded. The perceptual noise is substituted or out_of_phase intensity stereo processing is done according to the substitution mode. Otherwise, the perceptual noise is substituted only if noise flag is 1.

The detailed description of how to decode this side information will be given in subclause 8.3.4.2.

8.3.3.2.2.10 Decoding of scalefactors

The spectral coefficients are divided into scalefactor bands that contain a multiple of 4 quantized spectral coefficients. Each scalefactor band has a scalefactor. For all scalefactors the difference to the maximum scalefactor value, **max_scalefactor** is arithmetic-coded using the arithmetic model given in Table 95. The arithmetic model necessary for coding the differential scalefactors in the base layer is given as a 3-bit unsigned integer in the bitstream element, **base_scf_model**. The arithmetic model necessary for coding the differential scalefactors in the other enhancement layers is given as a 3-bit unsigned integer in the bitstream element, **enh_scf_model**. The maximum scalefactor value is given explicitly as a 8 bit PCM in the bitstream element **max_scalefactor**.

The detailed description of how to decode this side information will be given in subclause 8.3.4.3.

8.3.3.2.2.11 Bit-Sliced spectral data

In BSAC encoder, the absolute values of quantized spectral coefficients is mapped into a bit-sliced sequence. These slicded bits are the symbols of the arithmetic coding. Every siliced bits are binary arithmetic coded with the proper probability (arithmetic model) from the lowest-frequency coefficient to the highest-frequency coefficient of the scalability layer, starting the Most Significant Bit(MSB) plane and progressing to the Least Significant Bit(LSB) plane. The arithmetic coding of the sign bits associated with non-zero coefficient follows that of the sliced bit when the sliced bit is 1 for the first time.

The probability value should be defined in order to arithmetic-code the symbols (the sliced bits). Binary probability table is made up of probability values of the symbol “0”. First of all, probability table is selected using cband_si as shown Table 96. The probability value is selected among the several values in the selected table according to the context such as the remaining available bit size and the sliced bits of successive non-overlapping 4 spectral data.

For the case of multiple windows per block, the concatenated and possibly grouped and interleaved set of spectral coefficients is treated as a single set of coefficients that progress from low to high as described in subclause 8.3.3.2.6. This set of spectral coefficients needs to be de-interleaved after they are decoded. The set of bit-sliced sequence is divided into coding bands. The probability table index used for encoding the bit-sliced data within each coding band is included in the bitsteam element **cband_si** and transmitted starting from the lowest frequency coding band and progressing to the highest frequency coding band. The spectral information for all scalefactor bands equal to or greater than **max_sfb** is set to zero.

8.3.3.2.2.12 Decoding the sliced bits of the spectral data

The spectral bandwidth is increased in proportion to the scalability layer. So, the new spectral data is added to each layer. First of all, these new spectral data are coded in each layer (bsac_layer_spectra()). The coding process is continued until the data of each layer is not available or all the sliced bits of the new spectra are coded. The length of the available bitstream (*available_len[]*) is initialized at the beginning of each layer as described in subclause 8.3.3.2.5. The estimated length of the codeword (*est_cw_len*) to be decoded is calculated from the arithmetic decoding process as described in subclause 8.3.3.2.7. After the arithmetic decoding of a symbol, the length of the available bitstream should be updated by subtracting the estimated codeword length from it. We can detect whether the remaing bitstream of each layer is available or not by checking the array *available_len[]*.

From the lowest layer to the top layer, the new spectra are arithmetic-coded layer-by-layer in the above first process (bsac_layer_spectra()). Some sliced bits cannot be coded for lack of the codeword allocated to the layer. After the first coding process is finished, the current significances (cur_snf) are saved for the secondary coding processes (bsac_lower_spectra() and bsac_higher_spectra()). The sliced bits which remain uncoded is coded using the saved significances(unc_snf) in the secondary coding process.

If there remains the available codewords after the first coding, the next symbol to be decoded with these redundant codewords depends on whether the segmented binary arithmetic coding(SBA) mode is active.

In case of the non-SBA mode, the uncoded symbols of the lower spectra in the layers than the current layer are coded in the secondary coding process (bsac_lower_spectra()).

In case of the SBA mode for the error resilience, the next symbol is dependant upon whether the layer is the terminal layer of a segment or not. If the layer is not a terminal of the segment, the spectral data of the next layer (*bsac_layer_spectra(layer+1)*) should be decoded. That is to say, the redundant length of the layer is added to the available bitstream length (*available_len[layer+1]*) of the next layer in the first coding process.

If the layer is a terminal of the segment, the uncoded symbols of the lower spectra in the layers than the current layer are coded in the secondary coding process (bsac_lower_spectra()). The uncoded symbol of the spectra in the layers higher than the current layer are coded in the secondary coding process (bsac_higher_spectra()) if the codeword of the layer is available in spite of having coded the lower spectra. And the remaining symbols are continuously coded in the layers whose codeword is available, starting from the lowest layer and progressing to the top layer.

If there are the redundant bits after the secondary coding, the size of the redundant bits is added to the available bitstream length(*available_len[layer+1]*) of the next layer and the redundant bits are used in the first coding of the next layer.

8.3.3.2.2.13 Reconstruction of the decoded sample from bit-sliced data

In order to reconstruct the spectral data, a bit-sliced sequence that has been decoded should be mapped into quantized spectral values. An arithmetic decoded symbol is a sliced bit. A decoded symbol is translated to the bit values of quantized spectral coeffiecients, as specified in the following pseudo C code:

```
snf = the significance of the symbol (the sliced bit) to be decoded;
sliced_bit[ch][g][i][snf] = the decoded symbol (the sliced bits of the quantized spectrum);
sample[ch][g][i] = buffer for quantized spectral coeffients to be recontructed;
scaled_bit = sliced_bit[ch][g][i][snf] << (snf-1);
if (sample[ch][g][i] < 0)
    sample[ch][g][i] -= scaled_bit;
else
    sample[ch][g][i] += scaled_bit;
```

And if the sign bit of the decoded sample is 1, the decoded sample sample[ch][g][i] has the negative value as follows :

```
if (sample[ch][g][i] != 0) {
    if (sign_bit[ch][g][i] == 1)   sample[ch][g][i] = -sample[ch][g][i];
}
```

8.3.3.2.3 Windows and window sequences for BSAC

Quantization and coding is done in the frequency domain. For this purpose, the time signal is mapped into the frequency domain in the encoder. Depending on the signal, the coder may change the time/frequency resolution by using two different windows: LONG_WINDOW and SHORT_WINDOW. To switch between windows, the transition windows LONG_START_WINDOW and LONG_STOP_WINDOW are used. Refer to ISO/IEC 14496-3 General Audio Coding for more detailed information about the transform and the windows since BSAC has the same transform and windows with AAC.

8.3.3.2.4 Scalefactor bands, grouping and coding bands for BSAC

Many tools of the AAC/BSAC decoder perform operations on groups of consecutive spectral values called scalefactor bands (abbreviation "sfb"). The width of the scalefactor bands is built in imitation of the critical bands of the human auditory system. For that reason the number of scalefactor bands in a spectrum and their width depend on the transform length and the sampling frequency. Refer to ISO/IEC 14496-3 General Audio Coding for more detailed information about the scalefactor bands and grouping because BSAC has the same process with AAC.

BSAC decoding tool performs operations on groups of consecutive spectral values called coding bands (abbreviation "cband"). To increase the efficiency of the noiseless coding, the width of the coding bands is fixed as 32 irrespective of the transform length and the sampling frequency. In case of sequences which contain LONG_WINDOW, 32 spectral data are simply grouped into a coding band. Since the spectral data within a group are interleaved in an ascending spectral order in case of SHORT_WINDOW, the interleaved spectral data are grouped into a coding band. Each spectral index within a group is mapped into a coding band with a mapping function, *cband* = spectral_index/32.

Since scalefactor bands and coding bands are a basic element of the BSAC coding algorithm, some help variables and arrays are needed to describe the decoding process in all tools using scalefactor bands and coding bands. These help variables must be defined for BSAC decoding. These help variables depend on sampling_frequency, **window_sequence, scalefactor_grouping** and **max_sfb** and must be built up for each bsac_raw_data_block. The pseudo code shown below describes

- how to determine the number of windows in a window_sequence *num_windows*

- how to determine the number of window_groups *num_window_groups*
- how to determine the number of windows in each group *window_group_length[g]*
- how to determine the total number of scalefactor window bands *num_swb* for the actual window type
- how to determine *swb_offset[g][swb]*, the offset of the first coefficient in scalefactor window band *swb* of the window actually used

A long transform window is always described as a window_group containing a single window. Since the number of scalefactor bands and their width depend on the sampling frequency, the affected variables are indexed with sampling_frequency_index to select the appropriate table.

```
fs_index = sampling_frequency_index;
switch( window_sequence ) {
    case ONLY_LONG_SEQUENCE:
    case LONG_START_SEQUENCE:
    case LONG_STOP_SEQUENCE:
        num_windows = 1;
        num_window_groups = 1;
        window_group_length[num_window_groups-1] = 1;
        num_swb = num_swb_long_window[fs_index];
        for( sfb=0; sfb< max_sfb+1; sfb++ ) {
                swb_offset[0][sfb] = swb_offset_long_window[fs_index][sfb];
        }
        break;
    case EIGHT_SHORT_SEQUENCE:
        num_windows = 8;
        num_window_groups = 1;
        window_group_length[num_window_groups-1] = 1;
        num_swb = num_swb_short_window[fs_index];
        for( i=0; i< num_windows-1; i++) {
            if( bit_set(scale_factor_grouping,6-i)) == 0 ) {
                num_window_groups += 1;
                window_group_length[num_window_groups-1] = 1;
            }
            else {
                window_group_length[num_window_groups-1] += 1;
            }
        }

        for(g = 0; g < num_window_groups; g++)
            swb_offset[g][0] = 0;

        for(sfb = 0; sfb < max_sfb; sfb++) {
            for(g = 0; g < num_window_groups; g++) {
                swb_offset[g][sfb] = swb_offset_short_window[fs_index][sfb];
                swb_offset[g][sfb] = swb_offset[g][sfb] * window_group_length[g];
            }
        }
        break;
    default:
        break;
}
```

8.3.3.2.5 BSAC fine grain scalability layer

BSAC provides a 1-kits/sec/ch fine grain scalability which has the layered bitstream, one BSAC base layer and various enhancement layers. BSAC base layer is made up of the general side information for all the fine grain layers, the specific side information for only the base layer and the audio data. BSAC enhancement layers contain the layer side information and the audio data.

BSAC scalable coding scheme has the scalable band-limit according to the fine grain layer. First of all, the base band-limit is set. The base band-limit depends on the signal to be encoded and is in the syntax element, **base_band**. The actually limited spectral line is 4*base_band if the window sequence is SHORT_WINDOW. Otherwise, the limited spectral line is 32*base_band. In order to provide the fine grain scalability, BSAC extends the band-limit according to the fine grain layer. The band limit of each layer depends on the base band-limit, the transform lengths 1024(960) and 128(120) and the sampling frequencies. The spectral band is extended more and more as the number of the enhancement layer is increased. So, the new spectral components are added to each layer.

Some help variables and arrays are needed to describe the bit-sliced decoding process of the side information and spectral data in each BSAC fine grain layer. These help variables depend on sampling_frequency, layer, **nch, frame_length, top_layer, window_sequence** and **max_sfb** and must be built up for each bsac_layer_element. The pseudo code shown below describes

- how to determine *slayer_size*, the number of the sub-layers which the base layer is split into.

```
slayer_size = 0;
for ( g = 0; g < num_window_groups; g++) {
    if (window_sequence == EIGHT_SHORT_SEQUENCE) {
        end_index[g] = base_band * 4 * window_group_length[g];
        if (fs==44100 || fs==48000) {
            if (end_index[g]%32>=16)
                end_index[g] = (int)(end_index[g]/32)*32 + 20;
            else if ( end_index[g]%32 >= 4)
                end_index[g] = (int)(end_index[g]/32)*32 + 8;
        }
        else if (fs==22050 || fs==24000 || fs==32000) end_index[g] = (int)(end_index[g]/16)*16;
        else if (fs==11025 || fs==12000 || fs==16000) end_index[g] = (int)(end_index[g]/32)*32;
        else end_index[g] = (int)(end_index[g]/64)*64;
        end_cband[g] = (end_index[g] + 31) / 32;
    }
    else
        end_cband[g] = base_band;
    slayer_size += end_cband[g];
}
```

- how to determine *layer_group[]*, the group index of the spectral components to be added newly in the scalability layer

```
layer = 0
for ( g = 0; g < num_window_groups; g++)
    for ( w = 0; w < window_group_length[g]; w++)
        layer_group[layer++] = g;
for (layer = slayer_size+8; layer < (top_layer+slayer_size); layer++)
    layer_group[layer] = layer_group[layer-8];
```

- how to determine *layer_end_index[]*, the end offset of the spectral components to be added newly in each scalability layer
- how to determine *layer_end_cband[]*, the end coding band to be added newly in each scalability layer
- how to determine *layer_start_index[]*, the start offset of the spectral components to be added newly in each scalability layer
- how to determine *layer_start_cband[]*, the start coding band to be added newly in each scalability layer

```
layer = 0;
for ( g = 0; g < num_window_groups; g++) {
        for (cband = 0; cband < end_cband[g]; cband++) {
            layer_start_cband[layer] = cband;
            end_cband[g] = layer_end_cband[layer] = cband+1;
```

```
            layer_start_index[layer] = cband * 32;
            end_index[g] = layer_end_index[layer++] = (cband+1) * 32;
        }
        if (window_sequence == EIGHT_SHORT_SEQUENCE)
            last_index[g] = swb_offset_short_window[max_sfb] * window_group_length[g];
        else
            last_index[g] = swb_offset_long_window[max_sfb];
}

for (layer = slayer_size; layer < (top_layer+slayer_size); layer++) {
    g = layer_group[layer];
    layer_start_index[layer] = end_index[g];
    if (fs==44100 || fs==48000) {
        if (end_index[g]%32==0)
            end_index[g] += 8;
        else
            end_index[g] += 12;
    }
    else if (fs==22050 || fs==24000 || fs==32000)
        end_index[g] += 16;
    else if (fs==11025 || fs==12000 || fs==16000)
        end_index[g] += 32;
    else
        end_index[g] += 64;
    if ( end_index[g] > last_index[g] )
        end_index[g] = last_index[g];
    layer_end_index[layer] = end_index[g];
    layer_start_cband[g] = end_cband[g];
    end_cband[g] = layer_end_cband[layer] = (end_index[g] + 31) / 32;
}
```

where, fs is the sampling frequency.

- how to determine *layer_end_sfb[]*, the end scalefactor band to be added newly in each scalability layer
- how to determine *layer_start_sfb[]*, the start scalefactor band to be added newly in each scalability layer

```
for (g = 0; g < num_window_groups; g++)
    end_sfb[g] = 0;
    for (layer = 0; layer < (top_layer+slayer_size); layer++) {
        g = layer_group[layer];
        layer_start_sfb[layer] = end_sfb[g];
        layer_end_sfb[layer] = max_sfb;
        for (sfb = 0; sfb < max_sfb; sfb++) {
            if ( layer_end_index[layer] <= swb_offset_short_window[sfb] *
window_group_length[g]) {
                layer_end_sfb[layer] = sfb + 1;
                break;
            }
        }
    end_sfb[g] = layer_end_sfb[layer];
}
```

- how to determine *available_len[i]*, the available maximum size of the bitstream of the *i*-th layer. If the arithmetic coding was initialized at the beginning of the layer, 1 should subtracted from available_len[i] since the additional 1 bit is required at the arithmetic coding termination. The maximum length of the 0^{th} coding band side information(*max_cband0_si_len*) is defined as 11.

```
for (layer = 0; layer <(top_layer+slayer_size); layer++) {
    layer_si_maxlen[layer] = 0;
    for (cband = layer_start_cband[layer]; cband < layer_end_cband[layer]; cband++) {
        for (ch=0; ch <nch; ch++) {
            if (cband == 0)
```

```
                layer_si_maxlen[layer] += max_cband0_si_len;
            else
                layer_si_maxlen[layer] +=  max_cband_si_len[cband_si_type[ch]];
        }
    }
    for (sfb = layer_start_sfb[layer]; sfb < layer_end_sfb[layer]; sfb++)
        for (ch = 0; ch < nch; ch++)
            layer_si_maxlen[layer] +=  max_sfb_si_len[ch] + 5;
}

for (layer = slayer_size; layer <= (top_layer + slayer_size); layer++) {
layer_bitrate = nch * ( (layer-slayer_size) * 1000 + 16000);
    layer_bit_offset[layer] = layer_bitrate * BLOCK_SIZE_SAMPLES_IN_FRAME;
    layer_bit_offset[layer] = (int)(layer_bit_offset[layer] / SAMPLING_FREQUENCY / 8 ) * 8;
    if (layer_bit_offset[layer] > frame_length*8)
        layer_bit_offset[layer] = frame_length*8;
}

for (layer = (top_layer + slayer_size –1);  layer >= slayer_size; layer--) {
    bit_offset = layer_bit_offset[layer+1] – layer_si_maxlen[layer]
    if ( bit_offset < layer_bit_offset[layer] )
        layer_bit_offset[layer] = bit_offset
}

for (layer = slayer_size – 1; slayer_size >= 0; slayer--)
    layer_bit_offset[layer] = layer_bit_offset[layer+1] – layer_si_maxlen[layer];


overflow_size = (header_length + 7) * 8 – layer_bit_offset[0];
layer_bit_offset[0] = (header_length + 7) * 8;
if (overflow_size > 0) {
    for ( layer = (top_layer+slayer_size-1); layer >= slayer_size; layer--) {
        layer_bit_size = layer_bit_offset[layer+1] – layer_bit_offset[layer];
        layer_bit_size -= layer_si_maxlen[layer];
        if (layer_bit_size >= overflow_size) {
            layer_bit_size = overflow_size;
            overflow_size = 0;
        }
        else
            overflow_size = overflow_size – layer_bit_size;
        for (m=1; m<=layer; m++)
            layer_bit_offset[m] += layer_bit_size;
        if (overflow_size<=0)
            break;
    }
}
else {
    underflow_size = -overflow_size;
    for (m=1; m < slayer_size; m++)  {
        layer_bit_offset[m] = layer_bit_offset[m-1] + layer_si_maxlen[m-1];
        layer_bit_offset[m] += underflow_size / slayer_size;
        if (layer <= (underflow_size%slayer_size);
            layer_bit_offset[m] += 1;
    }
}
for (layer=0; layer <(top_layer+slayer_size); layer++)
    available_len[layer] = layer_bit_offset[layer+1] – layer_bit_offset[layer];
```

Some help variables and arrays are needed to describe the bit-sliced decoding process of the spectral values in each BSAC fine grain layer. cur_snf[ch][g][*i*] is initialized as the MSB plane (MSBplane[ch][g][cband]) allocated to the coding band *cband*, where we can get MSBplane[][] from **cband_si[ch][g][cband]** using Table 96. And, we start the decoding of the bit-sliced data in each layer from the maximum significance, *maxsnf*.

These help variables and arrays must be built up for each bsac_spectral_data(). The pseudo code shown below describes

- how to initialize *cur_snf[][][]*, the current significance of the spectra to be added newly due to the spectral band extension in each enhancement scalability layer.

```
/* set current snf */
g = layer_group[layer];
for(ch = 0; ch < nch; ch++) {
    for (i=layer_start_index[layer]; i<layer_end_index[layer]; i++) {
        cband = i/32;
        cur_snf[ch][g][i] = MSBplane[ch][g][cband];
    }
}
```

- how to determine *maxsnf*, the maximum significance of all vectors to be decoded.

```
maxsnf = 0;
for (g = start_g; g < end_g; g++)
for(ch = 0; ch < nch; ch++) {
    for(i = start_index[g]; i< end_index[g]; i++)
        if (maxsnf < cur_snf[ch][g][i])
            maxsnf = cur_snf[ch][g][i];
}
```

- how to store *cur_snf[][][]* for the secondary coding (bsac_lower_spectra() and bsac_higher_spectra()) after the sliced bits of the new spectra has been coded in each layer (bsac_layer_spectra()).

  /* store current snf */

```
for (g = 0; g <no_window_groups; g++)
for(ch = 0; ch < nch; ch++) {
    for (i=layer_start_index[layer]; i<layer_end_index[layer]; i++) {
        unc_snf[ch][g][i] = cur_snf[ch][g][i];
    }
}
```

8.3.3.2.6 Order of spectral coefficients in spectral_data

For ONLY_LONG_SEQUENCE windows (num_window_groups = 1, window_group_length[0] = 1) the spectral data is in ascending spectral order, as shown in Figure 9.

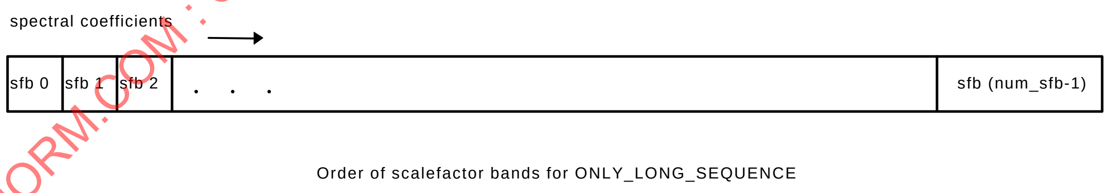


Figure 9: Order of scalefactor bands for ONLY_LONG_SEQUENCE

For the EIGHT_SHORT_SEQUENCE window, each 4 spectral coefficients of blocks within each group are interleaved in ascending spectral order and the interleaved spectral coefficients are interleaved in ascending group number, as shown in Figure 10.

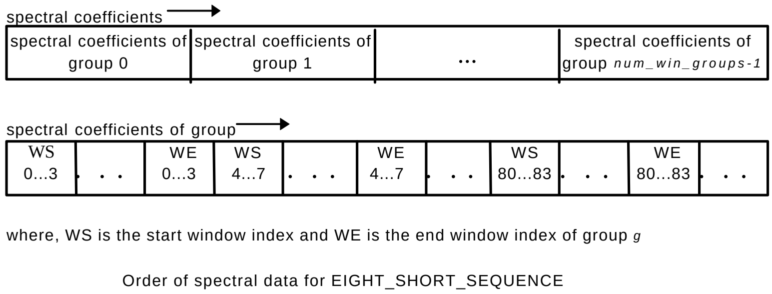


Figure 10: Order of spectral data for EIGHT_SHORT_SEQUENCE

8.3.3.2.7 Arithmetic coding procedure

Arithmetic Coding consists of the following 2 steps:

- Initialization which is performed prior to the coding of the first symbol
- Coding of the symbol themselves.

8.3.3.2.7.1 Registers, symbols and constants

Several registers, symbols and constants are defined to describe the arithmetic decoder.

- half[] : 32-bit fixed point array equal to ½
- range: 32-bit fixed point register. Contains the range of the interval.
- value: 32-bit fixed point register. Contains the value of the arithmetic code.
- est_cw_len: 16-bit fixed point register. Contains the estimated length of the arithmetic codeword to be decoded.
- p0: 16-bit fixed point register (Upper 6 MSBs are available, Other LSBs are 0). Probability of the “0” symbol.
- p1: 16-bit fixed point register(Upper 6 MSBs are available, Other LSBs are 0). Probability of the “1” symbol.
- cum_freq : 16-bit fixed point registers. Cummulative Probabilities of the symbols.

8.3.3.2.7.2 Initialization

The bitstreams of each segment are read in the buffer of each segment. And 32-bit zero is concatenated to the buffer of each segment. If the segmented arithmetic coding is not, all the bitstreams of a frame is a segment and the zero stuffing is used. See subclause 8.3.4.5.3 for the detailed description of the segment.

The register *value* is set to 0, *range* to 1 and *est_cw_len* to 30. Using these initialized regiters, the 30 bits are read in register *value* and registers are updated when the first symbol is decoded.

8.3.3.2.7.3 Decoding a symbol

Arithmetic decoding procedure varies on the symbol to be decoded. If the symbol is the sliced bit of the spectral data, the binary arithmetic decoding is used. Otherwise, the general arithmetic decoding is used.

When a symbol is binary arithmetic-decoded, the probability p0 of the “0” symbol is provided according to the context computed properly and using the probability table. p0 uses a 6-bit fixed-point number representation. Since the decoder is binary, the probability of the “1” symbol is defined to be 1 minus the probability of the “0” symbol, i.e. p1 = 1-p0.

When a symbol is arithmetic-decoded, the cummulative probability values of multiple symbols are provided. The probability values are regarded as the arithmetic model. The arithmetic model for decoding a symbol is given in the bitstream elements. For example, arithmetic models of scalefactor and cband_si are given in the bitstream elements, **base_scf_model**, **enh_scf_model** and **cband_si_type**. Each value of the arithmetic model uses a 14-bit fixed-point representation.

8.3.3.2.7.4 Software

```
unsigned long half[16] =
{
    0x20000000, 0x10000000, 0x08000000, 0x04000000,
    0x02000000, 0x01000000, 0x00800000, 0x00400000,
    0x00200000, 0x00100000, 0x00080000, 0x00040000,
    0x00020000, 0x00010000, 0x00008000, 0x00004000
};


/* Initialize the Parameteres of the Arithmetic Decoder */
void initArDecode()
{
   value = 0;
   range = 1;
   est_cw_len = 30;
}
/* GENEARL ARITHMETIC DECODE */
int decode_symbol (buf_idx, cum_freq, symbol)
int buf_idx;     /* buffer index to save the arithmetic code word */
int cum_freq[];  /* Cumulative symbol freqencies */
int *symbol;     /* Symbol decoded */
{
   if (est_cw_len) {
      range = (range<<est_cw_len);
      value = (value<<est_cw_len) | readBits(buf_idx, est_cw_len);
         /* read bitstream from the buffer */
   }

   range >>= 14;
   cum = value/range;          /* Find cum freq */

   /* Find symbol */
   for (sym=0; cum_freq[sym]>cum; sym++);
   *symbol = sym;

   /* Narrow the code region to that allotted to this symbol. */
   value -= (range * cum_freq[sym]);

   if (sym > 0) {
      range = range * (cum_freq[sym-1]-cum_freq[sym]);
   }
   else {
      range = range * (16384-cum_freq[sym]);
   }

   for (est_cw_len = 0; range < half[est_cw_len]; est_cw_len++) ;
   return est_cw_len;
}

/* BINARY ARITHMETIC-DECODE THE NEXT SYMBOL. */
int decode_symbol2 (buf_idx, freq0, symbol)
int buf_idx; /* buffer index to save the arithmetic code word */
int p0;      /* Normalized probability of symbol 0 */
int *symbol; /* Symbol decoded */
{
   if (est_cw_len) {
      range = (range<<est_cw_len);
      value = (value<<est_cw_len) | readBits(buf_idx, est_cw_len);
```

```
            /* read bitstream from the buffer */
    }

    range >>= 14;

    /* Find symbol */
    if ( (p0 * range) <= value ) {
        *symbol = 1;

        /* Narrow the code region to that allotted to this symbol. */
        value -= range * p0;
        p1 = 16384 – p0;
        range =  range * p1;
    }
    else {
        *symbol = 0;

        /* Narrow the code region to that allotted to this symbol. */
        range = range * p0;
    }

    for(est_cw_len=0; range<half[est_cw_len]; est_cw_len++);
    return est_cw_len;
}
```

8.3.4 Tool description

BSAC stands for bit sliced arithmetic coding and is the name of a noiseless coder and bitstream formatter that provides a fine grain scalability and error resilience in the MPEG-4 General Audio(GA) coder. The BSAC noiseless coding module is an alternative to the AAC coding module, with all other modules of the AAC-based coder remaining unchanged. The BSAC noiseless coding is used to make the bitstream scalable and error resilience and further reduce the redundancy of the scalefactors and the quantized spectrum. The BSAC noiseless decoding process is split into 4 subclauses. Subclause 8.3.4.1 to 8.3.4.5 describe the detailed decoding process of the spectral data, the stereo or pns related data, the scalefactors and the coding band side information.

8.3.4.1 Decoding of bit-sliced spectral data (bsac_spectral_data())

8.3.4.1.1 Description

BSAC uses the bit-slicing scheme of the quantized spectral coefficients in order to provide the fine grain scalability. And it encode the bit-sliced data using binary arithmetic coding scheme in order to reduce the average bits transmitted while suffering no loss of fidelity.

In BSAC scalable coding scheme, a quantized sequence is divided into coding bands, as shown in subclause 8.3.3.2.5. And, a quantized sequence is mapped into a bit-sliced sequence within a coding band. The noiseless coding of the sliced bits relies on the probability table of the coding band, the significance and the other contexts.

The significance of the bit-sliced data is the position of the sliced bit to be coded.

The flags, sign_is_coded[] are updated with coding the vectors from MSB to LSB. They are initialized to 0. And they are set to 1 when the sign of the quantized spectrum is coded.

The probability table for encoding the bit-sliced data within each coding band is included in the bistream element **cband_si_type** and transmitted starting from the lowest coding band and progressing to the highest coding band allocated to each layer. For the detailed description of the coding band side information **cband_si_type**, see subclause 8.3.4.4. Table 96 lists 23 probability tables which are used for encoding/decoding the bit-sliced data. The BSAC probability table consists of several sub-tables. sub-tables are classified and chosen according to the significance and the coded upper bits as shown Table 119 to Table 140. Every sliced bit is arithmetic encoded using the probability value chosen among several possible sub-tables of BSAC probability table.

8.3.4.1.2 Definitions

Bitstream elements:

acod_sliced_bit[ch][g][i]	Arithmetic codeword necessary for arithmetic decoding of the sliced bit. Using this decoded bit, we can reconstuct each bit value of the quantized spectral value. The actually reconstructed bit-value is dependent on the significance of the sliced bit.
acod_sign[ch][g][i]	Arithmetic codeword from binary arithmetic coding sign_bit. The probability of the "0" symbol is defined to 0.5 which uses 8192 as a 14-bit fixed-point number. sign_bit indicates sign bit for non-zero coefficient. A "1" indicates a negative coefficient, a "0" a positive one. When the bit value of the quantized signal is assigned 1 for the first time, sign bit is arithmetic coded and sent.

Help elements:

layer	scalability layer index
snf	significance of vector to be decoded.
ch	channel index
nch	the number of channel
cur_snf [i]	current significance of the *i*-th vector. cur_snf[] is initialized to Abit[cband]. See subclause 8.3.3.2.5.
maxsnf	maximum of current significance of the vectors to be decoded. See subclause 8.3.3.2.5.
snf	significance index
layer_data_available()	function that returns "1" as long as each layer's bitstream is available, otherwise "0". In other words, it indicates whether the remaing bitstream of each layer is available or not.
layer_group[layer]	indicates the group index of the spectral data to be added newly in the scalability layer. See subclause 8.3.3.2.5
layer_start_index[layer]	indicates the index of the lowest spectral component to be added newly in the scalability layer. See subclause 8.3.3.2.5
layer_end_index[layer]	indicates the index of the highest spectral component to be added newly in the scalability layer. See subclause 8.3.3.2.5
start_index[g]	indicates the index of the lowest spectral component to be coded in the group *g*
end_index[g]	indicates the index of the lowest spectral component to be coded in the group *g*
sliced_bit	the decoded value of the sliced bits of the quantized spectrum.
sample[ch][g][i]	quantized spectral coefficients reconstructed from the decoded bit-sliced data of spectral line i in channel *ch* and group index *g*. See subclause 8.3.3.2.2
sign_is_coded[ch][g][i]	flag that indicates whether the sign of the i^{th} quantized spectrum is already coded (1) or not (0) in channel *ch* and group index *g*.
sign_bit[ch][g][i]	sign bit for non-zero coefficient. A "1" indicates a negative coefficient, a "0" a positive one. When the bit value of the quantized signal is assigned 1 for the first time, sign bit is arithmetic coded and sent.

8.3.4.1.3 Decoding process

In BSAC encoder, the absolute values of quantized spectral coefficients is mapped into a bit-sliced sequence. These slicded bits are the symbols of the arithmetic coding. Every siliced bits are binary arithmetic coded from the lowest-frequency coefficient to the highest-frequency coefficient of the scalability layer, starting the Most Significant Bit(MSB) plane and progressing to the Least Significant Bit(LSB) plane. The arithmetic coding of the sign bits associated with non-zero coefficient follows that of the sliced bit when the bit-slice of the spectral coefficient is 1 for the first time.

For the case of multiple windows per block, the concatenated and possibly grouped and interleaved set of spectral coefficients is treated as a single set of coefficients that progress from low to high as described in subclause 8.3.3.2.6. This set of spectral coefficients may need to be de-interleaved after they are decoded. The spectral information for all scalefactor bands equal to or greater than max_sfb is set to zero.

After all MSB data are encoded from the lowest frequency line to the highest, the same encoding process is repeated until LSB data is encoded or the layer data is not available.

The length of the available bitstream (*available_len[]*) is initialized at the beginning of each layer as described in subclause 8.3.3.2.5. The estimated length of the codeword (*est_cw_len*) to be decoded is calculated from the arithmetic decoding process as described in subclause 8.3.3.2.7. After the arithmetic decoding of a symbol, the length of the available bitstream should be updated by subtracting the estimated codeword length from it. We can detect whether the remaing bitstream of each layer is available or not by checking the available_len.

The bit-sliced data is decoded with the probability which is selected among values listed Table 119 to Table 140.

The probability value should be defined in order to arithmetic-code the symbols (the sliced bits). Binary probability table is made up of probability values (p0) of the symbol ‘0’. First of all, probability table is selected using cband_si as shown in Table 94. Next, the sub-table is selected in the probability table according to the context such as the current significance of the spectral coefficient and the higher bit-slices that have been decoded. All the vector of the higher bit-slices, *higher_bit_vector* are initialized to 0 before the coding of the bit-sliced data is started. Whenever the bit-slice is coded, the vector, *higher_bit_vector* is updated as follows:

```
higher_bit_vector[ch][g][i] = (higher_bit_vector[ch][g][i]<<1) + decoded_bitslice;
if (higher_bit_vector[ch][g][i] > 15)
   higher_bit_vector[ch][g][i] = 15;
```

And, the probability (p0) is selected among the several values in the sub-table. In order to select one of the several probability values in the sub-table, the index of the probability should be decided. If the higher bit-slice vector is non-zero, the index of the probability (p0) is (hgiher_bit_vector[ch][g][i] - 1). Otherwise, it relies upon the sliced bits of successive non-overlapping 4 spectral data as shown in Table 97.

However if the available codeword size is smaller than 14, there is a constraints on the selected probability value as follows:

```
   if (available_len <14) {
       if (p0 < min_p0[available_len])
          p0 = min_p0[available_len];
       else if (p0 > max_p0[available_len])
          p0 = max_p0[available_len];
   }
```

The minimum probability min_p0[] and the maximum probability max_p0[] is listed in Table 98 and Table 99.

Detailed arithmetic decoding procedure is described in this subclause 8.3.3.2.7.

There are 23 probability tables which can be used for encoding/decoding the bit-sliced data. 23 probability table are provided to cover the different statistics of the bit-slices. In order to transmit the probability table used in encoding process, the probability table is included in the syntax element, **cband_si**. After **cband_si** is decoded, the probaiblity table is mapped from **cband_si** using Table 96 and the decoding of the bit-sliced data shall be started.

The current significance of the spectral coefficient represents the bit-plane of the bit-slice to be decoded. Table 96 shows the MSB plane of the decoded sample according to **cband_si**. Current significance, *cur_snf[]* of all spectral coefficient within a coding band are initialized to the MSB plane. For the detailed initialization process, see subclause 8.3.3.2.5

The arithmetic decoding of the sign bit associated with non-zero coefficient follows the arithmetic decoding of the sliced bit when the bit-value of the quantized spectral coefficient is 1 for the first time, with a 1 indicating a negative coefficient and a 0 indicating a positive one. The flag, *sign_is_coded[]* represents whether the sign bit of the quantized spectrum has been decoded or not. Before the decoding of the bit-sliced data is started, all the *sign_is_coded* flags are set to 0. The flag, *sign_is_coded* is set to 1 after the sign bit is decoded. The decoding process of the sign bit can be summarized as follows:

```
i =the spectral line index
if(sample[ch][g][i] && !sign_is_coded[ch][g][i]) {
    arithmetic decoding of the sign bit;
    sign_is_coded[ch][g][i] = 1;
}
```

Decoded symbol need to be reconstructed to the sample. For the detailed reconstruction of the blt-sliced data, see **Reconstruction of the decoded sample from bit-sliced data** part in subclause 8.3.3.2.2.

8.3.4.2 Decoding of stereo_info, ms_used and noise_flag

8.3.4.2.1 Descriptions

The BSAC scalable coding scheme includes the noiseless coding which is different from MPEG-4 AAC coding and further reduce the redundancy of the stereo-related data.

Decoding of the stereo-related data and Perceptual Noise Substitution(pns) data is depended on pns_data_present and stereo_info which indicates the stereo mask. Since the decoded data is the same value with MPEG-4 AAC, the MPEG-4 AAC stereo-related and pns processing follows the decoding of the stereo-related data and pns data.

8.3.4.2.2 Definitions

Bitstream elements:

acode_ms_used[g][sfb] arithmetic codeword from the arithmetic coding of ms_used which is one-bit flag per scalefactor band indicating that M/S coding is being used in window group g and scalefactor band sfb, as follows:

0 Independent

1 ms_used

acode_stereo_info[g][sfb] arithmetic codeword from the arithmetic coding of stereo_info which is two-bit flag per scalefactor band indicating that M/S coding or Intensity coding is being used in window group g and scalefactor band sfb, as follows:
00 Independent
01 ms_used
10 Intensity_in_phase
11 Intensity_out_of_phase or noise_flag_is_used
Note : If ms_mask_present is 3, noise_flag_l and noise_flag_r are 0 value, then stereo_info is interpreted as out-of-phase intensity stereo regardless the value of pns_data_present.

acode_noise_flag[g][sfb] arithmetic codeword from the arithmetic coding of noise_flag which is 1-bit flag per scalefactor band indicating whether the perceptual noise substitution is used(1) or not(0) in window group g and scalefactor band sfb.

acode_noise_flag_l[g][sfb] arithmetic codeword from the arithmetic coding of noise_flag_l which is 1-bit flag per scalefactor band indicating whether the perceptual noise substitution is used(1) or not(0) in the left channel, window group g and scalefactor band sfb .

acode_noise_flag_r[g][sfb] arithmetic codeword from the arithmetic coding of noise_flag which is 1-bit flag per scalefactor band indicating whether the perceptual noise substitution is used(1) or not(0) in the right channel, window group g and scalefactor band sfb.

acode_noise_mode[g][sfb] arithmetic codeword from the arithmetic coding of noise_mode which is two-bit flag per scalefactor band indicating that which noise substitution is being used in window group g and scalefactor band sfb, as follows:
00 Noise Subst L+R (independent)
01 Noise Subst L+R (correlated)
10 Noise Subst L+R (correlated, out-of-phase)
11 reserved

Help elements:

ch channel index

g group index

sfb scalefacotor band index within group

layer scalability layer index

nch the number of channel

ms_mask_present this two bit field indeicates that the stereo mask is
00 Independent
01 1 bit mask of ms_used is located in the layer sfb side information part.
10 All ms_used are ones
11 2 bit mask of stereo_info is located in the layer sfb side information part.

layer_group[layer] indicates the group index of the spectral data to be added newly in the scalability layer. See subclause 8.3.3.2.5

layer_start_sfb[layer] indicates the index of the lowest scalefactor band index to be added newly in the scalability layer. See subclause 8.3.3.2.5

layer_end_sfb[layer] indicates the highest scalefactor band index to be added newly in the scalability layer. See subclause 8.3.3.2.5

8.3.4.2.3 Decoding process

Decoding process of ms_mask_present, noise_flag or ms_used is depended on pns_data_present, number of channel and ms_mask_present. pns_data_present flag is conveyed as a element in syntax of general_header(). pns_data_present indicates whether pns tool is used or not at each frame. stereo_info indeicates the stereo mask as follows :

00 Independent

01 1 bit mask of ms_used is located in the layer sfb side information part.

10 All ms_used are ones

11 2 bit mask of stereo_info is located in the layer sfb side information part.

Detailed arithmetic decoding procedure is described in this subclause 8.3.3.2.7.

Decoding process is classified as follows :

- 1 channel, no pns data
- If the number of channel is 1 and pns data is not present, there is no bitstream elements related to stereo or pns.
- 1 channel, pns data

If the number of channel is 1 and pns data is present, noise flag of the scalefactor bands between **pns_start_sfb** to **max_sfb** is arithmetic decoded using model shown in Table 117. Perceptual noise substitution is done according to the decoded noise flag.

- 2 channel, ms_mask_present=0 (Independent), No pns data

If ms_mask_present is 0 and pns data is not present, arithmetic decoding of stereo_info or ms_used is not needed.

- 2 channel, ms_mask_present=0 (Independent), pns data

If ms_mask_present is 0 and pns data is present, noise flag for pns is arithmetic decoded using model shown in Table 117. Perceptual noise substitution of independent mode is done according to the decoded noise flag.

- 2 channel, ms_mask_present=2 (all ms_used), pns data or no pns data

All ms_used values are ones in this case. So, M/S stereo processing of AAC is done at all scalefactor band. And naturally there can be no pns processing regardless of pns_data_present flag.

- 2 channel, ms_mask_present=1 (optional ms_used), pns data or no pns data

1 bit mask of max_sfb bands of ms_used is conveyed in this case. So, ms_used is arithmetic decoded using the ms_used model given in Table 115. M/S stereo processing of AAC is done or not according to the decoded ms_used. And there is no pns processing regardless of pns_data_present flag

- 2 channel, ms_mask_present=3 (optional ms_used/intensity/pns), no pns data

At first, stereo_info is arithmetic decoded using the stereo_info model given in Table 116.

stereo_info is is two-bit flag per scalefactor band indicating that M/S coding or Intensity coding is being used in window group g and scalefactor band sfb as follows :

00 Independent

01 ms_used

10 Intensity_in_phase

11 Intensity_out_of_phase

If stereo_info is not 0, M/S stereo or intensity stereo of AAC is done with these decoded data. Since pns data is not present, we don_t have to process pns.

- 2 channel, ms_mask_present=3 (optional ms_used/intensity/pns), pns data stereo_info is arithmetic decoded using the stereo_info model given in Table 116.

If stereo_info is 1 or 2, M/S stereo or intensity stereo processing of AAC is done with these decoded data and there is no pns processing.

If stereo_info is 3 and scalefactor band is larger than or equal to pns_start_sfb, noise flag for pns is arithmetic decoded using model given in Table 117. And then if the both noise flags of two channel are 1, noise substitution mode is arithmetic decoded using model given in Table 118. The perceptual noise is substituted or out_of_phase intensity stereo processing is done according to the substitution mode. Otherwise, the perceptual noise is substituted only if noise flag is 1.

If stereo_info is 3 and scalefactor band is smaller than pns_start_sfb, out_of_phase intensity stereo processing is done.

8.3.4.3 Decoding of scalefactors, noise energy and intensity stereo position

8.3.4.3.1 Description

The BSAC scalable coding scheme includes the noiseless coding which is different from AAC and further reduce the redundancy of the scalefactors.

The max_scalefactor is coded as an 8 bit unsigned integer. The scalefactors are differentially coded relative to the max_scalefactor value and then Arithmetic coded using the differential scalefactor model.

8.3.4.3.2 Definitions

Bitstream element:

acode_scf[ch][g][sfb] Arithmetic codeword from the coding of the differential scalefactors.

acode_max_noise_energy [ch] Arithmetic codeword from the coding of the maximum of the noise energies.

acode_dpcm_noise_energy_index[ch][g][sfb] Arithmetic codeword from the coding of the differential noise energy index.

acode_is_position_index[g][sfb] Arithmetic codeword from the coding of the intensity stereo poistion index.

Help elements:

ch channel index

g group index

sfb scalefacotor band index within group

layer scalability layer index

nch the number of channel

layer_group[layer] indicates the group index of the spectral data to be added newly in the scalability layer. See subclause 8.3.3.2.5

layer_start_sfb[layer] indicates the index of the lowest scalefactor band index to be added newly in the scalability layer. See subclause 8.3.3.2.5

layer_end_sfb[layer] indicates the highest scalefactor band index to be added newly in the scalability layer. See subclause 8.3.3.2.5

scf[ch][g][sfb] indicates the scalefactors.

max _noise_energy[ch] indicates the maximum of the noise energy.

dpcm_noise_energy_index[ch][g][sfb] indicates the differential noise energy index.

is_position_index[g][sfb] indicates the intensity stereo poistion index.

8.3.4.3.3 Decoding process

The spectral coefficients are divided into scalefactor bands that contain a multiple of 4 quantized spectral coefficients. Each scalefactor band has a scalefactor.

The differential scalefactor index is arithmetic-decoded using the arithmetic model given in Table 95. The arithmetic model of the scalefactor for the base layer is given as a 3 bit unsigned integer bitstream element, **base_scf_model**. The arithmetic model of the scalefactor for the enhancement layers is given as a 3 bit unsigned integer bitstream element, **enh_scf_model**.

For all scalefactors the difference to the offset value is arithmetic-decoded. All scalefactors are calculated from the difference and the offset value. The offset value is given explicitly as a 8 bit PCM in the bitstream element **max_scalefactor[ch]**. Detailed arithmetic decoding procedure is described in this subclause 8.3.3.2.7.

The following pseudo code describes how to decode the scalefactors *scf[ch][g][sfb]* in base layer and each enhancement layer:

```
for (ch =0; ch<nch; ch++) {
    g = layer_group[ch][g][sfb];
    for (sfb = layer_start_sfb[layer]; sfb < layer_end_sfb[layer]; sfb++) {
        diff_scf = arithmetic_decoding();
        scf[ch][g][sfb] = max_scalefactor[ch] - diff_scf;
    }
}
```

If noise substituition coding is active for a particular group and scalefactor band, a noise energy value is transmitted instead of the scalefactor of the respective channel.

Noise energies are arithmetic-coded of differential values. For all noise energies the difference to the offset value is arithmetic-decoded. All noise energies are calculated from the difference and the offset value. The offset value, max_noise_energy[ch] is arithmetic-decoded before the first differential noise energy is decoded.

Noise substitution decoding process is same as the PNS part of MPEG-4 Audio General Audio. The noise energy decoding in each layer is defined by the following pseudo code:

```
for (ch =0; ch<nch; ch++) {
    g = layer_group[ch][g][sfb];
    for (sfb = layer_start_sfb[layer]; sfb < layer_end_sfb[layer]; sfb++) {
        if (noise_flag[ch][g][sfb]) {
            dpcm_noise_energy_index[ch][g][sfb] = arithmetic_decoding();
            noise_nrg[ch][g][sfb] = max_noise_energy[ch] - dpcm_noise_energy[ch][g][sfb];
        }
    }
}
```

The direction information for the intensity stereo decoding is represented by an “intensity stereo position” value indicating the relation between left and right channel scaling. If intensity stereois active for a particular group and scalefactor band, an intensity stereo position value is transmitted in stead of the scalefactor of the right channel.

When intensity positions are arithmetic-coded, the same arithmetic model is used. The intensity decoding process is same as the intensity stereo of MPEG-4 Audio General Audio. The intensity stereo position decoding in each layer is defined by the following pseudo code:

```
g = layer_group[1][g][sfb]
for (sfb = layer_start_sfb[layer]; sfb < layer_end_sfb[layer]; sfb++) {
    if (stereo_info[g][sfb] && ch==1) {
        is_position_index[g][sfb] = arithmetic_decoding();
        if (is_position_sign[g][sfb]%2)
            is_position[g][sfb] = -(int)((is_position_index[g][sfb]+1)/2);
        else
            is_position[g][sfb] = (int)(is_position_index[g][sfb]/2);
    }
}
```

8.3.4.4 Decoding of coding band side information

8.3.4.4.1 Descriptions

In BSAC scalable coding scheme, the spectral coefficients are divided into coding bands which contain 32 quantized spectral coefficients for the noiseless coding. Coding bands are the basic units used for the noiseless coding. The set of bit-sliced sequence is divided into coding bands. The MSB plane and the probability table of each coding band are included in this layer coding band side information, **cband_si** as shown in Table 96. The coding band side informations of each layer are transmitted starting from the lowest coding band (layer_start_cband[layer]) and progressing to the highest coding band (layer_end_cband[layer]). For all cband_si, it is arithmetic-coded using the arithmetic model as given in Table 94.

8.3.4.4.2 Definitions

Bitstream element:

acode_cband_si[ch][g][cband] Arithmetic codeword from the arithmetic coding of **cband_si** for each coding-band.

Help elements:

g group index

cband coding band index within group

ch channel index

nch the number of channel

layer_group[layer] indicates the group index of the spectral data to be added newly in the scalability layer. See subclause 8.3.3.2.5

layer_start_cband[layer] indicates the lowest coding band index to be added newly in the scalability layer. See subclause 8.3.3.2.5

layer_end_cband[layer] indicates the highest coding band index to be added newly in the scalability layer. See subclause 8.3.3.2.5

8.3.4.4.3 Decoding process

cband_si is arithmetic-coded using the arithmetic model as given in Table 94. The arithmetic model used for coding cband_si is dependent on a 5-bit unsigned integer in the bitstream element, **cband_si_type** as shown in Table 94. And, the largest value of the decodable cband_si is given in Table 94. If the decoded cband_si larger than this value, it can be considered that there was a bit-error in the bitream.Detailed arithmetic decoding procedure is described in this subclause 8.3.3.2.7.

The following pseudo code describes how to decode the cband_si *cband_si[ch][g][cband]* in base layer and each enhancement layer:

```
g = layer_group[layer];
for ( ch=0; ch<nch; ch++ ) {
    for ( cband=layer_start_cband[g][layer]; cband<layer_cband[g][layer+1]; cband++ ) {
        cband_si[ch][g][cband] = arithmetic_decoding();
        if (cband_si[ch][g][cband] > largest_cband_si)
            bit_error_is_generated;
    }
}
```

where, layer_cband[g][layer] is the start coding band and layer_cband[g][layer+1] is the end coding band for decoding the arithmetic model index in each layer.

8.3.4.5 Segmented binary arithmetic coding (SBA)

8.3.4.5.1 Tool description

Segmented Binary Arithmetic Coding (SBA) is based on the fact that the arithmetic codewords can be partitioned at known positions so that these codewords can be decoded independent of any error within other sections. Therefore, this tool avoids error propagation to those sections. The arithmetic coding should initialized at the beginning of these segments and terminated at the end of these segments in order to localize the arithmetic codewords. This tool is activated if the syntax element, **sba_mode** is 1. And this flag should be set to 1 if the BSAC is used in the error-prone environment.

8.3.4.5.2 Definitions

There is no definition because only the initialization and termination process are added at the beginning and the end of the segments in order to localize the arithmetic codewords.

8.3.4.5.3 Decoding process

The arithmetic coding is terminated at the end of the segments, and re-initialized at the beginning of the next segment. The segment is made up of the scalability layers. terminal_layer[layer] indicates whether each layer is the last layer of the segment, which is set as follows :

```
for (layer = 0; layer < (top_layer+slayer_size-1); layer++) {
    if (layer_start_cband[layer] != layer_start_cband[layer+1])
        terminal_layer[layer] = 1;
    else
        terminal_layer[layer] = 0;
    }
}
```

where, toplayer is the top layer to be encoded, layer_max_cband[] are the maximum coding band limit to be encoded and slayer_size is the sub-layer size of the base layer. Figure 11 shows an example of the segmented bistream to be made in the encoder.

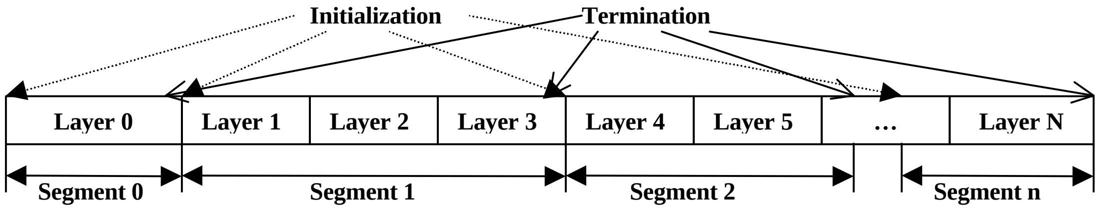


Figure 11: The structure of SBA coded bitstream

In the decoder, the bitstream of each layer is split from the total bitstream. If the previous layer is the last of the segment, the split bitstream is stored in the independent buffer and arithmetic decoding process is re-initialized. Otherwise, the split bitstream is concatenated to that of the previous layer and used for arithmetic decoding sequentially.

In order to do the arithmetic decoding perfectly, 32-bit zero value should be concatenated to the split bitstream if the layer is the last of the segment. Figure 12 shows an example of the bitstream spliting and zero stuffing in decoder part.

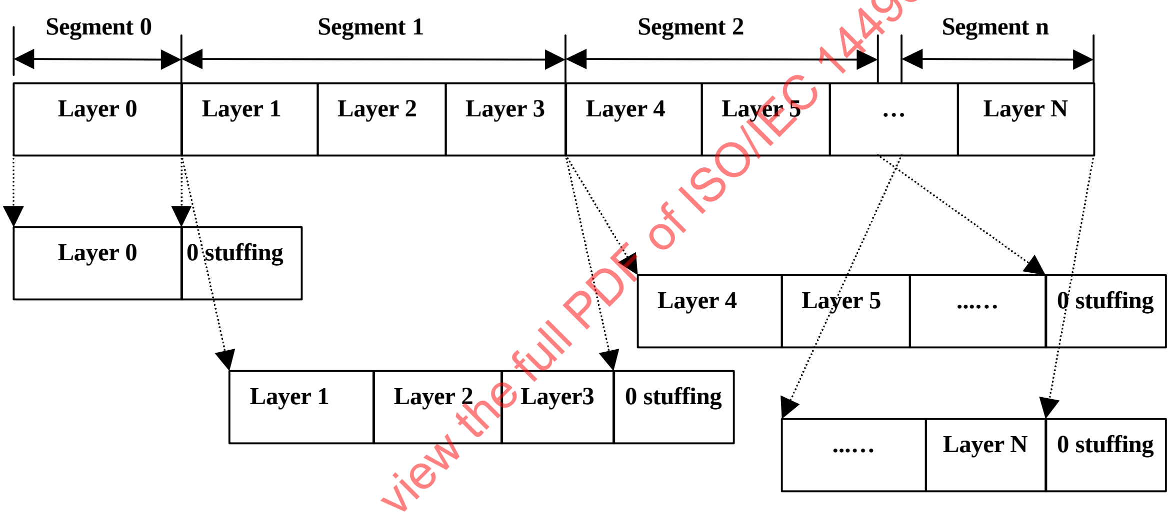


Figure 12: Decoding of SBA bitstream

8.3.4.6 Tables

Table 94: cband_si_type parameters

cband_si_type	max_cband_si_len	Largest cband_si		Model listed in	
		0th cband	Other cband	0th cband	Other cband
0	6	6	4	Table 114	Table 107
1	5	6	6	Table 114	Table 108
2	6	8	4	Table 114	Table 107
3	5	8	6	Table 114	Table 108
4	6	8	8	Table 114	Table 109
5	6	10	4	Table 114	Table 107
6	5	10	6	Table 114	Table 108
7	6	10	8	Table 114	Table 109
8	5	10	10	Table 114	Table 110
9	6	12	4	Table 114	Table 107
10	5	12	6	Table 114	Table 108
11	6	12	8	Table 114	Table 109
12	8	12	12	Table 114	Table 111
13	6	14	4	Table 114	Table 107
14	5	14	6	Table 114	Table 108
15	6	14	8	Table 114	Table 109
16	8	14	12	Table 114	Table 111
17	9	14	14	Table 114	Table 112
18	6	15	4	Table 114	Table 107
19	5	15	6	Table 114	Table 108
20	6	15	8	Table 114	Table 109
21	8	15	12	Table 114	Table 111
22	10	15	15	Table 114	Table 113
23	8	16	12	Table 114	Table 111
24	10	16	16	Table 114	Table 113
25	9	17	14	Table 114	Table 112
26	10	17	17	Table 114	Table 113
27	10	18	18	Table 114	Table 113
28	12	19	19	Table 114	Table 113
29	12	20	20	Table 114	Table 113
30	12	21	21	Table 114	Table 113
31	12	22	22	Table 114	Table 113

Table 95: Scalefactor model parameters

scf_model	Largest Differential ArModel	Model listed in
0	0	not used
1	3	Table 100
2	7	Table 101
3	15	Table 102
4	15	Table 103
5	31	Table 104
6	31	Table 105
7	63	Table 106

Table 96: BSAC cband_si parameters

cband_si	MSB plane	Table listed in	cband_si	MSB plane	Table listed in
0	0	-	12	6	Table 130
1	1	Table 119	13	7	Table 131
2	1	Table 120	14	7	Table 132
3	2	Table 121	15	8	Table 133
4	2	Table 122	16	9	Table 134
5	3	Table 123	17	10	Table 135
6	3	Table 124	18	11	Table 136
7	4	Table 125	19	12	Table 137
8	4	Table 126	20	13	Table 138
9	5	Table 127	21	14	Table 139
10	5	Table 128	22	15	Table 140
11	6	Table 129			

Table 97: Position of probability value in probability table

				h	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
				g	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
				f	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
				e	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
a	b	c	d																	
0	x	x	x		0	15	22	29	32	39	42	45								
1	x	x	0		1	16	23	30					46	53	56	59				
	x	x	1		2	17	24	31					46	53	56	59				
2	x	0	0		3	18			33	40			47	54			60	63		
	x	0	1		4	19			33	40			48	55			60	63		
	x	1	0		5	20			34	41			47	54			60	63		
	x	1	1		6	21			34	41			48	55			60	63		
3	0	0	0		7		25		35		43		49		57		61		64	
	0	0	1		8		25		36		43		50		57		62		64	
	0	1	0		9		26		35		43		51		58		61		64	
	0	1	1		10		26		36		43		52		58		62		64	
	1	0	0		11		27		37		44		49		57		61		64	
	1	0	1		12		27		38		44		50		57		62		64	
	1	1	0		13		28		37		44		51		58		61		64	
	1	1	1		14		28		38		44		52		58		62		64	

where, i = spectral index

a = i % 4

b = the sliced bit of (i-3)th spectral data whose significance is same with that of i-th spectral data

c = the sliced bit of (i-2)th spectral data whose significance is same with that of i-th spectral data

d = the sliced bit of (i-1)th spectral data whose significance is same with that of i-th spectral data

e = whether the higher bits of the (i-a+3)th spectral data whose sigificance is larger than that of i-th spectral data is nonzero (1) or zero(0)

f = whether the higher bits of the (i-a+2)th spectral data whose sigificance is larger than that of i-th spectral data is nonzero (1) or zero(0)

g = whether the higher bits of the (i-a+1)th spectral data whose sigificance is larger than that of i-th spectral data is nonzero (1) or zero(0)

h = whether the higher bits of the (i-a)th spectral data whose sigificance is larger than that of i-th spectral data is nonzero (1) or zero(0)

Table 98: The minimum probability(min_p0) in proportion to the available length of the layer

Available length	1	2	3	4	5	6	7	8	9	10	11	12	13
min_p0 (hexadecimal)	2000	1000	800	400	200	100	80	40	20	10	8	4	2

Table 99: The maximum probability(max_p0) in proportion to the available length of the layer

Available length	1	2	3	4	5	6	7	8	9	10	11	12	13
max_p0 (hexadecimal)	2	4	8	10	20	40	80	100	200	400	800	1000	2000

Table 100: Scalefactor arithmetic model 1

size	cumulative frequencies (hexadecimal)			
4	752,	3cd,	14d,	0,

Table 101: Scalefactor arithmetic model 2

size	cumulative frequencies (hexadecimal)							
8	112f,	de7,	a8b,	7c1,	47a,	23a,	d4,	0,

Table 102: Scalefactor arithmetic model 3

size	cumulative frequencies (hexadecimal)							
16	1f67,	1c5f,	18d8,	1555,	1215,	eb4,	adc,	742,
	408,	1e6,	df,	52,	32,	23,	c,	0,

Table 103: Scalefactor arithmetic model 4

size	cumulative frequencies (hexadecimal)							
16	250f,	22b8,	2053,	1deb,	1b05,	186d,	15df,	12d9,
	f77,	c01,	833,	50d,	245,	8c,	33,	0,

Table 104: Scalefactor arithmetic model 5

size	cumulative frequencies (hexadecimal)							
32	8a8,	74e,	639,	588,	48c,	3cf,	32e,	272,
	1bc,	13e,	e4,	97,	69,	43,	2f,	29,
	20,	1b,	18,	15,	12,	f,	d,	c,
	a,	9,	7,	6,	4,	3,	1,	0,

Table 105: Scalefactor arithmetic model 6

size	cumulative frequencies (hexadecimal)							
32	c2a,	99f,	809,	6ec,	603,	53d,	491,	40e,
	394,	30a,	2a5,	259,	202,	1bc,	170,	133,
	102,	c9,	97,	73,	4f,	37,	22,	16,
	f,	b,	9,	7,	5,	3,	1,	0,

Table 106: Scalefactor arithmetic model 7

size	cumulative frequencies (hexadecimal)							
64	3b5e,	3a90,	39d3,	387c,	3702,	3566,	33a7,	321c,
	2f90,	2cf2,	29fe,	26fa,	23e4,	20df,	1e0d,	1ac4,
	1804,	159a,	131e,	10e7,	e5b,	c9c,	b78,	a21,
	8fd,	7b7,	6b5,	62c,	55d,	4f6,	4d4,	44b,
	38e,	2e2,	29d,	236,	225,	1f2,	1cf,	1ad,
	19c,	179,	168,	157,	146,	135,	123,	112,
	101,	f0,	df,	ce,	bc,	ab,	9a,	89,
	78,	67,	55,	44,	33,	22,	11,	0,

Table 107: cband_si arithmetic model 0

size	cumulative frequencies (hexadecimal)							
5	3ef6,	3b59,	1b12,	12a3,	0,			

Table 108: cband_si arithmetic model 1

size	cumulative frequencies (hexadecimal)							
7	3d51,	33ae,	1cff,	fb7,	7e4,	22b,	0,	

Table 109: cband_si arithmetic model 2

size	cumulative frequencies (hexadecimal)							
9	3a47,	2aec,	1e05,	1336,	e7d,	860,	5e0,	44a,
	0,							

Table 110: cband_si arithmetic model 3

size	cumulative frequencies (hexadecimal)							
11	36be,	27ae,	20f4,	1749,	14d5,	d46,	ad3,	888,
	519,	20b,	0,					

Table 111: cband_si arithmetic model 4

size	cumulative frequencies (hexadecimal)							
13	3983,	2e77,	2b03,	1ee8,	1df9,	1307,	11e4,	b4d,
	94c,	497,	445,	40,	0,			

Table 112: cband_si arithmetic model 5

size	cumulative frequencies (hexadecimal)							
15	306f,	249e,	1f56,	1843,	161a,	102d,	f6c,	c81,
	af2,	7a8,	71a,	454,	413,	16,	0,	

Table 113: cband_si arithmetic model 6

size	cumulative frequencies (hexadecimal)							
23	31af,	2001,	162d,	127e,	f05,	c34,	b8f,	a61,
	955,	825,	7dd,	6a9,	688,	55b,	54b,	2f7,
	198,	77,	10,	c,	8,	4,	0,	

Table 114: cband_si arithmetic model for 0th coding band

size	cumulative frequencies (hexadecimal)							
23	3ff8,	3ff0,	3fe8,	3fe0,	3fd7,	3f31,	3cd7,	3bc9,
	3074,	2bcf,	231b,	13db,	d51,	603,	44c,	80,
	30,	28,	20,	18,	10,	8,	0,	

Table 115: MS_used model

size	cumulative frequencies (hexadecimal)	
2	2000,	0,

Table 116: stereo_info model

size	cumulative frequencies(hexadecimal)			
4	3666,	1000,	666,	0,

Table 117: noise_flag arithmetic model

size	cumulative frequencies(hexadecimal)	
2	2000,	0,

Table 118: noise_mode arithmetic model

size	cumulative frequencies(hexadecimal)			
4	3000,	2000,	1000,	0,

Table 119: BSAC probability table 1

(MSB plane = 1)

Significance	Probability Value of symbol ‚0‘ (Hexadecimal)							
1	3900,	3a00,	2f00,	3b00,	2f00,	3700,	2c00,	3b00,
	3000,	3600,	2d00,	3900,	2f00,	3700,	2c00,	

Table 120: BSAC probability table 2

(MSB plane = 1)

Significance	Probability Value of symbol ,'0' (Hexadecimal)							
1	2800,	2800,	2500,	2900,	2600,	2700,	2300,	2a00,
	2700,	2800,	2400,	2800,	2500,	2600,	2200,	

Table 121: BSAC probability table 3

(MSB plane = 2)

Significance	decoded higher bits	Probability Value of symbol ,'0' (Hexadecimal)							
2	zero	3d00,	3d00,	3300,	3d00,	3300,	3b00,	3300,	3d00,
		3200,	3b00,	3100,	3e00,	3700,	3c00,	3300,	
1	zero	3700,	3a00,	2800,	3b00,	2600,	2c00,	2400,	3a00,
		2500,	2b00,	2400,	3100,	2300,	2900,	2300,	3000,
		2c00,	1d00,	2200,	1a00,	1c00,	1600,	2700,	2200,
		1a00,	1d00,	1900,	1c00,	1e00,	2c00,	2400,	1900,
		1e00,	1f00,	1c00,	2b00,	2400,	2900,	2700,	2400,
		1300,	1a00,	2000,	1800,	2300,	2500,	1f00,	2c00,
		2300,	3600,	2800,	3100,	2500,	1400,	1200,	1800,
		1400,	2100,	2200,	1000,	1e00,	3000,	2600,	1200,
		2200,							
	non-zero	3100,							

Table 122: BSAC probability table 4

(MSB plane = 2)

Significance	decoded higher bits	Probability Value of symbol ,'0' (Hexadecimal)							
2	zero	3900,	3a00,	2e00,	3a00,	2f00,	3400,	2a00,	3a00,
		3000,	3500,	2c00,	3600,	2b00,	3100,	2500,	
1	zero	1e00,	1d00,	1c00,	1d00,	1c00,	1d00,	1b00,	1d00,
		1e00,	1e00,	1a00,	1e00,	1c00,	1d00,	1b00,	1a00,
		1a00,	1800,	1800,	1800,	1700,	1700,	1800,	1a00,
		1700,	1700,	1900,	1800,	1600,	1700,	1600,	1500,
		1700,	1800,	1600,	1c00,	1700,	1900,	1700,	1500,
		1c00,	1500,	1600,	f00,	1800,	1400,	1700,	1a00,
		1a00,	1e00,	1800,	1c00,	1b00,	1500,	1300,	1500,
		1400,	1600,	1500,	1700,	1600,	1b00,	1800,	1400,
		1400,							
	non-zero	3600,							

Table 123: BSAC probability table 5

(MSB plane = 3)

Significance	decoded higher bits	Probability Value of symbol ‚0‘ (Hexadecimal)							
3	zero	3d00,	3d00,	3200,	3d00,	3300,	3d00,	3600,	3d00,
		3500,	3c00,	3500,	3f00,	3b00,	3f00,	3d00,	
2	zero	3c00,	3d00,	2b00,	3d00,	2900,	3500,	2c00,	3d00,
		2b00,	3400,	2b00,	3800,	2b00,	3700,	2a00,	3900,
		3400,	2400,	2a00,	1c00,	1f00,	1600,	3500,	2500,
		1a00,	2a00,	2200,	2b00,	2a00,	3500,	2600,	1a00,
		2600,	2500,	2700,	3500,	2d00,	3800,	3200,	2e00,
		1800,	1600,	2900,	2500,	3100,	2c00,	2300,	3600,
		3000,	3c00,	3300,	3b00,	3400,	1700,	1a00,	1c00,
		1900,	2900,	2a00,	2400,	2700,	3c00,	3600,	1d00,
		3100,							
	non-zero	3100,							
1	zero	3400,	3800,	2700,	3900,	2700,	2f00,	2200,	3800,
		2500,	2d00,	2000,	3300,	2000,	2900,	1e00,	2b00,
		2300,	1a00,	1a00,	1b00,	1800,	1700,	1e00,	1c00,
		1b00,	1c00,	1b00,	1a00,	1800,	1d00,	1b00,	1800,
		1900,	1b00,	1a00,	1d00,	1e00,	1f00,	1b00,	1e00,
		1200,	1400,	1a00,	1300,	1c00,	1b00,	1900,	2000,
		1e00,	3000,	2900,	2d00,	2500,	1300,	1700,	1400,
		1300,	1e00,	1f00,	1100,	1900,	2100,	1e00,	1500,
		1a00,							
	non-zero	2a00,	2b00,	2800,					

Table 124: BSAC probability table 6

(MSB plane = 3)

Significance	decoded higher bits	Probability Value of symbol ‚'0' (Hexadecimal)							
3	zero	3800,	3a00,	2d00,	3a00,	2d00,	3600,	2d00,	3a00,
		2d00,	3600,	2b00,	3a00,	2800,	3600,	2700,	
2	zero	2b00,	3000,	2500,	2f00,	2600,	2d00,	2400,	3000,
		2500,	2b00,	2400,	2d00,	2500,	2800,	2500,	2a00,
		2900,	2300,	2200,	1e00,	1b00,	1900,	2600,	2300,
		1f00,	1d00,	2200,	1b00,	1800,	2100,	2100,	1d00,
		1d00,	1f00,	1f00,	2900,	2600,	2a00,	2100,	2300,
		1800,	1a00,	1d00,	2000,	1c00,	1a00,	1e00,	2900,
		2800,	2f00,	2300,	2f00,	2600,	1d00,	1700,	1d00,
		1c00,	1e00,	2100,	1700,	2200,	2300,	2300,	1400,
		1a00,							
	non-zero	3000,							
1	zero	1900,	1900,	1900,	1b00,	1700,	1b00,	1a00,	1000,
		1900,	1600,	1800,	1e00,	1900,	1a00,	1700,	1b00,
		1700,	1500,	1500,	1500,	1700,	1400,	1900,	1700,
		1600,	1600,	1200,	1300,	1200,	1600,	1500,	1500,
		1300,	1600,	1600,	1c00,	1400,	1700,	1600,	1400,
		1400,	1400,	1500,	1400,	1300,	1300,	1500,	1800,
		1600,	1f00,	1a00,	1e00,	1800,	1700,	1600,	1600,
		1300,	1400,	1300,	1100,	1500,	1600,	1500,	1200,
		1300,							
	non-zero	2b00,	2800,	2700,					

Table 125: BSAC probability table 7

(MSB plane = 4)

Significance	decoded higher bits	Probability Value of symbol ‚'0' (Hexadecimal)							
MSB	zero	3d00,	3d00,	3500,	3e00,	3500,	3f00,	3b00,	3e00,
		3200,	3f00,	3a00,	3f00,	3d00,	3f00,	3b00,	
MSB-1	zero	3f00,	3f00,	3200,	3f00,	3500,	3e00,	3700,	3f00,
		2d00,	3c00,	3000,	3f00,	3700,	3e00,	3400,	3f00,
		3900,	2600,	2f00,	1e00,	2400,	1500,	3700,	3100,
		1b00,	2600,	2300,	3a00,	3900,	3e00,	2b00,	2200,
		2800,	2f00,	2500,	3e00,	3700,	3e00,	3d00,	3900,
		1a00,	3300,	2500,	2800,	3c00,	3800,	2c00,	3d00,
		3800,	3f00,	3b00,	3f00,	3a00,	1e00,	1b00,	1800,
		1800,	3b00,	3a00,	1200,	2f00,	3f00,	3b00,	1b00,
		3500,							
	non-zero	2100,							
MSB-2	zero	3c00,	3e00,	3000,	3e00,	3100,	3a00,	3100,	3d00,
		2c00,	3900,	2e00,	3c00,	2d00,	3c00,	3100,	3d00,
		3100,	2100,	2c00,	2600,	2800,	1d00,	2b00,	2800,
		2800,	2400,	2200,	2100,	2300,	2d00,	2500,	1f00,
		2100,	2b00,	2700,	3200,	2d00,	3400,	2a00,	3500,
		1800,	1800,	1f00,	1e00,	2e00,	2a00,	2400,	3000,
		2b00,	3e00,	3d00,	3d00,	3a00,	1e00,	2b00,	2600,
		1900,	3400,	3500,	1c00,	2600,	3300,	2a00,	1c00,
		2b00,							
	non-zero	2800,	2900,	2400,					
Others	zero	3500,	3b00,	2900,	3b00,	2a00,	3100,	2700,	3b00,
		2600,	2f00,	2400,	3400,	2300,	2d00,	2000,	3300,
		2700,	1c00,	2400,	1c00,	1c00,	1900,	2700,	2800,
		1b00,	1d00,	2000,	1b00,	1a00,	2300,	1d00,	1700,
		1e00,	2400,	2100,	2b00,	2100,	2800,	2000,	2300,
		1b00,	1500,	1b00,	1400,	1a00,	1a00,	2000,	2a00,
		2200,	3700,	2f00,	3200,	2a00,	1700,	1700,	1600,
		1900,	2500,	2300,	1500,	1900,	2500,	2200,	1400,
		1b00,							
	non-zero	2d00,	2500,	2300,	2500,	2500,	2600,	2400,	

Table 126: BSAC probability table 8

(MSB plane = 4)

Significance	decoded higher bits	Probability Value of symbol ‚0' (Hexadecimal)							
MSB	zero	3b00,	3c00,	3400,	3c00,	3400,	3a00,	3000,	3c00,
		3200,	3a00,	3100,	3c00,	3000,	3900,	2f00,	
MSB-1	zero	3500,	3800,	2c00,	3900,	2c00,	3400,	2b00,	3800,
		2e00,	3400,	2d00,	3600,	2a00,	3300,	2800,	3100,
		3100,	2600,	2900,	2000,	2300,	1f00,	2d00,	2600,
		2000,	2600,	2300,	2500,	2100,	2c00,	2400,	1d00,
		2500,	2400,	2400,	3000,	2800,	3000,	2900,	2200,
		1e00,	1c00,	2500,	1d00,	2300,	2300,	2500,	3300,
		2c00,	3700,	2b00,	3400,	2c00,	1e00,	1c00,	2100,
		1b00,	2900,	2a00,	1d00,	2600,	3200,	2a00,	2000,
		2400,							
	non-zero	3200,							
MSB-2	zero	2900,	2e00,	2600,	2f00,	2600,	2d00,	2600,	2e00,
		2500,	2b00,	2600,	2f00,	2300,	2a00,	2300,	2800,
		2800,	2100,	2400,	2000,	2000,	1b00,	2400,	1f00,
		1c00,	2100,	2200,	1d00,	1c00,	1f00,	1c00,	1900,
		1e00,	2100,	2100,	2900,	2200,	2300,	2100,	1c00,
		1a00,	1a00,	2100,	2100,	1c00,	1c00,	1f00,	2700,
		2500,	2d00,	2700,	2a00,	2300,	1c00,	1d00,	1a00,
		1a00,	1b00,	1d00,	1800,	2000,	2300,	1f00,	1900,
		1c00,							
	non-zero	2b00,	2900,	2800,					
Others	zero	1c00,	1e00,	1b00,	1e00,	1c00,	1e00,	1900,	1a00,
		1f00,	1f00,	1900,	2000,	1a00,	1f00,	1700,	1b00,
		1a00,	1900,	1800,	1900,	1800,	1600,	1900,	1a00,
		1900,	1700,	1800,	1700,	1800,	1600,	1700,	1400,
		1600,	1800,	1a00,	1c00,	1c00,	1c00,	1700,	1700,
		1500,	1500,	1600,	1600,	1500,	1400,	1700,	1b00,
		1a00,	2300,	1c00,	1d00,	1a00,	1600,	1600,	1500,
		1400,	1800,	1500,	1300,	1700,	1900,	1600,	1400,
		1400,							
	non-zero	2800,	2500,	2500,	2700,	2500,	2600,	2500,	

Table 127: BSAC probability table 9

(MSB plane = 5)

Significance	decoded higher bits	Probability Value of symbol ‚0' (Hexadecimal)							
MSB	zero	3d00,	3e00,	3300,	3e00,	3500,	3e00,	3700,	3e00,
		3400,	3e00,	3500,	3f00,	3d00,	3f00,	3c00,	
MSB-1	zero	same as BSAC probability table 8 (see Table 126)							
	non-zero	2e00,							
MSB-2	zero	same as BSAC probability table 8 (see Table 126)							
	non-zero	2900,	2a00,	2700,					
MSB-3	zero	same as BSAC probability table 8 (see Table 126)							
	non-zero	2d00,	2500,	2400,	2500,	2400,	2500,	2300,	
Others	zero	same as BSAC probability table 8 (see Table 126)							
	non-zero	2800,	2500,	2300,	2300,	2200,	2200,	2200,	2200,
		2200,	2200,	2200,	2100,	2000,	2200,	2100,	

Table 128: BSAC probability table 10

(MSB plane = 5)

Significance	decoded higher bits	Probability Value of symbol ‚0‘ (Hexadecimal)
MSB	zero	3b00, 3c00, 3400, 3c00, 3200, 3900, 2e00, 3d00, 3400, 3900, 2f00, 3c00, 2d00, 3700, 2d00,
MSB-1	zero	same as BSAC probability table 8 (see Table 126)
	non-zero	3100,
MSB-2	zero	same as BSAC probability table 8 (see Table 126)
	non-zero	2b00, 2a00, 2900,
MSB-3	zero	same as BSAC probability table 8 (see Table 126)
	non-zero	2700, 2600, 2500, 2500, 2500, 2200, 2200,
Others	zero	same as BSAC probability table 8 (see Table 126)
	non-zero	2200, 2300, 2300, 2300, 2200, 2300, 2200, 2300, 2200, 2200, 2200, 2200, 2200, 2000, 2100,

Table 129: BSAC probability table 11

same as BSAC probability table 10, but MSB plane = 6

Table 130: BSAC probability table 12

same as BSAC probability table 11, but MSB plane = 6

Table 131: BSAC probability table 13

same as BSAC probability table 10, but MSB plane = 7

Table 132: BSAC probability table 14

same as BSAC probability table 11, but MSB plane = 7

Table 133: BSAC probability table 15

same as BSAC probability table 10, but MSB plane = 8

Table 134: BSAC probability table 16

same as BSAC probability table 10, but MSB plane = 9

Table 135: BSAC probability table 17

same as BSAC probability table 10, but MSB plane = 10

Table 136: BSAC probability table 18

same as BSAC probability table 10, but MSB plane = 11

Table 137: BSAC probability table 19

same as BSAC probability table 10, but MSB plane = 12

Table 138: BSAC probability table 20

same as BSAC probability table 10, but MSB plane = 13

Table 139: BSAC probability table 21

same as BSAC probability table 10, but MSB plane = 14

Table 140: BSAC probability table 22

same as BSAC probability table 10, but MSB plane = 15

8.4 Low delay coding mode

8.4.1 Introduction

The low delay coding functionality provides the ability to extend the usage of generic low bitrate audio coding to applications requiring a very low delay of the encoding / decoding chain (e.g. full-duplex real-time communications).

This subpart specifies a low delay audio coder providing a mode with an algorithmic delay not exceeding 20 ms.

The overall algorithmic delay of a general audio coder is determined by the following factors:

- **Frame length**
  For block-based processing, a certain amount of time has to pass to collect the samples belonging to one block
- **Filterbank delay**
  Use of an analysis-synthesis filterbank pair causes a certain amount of delay.
- **Look-ahead for block switching decision**
  Due to the underlying principles of the block switching scheme, the detection of transients has to use a certain amount of "look-ahead" in order to ensure that all transient signal parts are covered properly by short windows.
- **Use of bit reservoir**
  While the bit reservoir facilitates the use of a locally varying bitrate, this implies an additional delay depending on the size of the bit reservoir relative to the average bitrate per block.

The overall algorithmic delay can be calculated as

$$t_{delay} = \frac{N_{Frame} + N_{FB} + N_{look\_ahead} + N_{bitres}}{F_s}$$

where F_S is the coder sampling rate, N_{Frame} is the frame size, N_{FB} is the delay due to the filterbank (s), N_{look_ahead} corresponds to the look-ahead delay for block switching and N_{bitres} is the delay due to the use of the bit reservoir.

The basic idea of the low delay coder is to make use of the tools defined in 14496-3 as far as possible. The low delay codec is derived from the MPEG-4 AAC LTP object type, i.e. a coder consisting of the low complexity AAC codec plus the PNS (Perceptual Noise Substitution) and the LTP (Long Term Predictor) tools.

Figure 13 shows the overall structure of the decoder.

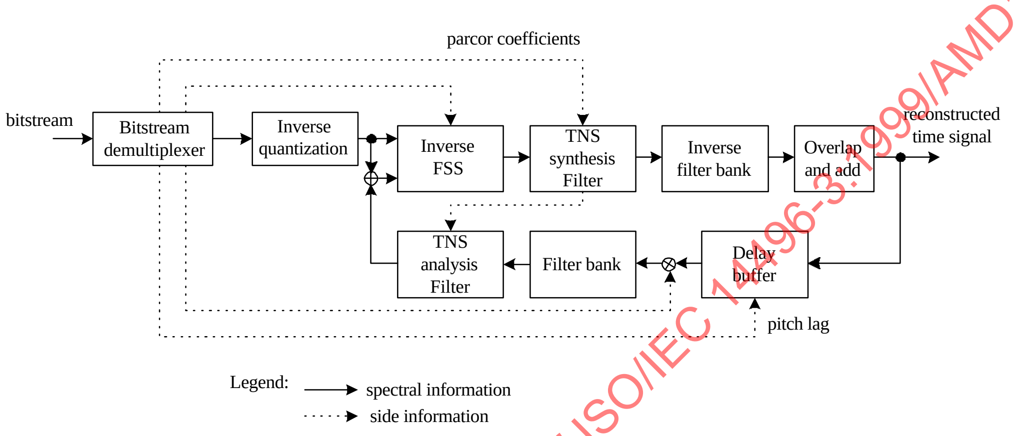


Figure 13: Decoder structure

Figure 14 shows the overall structure of the encoder.

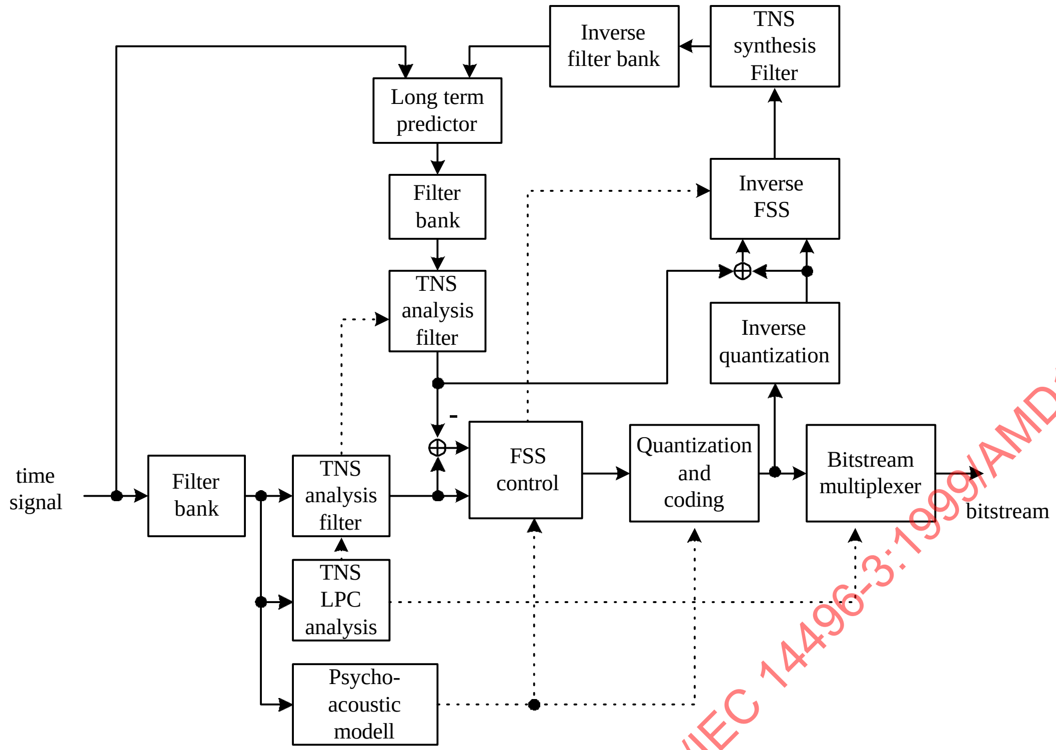


Figure 14: Encoder structure

8.4.2 Syntax

This part defines the syntax of the low delay codec.

8.4.2.1 Bitstream syntax

The bitstream syntax for the low delay GA codec is identical to the syntax used for the AAC LTP object type as defined in IS 14496-3 with the exception of ltp_data() which is modified as follows:

- The field for the LTP lag (ltp_lag) is reduced in size from 11 to 10 bits.
- A field is added (ltp_lag_update) allowing to keep the previous ltp_lag value.

Table 141: Syntax of ltp_data() for low delay codec

Syntax	No. of bits	Mnemonic
ltp_data()		
{		
ltp_lag_update;	**1**	**uimsbf**
if (ltp_lag_update) {		
ltp_lag;	**10**	**uimsbf**
} else {		
ltp_lag = ltp_prev_lag;		
}		
ltp_coef;	**3**	**uimsbf**
for (sfb = 0; sfb < max_sfb; sfb++) {		
ltp_long_used[sfb];	**1**	**uimsbf**
}		
}		

In order to retrieve the ltp_data in case of AAC_LD, the ics_info defined in IS 14496-3 must be extended to:

Table 142: Syntax of ics_info()

Syntax	No. of bits	Mnemonic
ics_info()		
{		
ics_reserved_bit;	**1**	**bslbf**
window_sequence;	**2**	**uimsbf**
window_shape;	**1**	**uimsbf**
if(window_sequence == EIGHT_SHORT_SEQUENCE) {		
max_sfb;	**4**	**uimsbf**
scale_factor_grouping;	**7**	**uimsbf**
}		
else {		
max_sfb;	**6**	**uimsbf**
predictor_data_present;	**1**	**uimsbf**
if (predictor_data_present) {		
if ((audioObjectType == 1) {		
predictor_reset;	**1**	**uimsbf**
if (predictor_reset) {		
predictor_reset_group_number;	**5**	**uimsbf**
}		
for (sfb = 0; sfb < min (max_sfb, PRED_SFB_MAX); sfb++) {		
prediction_used[sfb]**;**	**1**	**uimsbf**
}		
}		
else {		
ltp_data_present;	**1**	**uimsbf**
if (ltp_data_present) {		
ltp_data();		
}		
if (common_window) {		
ltp_data_present;	**1**	**uimsbf**
if (ltp_data_present) {		
ltp_data();		
}		
}		
}		
}		
}		
}		

8.4.2.2 Frame length in GA specific configuration

The *frameLengthFlag* is interpreted as follows:

Table 143: Frame length depending on frameLengthFlag

frameLengthFlag	frame length in samples
0x0	512 (instead of 1024)
0x1	480 (instead of 960)

8.4.3 General information

8.4.3.1 Definitions

(no additional definitions)

8.4.4 Coder description

The low delay codec is defined by the following modifications with respect to the standard algorithm (i.e. IS 14496-3 AAC LTP object) to achieve low delay operation:

8.4.4.1 Frame size/window length

The length of the analysis window is reduced to 1024 or 960 time domain samples corresponding to 512 and 480 spectral values, respectively. The latter choice enables the coder to have a frame size that is commensurate with widely used speech codecs (20 ms). The corresponding scalefactor band tables are shown below:

Table 144: Scalefactor bands for 44.1 and 48 kHz, N=480

fs [kHz]	44.1, 48
num_swb_long_window	35
swb	swb_offset_long_window
0	0
1	4
2	8
3	12
4	16
5	20
6	24
7	28
8	32
9	36
10	40
11	44
12	48
13	52
14	56
15	64
16	72
17	80

swb	swb_offset_long_window
18	88
19	96
20	108
21	120
22	132
23	144
24	156
25	172
26	188
27	212
28	240
29	272
30	304
31	336
32	368
33	400
34	432
	480

Table 145: Scalefactor bands for 44.1 and 48 kHz, N=512

fs [kHz]	44.1, 48
num_swb_long_window	36
swb	swb_offset_long_window
0	0
1	4
2	8
3	12
4	16
5	20
6	24
7	28
8	32
9	36
10	40
11	44
12	48
13	52
14	56
15	60
16	68
17	76
18	84
19	92
20	100
21	112
22	124
23	136
24	148
25	164
26	184
27	208
28	236
29	268
30	300
31	332
32	364
33	396
34	428
35	460
	512

Table 146: Scalefactor bands for 32 kHz, N=480

fs [kHz]	32
num_swb_long_window	37
Swb	swb_offset_long_window
0	0
1	4
2	8
3	12
4	16
5	20
6	24
7	28
8	32
9	36
10	40
11	44
12	48
13	52
14	56
15	60
16	64
17	72
18	80
19	88
20	96
21	104
22	112
23	124
24	136
25	148
26	164
27	180
28	200
29	224
30	256
31	288
32	320
33	352
34	384
35	416
36	448
	480

Table 147: Scalefactor bands for 32 kHz, N=512

fs [kHz]	32		
num_swb_long_ window	37		
Swb	swb_offset_long_ window	swb	swb_offset_long_ window
0	0	19	96
1	4	20	108
2	8	21	120
3	12	22	132
4	16	23	144
5	20	24	160
6	24	25	176
7	28	26	192
8	32	27	212
9	36	28	236
10	40	29	260
11	44	30	288
12	48	31	320
13	52	32	352
14	56	33	384
15	64	34	416
16	72	35	448
17	80	36	480
18	88		512

Table 148: Scalefactor bands for 22.05 and 24 kHz, N=480.

fs [kHz]	24, 22.05		
num_swb_long_ window	30		
swb	swb_offset_long_ window	swb	swb_offset_long_ window
0	0	16	92
1	4	17	104
2	8	18	120
3	12	19	140
4	16	20	164
5	20	21	192
6	24	22	224
7	28	23	256
8	32	24	288
9	36	25	320
10	40	26	352
11	44	27	384
12	52	28	416
13	60	29	448
14	68		480
15	80		

Table 149: Scalefactor bands for 22.05 and 24 kHz, N=512

fs [kHz]	24, 22.05		
num_swb_long_window	31		
swb	swb_offset_long_window	swb	swb_offset_long_window
0	0	16	92
1	4	17	104
2	8	18	120
3	12	19	140
4	16	20	164
5	20	21	192
6	24	22	224
7	28	23	256
8	32	24	288
9	36	25	320
10	40	26	352
11	44	27	384
12	52	28	416
13	60	29	448
14	68	30	480
15	80		512

8.4.4.2 Block switching

Due to the contribution of the look-ahead time to the overall delay, no block switching is used.

8.4.4.3 Window shape

As stated in the previous chapter, block switching is not used in the low delay coder to keep the delay as low as possible. As an alternative tool to improve coding of transient signals, the low delay coder uses the window shape switching feature with a slight modification compared to normal-delay AAC: The low delay coder still uses the sine window shape, but the Kaiser-Bessel derived window is replaced by a low-overlap window. As indicated by its name, this window has a rather low overlap with the following window, thus being optimized for the use of the TNS tool to prevent preecho artefacts in case of transient signals. For normal coding of non-transient signals the sine window is used because of its advantageous frequency response.

In line with normal-delay AAC, the window_shape indicates the shape of the trailing part (i.e. the second half) of the analysis window. The shape of the leading part (i.e. the first half) of the analysis window is identical to the window_shape of the last block.

Table 150: window depending on window_shape

window_shape	window
0x0	sine
0x1	low-overlap

The low-overlap window is defined by:

$$W(i)=\begin{cases}0 & i=0..3\cdot N/16-1\\ \sin\left[\dfrac{p(i-3\cdot N/16+0.5)}{N/4}\right] & i=3\cdot N/16..5\cdot N/16-1\\ 1 & i=5\cdot N/16..11\cdot N/16-1\\ \sin\left[\dfrac{p(i-9\cdot N/16+0.5)}{N/4}\right] & i=11\cdot N/16..13\cdot N/16-1\\ 0 & i=13\cdot N/16..N-1\end{cases}$$

with $N=1024$ or $N=960$.

8.4.4.4 Bit reservoir use

Use of the bit reservoir is minimized in order to reach the desired target delay. As one extreme case, no bit reservoir is used at all.

8.4.4.5 Tables for temporal noise shaping (TNS)

The following tables specify the value of TNS_MAX_BANDS for the low delay coder:

Table 151: TNS_MAX_BANDS in case of 480 samples per frame

Sampling Rate	TNS_MAX_BANDS
48000	31
44100	32
32000	37
24000	30
22050	30

Table 152: TNS_MAX_BANDS in case of 512 samples per frame

Sampling Rate	TNS_MAX_BANDS
48000	31
44100	32
32000	37
24000	31
22050	31

8.4.4.6 Further differences

Since the low delay codec is derived from the GA object, all used tools are defined already and the standard bitstream syntax is used. In addition, the following optimizations of the LTP tool apply:

- The size of the LTP delay buffer size is scaled down proportionally with the frame size. Thus, the size is 2048 and 1920 samples for frame sizes of N=512 and N=480, respectively.
- Accordingly, the field for the LTP lag (ltp_lag in ltp_data()) is reduced in size from 11 to 10 bits.

- Due to the high consistency of the LTP lag for many signals, one additional bit is introduced signaling that the lag of the previous frame is repeated (ltp_lag_update==0). Otherwise, a new value for the ltp lag is transmitted (ltp_lag_update==1).

8.4.4.7 Adaptation to systems using lower sampling rates

In certain applications it may be necessary to integrate the low delay decoder into an audio system running at lower sampling rates (e.g. 16 kHz) while the nomnial sampling rate of the bitstream is much higher (e.g. 48 kHz, corresponding to an algorithmic codec delay of approx. 20 ms). In such cases, it is favorable to decode the output of the low delay codec directly at the target sampling rate rather than using an additional sampling rate conversion operation after decoding.

This can be approximated by appropriate downscaling of both, the frame size and the sampling rate, by some integer factor (e.g. 2, 3), resulting in the same time/frequency resolution of the codec. For example, the codec output can be generated at 16 kHz sampling rate instead of the nominal 48 kHz by retaining only the lowest third (i.e. 480/3=160) of the spectral coefficients prior to the synthesis filterbank and reducing the inverse transform size to one third (i.e. window size 960/3=320).

As a consequence, decoding for lower sampling rates reduces both memory and computational requirements, but may not produce exactly the same output as a full-bandwidth decoding, followed by band limiting and sample rate conversion.

Please note that decoding at a lower sampling rate, as described above, does not affect the interpretation of levels, which refers to the nominal sampling rate of a low delay coder bitstream.

8.5 AAC Error resilience

8.5.1 Overview of tools

The virtual codebooks (VCB11) tool can extend the part of the bitstream demultiplexer that decodes the sectioning information. The VCB11 tool gives the opportunity to detect serious errors within the spectral data of an MPEG-4 AAC bitstream.

The input of the VCB11 tool is:

- The encoded section data using virtual codebooks

The output of the VCB11 tool is:

- The decoded sectioning information as described in ISO/IEC 14496-3, subpart 4 (GA)

The reversible variable length coding (RVLC) tool can replace the part of the noiseless coding tool that decodes the Huffman and DPCM coded scalefactors. The RVLC tool is used to increase the error resilience for the scalefactor data within an MPEG-4 AAC bitstream.

The input of the RVLC tool is:

- The noiselessly coded scalefactors using RVLC

The output of the RVLC tool is:

- The decoded integer representation of the scalefactors as described in ISO/IEC 14496-3, subpart 4 (GA)

The Huffman codeword reordering (HCR) tool can extend the part of the noiseless coding tool that decodes the Huffman coded spectral data. The HCR tool is used to increase the error resilience for the spectral data within an MPEG-4 AAC bitstream.

The input of the HCR tool is:

- The sectioning information for the noiselessly coded spectra as described in ISO/IEC 14496-3, subpart 4 (GA)
- The noiselessly coded spectral data in an error resilient reordered manner
- The length of the longest codeword within spectral_data
- The length of spectral_data

The output of the HCR tool is:

- The quantized value of the spectra as described in ISO/IEC 14496-3, subpart 4 (GA)

8.5.2 Bitstream payload

Table 153: Syntax of individual_channel_stream ()

Syntax	No. of bits	Mnemonic
individual_channel_stream (common_window, scale_flag)		
{		
global_gain;	**8**	**uimsbf**
if (! common_window && ! scale_flag) {		
ics_info ();		
}		
section_data ();		
scale_factor_data ();		
pulse_data_present;	**1**	**uimsbf**
if (pulse_data_present) {		
pulse_data ();		
}		
if (! scale_flag) {		
tns_data_present;	**1**	**uimsbf**
if (tns_data_present) {		
tns_data ();		
}		
gain_control_data_present;	**1**	**uimsbf**
if (gain_control_data_present) {		
gain_control_data ();		
}		
}		
if (! aacSpectralDataResilienceFlag) {		
spectral_data ();		
}		
else {		
length_of_reordered_spectral_data;	**14**	**uimsbf**
length_of_longest_codeword;	**6**	**uimsbf**
reordered_spectral_data ();		
}		
}		

Table 154: Syntax of section_data ()

```
Syntax                                                            No. of bits  Mnemonic
section_data()
{
    if ( window_sequence == EIGHT_SHORT_SEQUENCE ) {
        sect_esc_val = ( 1 << 3 ) – 1;
    }
    else {
        sect_esc_val = ( 1 << 5 ) – 1;
    }
    for ( g = 0; g < num_window_groups; g++ ) {
        k = 0;
        i = 0;
        while ( k < max_sfb ) {
            if ( aacSectionDataResilienceFlag )
                sect_cb[g][i];                                    5            uimsbf
            }
            else {
                sect_cb[g][i];                                    4            uimsbf
            }
            sect_len = 0;
            if ( ! aacSectionDataResilienceFlag ||
                sect_cb < 11 || ( sect_cb > 11 && sect_cb < 16 ) ) {
                while (sect_len_incr == sect_esc_val) {           3/5          uimsbf
                    sect_len += sect_esc_val;
                }
            }
            else {
                sect_len_incr = 1;
            }
            sect_len += sect_len_incr;
            sect_start[g][i] = k;
            sect_end[g][i] = k + sect_len;
            for ( sfb = k; sfb < k + sect_len; sfb++ ) {
                sfb_cb[g][sfb] = sect_cb[g][i];
            }
            k += sect_len;
            i++;
        }
        num_sec[g] = i;
    }
}
```

Table 155: Syntax of scalefactor_data ()

```
Syntax                                                            No. of bits  Mnemonic
scale_factor_data()
{
    if ( ! aacScalefactorDataResilienceFlag ) {
        noise_pcm_flag = 1;
        for ( g = 0; g < num_window_groups; g++ ) {
            for ( sfb = 0; sfb < max_sfb; sfb++ ) {
                if ( sect_cb[g][sfb] != ZERO_HCB ) {
                    if ( is_intensity ( g, sfb) ) {
                        hcod_sf[dpcm_is_position[g][sfb]];        1..19        vlclbf
                    }
                    else {
                        if ( is_noise(g, sfb) ) {
```

```
                        if ( noise_pcm_flag ) {
                            noise_pcm_flag = 0;
                            dpcm_noise_nrg[g][sfb];                 9       uimsbf
                        }
                        else {
                        hcod_sf[dpcm_noise_nrg[g][sfb]];            1..19   vlclbf
                        }
                    }
                    else {
                        hcod_sf[dpcm_sf[g][sfb]];                   1..19   vlclbf
                    }
                }
            }
        }
    }
}
else {
    intensity_used = 0;
    noise_used = 0;
    sf_concealment;                                                 1       uimsbf
    rev_global_gain;                                                8       uimsbf
    length_of_rvlc_sf;                                              11/9    uimsbf
    for ( g = 0; g < num_window_groups; g++ ) {
        for ( sfb=0; sfb < max_sfb; sfb++ ) {
            if ( sect_cb[g][sfb] != ZERO_HCB ) {
                if ( is_intensity ( g, sfb) ) {
                    intensity_used = 1;
                    rvlc_cod_sf[dpcm_is_position[g][sfb]];          1..9    vlclbf
                }
                else {
                    if ( is_noise(g,sfb) ) {
                        if ( ! noise_used ) {
                            noise_used = 1;
                            dpcm_noise_nrg[g][sfb];                 9       uimsbf
                        }
                        else {
                            rvlc_cod_sf[dpcm_noise_nrg[g][sfb]];    1..9    vlclbf
                        }
                    }
                    else {
                        rvlc_cod_sf[dpcm_sf[g][sfb]];               1..9    vlclbf
                    }
                }
            }
        }
    }
    if ( intensity_used ) {
        rvlc_cod_sf[dpcm_is_last_position];                         1..9    vlclbf
    }
    sf_escapes_present;                                             1       uimsbf
    if ( sf_escapes_present ) {
        length_of_rvlc_escapes;                                     8       uimsbf
        for ( g = 0; g < num_window_groups; g++ ) {
            for ( sfb = 0; sfb < max_sfb; sfb++ ) {
                if ( sect_cb[g][sfb] != ZERO_HCB ) {
                    if ( is_intensity ( g, sfb ) &&
                        dpcm_is_position[g][sfb] == ESC_FLAG ) {
                        rvlc_esc_sf[dpcm_is_position[g][sfb]];      2..20   vlclbf
                    }
                    else {
```

if (is_noise (g, sfb) &&		
dpcm_noise_nrg[g][sfb] == ESC_FLAG) {		
rvlc_esc_sf[dpcm_noise_nrg[g][sfb]];	**2..20**	**vlclbf**
}		
else {		
if (dpcm_sf[g][sfb] == ESC_FLAG) {		
rvlc_esc_sf[dpcm_sf[g][sfb]];	**2..20**	**vlclbf**
}		
}		
}		
}		
}		
}		
if (intensity_used &&		
dpcm_is_position[g][sfb] == ESC_FLAG) {		
rvlc_esc_sf[dpcm_is_last_position];	**2..20**	**vlclbf**
}		
}		
if (noise_used) {		
dpcm_noise_last_position;	**9**	**uimsbf**
}		
}		
}		

Table 156: Syntax of reordered_spectral_data ()

Syntax	No. of bits	Mnemonic
reordered_spectral_data ()		
{		
/* complex reordering, see tool description of Huffman codeword reordering */		
}		

8.5.3 Tool descriptions

8.5.3.1 Virtual codebooks for AAC section data

8.5.3.1.1 Tool description

Virtual codebooks are used to limit the largest absolute value permitted within a certain scale factor band where escape values are allowed, i. e. where codebook 11 is used originally. This tool allows 17 different codebook indices (11, 16...31) for the escape codebook. All these codebook indices refer to codebook 11. They are therefore called virtual codebooks. The difference between these codebook indices is the allowed maximum of spectral values belonging to the appropriate section. Due to this, errors within spectral data resulting in too large spectral values can be located and the according spectral lines can be concealed.

8.5.3.1.2 Decoding process

Within ISO/IEC 14496-3, subpart 4 (GA), section 5 (General Information), subclause 5.2 (Decoding of the GA bitstream payloads), sub-subclause 5.2.3 (Decoding of an individual_channel_stream (ICS) and ics_info), sub-sub-subclause 5.2.3.2 (Decoding process) has to be applied. The paragraph (Recovering section_data ()) needs to be extended as follows:

If the aacSectionDataResilienceFlag is set, **sect_len** is not transmitted but is set to one per default in case the codebook for a section is 11 or in the range of 16 and 31.

8.5.3.1.3 Tables

In section 6 (GA-Tool Descriptions), subclause 6.3 (Noiseless coding), sub-subclause 6.3.4 (Tables), table 6.2 (Spectrum Huffman codebooks parameters) needs to be extended as follows:

Table 157: Spectrum Huffman codebooks parameters

Codebook number, i	unsigned_cb[i]	Dimension of codebook	LAV for codebook	Codebook listed in Table
0	-	-	0	-
1	0	4	1	A.2
2	0	4	1	A.3
3	1	4	2	A.4
4	1	4	2	A.5
5	0	2	4	A.6
6	0	2	4	A.7
7	1	2	7	A.8
8	1	2	7	A.9
9	1	2	12	A.10
10	1	2	12	A.11
11	1	2	16 (with ESC 8191)	A.12
12	-	-	(reserved)	-
13	-	-	perceptual noise substitution	-
14	-	-	intensity out-of-phase	-
15	-	-	intensity in-phase	-
16	1	2	16 (w/o ESC 15)	A.12
17	1	2	16 (with ESC 31)	A.12
18	1	2	16 (with ESC 47)	A.12
19	1	2	16 (with ESC 63)	A.12
20	1	2	16 (with ESC 95)	A.12
21	1	2	16 (with ESC 127)	A.12
22	1	2	16 (with ESC 159)	A.12
23	1	2	16 (with ESC 191)	A.12
24	1	2	16 (with ESC 223)	A.12
25	1	2	16 (with ESC 255)	A.12
26	1	2	16 (with ESC 319)	A.12
27	1	2	16 (with ESC 383)	A.12
28	1	2	16 (with ESC 511)	A.12
29	1	2	16 (with ESC 767)	A.12
30	1	2	16 (with ESC 1023)	A.12
31	1	2	16 (with ESC 2047)	A.12

8.5.3.2 RVLC for AAC scalefactors

8.5.3.2.1 Tool description

RVLC (reversible variable length coding) is used instead of Huffman coding to achieve entropy coding of the scalefactors, because of its better performance in terms of error resilience. It can be considered to be a plug-in of the noiseless coding tool defined in ISO/IEC 14496-3, subpart 4 (GA), which allows decoding error resilient encoded scalefactor data.

RVLC enables additional backward decoding. Some error detection is possible in addition because not all nodes of the coding tree are used as codewords. The error resilience performance of the RVLC is as better as smaller the number of codewords. Therefore the RVLC table contains only values from -7 to +7, whereas the original Huffman codebook contains values from -60 to +60. 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.

It is necessary to transmit an additional value in order to have a starting point for backward decoding for the DPCM encoded scalefactors. This value is called reversible global gain. If intensity stereo coding or PNS is used, additional values are also necessary for them. The length of the RVLC bitstream part has to be troansmitted to allow backward decoding. Furthermore the length of the bitstream part containing the escape codewords should be transmitted to keep synchronization in case of bitstream errors.

8.5.3.2.2 Definitions

The following bitstream elements are available within the bitstream, if the GASpecificConfig enables the RVLC tool.

sf_concealment is a data field that signals whether the scalefactors of the last frame are similar to the current ones or not. The length of this data field is 1 bit.

rev_global_gain contains the last scalefactor value as a start value for the backward decoding. The length of this data field is 8 bits.

length_of_rvlc_sf is a data field that contains the length of the current RVLC data part in bits, including the DPCM start value for PNS. The length of this data field depends on window_sequence: If window_sequence == EIGHT_SHORT_SEQUENCE, the field consists of 11 bits, otherwise it consists of 9 bits.

rvlc_cod_sf RVLC word from the RVLC table used for coding of scalefactors, intensity positions or noise energy.

sf_escapes_present is a data field that signals whether there are escapes coded in the bitstream or not. The length of this data is 1 bit.

length_of_rvlc_escapes is a data field that contains the length of the current RVLC escape data part in bits. The length of this data is 8 bits.

rvlc_esc_sf Huffman codeword from the Huffman table for RVLC-ESC-values used for coding values larger than ±6.

dpcm_is_last_position DPCM value allowing backward decoding of intensity stereo data part. It is the symmetric value to dpcm_is_position.

dpcm_noise_last_position DPCM value allowing backward decoding of PNS data part. The length of this data is 9 bit. It is the symmetric value to dpcm_noise_nrg.

8.5.3.2.3 Decoding process

Within ISO/IEC 14496-3, subpart 4 (GA), section 6 (GA-Tool Descriptions), subclause 6.2 (Scalefactors), subclause 6.2.3 (Decoding process), subclause 6.2.3.2 (Decoding of scalefactors) has to be applied. The following paragraphs have to be added:

In case of error resilient scalefactor coding, a RVLC has been used instead of a Huffman code. The decoding process of the RVLC words is the same as for the Huffman codewords; just another codebook has to be used. This codebook uses symmetric codewords. Due to this it is possible to detect errors, because asymmetric codewords are illegal. Furthermore, decoding can be started at both sides. To allow backward decoding, an additional value is available within the bitstream, which contains the last scalefactor value. In case of intensity an additional codeword is available, which allows backwards decoding. In case of PNS an additional DPCM value is available for the same reason.

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.

8.5.3.2.4 Tables

Table 158: RVLC codebook

index	length	codeword
-7	7	65
-6	9	257
-5	8	129
-4	6	33
-3	5	17
-2	4	9
-1	3	5
0	1	0
1	3	7
2	5	27
3	6	51
4	7	107
5	8	195
6	9	427
7	7	99

Table 159: Asymmetric (forbidden) codewords

length	codeword
6	50
7	96
9	256
8	194
7	98
6	52
9	426
8	212

Table 160: Huffman codebook for RVLC escape values

index	length	codeword	index	length	codeword
0	2	2	27	20	473482
1	2	0	28	20	473483
2	3	6	29	20	473484
3	3	2	30	20	473485
4	4	14	31	20	473486
5	5	31	32	20	473487
6	5	15	33	20	473488
7	5	13	34	20	473489
8	6	61	35	20	473490
9	6	29	36	20	473491
10	6	25	37	20	473492
11	6	24	38	20	473493
12	7	120	39	20	473494
13	7	56	40	20	473495
14	8	242	41	20	473496
15	8	114	42	20	473497
16	9	486	43	20	473498
17	9	230	44	20	473499
18	10	974	45	20	473500
19	10	463	46	20	473501
20	11	1950	47	20	473502
21	11	1951	48	20	473503
22	11	925	49	19	236736
23	12	1848	50	19	236737
24	14	7399	51	19	236738
25	13	3698	52	19	236739
26	15	14797	53	19	236740

8.5.3.3 Huffman Codeword Reordering for AAC spectral data

8.5.3.3.1 Tool description

The Huffman codeword reordering (HCR) algorithm for AAC spectral data is based on the fact that some of the codewords can be placed at known positions so that these codewords can be decoded independent of any error within other codewords. Therefore, this algorithm avoids error propagation to those codewords, the so-called priority codewords (PCW).. To achieve this, segments of known length are defined and those codewords are placed at the beginning of these segments.

The remaining codewords (non-priority codewords, non-PCW) are filled into the gaps left by the PCWs using a special algorithm that minimizes error propagation to the non-PCWs codewords.

This reordering algorithm does not increase the size of spectral data.

Before applying the reordering algorithm itself, a pre-sorting process is applied to the codewords. It sorts all codewords depending on their importance, i. e. it determines the PCWs.

8.5.3.3.2 Definitions

The following bitstream elements are available within the bitstream, if the GASpecificConfig enables the HCR tool.

length_of_reordered_spectral_data is a 14-bit data field that contains the length of spectral data in bits. The maximum value is 6144 in case of a single_channel_element, a coupling_channel_element and a lfe_channel_element and 12288 in case of a channel_pair_element. Larger values are reserved for future use. If those values occur, current decoders have to replace them by the valid maximum value.

length_of_longest_codeword is a 6-bit data field that contains the length of the longest codeword available within the current spectral data in bits. This field is used to decrease the distance between protected codewords. Valid values are between 0 and 49. Values between 50 and 63 are reserved for future use. If those values occur, current decoders have to replace them by 49.

8.5.3.3.3 Bitstream structure

8.5.3.3.3.1 Pre-sorting

Within ISO/IEC 14496-3, subpart 4 (GA), clause 5 (General Information), subclause 5.2 (Decoding of the GA Bitstream Payloads), subclause 5.2.3 (Decoding of an individual_channel_stream (ICS) and ics_info), subclause 5.2.3.5 (Order of spectral coefficients in spectral data), is not valid if this tool is used. Instead, the procedure described in the following paragraphs has to be applied:

For explanation of the pre-sorting steps the term “unit” is introduced. A unit covers four spectral lines, i. e. two two-dimensional codewords or one four-dimensional codeword.

In case of one long window (1024 spectral lines per long block, one long block per frame), each window contains 256 units.

In case of eight short windows (128 spectral lines per short block, eight short blocks per frame), each window contains 32 units.

First pre-sorting step:

Units representing the same part of the spectrum are collected together in temporal order and denoted as unit group. In case of one long window, each unit group contains one unit. In case of eight short windows, each unit group contains eight units.

Unit groups are collected ascending in spectral direction. For one long window, that gives the original codeword order, but for eight short windows a unit based window interleaving has been applied.

Using this scheme, the codewords representing the lowest frequencies are the first codewords within spectral data for both, long and short blocks.

Table 161 shows an example output of the first pre-sorting step for short blocks, assuming two-dimensional codebooks for window 0, 1, 6, and 7 and four-dimensional codebooks for window 2, 3, 4, and 5.

Second pre-sorting step:

The more energy a spectral line contains, the more audible is its distortion. The energy within spectral lines is related to the used codebook. Codebooks with low numbers can represent only low values and allow only small errors, while codebooks with high numbers can represent high values and allow large errors.

Therefore, the codewords are pre-sorted depending on the used codebook. If the error resilient section data is used, the order is 11, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 9/10, 7/8, 5/6, 3/4, 1/2. If the normal section data is used, the order is 11, 9/10, 7/8, 5/6, 3/4, 1/2. This order is based on the largest absolute value of the tables. This second pre-sorting step is done on the described unit by unit base used in the first pre-sorting step. The output of the first pre-sorting step is scanned in a consecutive way for each codebook.

Assigning numbers to units according the following metric can perform these two pre-sorting steps:

codebookPriority[32] = {x,0,0,1,1,2,2,3,3,4,4,21,x,x,x,x,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20}

assignedUnitNr = (codebookPriority[cb] * maxNrOfLinesInWindow
+ nrOfFirstLineInUnit) * MaxNrOfWindows + window

with:

codebookPriority[cb]	codebook priority according the second pre-sorting step.
maxNrOfLinesInWindow	constant number: 1024 in case of one long window and 128 in case of eight short windows
nrOfFirstLineInUnit	a number between 0 and 1020 in case of one long window and between 0 and 124 in case of eight short windows (this number is always a multiple of four)
maxNrOfWindows	constant number: 1 in case of one long window and 8 in case of eight short windows
window	always 0 in case of one long window, a number between 0 and 7 in case of eight short windows

and sort the units in ascending order using these assigned unit numbers.

Encoder note: In order to reduce audible artifacts in case of errors within spectral data it is strongly recommended to use codebook 11 only if necessary!

8.5.3.3.3.2 Derivation of segment width

The segment widths depend on the Huffman codebook used. They are derived as the minimums of the (codebook dependent) maximum codeword length and the transmitted longest codeword length:

```
segmentWidth = min ( maxCwLen, length_of_longest_codeword )
```

Table 162 shows the values of maxCwLen depending on the Huffman codebook.

8.5.3.3.3.3 Order of Huffman codewords in spectral data

Figure 15 shows the general scheme of the segmentation and the arrangement of the PCWs. In this Figure 15, five segments can be provided to protect codewords from section 0 and section 1 against error propagation. Segment widths are different, because the length of the longest possible codeword depends on the current codebook.

The writing scheme for the non-PCWs is as follows (PCWs have been written already):

The proposed scheme introduces the term set. A set contains a certain number of codewords. Assuming N is the number of segments, all sets except the last one contain N codewords. Non-PCWs are written consecutively into these sets. Due to the pre-sorting algorithm set one contains the most important non-PCWs. The importance of the codewords stored within a set is the smaller the higher the set number.

Sets are written consecutively. Writing of a set might need several trials.

First trial: The first codeword of the current set is written into the remaining part of the first segment, the second codeword into the remaining part of the second segment and so on. The last codeword of the current set is written into the remaining part of the last segment.

Second trial: The remaining part of the first codeword (if any) is written into the remaining part of the second segment, the remaining part of the second codeword (if any) into the remaining part of the third segment and so on. The remaining part of the last codeword (if any) is written into the remaining part of the first segment (modulo shift).

If a codeword does not fit into the remaining part of a segment, it is only partly written and its remaining part is stored. At least after a maximum of N trials all codewords are completely written into segments.

If one set was written completely, writing of the next set starts. To improve the error propagation behavior between consecutive sets, the writing direction within segments changes from set to set. While PCWs are written from left to right, codewords of set one are written from right to left, codewords of set two are again written from left to right and so on.

8.5.3.3.3.4 Encoding process

The structure of the reordered spectral data cannot be described within the C like syntax commonly used. Therefore, Figure 16 shows an example and the following c-like description is provided:


```
/* helper functions */
void InitReordering(void);
  /* Initializes variables used by the reordering functions like the segment
     widths and the used offsets in segments and codewords. */

void InitRemainingBitsInCodeword(void);
  /* Initializes remainingBitsInCodeword[] array for each codeword with
     the total size of the codeword. */

int  WriteCodewordToSegment(codewordNr, segmentNr, direction);
  /* Writes a codeword or only a part of a codeword indexed by codewordNr
     to the segment indexed by segmentNr with a given direction.
     Write offsets for each segment are handled internally.
     The function returns the number of bits written to the segment.
     This number may be lower than the codeword length.
     WriteCodewordToSegment handles already written parts of the codeword
     internally. */

void ToggleWriteDirection(void);
  /* Toggles the write direction in the segments between forward and backward. */

/* (input) variables */
numberOfCodewords;  /* 15 in the example */
numberOfSegments;   /*  6 in the example */
numberOfSets;       /*  3 in the example */

ReorderSpectralData()
{
  InitReordering();
  InitRemainingBitsInCodeword();

  /* first step: write PCWs (set 0) */
  writeDirection = forward;
  for (codeword = 0; codeword < numberOfSegments; codeword ++) {
    WriteCodewordToSegment(codeword, codeword, writeDirection);
  }

  /* second step: write nonPCWs */
  for (set = 1; set < numberOfSets; set++) {
    ToggleWriteDirection();
    for (trial = 0; trial < numberOfSegments; trial++) {
      for (codewordBase = 0; codewordBase < numberOfSegments; codewordBase++) {
        segment  = (trial + codewordBase) % numberOfSegments;
        codeword = codewordBase + set*numberOfSegments;

        if (remainingBitsInCodeword[codeword] > 0) {
          remainingBitsInCodeword[codeword] -= WriteCodewordToSegment(codeword,
                                                                      segment,
                                                                      writeDirection);
        }
      }
    }
  }
}
```

8.5.3.3.4 Decoding process

Within ISO/IEC 14496-3, subpart 4 (GA), section 5 (General Information), subclause 5.2 (Decoding of the GA Bitstream Payloads), subclause 5.2.3 (Decoding of an individual_channel_stream (ICS) and ics_info), subclause 5.2.3.2 (Decoding process), has to be applied. The paragraph (Decoding an individual_channel_stream (ICS)) needs to be extended as follows:

In the individual_channel_stream, the order of decoding is:

- get global_gain
- get ics_info (parse bitstream if common information is not present)
- get section_data, if present
- get scalefactor_data, if present
- get pulse_data, if present
- get tns_data, if present
- get gain control data, if present
- get length_of_reordered_spectral_data, if present
- get length_of_longest_codeword, if present
- get reordered_spectral_data, if present.

Within ISO/IEC 14496-3, subpart 4 (GA), section 5 (General Information), subclause 5.2 (Decoding of the GA Bitstream Payloads), subclause 5.2.3 (Decoding of an individual_channel_stream (ICS) and ics_info), subclause 5.2.3.2 (Decoding process) has to be applied. The paragraph (spectral_data () parsing and decoding) needs to be extended as follows:

If the HCR tool is used, spectral data does not consist of consecutive codewords anymore. Concerning HCR, the whole data necessary to decode two or four lines are referred as codeword. This includes Huffman codeword, sign bits, and escape sequences.

Within ISO/IEC 14496-3, subpart 4 (GA), section 6 (GA-Tool Descriptions), subclause 6.3 (Noiseless coding), sub-subclause 6.3.3 (Decoding process) has to be applied. The following paragraphs have to be added:

Decoding of reordered spectral data cannot be done straightforward. The following c-like description shows the decoding process:

```
/* helper functions */
void InitReordering(void);
  /* Initializes variables used by the reordering functions like the segment
     widths and the used offsets in segments and codewords */

void InitRemainingBitsInSegment(void);
  /* Initializes remainingBitsInSegment[] array for each segment with the
     total size of the segment */

int  DecodeCodeword(codewordNr, segmentNr, direction);
  /* Try to decode the codeword indexed by codewordNr using data already read
     for this codeword and using data from the segment index by segmentNr.
     The read direction in the segment is given by direction.
     DecodeCodeword returns the number of bits read from the indexed segment. */

void MoveFromSegmentToCodeword(codewordNr, segmentNr, bitLen, direction);
  /* Move bitLen bits from the segment indexed by segmentNr to the codeword
```

```
     indexed by codewordNr using direction as read direction in the segment.
     The bits are appended to existing bits for the codeword and the codeword
     length is adjusted. */

void AdjustOffsetsInSegment(segmentNr, bitLen, direction);
  /* Like MoveFromSegmentToCodeword(), but no bits are moved. Only the offsets
     for the segment indexed by segmentNr are adjusted according bitLen and
     direction. */

void MarkCodewordAsDecoded(codewordNr);
  /* Marks the codeword indexed by codewordNr as decoded. */

bool CodewordIsNotDecoded(codewordNr);
  /* Returns TRUE if the codeword indexed by codewordNr is not decoded. */

void ToggleReadDirection(void);
  /* Toggles the read direction in the segments between forward and backward. */

/* (input) variables */
numberOfCodewords;
numberOfSegments;
numberOfSets;

DecodeReorderedSpectralData()
{
  InitReordering();
  InitRemainingBitsInSegment();

  /* first step: decode PCWs (set 0) */
  readDirection = forward;
  for (codeword = 0; codeword < numberOfSegments; codeword++) {
    cwLen = DecodeCodeword(codeword, codeword, readDirection);
    if (cwLen <= remainingBitsInSegment[codeword]) {
      AdjustOffsetsInSegment(codeword, cwLen, readDirection);
      MarkCodewordAsDecoded(codeword);
      remainingBitsInSegment[codeword] -= cwLen;
    }
    else {
      /* error !!! (PCWs do always fit into segments) */
    }
  }

  /* second step: decode nonPCWs */
  for (set = 1; set < numberOfSets; set++) {
    ToggleReadDirection();
    for (trial = 0; trial < numberOfSegments; trial++) {
      for (codewordBase = 0; codewordBase < numberOfSegments; codewordBase++) {
        segment  = (trial + codewordBase) % numberOfSegments;
        codeword = codewordBase + set*numberOfSegments;

        if (CodewordIsNotDecoded(codeword) &&
            (remainingBitsInSegment[segment] > 0)) {
          cwLenInSegment = DecodeCodeword(codeword, segment, readDirection);
          if (cwLenInSegment <= remainingBitsInSegment[segment]) {
            AdjustOffsetsInSegment(segment, cwLenInSegment, readDirection);
            MarkCodewordAsDecoded(codeword);
            remainingBitsInSegment[segment] -= cwLenInSegment;
          }
          else {  /* only part of codeword in segment */
            MoveFromSegmentToCodeword(codeword,
                                      segment,
                                      remainingBitsInSegment[segment],
                                      readDirection);
            remainingBitsInSegment[segment] = 0;
          }
        }
      }
    }
  }
```

```
  }
}
```

8.5.3.3.5 Tables

Table 161: Example output of the first pre-sorting step for short blocks, assuming two-dimensional codebooks for window 0, 1, 6, and 7 and four-dimensional codebooks for window 2, 3, 4, and 5

index	codeword entry	
	window	window index
0	0	0
1	0	1
2	1	0
3	1	1
4	2	0
5	3	0
6	4	0
7	5	0
8	6	0
9	6	1
10	7	0
11	7	1
12	0	2
13	0	3
14	1	2
15	1	3
16	2	1
17	3	1
18	4	1
19	5	1
20	6	2
21	6	3
22	7	2
23	7	3
…	…	…

Table 162: Values of maxCwLen depending on the Huffman codebook

codebook	maximum codeword length (maxCwLen)
0	0
1	11
2	9
3	20
4	16
5	13
6	11
7	14
8	12
9	17
10	14
11	49
16	14
17	17
18	21
19	21
20	25
21	25
22	29
23	29
24	29
25	29
26	33
27	33
28	33
29	37
30	37
31	41

8.5.3.3.6 Figures

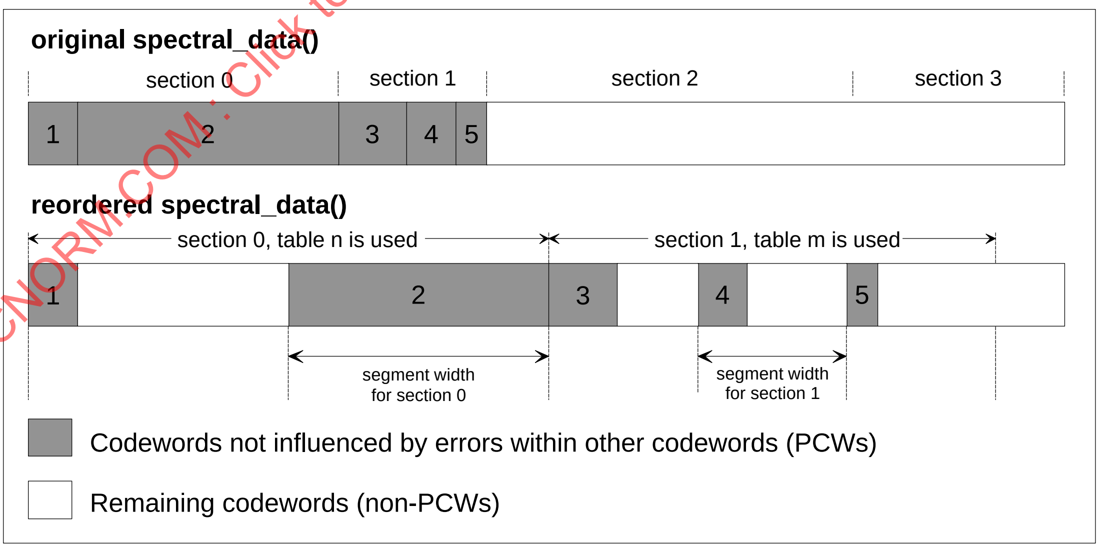


Figure 15: General scheme of segmentation and arrangement of PCWs

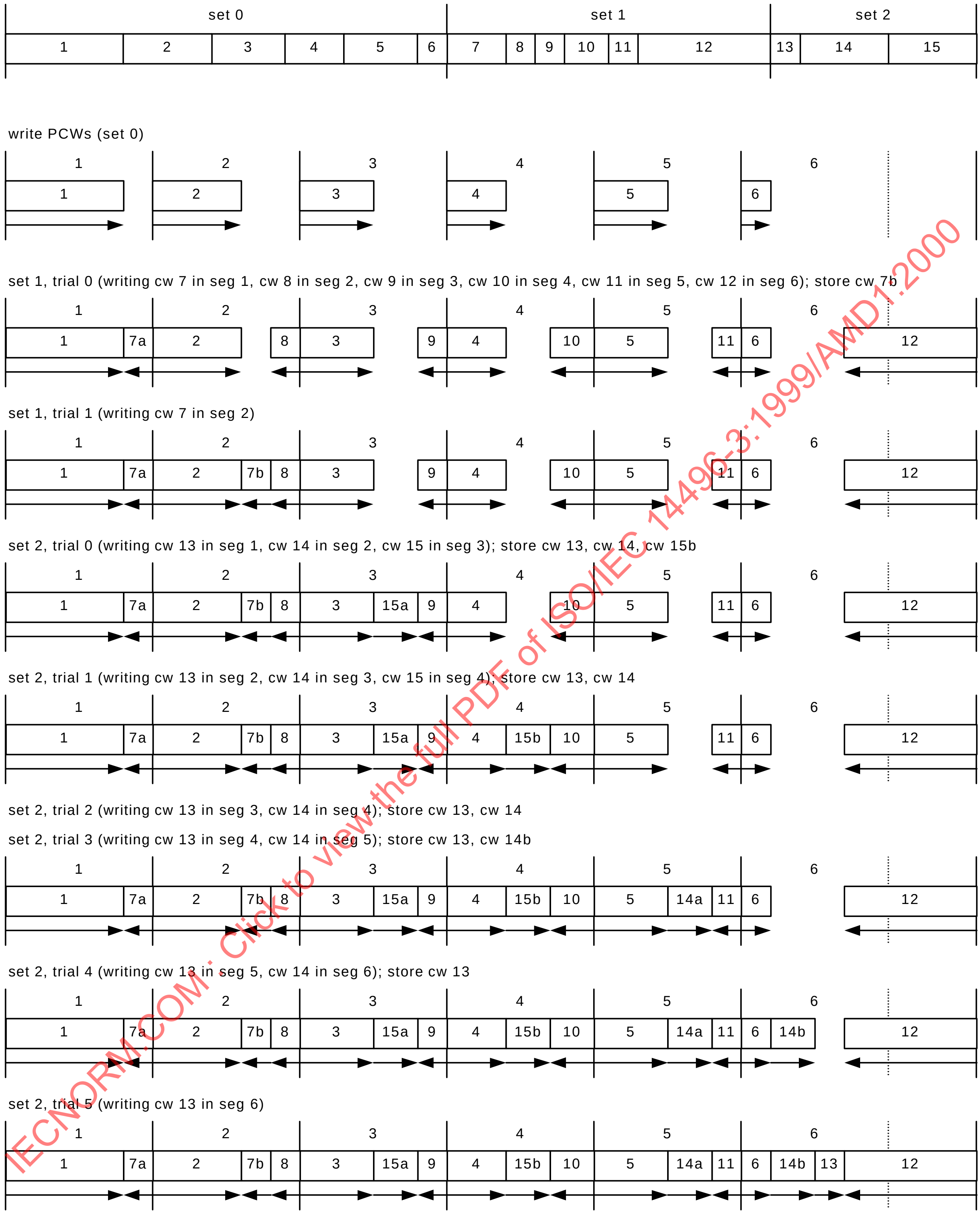


Figure 16: Example for HCR encoding algorithm (only one segment width, pre-sorting has been done before)

9 Error protection

9.1 Overview of the tools

The error protection tool (EP tool) provides the unequal error protection (UEP) capability to the ISO/IEC 14496-3 codecs. The main features of EP tool are follows:

- providing a set of error correcting/detecting codes with wide and small-step scalability, in performance and in redundancy
- providing a generic and bandwidth-efficient error protection framework, which covers both fixed-length frame bitstreams and variable-length frame bitstreams
- providing a UEP configuration control with low overhead

The stream type ERROR_PROTECTION_STREAM is defined. This stream consists of error protection frames.

The basic idea of UEP is to divide the frame into sub-frames according to the bit error sensitivities (these sub-frames are referred to be as classes in the following subclauses), and to protect these sub-frames with appropriate strength of FEC and/or CRC. If this would not be done, the decoded audio quality is determined by how the most error sensitive part is corrupted, and thus the strongest FEC/CRC has to be applied to the whole frame, requiring much more redundancy.

In order to apply UEP to audio frames, the following information is required:

1. Number of classes
2. Number of bits each class contains
3. The CRC code to be applied for each class, which can be presented as a number of CRC bits
4. The FEC code to be applied for each class

This information is called as “frame configuration parameters” in the following sections. The same information is used to decode the UEP encoded frames; thus they have to be transmitted. To transmit them effectively, the frame structures of MPEG-4 audio algorithms have been taken into account for this EP tool.

The MPEG-4 audio frame structure can be categorized into three different approaches from the viewpoint of UEP application:

1. All the frame configurations are constant while the transmission (as CELP).
2. The frame configurations are restricted to be one of the several patterns (as Twin-VQ).
3. Most of the parameters are constant during the transmission, but some parameters can be different frame by frame (as AAC).

To utilize these characteristics, the EP tool uses two paths to transmit the frame configuration parameters. One is the out-of-band signaling, which is the same way as the transmission of codec configuration parameters. The parameters that are shared by the frames are transmitted through this path. In case there are several patterns of configuration, all these patterns are transmitted with indices. The other is the in-band transmission, which is made by defining the EP-frame structure with a header. Only the parameters that are not transmitted out-of-band are transmitted through this path. With this parameter transmission technique, the amount of in-band information, which is a part of the redundancy caused by the EP tool, is minimized.

With these parameters, each class is FEC/CRC encoded and decoded. To enhance the performance of this error protection, an interleaving technique is adopted. The objective of interleaving is to randomize burst errors within the frames, and this is not desirable for the class that is not protected. This is because there are other error resilience tools whose objective is to localize the effect of the errors, and randomization of errors with interleaving would have a harmful influence on such part of bitstream.

The outline of the EP encoder and EP decoder is figured out in Figure 17 and Figure 18.

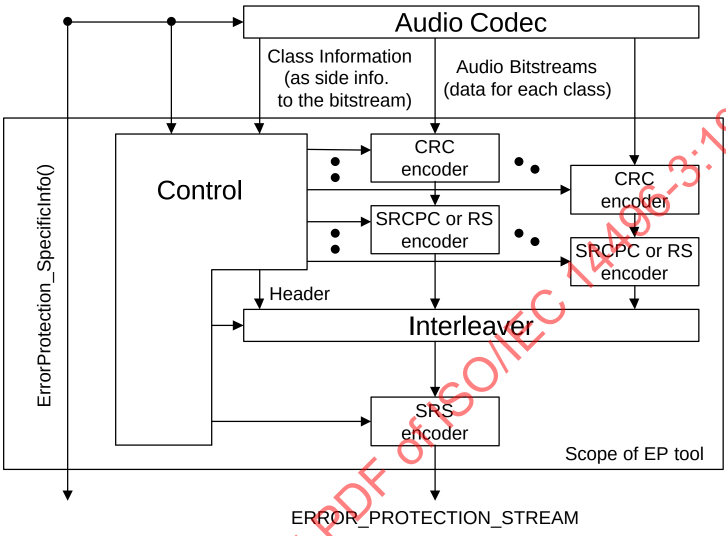


Figure 17: Outline of EP encoder

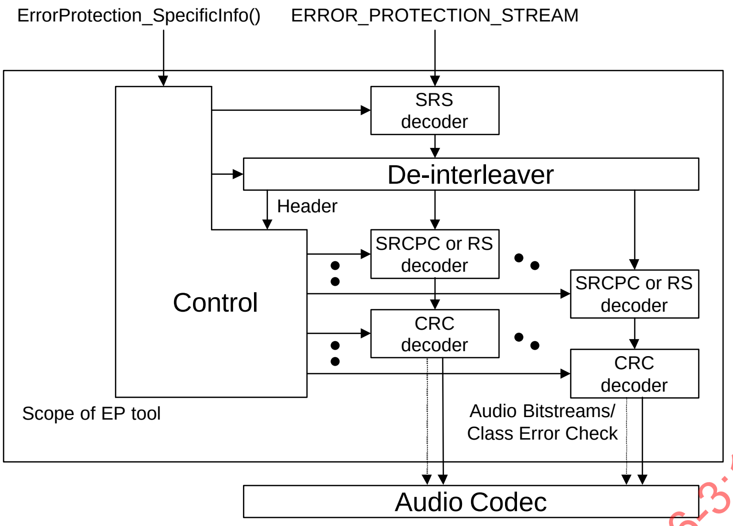


Figure 18: Outline of EP decoder

9.2 Syntax

9.2.1 Error protection specific configuration

This part defines the syntax of the specific configuration for error protection.

Table 163: Syntax of ErrorProtectionSpecificConfig()

Syntax	No. of bits	Mnemonic
ErrorProtectionSpecificConfig()		
{		
number_of_predefined_set;	**8**	**uimsbf**
interleave_type;	**2**	**uimsbf**
bit_stuffing;	**3**	**uimsbf**
number_of_concatenated_frame;	**3**	**uimsbf**
for (i = 0; i < number_of_predefined_set; i++) {		
number_of_class[i];	**6**	**uimsbf**
for (j = 0; j < number_of_class[i]; j++) {		
length_escape[i][j];	**1**	**uimsbf**
rate_escape[i][j];	**1**	**uimsbf**
crclen_escape[i][j];	**1**	**uimsbf**
if (number_of_concatenated_frame != 1) {		
concatenate_flag[i][j];	**1**	**uimsbf**
}		
fec_type[i][j];	**2**	**uimsbf**
if(fec_type[i][j] == 0) {		
termination_switch[i][j];	**1**	**uimsbf**
}		
if (interleave_type == 2) {		
interleave_switch[i][j];	**2**	**uimsbf**
}		
class_optional;	**1**	**uimsbf**
if (length_escape[i][j] == 1) { /* ESC */		
number_of_bits_for_length[i][j];	**4**	**uimsbf**
}		
else {		
class_length[i][j];	**16**	**uimsbf**
}		
if (rate_escape[i][j] != 1) { /* not ESC */		
class_rate[i][j];	**5**	**uimsbf**
}		
if (crclen_escape[i][j] != 1) { /* not ESC */		
class_crclen[i][j];	**5**	**uimsbf**
}		
}		
class_reordered_output;	**1**	**uimsbf**
if (class_reordered_output == 1) {		
for (j = 0; j < number_of_class[i]; j++) {		
class_output_order[i][j];	**6**	**uimsbf**
}		
}		
}		
header_protection;	**1**	**uimsbf**
if (header_protection == 1) {		
header_rate;	**5**	**uimsbf**
header_crclen;	**5**	**uimsbf**
}		
rs_fec_capability;	**7**	**uimsbf**
}		

9.2.2 Error protection bitstream payloads

This part defines the syntax of the stream type ERROR_PROTECTION_STREAM which consists of error protection frames. Note that this stream is common for all algorithms. In case this stream is transported by 14496-1 Systems, one rs_ep_frame() is mapped to one access unit.

Table 164: Syntax of rs_ep_frame ()

Syntax	No. of bits	Mnemonic
rs_ep frame()		
{		
ep_frame();		
rs_parity_bits;	**Nrsparity**	**bslbf**
}		

Nrsparity: see subclause 9.4.7

Table 165: Syntax of ep_frame ()

Syntax	No. of bits	Mnemonic
ep_frame()		
{		
if (interleave_type == 0){		
ep_header();		
ep_encoded_classes();		
}		
if (interleave_type == 1){		
interleaved_frame_mode1;	**1 -**	**bslbf**
}		
if (interleave_type == 2){		
interleaved_frame_mode2;	**1 -**	**bslbf**
}		
stuffing_bits;	**Nstuff**	**bslbf**
}		

Table 166: Syntax of ep_header ()

Syntax	No. of bits	Mnemonic
ep_header()		
{		
choice_of_pred;	$\mathbf{N_{pred}}$	**uimsbf**
choice_of_pred_parity;	$\mathbf{N_{pred\_parity}}$	**bslbf**
class_attrib();		
class_attrib_parity;	$\mathbf{N_{attrib\_parity}}$	**bslbf**
}		

Npred: the smallest integer value greater than log2 (# of pre-defined set).
Npred_parity: See subclause 9.4.3
Nattrib_parity: See subclause 9.4.3

Table 167: Syntax of class_attrib ()

Syntax	No. of bits	Mnemonic
class_attrib(class_count, length_escape, rate_escape, crclen_escape, frame_pred)		
{		
for(j=0; j<class_count; j++){		
if (length_escape[frame_pred][j] == 1){		
class_bit_count[j];	**Nbitcount**	**uimsbf**
}		
if (rate_escape[frame_pred][j] == 1){		
class_code_rate[j];	**3**	**uimsbf**
}		
if (crclen_escape[frame_pred][j] == 1){		
class_crc_count[j];	**3**	**uimsbf**
}		
}		
if (bit_stuffing == 1){		
num_stuffing_bits;	**3**	**uimsbf**
}		
}		

class_count: number of class for this frame
frame_pred: selected predefined set for this frame

Table 168: Syntax of ep_encoded_classes ()

Syntax	No. of bits	Mnemonic
ep_encoded_classes(class_count)		
{		
for(j=0; j<class_count; j++){		
ep_encoded_class[j];		**bslbf**
}		
}		

9.3 General information

9.3.1 Definitions

ErrorProtectionSpecificConfig (): Error protection specific configuration that is out-of-band information.

number_of_predefined_set The number of pre-defined set.

interleave_type This variable defines the interleave type. (interleave_type == 0) means no interleaving, (interleave_type == 1) means intra-frame interleaving and (interleave_type == 2) enables interleaving fine tuning for each class. For details see subclause 9.4.8. (interleave_type==3) is reserved.

bit_stuffing Signals whether the bit stuffing to ensure the byte alignment is used with the in-band information or not:
1 indicates the bit stuffing is used.
0 indicates the bit stuffing is not used. This implies that the configuration provided with the out-of-band information ensure the EP-frame is byte-aligned.

number_of_concatenated_frame The number of concatenated source coder frames for the constitution of one error protected frame.

Table 169: concatenated frames depending on number_of_concatenated_frame

Codeword	000	001	010	011	100	101	110	111
number of concatenated frame	reserved	1	2	3	4	5	6	7

number_of_class[i] The number of classes for i-th pre-defined set.

length_escape[i][j] If 0, the length of j-th class in i-th pre-defined set is fixed value. If 1, the length is variable. Note that in case "until the end", this value should be 1, and the **number_of_bits_for_length[i][j]** value should be 0.

rate_escape[i][j] If 0, the SRCPC code rate of j-th class in i-th pre-defined set is fixed value. If 1, the code rate is signaled in-band.

crclen_escape[i][j] If 0, the CRC length of j-th class in i-th pre-defined set is fixed value. If 1, the CRC length is signaled in-band.

concatenate_flag[i][j] This parameter defines whether j-th class of i-th pre-defined set is concatenated or not. 0 indicates "not concatenated" and 1 indicates "concatenated". (See subclause 9.4.4)

fec_type[i][j] This parameter defines whether SRCPC code ("0") or RS code ("1" or "2") are used to protect the j-th class of i-th pre-defined set. Note that the class length which is signaled to be protected by RS code shall be byte aligned, in either case that the length is signaled in the out-of-band information or that the length is signaled as in-band information. If this field is set to "2", it indicates that this class is RS encoded in conjunction with next class as one RS code. Note that more than two succeeding classes have the value "2" for this field, it means these classes are concatenated and RS encoded as one RS code. If this field is "1", it indicates that this class is not concatenated with next class. This means this class is the last class to be concatenated before RS encoding, or this class is RS encoded independently.

termination_switch[i][j] This parameter defines whether j-th class of i-th pre-defined set is terminated or not when it is SRCPC encoded. See subclause 9.4.6.2.

interleave_switch[i][j] This parameter defines how to interleave j-th class of i-th pre-defined set.
0 – not interleaved
1 – interleaved without intra-interleaving
2 – interleaved with intra-interleaving
3 – concatenated
(see subclause 9.4.8.2.2)

class_optional This flag signals, whether the class is mandatory (class_optional == 0) or optional (class_optional == 1). This flag can be used to reduce the redundancy within ErrorProtectionSpecificConfig. Usually it would be necessary to define 2^N predefinition sets, where N equals the number of optional classes.(See subclause 9.4.2)

number_of_bits_for_length[i][j] This field exists only when the **length_escape[i][j]** is 1. This value shows the number of bits for the class length in-band signaling. This value should be set considering possible maximum length of the class.

class_length[i][j] This field exists only when the **length_escape[i][j]** is 0. This value shows the length of the j-th class in i-th pre-defined set, which is the fixed value while the transmission.

class_rate[i][j] This field exists only when the **rate_escape[i][j]** is 0. In case **fec_type[i][j]** is 0, this value shows the SRCPC code rate of the j-th class in i-th pre-defined set, which is the fixed value while the transmission. The value from 0 to 24 corresponds

	to the code rate from 8/8 to 8/32, respectively. In case **fec_type[i][j]** is 1 or 2, this value shows the number of erroneous bytes which can be corrected by RS code (see subclause 9.4.7). All the classes which is signaled to be concatenated with **fec_type[i][j]** shall have the same value of **class_rate[i][j]**.
class_crclen[i][j]	This field exists only when the **crclen_escape[i][j]** is 0. This value shows the CRC length of the j-th class in i-th pre-defined set, which is the fixed value while the transmission. The value should be 0 – 18, which represents CRC length 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 24 or 32. (See subclause 9.4.8.2.2)
class_reordered_output	If this value is "1", the classes output from ep decoder is re-ordered. If "0", no such processing is made. See subclause 9.4.9.
class_output_order[i][j]	This field only exists when class_reordered_output is set to "1", to signal the order of the class after re-ordering. The j-th class of i-th pre-defined set is output as (**class_output_order[i][j]**)-th class from ep decoder. See subclause 9.4.9.
header_protection	This value indicates the header error protection mode. 0 indicates the use of basic set of FEC, and 1 indicates the use of extended header error protection, as defined in subclause 9.4.3. The extended header error protection is applied only if the length of the header exceeds 16 bits.
header_rate, header_crclen	These values have the same semantics with **class_rate[i][j]** and **class_crclen[i][j]** respectively, while these error protection is utilized for the protection of header part.
rs_fec_capability	This field indicates the correction capability of SRS code to protect the whole frame (see subclause9.4.7). This capability is given as number of erroneous bytes that can be corrected. The value "0" indicates SRS is not used.
rs_ep_frame()	Reed-Solomon error protected frame that is applied Reed-Solomon code.
rs_parity_bits	The Reed-Solomon parity bits for ep_frame(). See subclause 9.4.7.
ep_frame()	error protected frame.
ep_header()	EP frame header information.
ep_encoded_classes()	The EP encoded audio information.
interleaved_frame_mode1	The information bits after interleaving with interleaving mode 1. See subclause 9.4.1 and subclause 9.4.8.
interleaved_frame_mode2	The information bits after interleaving with interleaving mode 2. See subclause 9.4.1 and subclause 9.4.8.
stuffing_bits	The stuffing bits for the EP frame octet alignment. The number of bits Nstuff is signaled in class_attrib(), and should be in the range of 0...7.
choice_of_pred	The choice of pre-defined set. See subclause 9.4.2.
choice_of_pred_parity	The parity bits for **choice_of_pred**. See subclause 9.4.2.
class_attrib_parity	The parity bits for class_attrib(). See subclause 9.4.2.
class_attrib()	Attribution information for each class
class_bit_count[j]	The number of information bits included in the class. This field only exists in case the length_escape in out-of-band information is 1 (escape). The number of bits of this parameter Nbitcount is also signaled in the out-of-band information.

class_code_rate[j] The coding rate for the audio data belonging to the class, as defined in the table below. This field only exists in case the rate_escape in out-of-band information is 1 (escape).

Table 170: The coding rate for the audio data belonging to the class

Codeword	000	001	010	011	100	101	110	111
Puncture Rate	8/8	8/11	8/12	8/14	8/16	8/20	8/24	8/32
Puncture Pattern	FF, 00 00, 00	FF, A8 00, 00	FF, AA 00, 00	FF, EE 00, 00	FF, FF 00, 00	FF, FF AA, 00	FF, FF FF, 00	FF, FF FF, FF

class_crc_count[j] The number of CRC bits for the audio data belonging to the class, as defined in the table below. This field only exists in case the crclen_escape in out-of-band information is 1 (escape).

Table 171: The number of CRC bits for the audio data belonging to the class

Codeword	000	001	010	011	100	101	110	111
CRC bits	0	6	8	10	12	14	16	32

num_stuffing_bits the number of stuffing bits for the EP frame octet alignment. This field only exists in case the **bit_stuffing** in out-of-band information is 1.

ep_encoded_class[j] CRC/SRCPC encoded audio data of j-th class.

9.4 Tool description

9.4.1 Out-of-band information

In this subclause, the content of out-of band information is described. In the real transmission environment, these parameters should be sent during the channel configuration, using such as ObjectDescriptor.

For the class length, the length of the last class can be defined as “until the end”, which means this class lasts until the end of this frame. In the MPEG-4 systems, the systems layer guarantees the audio frame boundary by mapping one audio frame to one access unit. Therefore, the length of “until the end” class can be calculated from the length of other classes and the total EP-encoded audio frame length.

This implies the following two aspects, which should be carefully considered while generating the out-of-band information:

- The “until the end” definition is only allowed for the last class of each pre-defined set.
- If the length of the last class is fixed, this value should be set in the pre-definition file, and should not use the “until the end” definition. If the decoder knows this fixed value, the decoder can find the violation of the frame length. This may occur when the error protected audio frame is partially dropped at the de-multiplexing, or when the choice of the pre-defined set in the error-protected audio frame is corrupted due to channel error, and finding these violations will enhance the error resiliency.

The text file format and examples of this information can be found in the informative part.

While the flag class_optional can reduce the redundancy within ErrorProtectionSpecificConfig, the EP tool still works with the same number of pre-defined sets. If there are N classes with (class_optional == 1), this pre-defined set is extended to 2^N pre-defined sets. Unwrapping of the predefinition sets is described within the following subclause.

9.4.2 Derivation of pre-defined sets

This subclause describes the post processing, whose input is ErrorProtectionSpecificConfig() with “class_optional” switch and whose output are pre-defined sets used for the ep_frame() parameters.

General procedure:

- Each pre-defined set expands $2^{NCO[i]}$ pre-defined sets, where NCO[i] is the number of classes with (class_optional == 1) in i-th original pre-defined set. Hereafter, any class with (class_optional == 1) is referred to as optClass.
- These expanded pre-defined sets start from “all the optClasses don’t exist” to “all the optClasses exists”.

Algorithm:

```
transPred = 0;
for ( i = 0; i < nPred; i++ ) {          /* for all predefinition sets */
    for ( j = 0; j < pow ( 2, NCO[i] ); j++ ) {  /* unwraping */
        for ( k = 0; k < NCO[i]; k++ ) {      /* for all optional classes */
            if ( j & ( 0x01 << k ) ) {
                optClassExists[k] = 1;
            }
            else {
                optClassExists[k] = 0;
            }
        }
        DefineTransPred(transPred, i, optClassExists);
        transPred ++;
    }
}
```

where,

optClassExists[k] signals whether k-th optClass of the pre-defined set exists (1) or not (0) in the defining new pre-defined set.

DefineTransPred (transPred, i, optClassExists) defines transPred-th new pre-defined set used for the transmission. This new pre-defined set is a copy of i-th original pre-defined set, except it don’t have optClasses whose optClassExists equals to 0.

<u>Example</u>

ErrorProtectionSpecificConfig() defines pre-defined sets as follows:

Table 172: The example of pre-defined set

Pred #0		Pred #1	
Class A	class_optional = 1	Class E	class_optional = 1
Class B	class_optional = 0	Class F	class_optional = 0
Class C	class_optional = 1		
Class D	class_optional = 0		

After the pre-processing described above, the pre-defined sets used for ep_frame() becomes as follows:

Table 173: The example of pre-defined sets after the pre-processing

Pred #0	Pred #1	Pred #2	Pred #3	Pred #4	Pred #5
Class B	Class A	Class B	Class A	Class F	Class E
Class D	Class B	Class C	Class B		Class F
	Class D	Class D	Class C		
			Class D		

9.4.3 In-band information

The EP frame information, which is not included in the out-of-band information, is the in-band information. The parameters belonging to this information are transmitted as an EP frame header. The parameters are:

- The choice of pre-defined set
- The number of stuffing bits for byte alignment
- The class information which is not included in the out-of-band information

The EP decoder cannot decode the audio frame information without these parameters, and thus they have to be error protected stronger than or equal to the other parts. On this error protection, the choice of pre-defined set has to be treated differently from the other parts. This is because the length of the class information can be changed according to which pre-defined set is chosen. For this reason, this parameter is FEC encoded independently from the other parts. At decoder side, the choice of pre-defined set is decoded first, and then the length of the remaining header part is calculated with this information, and decodes that.

The FEC applied for these parts are as follows:

Basic set of FEC codes:

Table 174: Basic set of FEC codes for in-band information

Number of bit to be protected	FEC code	total number of bits	Length of codeword
1-2	majority (repeat 3 times)	3-6	3
3-4	BCH(7,4)	6-7	6-7
5-7	BCH(15,7)	13-15	13-15
8-12	Golay(23,12)	19-23	19-23
13-16	BCH(31,16)	28-31	28-31
17-	RCPC 8/16 + 4-bit CRC	50 -	-

Extended FEC:

If a header length exceeds 16 bits, this header is protected in the same way as the class information. The SRCPC code rate and the number of CRC bits are signaled. The encoding and decoding method for this is the same as described below within the CRC/SRCPC description.

The generation polynomials for each FEC is as follows:

BCH(7,4): x^3+x+1

BCH(15,7): $x^8+x^7+x^6+x^4+1$

Golay(23,12): $x^{11}+x^9+x^7+x^6+x^5+x^3+x+1$

BCH(31,16): $x^{15}+x^{11}+x^{10}+x^9+x^8+x^7+x^5+x^3+x^2+x+1$

With these polynomials, the FEC (*n, k*) for *l*-bit information encoding is made as follows:

Calculate the polynomial R(x) that satisfies

$$M(x)\, x^{n-l} = Q(x)G(x) + R(x)$$

M(x): Information bits. Highest order corresponds to the first bit to be transmitted

G(x): The generation polynomial from the above definition

This polynomial R(x) represents parity to **choice_of_pred** or class_attrib(), and set to **choice_of_pred_parity** or **class_attrib_parity** respectively. The highest order corresponds to the first bit. The decoder can perform error correction using these parity bits , while it is optional operation.

9.4.4 Concatenation functionality

EP tool has a functionality to concatenate several source coder frames to build up a new frame for the EP tool. In this concatenation, the groups of bits belonging to the same class in the different source coder frames are concatenated in adjacent, class by class basis. The concatenated groups belonging to the same class is either treated as a single new one class or independent class in the same manner as before the concatenation.

The number of frames to be concatenated is signaled as number_of_concatenated_frame in ErrorProtectionSpecificConfig(), and the choice whether the concatenated groups belonging to the same class is treated as single new one class or independent class is signaled by concatenate_flag[i][j] (1 indicate “single new one class”, and 0 indicates “independent class”). This process is illustrated in Figure 19.

The same pre-defined set shall be used for all concatenated frames. No escape mechanism shall be used for any class parameter.

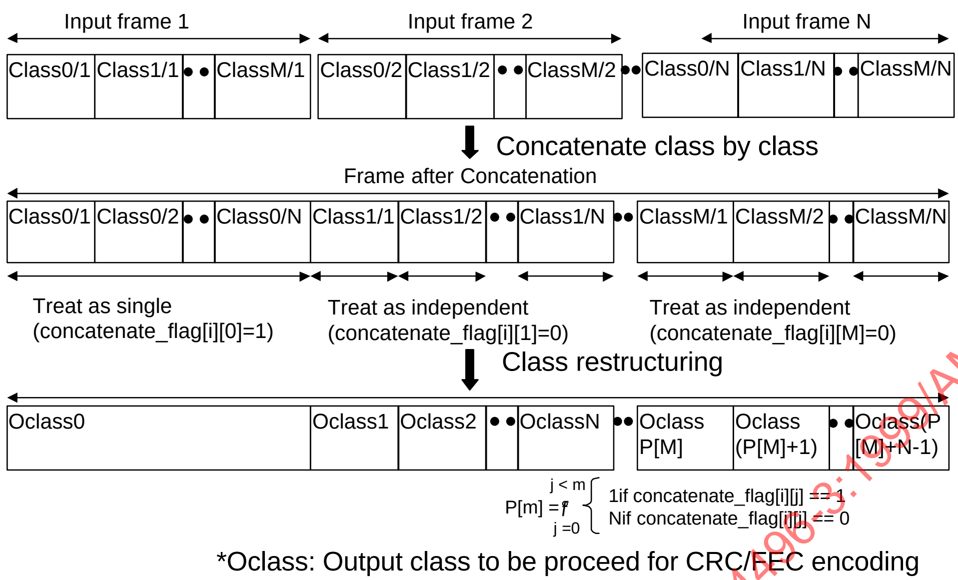


Figure 19: Concatenation procedure

9.4.5 CRC

The CRC provides error detection capability. The information bits of each class is CRC encoded as a first process. In this tool, the following set of the CRC is defined:

1-bit CRC ***CRC1***: $x+1$

2-bit CRC ***CRC2***: x^2+x+1

3-bit CRC ***CRC3***: x^3+x+1

4-bit CRC ***CRC4***: $x^4+x^3+x^2+1$

5-bit CRC ***CRC5***: $x^5+x^4+x^2+x+1$

6-bit CRC ***CRC6*** : $x^6+x^5+x^3+x^2+x+1$

7-bit CRC ***CRC7*** : $x^7+x^6+x^2+1$

8-bit CRC ***CRC8*** : x^8+x^2+x+1

9-bit CRC ***CRC9*** : $x^9+x^8+x^5+x^2+x+1$

10-bit CRC ***CRC10*** : $x^{10}+x^9+x^5+x^4+x+1$

11-bit CRC ***CRC11*** : $x^{11}+x^{10}+x^4+x^3+x+1$

12-bit CRC ***CRC12*** : $x^{12}+x^{11}+x^3+x^2+x+1$

13-bit CRC ***CRC13*** : $x^{13}+x^{12}+x^7+x^6+x^5+x^4+x^2+1$

14-bit CRC ***CRC14*** : $x^{14}+x^{13}+x^5+x^3+x^2+1$

15-bit CRC ***CRC15*** : $x^{15}+x^{14}+x^{11}+x^{10}+x^{7}+x^{6}+x^{2}+1$

16-bit CRC ***CRC16*** : $x^{16}+x^{12}+x^{5}+1$

24-bit CRC ***CRC24*** : $x^{24}+x^{23}+x^{6}+x^{5}+x+1$

32-bit CRC ***CRC32*** : $x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^{8}+x^{7}+x^{5}+x^{4}+x^{2}+x+1$

With these polynomials, the CRC encoding is made as follows:

Calculate the polynomial $R(x)$ that satisfies

$$M(x)x^k = Q(x)G(x) + R(x)$$

$M(x)$: Information bits. Highest order corresponds to the first bit to be transmitted

G(x): The generation polynomial from the above definition

k: The number of CRC bits.

With this polynomial $R(x)$, the CRC encoded bits $W(x)$ is represented as:

$$W(x) = M(x)x^k + R(x)$$

Note that the value k should be chosen so that the number of CRC encoded bits does not exceed 2^{k-1}.

Using these CRC bits, the decoder should perform error detection. When an error is detected through CRC, error concealment may be applied to reduce the quality degradation caused by the error. The error concealment method depends on MPEG-4 audio algorithms. See the informal annex (example of error concealment).

9.4.6 Systematic rate-compatible punctured convolutional (SRCPC) codes

Following to the CRC encoding, FEC encoding is made with the SRCPC codes. This subclause describes the SRCPC encoding process.

The channel encoder is based on a systematic recursive convolutional (SRC) encoder with rate R=1/4. The CRC encoded classes are concatenated and input into this encoder. Then, with the puncturing procedure described in the subclause later, we obtain a Rate Compatible Punctured Convolutional (RCPC) code whose code rate varies for each class according to the error sensitivity.

9.4.6.1 SRC code generation

The SRC code is generated from a rational generator matrix by using a feedback loop. A shift register realization of the encoder is shown in Figure 20.

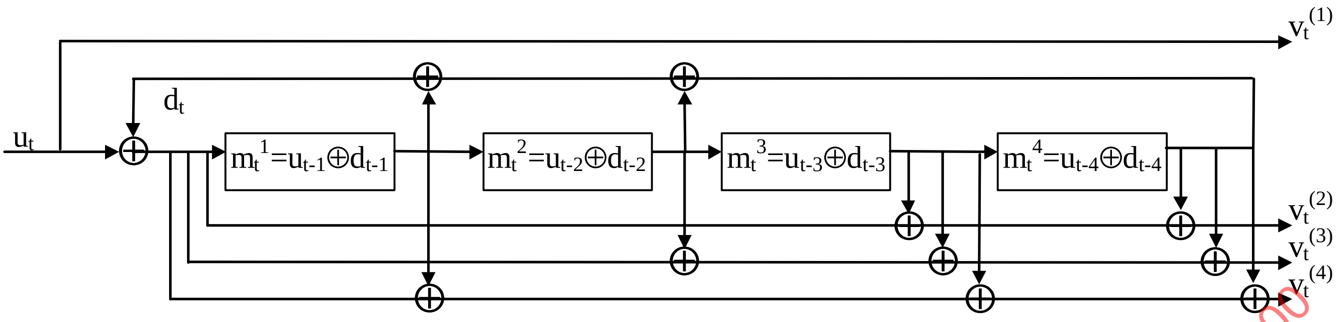


Figure 20: Shift register realization for systematic recursive convolutional encoder

To obtain the output vectors v_t at each time instant t, one has to know the content of the shift registers m_t^1, m_t^2, m_t^3, m_t^4 (corresponds to the state) and the input bit u_t at time t.

We obtain the output $v_t^{(2)}$, $v_t^{(3)}$ and $v_t^{(4)}$

$$v_t^{(2)} = m_t^4 \oplus m_t^3 \oplus (u_t \oplus d_t)$$

$$v_t^{(3)} = m_t^4 \oplus m_t^3 \oplus m_t^2 \oplus (u_t \oplus d_t)$$

$$v_t^{(4)} = m_t^4 \oplus m_t^3 \oplus m_t^1 \oplus (u_t \oplus d_t)$$

with

$$d_t = m_t^4 \oplus m_t^2 \oplus m_t^1,\ m_t^4 = u_{t-4} \oplus d_{t-4},\ m_t^3 = u_{t-3} \oplus d_{t-3},\ m_t^2 = u_{t-2} \oplus d_{t-2},\ m_t^1 = u_{t-1} \oplus d_{t-1}$$

Finally we obtain for the output vector $\underline{\boldsymbol{v}}_t = (v_t^{(1)}, v_t^{(2)}, v_t^{(3)}, v_t^{(4)})$ at time t depending on the input bit u_t and the current state $\underline{\boldsymbol{m}}_t = (m_t^1, m_t^2, m_t^3, m_t^4)$:

$$\boxed{\begin{aligned} V_t^{(1)} &= u_t \\ V_t^{(2)} &= m_t^4 \oplus m_t^3 \oplus (u_t \oplus d_t) = m_t^3 \oplus m_t^2 \oplus m_t^1 \oplus u_t \\ V_t^{(3)} &= m_t^4 \oplus m_t^3 \oplus m_t^2 \oplus (u_t \oplus d_t) = m_t^3 \oplus m_t^1 \oplus u_t \\ V_t^{(4)} &= m_t^4 \oplus m_t^3 \oplus m_t^1 \oplus (u_t \oplus d_t) = m_t^3 \oplus m_t^2 \oplus u_t \end{aligned}}$$

with $\underline{\mathbf{m}}_1 = (m_1^1, m_1^2, m_1^3, m_1^4) = (0, 0, 0, 0) = \underline{\mathbf{0}}$

The initial state is always $\underline{\mathbf{0}}$, i.e. each memory cell contains a 0 before the input of the first information bit u_t.

9.4.6.2 Termination of SRC code

In case the SRC coded class is indicated as terminated with termination_switch[i] in ErrorProtectionSpecificConfig(), or SRC code is used for the protection of in-band information, the SRC encoder shall add the tail bits at the end of this class, and start the succeeding SRC encoding with initial state (all the encoder shift register shall reset to be 0).

The tail bits following the information sequence $\boldsymbol{u}$ for returning to state $\underline{\boldsymbol{m}}_n = \underline{\boldsymbol{0}}$ (termination) depends on the last state $\underline{\boldsymbol{m}}_{n-3}$ (state after the input of the last information bit u_{n-4}.). The termination sequence for each state described by $\underline{\boldsymbol{m}}_{n-3}$ is given in Table 175. The receiver may use these tail bits (TB) for additional error detection.

The appendix $(u_{n-3}, u_{n-2}, u_{n-1}\ \ u_n)$ to the information sequence can be calculated with the following condition:

for all t with $n\text{-}3£\ t\ £\ n\text{: } u_t\ Å\ d_t = 0$

Hence we obtain for the tail bit vector $\underline{\boldsymbol{u}}'=(u_{n-3}, u_{n-2}, u_{n-1}, u_n)$ depending on the state $\underline{\boldsymbol{m}}_{n-3} =(m_{n-3}^{1}, m_{n-3}^{2}, m_{n-3}^{3}, m_{n-3}^{4})$

$$u_{n-3} = d_{n-3} = m_{n-3}^{4}\ Å\ m_{n-3}^{2}\ Å\ m_{n-3}^{1}$$
$$u_{n-2} = d_{n-2} = m_{n-2}^{4}\ Å\ m_{n-2}^{2}\ Å\ m_{n-2}^{1} = m_{n-3}^{3}\ Å\ m_{n-3}^{1}\ Å\ 0 = m_{n-3}^{3}\ Å\ m_{n-3}^{1}$$
$$u_{n-1} = d_{n-1} = m_{n-1}^{4}\ Å\ m_{n-1}^{3}\ Å\ m_{n-1}^{2} = m_{n-3}^{2}\ Å\ 0\ Å\ 0 = m_{n-3}^{2}$$
$$u_n = d_n = m_{n-3}^{1}\ Å\ 0\ Å\ 0 = m_{n-3}^{1}$$

Table 175: Tail bits for systematic recursive convolutional code

state $\underline{m}_{n-3}$	m_{n-3}^{4}	m_{n-3}^{3}	m_{n-3}^{2}	m_{n-3}^{1}	u_{n-3}	u_{n-2}	u_{n-1}	u_n
0	0	0	0	0	0	0	0	0
1	0	0	0	1	1	1	0	1
2	0	0	1	0	1	0	1	0
3	0	0	1	1	0	1	1	1
4	0	1	0	0	0	1	0	0
5	0	1	0	1	1	0	0	1
6	0	1	1	0	1	1	1	0
7	0	1	1	1	0	0	1	1
8	1	0	0	0	1	0	0	0
9	1	0	0	1	0	1	0	1
10	1	0	1	0	0	0	1	0
11	1	0	1	1	1	1	1	1
12	1	1	0	0	1	1	0	0
13	1	1	0	1	0	0	0	1
14	1	1	1	0	0	1	1	0
15	1	1	1	1	1	0	1	1

9.4.6.3 Puncturing of SRC for SRCPC code

Puncturing of the output of the SRC encoder allows different rates for transmission. The puncturing tables are listed in Table 176.

Table 176: Puncturing tables (all values in hexadecimal representation)

Rate r	8/8	8/9	8/10	8/11	8/12	8/13	8/14	8/15	8/16	8/17	8/18	8/19	8/20
$P_r(0)$	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
$P_r(1)$	00	80	88	A8	AA	EA	EE	FE	FF	FF	FF	FF	FF
$P_r(2)$	00	00	00	00	00	00	00	00	00	80	88	A8	AA
$P_r(3)$	00	00	00	00	00	00	00	00	00	00	00	00	00

Rate r	8/21	8/22	8/23	8/24	8/25	8/26	8/27	8/28	8/29	8/30	8/31	8/32
$P_r(0)$	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
$P_r(1)$	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
$P_r(2)$	EA	EE	FE	FF	FF	FF	FF	FF	FF	FF	FF	FF
$P_r(3)$	00	00	00	00	80	88	A8	AA	EA	EE	FE	FF

The puncturing is made with the period of 8, and each bit of Pr(i) indicates the corresponding vt(i) from the SRC encoder is punctured (not transmitted) or not (transmitted). Each bit of Pr(i) is used from MSB to LSB, and 0/1 indicates not-punctured/punctured respectively. The code rate is a property of the class, thus the choice of the table is made according which class the current bit belongs to. After this decision which bits from vt(i) is transmitted, they are output in the order from vt(0) to vt(3).

9.4.6.4 Decoding process of SRCPC code

At the decoder, the error correction should be performed using this SRCPC code, while it is the optional operation and the decoder may extract the original information by just ignoring parity bits.

Decoding of SRCPC can be achieved using Viterbi algorithm for the punctured convolutional coding.

9.4.7 Shortened Reed-Solomon codes

Shortened RS codes RS(255-*l*, 255-2*k*-*l*) defined over GF(2^8) is used to protect EP encoded frame or to protect each class. Here, *k* is the number of correctable errors in one RS codeword. *l* is for the shortening.

First, the EP encoded frame is divided into N parts, so that its length is less than or equal to (255-2*k*) octets. This division is made from the beginning of the frame so that the length of the sub-frame becomes (255-2*k*) octets, except the last part. Then for each of N sub-frames, the parity digits are calculated. For the transmission, these N parity digits are appended at the end of the EP frame. This process is illustrated in Figure 21.

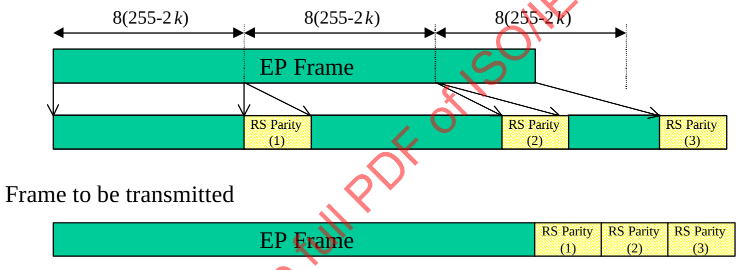


Figure 21: RS encoding of EP frame

In case RS code is used for EP for whole frame, the correction capability of SRS code *t* is transmitted within the out-of-band information as **rs_fec_capability**. This value can be selected as an arbitrary integer value satisfying $0 \le 2k \le 254$. The SRS code defined in the Galois Field GS(2^8) is generated from a generator polynomial $g(x) = (x-a)(x-a^2)\cdots(x-a^{2k})$, where a denotes a root of the primitive polynomial $m(x)=x^8+x^4+x^3+x^2+1$. The binary representative of a^i is shown in the Table 177 below, where the MSB of the octet is transmitted first.

Table 177: Binary representation for a^i $(0 \le i \le 254)$ over GF(2^8)

a^i	binary rep.	a^i	binary rep.	a^i	binary rep.	a^i	binary rep.
0	00000000	a^{63}	10100001	a^{127}	11001100	a^{191}	01000001
a^{0}	00000001	a^{64}	01011111	a^{128}	10000101	a^{192}	10000010
a^{1}	00000010	a^{65}	10111110	a^{129}	00010111	a^{193}	00011001
a^{2}	00000100	a^{66}	01100001	a^{130}	00101110	a^{194}	00110010
a^{3}	00001000	a^{67}	11000010	a^{131}	01011100	a^{195}	01100100
a^{4}	00010000	a^{68}	10011001	a^{132}	10111000	a^{196}	11001000
a^{5}	00100000	a^{69}	00101111	a^{133}	01101101	a^{197}	10001101
a^{6}	01000000	a^{70}	01011110	a^{134}	11011010	a^{198}	00000111
a^{7}	10000000	a^{71}	10111100	a^{135}	10101001	a^{199}	00001110
a^{8}	00011101	a^{72}	01100101	a^{136}	01001111	a^{200}	00011100
a^{9}	00111010	a^{73}	11001010	a^{137}	10011110	a^{201}	00111000
a^{10}	01110100	a^{74}	10001001	a^{138}	00100001	a^{202}	01110000
a^{11}	11101000	a^{75}	00001111	a^{139}	01000010	a^{203}	11100000
a^{12}	11001101	a^{76}	00011110	a^{140}	10000100	a^{204}	11011101
a^{13}	10000111	a^{77}	00111100	a^{141}	00010101	a^{205}	10100111
a^{14}	00010011	a^{78}	01111000	a^{142}	00101010	a^{206}	01010011
a^{15}	00100110	a^{79}	11110000	a^{143}	01010100	a^{207}	10100110
a^{16}	01001100	a^{80}	11111101	a^{144}	10101000	a^{208}	01010001
a^{17}	10011000	a^{81}	11100111	a^{145}	01001101	a^{209}	10100010
a^{18}	00101101	a^{82}	11010011	a^{146}	10011010	a^{210}	01011001
a^{19}	01011010	a^{83}	10111011	a^{147}	00101001	a^{211}	10110010
a^{20}	10110100	a^{84}	01101011	a^{148}	01010010	a^{212}	01111001
a^{21}	01110101	a^{85}	11010110	a^{149}	10100100	a^{213}	11110010
a^{22}	11101010	a^{86}	10110001	a^{150}	01010101	a^{214}	11111001
a^{23}	11001001	a^{87}	01111111	a^{151}	10101010	a^{215}	11101111
a^{24}	10001111	a^{88}	11111110	a^{152}	01001001	a^{216}	11000011
a^{25}	00000011	a^{89}	11100001	a^{153}	10010010	a^{217}	10011011
a^{26}	00000110	a^{90}	11011111	a^{154}	00111001	a^{218}	00101011
a^{27}	00001100	a^{91}	10100011	a^{155}	01110010	a^{219}	01010110
a^{28}	00011000	a^{92}	01011011	a^{156}	11100100	a^{220}	10101100
a^{29}	00110000	a^{93}	10110110	a^{157}	11010101	a^{221}	01000101
a^{30}	01100000	a^{94}	01110001	a^{158}	10110111	a^{222}	10001010
a^{31}	11000000	a^{95}	11100010	a^{159}	01110011	a^{223}	00001001
a^{32}	10011101	a^{96}	11011001	a^{160}	11100110	a^{224}	00010010
a^{33}	00100111	a^{97}	10101111	a^{161}	11010001	a^{225}	00100100
a^{34}	01001110	a^{98}	01000011	a^{162}	10111111	a^{226}	01001000
a^{35}	10011100	a^{99}	10000110	a^{163}	01100011	a^{227}	10010000
a^{36}	00100101	a^{100}	00010001	a^{164}	11000110	a^{228}	00111101
a^{37}	01001010	a^{101}	00100010	a^{165}	10010001	a^{229}	01111010
a^{38}	10010100	a^{102}	01000100	a^{166}	00111111	a^{230}	11110100
a^{39}	00110101	a^{103}	10001000	a^{167}	01111110	a^{231}	11110101
a^{40}	01101010	a^{104}	00001101	a^{168}	11111100	a^{232}	11110111
a^{41}	11010100	a^{105}	00011010	a^{169}	11100101	a^{233}	11110011
a^{42}	10110101	a^{106}	00110100	a^{170}	11010111	a^{234}	11111011
a^{43}	01110111	a^{107}	01101000	a^{171}	10110011	a^{235}	11101011
a^{44}	11101110	a^{108}	11010000	a^{172}	01111011	a^{236}	11001011
a^{45}	11000001	a^{109}	10111101	a^{173}	11110110	a^{237}	10001011
a^{46}	10011111	a^{110}	01100111	a^{174}	11110001	a^{238}	00001011
a^{47}	00100011	a^{111}	11001110	a^{175}	11111111	a^{239}	00010110
a^{48}	01000110	a^{112}	10000001	a^{176}	11100011	a^{240}	00101100
a^{49}	10001100	a^{113}	00011111	a^{177}	11011011	a^{241}	01011000
a^{50}	00000101	a^{114}	00111110	a^{178}	10101011	a^{242}	10110000
a^{51}	00001010	a^{115}	01111100	a^{179}	01001011	a^{243}	01111101

a^{52}	00010100	a^{116}	11111000	a^{180}	10010110	a^{244}	11111010
a^{53}	00101000	a^{117}	11101101	a^{181}	00110001	a^{245}	11101001
a^{54}	01010000	a^{118}	11000111	a^{182}	01100010	a^{246}	11001111
a^{55}	10100000	a^{119}	10010011	a^{183}	11000100	a^{247}	10000011
a^{56}	01011101	a^{120}	00111011	a^{184}	10010101	a^{248}	00011011
a^{57}	10111010	a^{121}	01110110	a^{185}	00110111	a^{249}	00110110
a^{58}	01101001	a^{122}	11101100	a^{186}	01101110	a^{250}	01101100
a^{59}	11010010	a^{123}	11000101	a^{187}	11011100	a^{251}	11011000
a^{60}	10111001	a^{124}	10010111	a^{188}	10100101	a^{252}	10101101
a^{61}	01101111	a^{125}	00110011	a^{189}	01010111	a^{253}	01000111
a^{62}	11011110	a^{126}	01100110	a^{190}	10101110	a^{254}	10001110

In case RS is used for UEP for each class, this is indicated by **fec_type** in ErrorProtectionSpecificConfig(). The SRS code shall be applied for each class, as the same manner with SRCPC encoding. One limitation for SRS code is that it can only be applied for the class whose length is known with ErrorProtectionSpecificConfig() or ep_header(), i. e. SRS code cannot be applied to the class whose length is defined as “Until the End”.

If the target information bits for SRS code are not byte-aligned, bits with value ‘0’ shall be added before SRS encoding, and deleted before transmission. At the decoder side, if SRS decoding is performed, same number of ‘0’s should be added before SRS decoding procedure, and deleted again after SRS decoding.

The decoder should perform error correction using these parity bytes, while this is an optional operation and the decoder may ignore these parity bytes added.

Before the SRS encoding, the EP frame is divided into sub-frames so that the length is less than or equal to 255-2*k*. The length of sub-frames are calculated with as follows:

L: The length of EP frame in octet

N: The number of sub-frames

l_i: The length of i-th sub-frame

N = minimum integer small than $(L / (255\text{-}2k))$

$l_i = 255\text{-}2k$, for $i<N$

$L \bmod (255\text{-}2k)$, for $i = N$

For each of these sub-frames, the SRC parity digits with length of 2*k* octets are calculated using $g(x)$ as follows:

$u(x)$: polynomial representative of a sub-frame. Lowest order corresponds to the first octet.

$p(x)$: polynomial representative of the parity digits. Lowest order corresponds to the first octet.

$$p(x) = x^{2k} \cdot u(x) \bmod g(x)$$

9.4.8 Recursive interleaving

The interleaving is applied in multi-stage manner. Figure 22 shows the interleaving method.

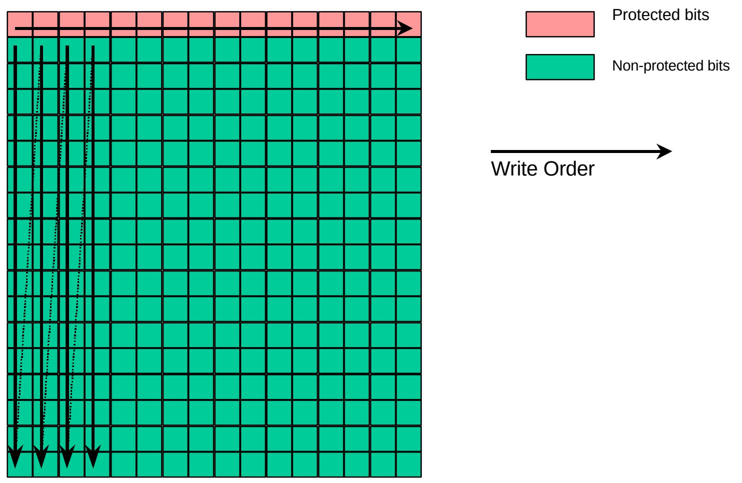


Figure 22: One stage of interleaving

In the multistage interleaving, the output of this one stage of interleaving is treated as a non-protected part in the next stage. Figure 23 shows the example of 2 stage interleaving.

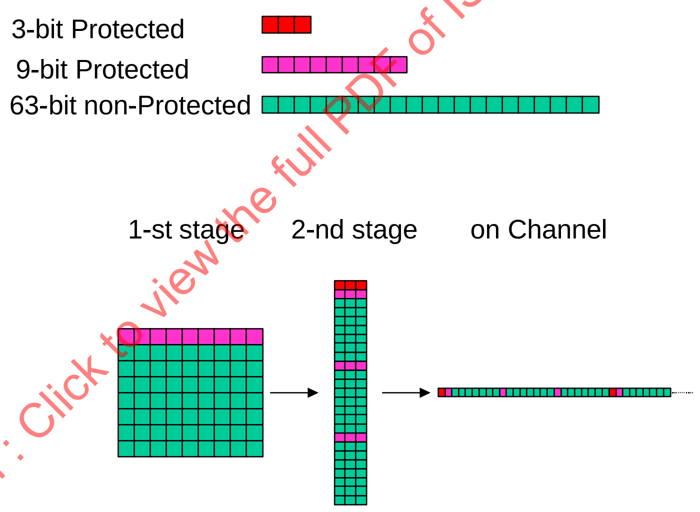


Figure 23: Example of multi-stage interleaving

By choosing the width *W* of the interleave-matrix to be the same as the FEC code length (or the value 28 in case of SRCPC codes), the interleaving size can be optimized for all the FEC codes.

In actual case, the total number of bits for the interleaving may not allow to use such rectangular. In such case, the matrix as shown in Figure 24 is used.

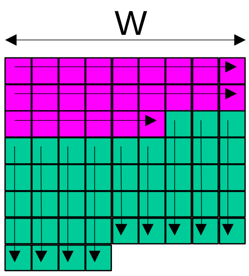


Figure 24: Interleave matrix in non-rectangular case

9.4.8.1 Definition of recursive interleaver

Two information streams are input to this interleaver, X_i and Y_i.

X_i, $0<=i<l_x$

Y_j, $0<=j<l_y$,

where l_x and l_y is the number of bits for each input streams X_i and Y_j, respectively. X_i is set to the interleaving matrix from the top left to the bottom right, into the horizontal direction. Then Y_i is set into the rest place in vertical direction.

With the width of interleaver W, the size of interleaving matrix is shown as Figure 25. Where,

$D = (l_x + l_y) / W$

$d = l_x + l_y - D * W$

Where '/' indicates division by truncation.

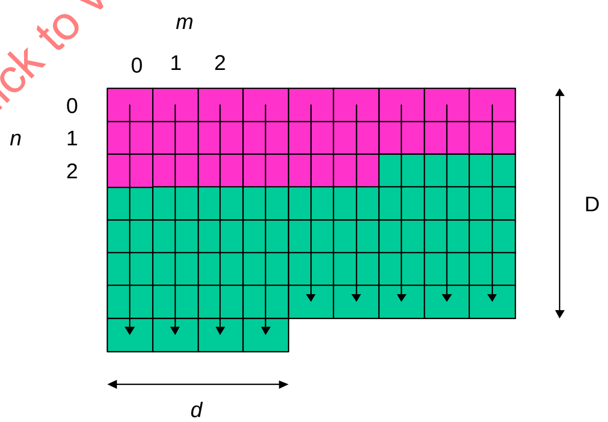


Figure 25: The size of interleaving matrix

The output bitstream Z_k ($0 < k <= l_x+l_y$) is read from this matrix from top left to bottom right, column by column in horizontal direction. Thus the bit placed *m*-th column, *n*-th row (m and n starts from 0) corresponds to Z_k where:

$$k = m * D + \min(m, d) + n$$

In the matrix, X_i is set to

$$m = i \bmod W, \quad n = i / W,$$

Thus Z_k which is set by the X_i becomes:

$$Z_k = X_i, \text{ where } k = (i \bmod W) * D + \min(i \bmod W, d) + i / W$$

The bits which are set with X_i in the interleaving matrix are shown as Figure 26 where:

$$D' = l_x / W$$

$$d' = l_x - D' * W$$

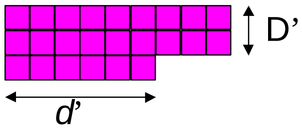


Figure 26: The bits which are set with Xi in the interleaving matrix

Thus, in the m-th row, Y_j is set from the *n*-th row where $n = D' + (m < d' ? 1 : 0)$ to the bottom. Thus Z_k set by Y_j is represented as follows:

```
Set j to 0;
for m = 0 to D-1 {
    for k = m * D + min(m, d) + D' + (m < d' ? 1 : 0) to (m+1) * D + min(m+1, d) - 1 {
        Z_k = Y_j;
        j ++;
    }
}
```

9.4.8.2 Modes of interleaving

Two modes of interleaving, mode 1 and mode 2 are defined in the following subclauses.

9.4.8.2.1 Interleaving operation in mode 1

Multi-stage interleaving is processed for **ep_encoded_class** from the last class to first class, and then class attribution part of ep_header() (which is class_attrib() + **class_attrib_parity**), and the pre-defined part of ep_header() (which is **choice_of_pred** + **choice_of_pred_parity**), as illustrated in Figure 27.

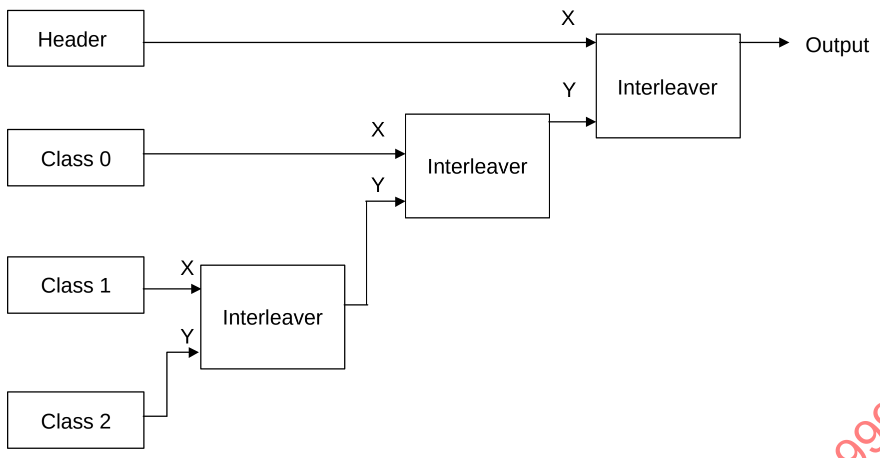


Figure 27: Interleaving process of mode1 specification

9.4.8.2.2 Interleaving operation in mode 2

In mode 2, a flag indicates whether the class is processed with interleaver, and how it is interleaved. This flag interleave_switch is signaled within the out-of-band information. The value 0 indicates the class is not processed by the interleaver. The value 1 indicates the class is interleaved by the recursive interleaver, and the length of the class is used as the width of the interleaver. The value 2 indicates the class is interleaved by the recursive interleaver, and the width is set to be equal to 28. The value 3 indicates the class is concatenated but not interleaved by the recursive interleaver. The interleaving operation for the ep_header is same as mode 1.

If the class is encoded with RS codes, this value is not used. Such class is interleaved with the recursive interleaver, and the width is set to be the number of bytes in the class. The bits in the class are written into the interleaving matrix by byte to byte for each column. (See Figure 28)

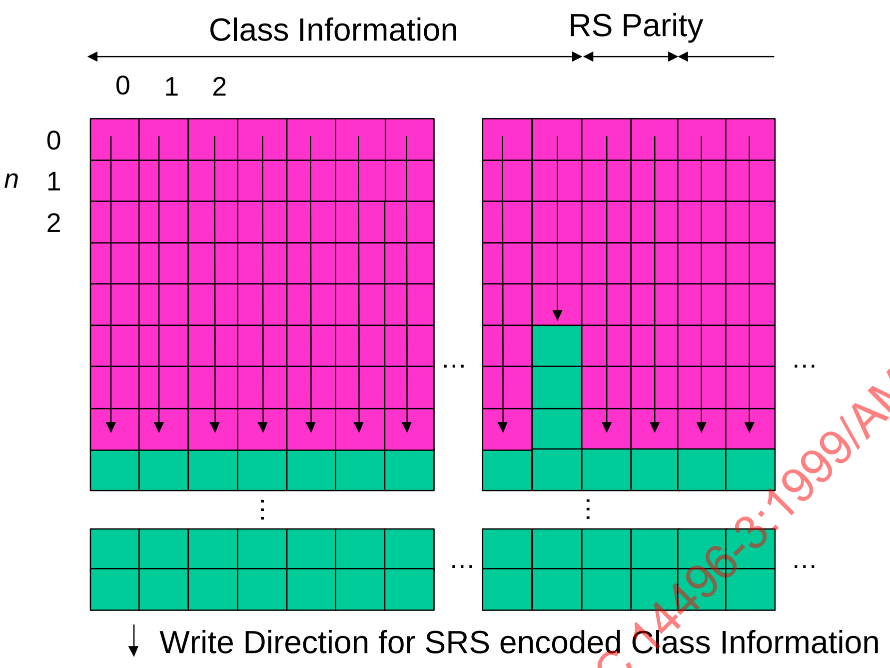
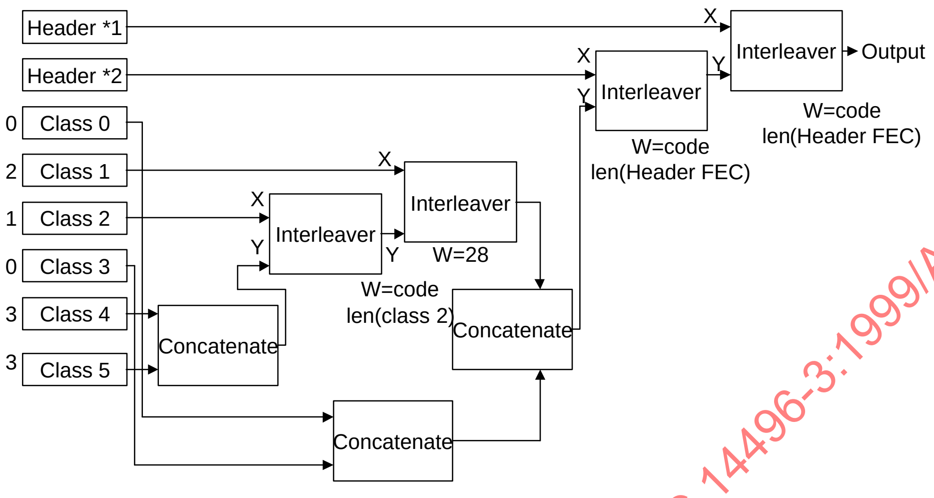


Figure 28: Interleaving matrix in RS encoded class case

The interleaving process to obtain **interleaved_frame_mode2** is as follows (N: number of classes):

```
clear buffer BUF_NO /* Buffer for non-interleaved part. */
clear buffer BUF_Y  /* Buffer for Y input in the next stage */
for j = 0 to N-1 {
    if ( interleave_switch[i][j] == 3) {
    concatenate ep_encoded_class[j] at the end of BUF_N0;
    }
}
set BUF_N0 into BUF_Y;
clear buffer BUF_N0
for j = N-1 to 0 {
    if ( interleave_switch[i][j] == 0 ) {
        concatenate ep_encoded_class[j] at the end of BUF_NO;
    } else if ( interleave_switch[i][j] != 3){
        if ( interleave_switch[i][j] == 1 ) {
            set the size of the interleave window to be the length of ep_encoded_class[j];
        } else if ( interleave_switch[i][j] == 2 ) {
            set the size of the interleave window to be 28;
        }
        input ep_encoded_class[j] into the recursive interleaver as X input;
        input BUF_Y into the recursive interleaver as Y input;
        set the output of the interleaver into BUF_Y;
    }
}
concatenate BUF_NO at the end of BUF_Y;
input class_attrib() followed by class_attrib_parity into the recursive interleaver as X
input;
input BUF_Y into the recursive interleaver as Y input;
set the output of the interleaver into BUF_Y;
input choice_of_pred followed by choice_of_pred_parity into the recursive interleaver as X
input;
input BUF_Y into the recursive interleaver as Y input;
set the output of the interleaver into BUF_Y;
set BUF_Y into interleaved_frame_mode2;
```

Interleave Switch ? (0: No interleaving, 1:Inter without Intra, 2: Inter with Intra)



*1 First part of Header : class_attrib() followed by class_attrib_parity
*2 Second part of Header : choice_of_pred followed by choice_of_pred_parity

Figure 29: Interleave process with class-wise control of interleaving

The width of interleave matrix is chosen according to the FEC used. In case block codes are used, i.e. In-band information is protected with "Basic set of FEC codes" as shown in Table 174, the length of the codeword is used as this width (see subclause 9.4.3). In case the RCPC code is used, 28-bit is used as this width.

9.4.9 Class reordered output

EP tool has a function to re-order the class output to the audio decoder, in order to align the bitstream order as defined for that codec, independently from the EP tool configuration and audio encoder to EP tool interface. Note that this interface is out of the scope of this specification, and is up to the implementation, while the interface to the audio decoder shall be aligned to the standard bitstream to avoid additional signaling from the encoder side.

The order of the class after this re-ordering is signaled as **class_output_order[i][j]** in the out-of-band information. The ep decoder re-orders the classes in the EP frame transmitted using i-th pre-defined set, so that the j-th class of EP frame is output as (**class_output_order[i][j]**)-th class when output to the audio decoder.

10 Error resilient bitstream payloads

10.1 Overview of the tools

Error resilient bitstream payloads allow the effective usage of advanced channel coding techniques like unequal error protection (UEP), which can be perfectly adapted to the needs of the different coding tools. The basic idea is to rearrange the conventional bitstream payload (as described in version 1 as well as within the tool descriptions in the previous chapters) depending on its error sensitivity in one or more instances belonging to different error sensitivity categories (ESC). This re-arrangement works either data element-wise or even bit-wise. The error resilient bitstream payload is build by concatenating these instances.

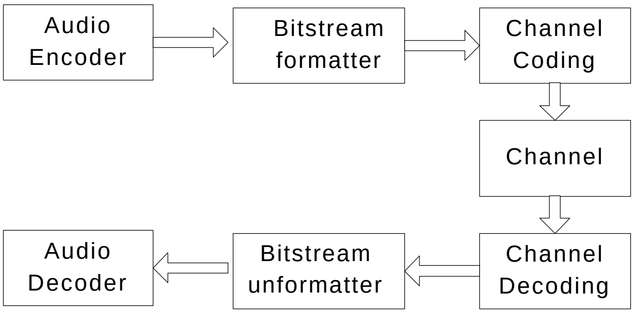


Figure 30: Basic principle of error resilient bitstream reordering

The basic principle is depicted in Figure 30. A bitstream, as defined in version 1 is reordered according to the error sensitivity of single bitstream elements or even single bits. This new arranged bitstream is channel coded, transmitted and channel decoded. Prior decoding, the bitstream is rearranged to its original order. Instead of performing the reordering in the described way, the reordered syntax, that is the bitstream order prior the bitstream formatter at the decoder site, is defined in this amendment.

In the subsequent sections, a detailed description of error resilient bitstream reordering for these tools can be found.

10.2 CELP

In order to describe the bit error sensitivity of bitstream elements, error sensitivity categories (ESC) are introduced. To describe single bits of elements, the following notation is used.

gain, x-y

Denotes bit x to bit y of element gain, whereby x is transmitted first. The LSB is bit zero and the MSB of an element that consist of N bit is N-1. The MSB is always the first bit in the bitstream.

The following syntax is a replacement for CelpBaseFrame as defined in 14494-3 section 3. The syntax for enhancement layer for bitrate and bandwidth scalability is not affected.

10.2.1 Syntax

10.2.1.1 Error resilient frame syntax

Table 178: Syntax of ER_CelpBaseFrame ()

Syntax	No. of bits	Mnemonic
ER_CelpBaseFrame()		
{		
if (ExcitationMode==MPE) {		
if (SampleRateMode == 8kHz) {		
MPE_NarrowBand_ESC0();		
MPE_NarrowBand_ESC1();		
MPE_NarrowBand_ESC2();		
MPE_NarrowBand_ESC3();		
MPE_NarrowBand_ESC4();		
}		
if (SampleRateMode == 16kHz) {		
MPE_WideBand_ESC0();		
MPE_WideBand_ESC1();		
MPE_WideBand_ESC2();		
MPE_WideBand_ESC3();		
MPE_WideBand_ESC4();		
}		
}		
if ((ExcitationMode==RPE) && (SampleRateMode==16kHz)) {		
RPE_WideBand_ESC0();		
RPE_WideBand_ESC1();		
RPE_WideBand_ESC2();		
RPE_WideBand_ESC3();		
RPE_WideBand_ESC4();		
}		
}		

10.2.1.2 MPE narrowband syntax

Table 179: Syntax of MPE_NarrowBand_ESC0()

Syntax	No. of bits	Mnemonic
MPE_NarrowBand_ESC0()		
{		
if (FineRateControl == ON) {		
Interpolation_flag;	**1**	**uimsbf**
LPC_Present;	**1**	**uimsbf**
}		
rms_index, 5-4;	**2**	**uimsbf**
for (subframe = 0; subframe < nrof_subframes; subframe++) {		
shape_delay[subframe], 7;	**1**	**uimsbf**
}		
}		

Table 180: Syntax of MPE_NarrowBand_ESC1()

Syntax	No. of bits	Mnemonic
MPE_NarrowBand_ESC1()		
{		
if (FineRateControl == ON) {		
if (LPC_Present == YES) {		
lpc_indices [0], 1-0;	**2**	**uimsbf**
lpc_indices [1], 0;	**1**	**uimsbf**
}		
} else {		
lpc_indices [0], 1-0;	**2**	**uimsbf**
lpc_indices [1], 0;	**1**	**uimsbf**
}		
signal_mode;	**2**	**uimsbf**
for (subframe = 0; subframe < nrof_subframes; subframe++) {		
shape_delay[subframe], 6-5;	**2**	**uimsbf**
}		
}		

Table 181: Syntax of MPE_NarrowBand_ESC2()

Syntax	No. of bits	Mnemonic
MPE_NarrowBand_ESC2()		
{		
if (FineRateControl == ON) {		
if (LPC_Present == YES) {		
lpc_indices [2], 6;	**1**	**uimsbf**
lpc_indices [2], 0;	**1**	**uimsbf**
lpc_indices [4];	**1**	**uimsbf**
}		
} else {		
lpc_indices [2], 6;	**1**	**uimsbf**
lpc_indices [2], 0;	**1**	**uimsbf**
lpc_indices [4];	**1**	**uimsbf**
}		
rms_index, 3	**1**	**uimsbf**
for (subframe = 0; subframe < nrof_subframes; subframe++) {		
shape_delay[subframe], 4-3;	**2**	**uimsbf**
gain_index[subframe], 1-0;	**2**	**uimsbf**
}		
}		

Table 182: Syntax of MPE_NarrowBand_ESC3()

Syntax	No. of bits	Mnemonic
MPE_NarrowBand_ESC3()		
{		
if (FineRateControl == ON) {		
if (LPC_Present == YES) {		
lpc_indices [0], 3-2;	**2**	**uimsbf**
lpc_indices [1], 2-1;	**2**	**uimsbf**
lpc_indices [2], 5-1;	**5**	**uimsbf**
}		
} else {		
lpc_indices [0], 3-2;	**2**	**uimsbf**
lpc_indices [1], 2-1;	**2**	**uimsbf**
lpc_indices [2], 5-1;	**5**	**uimsbf**
}		
for (subframe = 0; subframe < nrof_subframes; subframe++) {		
shape_delay[subframe], 2-0;	**3**	**uimsbf**
shape_signs[subframe];	**3 ... 12**	**uimsbf**
gain_index[subframe], 2;	**1**	**uimsbf**
}		
}		

Table 183: Syntax of MPE_NarrowBand_ESC4()

Syntax	No. of bits	Mnemonic
MPE_NarrowBand_ESC4()		
{		
if (FineRateControl == ON) {		
if (LPC_Present == YES) {		
lpc_indices [1], 3;	**1**	**uimsbf**
lpc_indices [3];	**6**	**uimsbf**
}		
} else {		
lpc_indices [1], 3;	**1**	**uimsbf**
lpc_indices [3];	**6**	**uimsbf**
}		
rms_index, 2-0	**3**	**uimsbf**
for (subframe = 0; subframe < nrof_subframes; subframe++) {		
shape_positions[subframe];	**13 ... 32**	**uimsbf**
gain_index[subframe], 5-3;	**3**	**uimsbf**
}		
}		

10.2.1.3 MPE wideband syntax

Table 184: Syntax of MPE_WideBand_ESC0()

Syntax	No. of bits	Mnemonic
MPE_WideBand_ESC0()		
{		
if (FineRateControl == ON) {		
Interpolation_flag;	**1**	**uimsbf**
LPC_Present;	**1**	**uimsbf**
if (LPC_Present == YES) {		
lpc_indices [0];	**5**	**uimsbf**
lpc_indices [1], 1-0;	**2**	**uimsbf**
lpc_indices [2], 6;	**1**	**uimsbf**
lpc_indices [2], 4-0;	**5**	**uimsbf**
lpc_indices [4];	**1**	**uimsbf**
lpc_indices [5], 0;	**1**	**uimsbf**
}		
} else {		
lpc_indices [0];	**5**	**uimsbf**
lpc_indices [1], 1-0;	**2**	**uimsbf**
lpc_indices [2], 6;	**1**	**uimsbf**
lpc_indices [2], 4-0;	**5**	**uimsbf**
lpc_indices [4];	**1**	**uimsbf**
lpc_indices [5], 0;	**1**	**uimsbf**
}		
rms_index, 4-5;	**2**	**uimsbf**
}		

Table 185: Syntax of MPE_WideBand_ESC1()

Syntax	No. of bits	Mnemonic
MPE_WideBand_ESC1()		
{		
if (FineRateControl == ON) {		
if (LPC_Present == YES) {		
lpc_indices [1], 3-2;	**2**	**uimsbf**
lpc_indices [2], 5;	**1**	**uimsbf**
lpc_indices [5], 1;	**1**	**uimsbf**
lpc_indices [6], 1-0;	**2**	**uimsbf**
}		
} else {		
lpc_indices [1], 3-2;	**2**	**uimsbf**
lpc_indices [2], 5;	**1**	**uimsbf**
lpc_indices [5], 1;	**1**	**uimsbf**
lpc_indices [6], 1-0;	**2**	**uimsbf**
}		
signal_mode;	**2**	**uimsbf**
for (subframe=0; subframe < nrof_subframes; subframe++) {		
shape_delay[subframe], 8-6;	**3**	**uimsbf**
}		
}		

Table 186: Syntax of MPE_WideBand_ESC2()

Syntax	No. of bits	Mnemonic
MPE_WideBand_ESC2()		
{		
if (FineRateControl == ON) {		
if (LPC_Present == YES) {		
lpc_indices [1], 4;	**1**	**uimsbf**
lpc_indices [3], 6;	**1**	**uimsbf**
lpc_indices [3], 1;	**1**	**uimsbf**
lpc_indices [5], 2;	**1**	**uimsbf**
lpc_indices [6], 3;	**1**	**uimsbf**
lpc_indices [7], 6;	**1**	**uimsbf**
lpc_indices [7], 4;	**1**	**uimsbf**
lpc_indices [7], 1-0;	**2**	**uimsbf**
lpc_indices [9];	**1**	**uimsbf**
}		
} else {		
lpc_indices [1], 4;	**1**	**uimsbf**
lpc_indices [3], 6;	**1**	**uimsbf**
lpc_indices [3], 1;	**1**	**uimsbf**
lpc_indices [5], 2;	**1**	**uimsbf**
lpc_indices [6], 3;	**1**	**uimsbf**
lpc_indices [7], 6;	**1**	**uimsbf**
lpc_indices [7], 4;	**1**	**uimsbf**
lpc_indices [7], 1-0;	**2**	**uimsbf**
lpc_indices [9];	**1**	**uimsbf**
}		
rms_index, 3;	**1**	**uimsbf**
for (subframe=0; subframe < nrof_subframes; subframe++) {		
shape_delay[subframe], 5-4;	**2**	**uimsbf**
gain_index[subframe], 1-0;	**2**	**uimsbf**
}		
}		

Table 187: Syntax of MPE_WideBand_ESC3()

Syntax	No. of bits	Mnemonic
MPE_WideBand_ESC3()		
{		
if (FineRateControl == ON) {		
if (LPC_Present == YES) {		
lpc_indices [3], 4-2;	**3**	**uimsbf**
lpc_indices [3], 0;	**1**	**uimsbf**
lpc_indices [5], 3;	**1**	**uimsbf**
lpc_indices [6], 2;	**1**	**uimsbf**
lpc_indices [7], 5;	**1**	**uimsbf**
lpc_indices [7], 3-2;	**2**	**uimsbf**
lpc_indices [8], 4-1;	**4**	**uimsbf**
}		
} else {		
lpc_indices [3], 4-2;	**3**	**uimsbf**
lpc_indices [3], 0;	**1**	**uimsbf**
lpc_indices [5], 3;	**1**	**uimsbf**
lpc_indices [6], 2;	**1**	**uimsbf**
lpc_indices [7], 5;	**1**	**uimsbf**
lpc_indices [7], 3-2;	**2**	**uimsbf**
lpc_indices [8], 4-1;	**4**	**uimsbf**
}		
for (subframe=0; subframe < nrof_subframes; subframe++) {		
shape_delay[subframe], 3-2;	**2**	**uimsbf**
shape_signs[subframe];	**3 ... 12**	**uimsbf**
gain_index[subframe], 2;	**1**	**uimsbf**
}		
}		

Table 188: Syntax of MPE_WideBand_ESC4()

Syntax	No. of bits	Mnemonic
MPE_WideBand_ESC4()		
{		
if (FineRateControl == ON) {		
if (LPC_Present == YES) {		
lpc_indices [3], 5;	**1**	**uimsbf**
lpc_indices [8], 0;	**1**	**uimsbf**
}		
} else {		
lpc_indices [3], 5;	**1**	**uimsbf**
lpc_indices [8], 0;	**1**	**uimsbf**
}		
rms_index, 2-0;	**3**	**uimsbf**
for (subframe=0; subframe < nrof_subframes; subframe++) {		
shape_delay[subframe], 1-0;	**2**	**uimsbf**
shape_positions[subframe];	**14 ... 32**	**uimsbf**
gain_index[subframe], 6-3;	**4**	**uimsbf**
}		
}		

10.2.1.4 RPE wideband syntax

Table 189: Syntax of RPE_WideBand_ESC0()

Syntax	No. of bits	Mnemonic
RPE_WideBand_ESC0()		
{		
if (FineRateControl == ON){		
Interpolation_flag;	**1**	**uimsbf**
LPC_Present;	**1**	**uimsbf**
if (LPC_Present == YES) {		
lpc_indices [0];	**5**	**uimsbf**
lpc_indices [1], 1-0;	**2**	**uimsbf**
lpc_indices [2], 6;	**1**	**uimsbf**
lpc_indices [2], 4-0;	**5**	**uimsbf**
lpc_indices [4];	**1**	**uimsbf**
lpc_indices [5], 0;	**1**	**uimsbf**
}		
} else {		
lpc_indices [0];	**5**	**uimsbf**
lpc_indices [1], 1-0;	**2**	**uimsbf**
lpc_indices [2], 6;	**1**	**uimsbf**
lpc_indices [2], 4-0;	**5**	**uimsbf**
lpc_indices [4];	**1**	**uimsbf**
lpc_indices [5], 0;	**1**	**uimsbf**
}		
for (subframe = 0; subframe < nrof_subframes; subframe++) {		
gain_indices[0][subframe], 5-3;	**3**	**uimsbf**
if (subframe == 0) {		
gain_indices[1][subframe], 4-3;	**2**	**uimsbf**
} else{		
gain_indices[1][subframe], 2;	**1**	**uimsbf**
}		
}		
}		

Table 190: Syntax of RPE_WideBand_ESC1()

Syntax	No. of bits	Mnemonic
RPE_WideBand_ESC1()		
{		
if (FineRateControl == ON) {		
if (LPC_Present == YES) {		
lpc_indices [1], 3-2;	**2**	**uimsbf**
lpc_indices [2], 5;	**1**	**uimsbf**
lpc_indices [5], 1;	**1**	**uimsbf**
lpc_indices [6], 1-0;	**2**	**uimsbf**
}		
} else {		
lpc_indices [1], 3-2;	**2**	**uimsbf**
lpc_indices [2], 5;	**1**	**uimsbf**
lpc_indices [5], 1;	**1**	**uimsbf**
lpc_indices [6], 1-0;	**2**	**uimsbf**
}		
for (subframe = 0; subframe < nrof_subframes; subframe++) {		
shape_delay[subframe], 7-5;	**3**	**uimsbf**
}		
}		

Table 191: Syntax of RPE_WideBand_ESC2()

Syntax	No. of bits	Mnemonic
RPE_WideBand_ESC2()		
{		
if (FineRateControl == ON) {		
if (LPC_Present == YES) {		
lpc_indices [1], 4;	**1**	**uimsbf**
lpc_indices [3], 6;	**1**	**uimsbf**
lpc_indices [3], 1;	**1**	**uimsbf**
lpc_indices [5], 2;	**1**	**uimsbf**
lpc_indices [6], 3;	**1**	**uimsbf**
lpc_indices [7], 6;	**1**	**uimsbf**
lpc_indices [7], 4;	**1**	**uimsbf**
lpc_indices [7], 1-0;	**2**	**uimsbf**
lpc_indices [9];	**1**	**uimsbf**
}		
} else {		
lpc_indices [1], 4;	**1**	**uimsbf**
lpc_indices [3], 6;	**1**	**uimsbf**
lpc_indices [3], 1;	**1**	**uimsbf**
lpc_indices [5], 2;	**1**	**uimsbf**
lpc_indices [6], 3;	**1**	**uimsbf**
lpc_indices [7], 6;	**1**	**uimsbf**
lpc_indices [7], 4;	**1**	**uimsbf**
lpc_indices [7], 1-0;	**2**	**uimsbf**
lpc_indices [9];	**1**	**uimsbf**
}		
for (subframe = 0; subframe < nrof_subframes; subframe++) {		
shape_delay[subframe], 4-3;	**2**	**uimsbf**
gain_index[0][subframe], 2;	**1**	**uimsbf**
if (subframe == 0) {		
gain_indices[1][subframe], 2;	**1**	**uimsbf**
} else{		
gain_indices[1][subframe], 1;	**1**	**uimsbf**
}		
}		
}		

Table 192: Syntax of RPE_WideBand_ESC3()

Syntax	No. of bits	Mnemonic
RPE_WideBand_ESC3()		
{		
if (FineRateControl == ON) {		
if (LPC_Present == YES) {		
lpc_indices [3], 4-2;	**3**	**uimsbf**
lpc_indices [3], 0;	**1**	**uimsbf**
lpc_indices [5], 3;	**1**	**uimsbf**
lpc_indices [6], 2;	**1**	**uimsbf**
lpc_indices [7], 5;	**1**	**uimsbf**
lpc_indices [7], 3-2;	**2**	**uimsbf**
lpc_indices [8], 4-1;	**4**	**uimsbf**
}		
} else {		
lpc_indices [3], 4-2;	**3**	**uimsbf**
lpc_indices [3], 0;	**1**	**uimsbf**
lpc_indices [5], 3;	**1**	**uimsbf**
lpc_indices [6], 2;	**1**	**uimsbf**
lpc_indices [7], 5;	**1**	**uimsbf**
lpc_indices [7], 3-2;	**2**	**uimsbf**
lpc_indices [8], 4-1;	**4**	**uimsbf**
}		
for (subframe = 0; subframe < nrof_subframes; subframe++) {		
shape_delay[subframe], 2-1;	**2**	**uimsbf**
}		
}		

Table 193: Syntax of RPE_WideBand_ESC4()

Syntax	No. of bits	Mnemonic
RPE_WideBand_ESC4()		
{		
if (FineRateControl == ON) {		
if (LPC_Present == YES) {		
lpc_indices [3], 5;	**1**	**uimsbf**
lpc_indices [8], 0;	**1**	**uimsbf**
}		
} else {		
lpc_indices [3], 5;	**1**	**uimsbf**
lpc_indices [8], 0;	**1**	**uimsbf**
}		
for (subframe = 0; subframe < nrof_subframes; subframe++) {		
shape_delay[subframe], 0;	**1**	**uimsbf**
shape_index[subframe];	**11, 12**	**uimsbf**
gain_index[0][subframe], 1-0;	**2**	**uimsbf**
if (subframe == 0) {		
gain_indices[1][subframe], 1-0;	**2**	**uimsbf**
} else{		
gain_indices[1][subframe], 0;	**1**	**uimsbf**
}		
}		
}		

10.2.2 General information

See ISO/IEC 14496-3 Subpart 3 speech coding - CELP

10.2.3 Tool description

See ISO/IEC 14496-3 Subpart 3 speech coding - CELP

10.3 HVXC

10.3.1 Syntax

When the HVXC tool is used with an error protection tool, such as an MPEG-4 EP Tool, the bit order arranged in accordance with the error sensitivity shown below should be used. The HVXC with the error resilient syntax shown below and the 4.0 kbit/s variable bitrate mode described in clause 12 are called ER_HVXC. ErHVXCframe(), ErHVXCfixframe(), ErHVXCvarframe(), ErHVXCenhaframe(), ErHVXCenh_fixframe(), and ErHVXCenh_varframe() are used in ER HVXC. Decoder configuration, ER_HvxcSpecificConfig() and Access Unit Payload of ER HVXC object are described in clause 12.

The same notation as in the CELP part is used to describe single bits of elements.

Definitions of the parameters are the same as in ISO/IEC 14496-3 subpart 2 as shown below.

Parameters used for 2/4 kbit/s

LSP1	LSP index 1	(5 bit)
LSP2	LSP index 2	(7 bit)
LSP3	LSP index 3	(5 bit)
LSP4	LSP index 4	(1 bit)
VUV	voiced/unvoiced flag	(2 bit)
Pitch	pitch parameter	(7 bit)
SE_shape1	spectrum index 1	(4 bit)
SE_shape2	spectrum index 2	(4 bit)
SE_gain	spectrum gain index	(5 bit)
VX_shape1[0]	stochastic codebook index 0	(6 bit)
VX_shape1[1]	stochastic codebook index 1	(6 bit)
VX_gain1[0]	gain codebook index 0	(4 bit)
VX_gain1[1]	gain codebook index 1	(4 bit)

Parameters used only for 4 kbit/s

LSP5	LSP index 5	(8 bit)
SE_shape3	4k spectrum index 3	(7 bit)
SE_shape4	4k spectrum index 4	(10 bit)
SE_shape5	4k spectrum index 5	(9 bit)
SE_shape6	4k spectrum index 6	(6 bit)
VX_shape2[0]	4k stochastic codebook index 0	(5 bit)
VX_shape2[1]	4k stochastic codebook index 1	(5 bit)
VX_shape2[2]	4k stochastic codebook index 2	(5 bit)
VX_shape2[3]	4k stochastic codebook index 3	(5 bit)
VX_gain2[0]	4k gain codebook index 0	(3 bit)
VX_gain2[1]	4k gain codebook index 1	(3 bit)
VX_gain2[2]	4k gain codebook index 2	(3 bit)

VX_gain2[3] 4k gain codebook index 3 (3 bit)

Parameters used only for 4 kbit/s variable rate mode

UpdateFlag a flag to indicate update noise frame (1 bit)

Table 194: Syntax of ErHVXCframe()

Syntax	No. of bits	Mnemonic

ErHVXCframe()
{
    if (HVXCvarMode ==0) {
        ErHVXCfixframe(HVXCrate);
    }
    else {
        ErHVXCvarframe(HVXCrate);
    }
}		

Table 195: Syntax of ErHVXCenhaframe()

Syntax	No. of bits	Mnemonic

ErHVXCenhaframe()
{
    if (HVXCvarMode ==0) {
        ErHVXCenh_fixframe();
    }
    else {
        ErHVXCenh_varframe();
    }
}		

10.3.1.1 Bitstream syntax of the fixed bitrate mode

Table 196: Syntax of ErHVXCfixframe ()

Syntax	No. of bits	Mnemonic
ErHVXCfixframe(rate)		
{		
if(rate == 2000){		
2k_ESC0();		
if(VUV!=0){		
2kV_ESC1();		
2kV_ESC2();		
2kV_ESC3();		
2kV_ESC4();		
}		
else{		
2kUV_ESC1();		
2kUV_ESC2();		
2kUV_ESC3();		
}		
}		
else if (rate >= 3700){		
4k_ESC0();		
if(VUV!=0){		
4kV_ESC1(rate);		
4kV_ESC2(rate);		
4kV_ESC3();		
4kV_ESC4();		
}		
else{		
4kUV_ESC1(rate);		
4kUV_ESC2(rate);		
}		
}		
}		

10.3.1.1.1 2.0 kbit/s frame

Table 197: Syntax of 2k_ESC0()

Syntax	No. of bits	Mnemonic
2k_ESC0()		
{		
VUV , 1-0;	**2**	**uimsbf**
}		

10.3.1.1.2 2.0 kbit/s voiced frame

Table 198: Syntax of 2kV_ESC1()

Syntax	No. of bits	Mnemonic
2kV_ESC1()		
{		
LSP4, 0;	**1**	**uimsbf**
SE_gain, 4-0;	**5**	**uimsbf**
LSP1, 4-0;	**5**	**uimsbf**
Pitch, 6-1;	**6**	**uimsbf**
LSP2, 6;	**1**	**uimsbf**
}		

Table 199: Syntax of 2kV_ESC2()

Syntax	No. of bits	Mnemonic
2kV_ESC2()		
{		
LSP3, 4;	**1**	**uimsbf**
LSP2, 5;	**1**	**uimsbf**
}		

Table 200: Syntax of 2kV_ESC3()

Syntax	No. of bits	Mnemonic
2kV_ESC3()		
{		
SE_shape1, 3-0;	**4**	**uimsbf**
SE_shape2, 3-0;	**4**	**uimsbf**
}		

Table 201: Syntax of 2kV_ESC4()

Syntax	No. of bits	Mnemonic
2kV_ESC4()		
{		
LSP2, 4-0;	**5**	**uimsbf**
LSP3, 3-0;	**4**	**uimsbf**
Pitch, 0;	**1**	**uimsbf**
}		

10.3.1.1.3 2.0 kbit/s unvoiced frame

Table 202: Syntax of 2kUV_ESC1()

Syntax	No. of bits	Mnemonic
2kUV_ESC1()		
{		
LSP4, 0;	**1**	**uimsbf**
VX_gain1[0], 3-0;	**4**	**uimsbf**
VX_gain1[1], 3-0;	**4**	**uimsbf**
LSP1, 4-0;	**5**	**uimsbf**
LSP2, 6-3;	**4**	**uimsbf**
}		

Table 203: Syntax of 2kUV_ESC2()

Syntax	No. of bits	Mnemonic
2kUV_ESC2()		
{		
LSP3, 4-3;	**2**	**uimsbf**
}		

Table 204: Syntax of 2kUV_ESC3()

Syntax	No. of bits	Mnemonic
2kUV_ESC3()		
{		
LSP2, 2-0;	**3**	**uimsbf**
LSP3, 2-0;	**3**	**uimsbf**
VX_shape1[0], 5-0;	**6**	**uimsbf**
VX_shape1[1], 5-0;	**6**	**uimsbf**
}		

10.3.1.1.4 4.0 kbit/s frame

Table 205: Syntax of 4k_ESC0()

Syntax	No. of bits	Mnemonic
4k_ESC0()		
{		
VUV, 1-0;	**2**	**uimsbf**
}		

10.3.1.1.5 4.0 kbit/s voiced frame

Table 206: Syntax of 4kV_ESC1()

Syntax	No. of bits	Mnemonic
4kV_ESC1(rate)		
{		
LSP4, 0;	**1**	**uimsbf**
SE_gain, 4-0;	**5**	**uimsbf**
LSP1, 4-0;	**5**	**uimsbf**
Pitch, 6-1;	**6**	**uimsbf**
LSP2, 6-3;	**4**	**uimsbf**
SE_shape3, 6-2;	**5**	**uimsbf**
LSP3, 4;	**1**	**uimsbf**
LSP5, 7;	**1**	**uimsbf**
SE_shape4, 9;	**1**	**uimsbf**
SE_shape5, 8;	**1**	**uimsbf**
if(rate>=4000){		
SE_shape6, 5;}	**1**	**uimsbf**
}		
}		

Table 207: Syntax of 4kV_ESC2()

Syntax	No. of bits	Mnemonic
4kV_ESC2(rate)		
{		
SE_shape4, 8-0;	**9**	**uimsbf**
SE_shape5, 7-0;	**8**	**uimsbf**
if(rate>=4000){		
SE_shape6, 4-0;	**5**	**uimsbf**
}		
}		

Table 208: Syntax of 4kV_ESC3()

Syntax	No. of bits	Mnemonic
4kV_ESC3()		
{		
SE_shape1, 3-0;	**4**	**uimsbf**
SE_shape2, 3-0;	**4**	**uimsbf**
}		

Table 209: Syntax of 4kV_ESC4()

Syntax	No. of bits	Mnemonic
4kV_ESC4()		
{		
LSP2, 2-0;	**3**	**uimsbf**
LSP3, 3-0;	**4**	**uimsbf**
LSP5, 6-0;	**7**	**uimsbf**
Pitch, 0;	**1**	**uimsbf**
SE_shape3, 1-0;	**2**	**uimsbf**
}		

10.3.1.1.6 4.0 kbit/s unvoiced frame

Table 210: Syntax of 4kUV_ESC1()

Syntax	No. of bits	Mnemonic
4kUV_ESC1(rate)		
{		
LSP4, 0;	**1**	**uimsbf**
VX_gain1[0], 3-0;	**4**	**uimsbf**
VX_gain1[1], 3-0;	**4**	**uimsbf**
LSP1, 4-0;	**5**	**uimsbf**
LSP2, 6-3;	**4**	**uimsbf**
LSP3, 4;	**1**	**uimsbf**
LSP5, 7;	**1**	**uimsbf**
VX_gain2[0], 2-0;	**3**	**uimsbf**
VX_gain2[1], 2-0;	**3**	**uimsbf**
VX_gain2[2], 2-0;	**3**	**uimsbf**
if(rate>=4000){		
VX_gain2[3], 2-0;	**3**	**uimsbf**
}		
}		

Table 211: Syntax of 4kUV_ESC2()

Syntax	No. of bits	Mnemonic
4kUV_ESC2(rate)		
{		
LSP2, 2-0;	**3**	**uimsbf**
LSP3, 3-0;	**4**	**uimsbf**
LSP5, 6-0;	**7**	**uimsbf**
VX_shape1[0], 5-0;	**6**	**uimsbf**
VX_shape1[1], 5-0;	**6**	**uimsbf**
VX_shape2[0], 4-0;	**5**	**uimsbf**
VX_shape2[1], 4-0;	**5**	**uimsbf**
VX_shape2[2], 4-0;	**5**	**uimsbf**
if(rate>=4000){		
VX_shape2[3], 4-0;	**5**	**uimsbf**
}		
}		

10.3.1.2 Bitstream syntax for the scalable mode

Bitstream syntax of the base layer for scalable mode is the same as that of ErHVXCfixframe(2000). Bitstream syntax of enhancement layer, ErHVXCenhaFrame(), for scalable mode is shown below.

Table 212: Syntax of ErHVXCenh_fixframe()

Syntax	No. of bits	Mnemonic
ErHVXCenh_fixframe()		
{		
if(VUV!=0){		
EnhV_ESC1();		
EnhV_ESC2();		
EnhV_ESC3();		
}		
else{		
EnhUV_ESC1();		
EnhUV_ESC2();		
}		
}		

10.3.1.2.1 Enhancement layer of voiced frame

Table 213: Syntax of EnhV_ESC1()

Syntax	No. of bits	Mnemonic
EnhV_ESC1()		
{		
SE_shape3, 6-2;	**5**	**uimsbf**
LSP5, 7;	**1**	**uimsbf**
SE_shape4, 9;	**1**	**uimsbf**
SE_shape5, 8;	**1**	**uimsbf**
SE_shape6, 5;	**1**	**uimsbf**
}		

Table 214: Syntax of EnhV_ESC2()

Syntax	No. of bits	Mnemonic
EnhV_ESC2()		
{		
SE_shape4, 8-0;	**9**	**uimsbf**
SE_shape5, 7-0;	**8**	**uimsbf**
SE_shape6, 4-0;	**5**	**uimsbf**
}		

Table 215: Syntax of EnhV_ESC3()

Syntax	No. of bits	Mnemonic
EnhV_ESC3()		
{		
LSP5, 6-0;	**7**	**uimsbf**
SE_shape3, 1-0;	**2**	**uimsbf**
}		

10.3.1.2.2 Enhancement layer of unvoiced frame

Table 216: Syntax of EnhUV_ESC1()

Syntax	No. of bits	Mnemonic
EnhUV_ESC1()		
{		
LSP5, 7;	**1**	**uimsbf**
VX_gain2[0], 2-0;	**3**	**uimsbf**
VX_gain2[1], 2-0;	**3**	**uimsbf**
VX_gain2[2], 2-0;	**3**	**uimsbf**
VX_gain2[3], 2-0;	**3**	**uimsbf**
}		

Table 217: Syntax of EnhUV_ESC2()

Syntax	No. of bits	Mnemonic
EnhUV_ESC2()		
{		
LSP5, 6-0;	**7**	**uimsbf**
VX_shape2[0], 4-0;	**5**	**uimsbf**
VX_shape2[1], 4-0;	**5**	**uimsbf**
VX_shape2[2], 4-0;	**5**	**uimsbf**
VX_shape2[3], 4-0;	**5**	**uimsbf**
}		

10.3.1.3 Bitstream syntax of variable bitrate mode

Table 218: Syntax of ErHVXCvarframe()

Syntax	No. of bits	Mnemonic
ErHVXCvarframe(rate)		
{		
if (rate == 2000) {		
if(var_ScalableFlag == 1) {		
BaseVar_ESC0()		
if(VUV==2 || VUV==3) {		
BaseVarV_ESC1();		
BaseVarV_ESC2();		
BaseVarV_ESC3();		
BaseVarV_ESC4();		
} else if(VUV == 0) {		
BaseVarUV_ESC1();		
BaseVarUV_ESC2();		
BaseVarUV_ESC3();		
} else {		
BaseVarBGN_ESC1();		
if(UpdateFlag == 1) {		
BaseVarBGN_ESC2();		
BaseVarBGN_ESC3();		
}		
}		
} else {		
Var2k_ESC0();		
if (VUV!=1) {		
if (VUV!=0) {		

```
                    Var2kV_ESC1();
                    Var2kV_ESC2();
                    Var2kV_ESC3();
                    Var2kV_ESC4();
                } else {
                    Var2kUV_ESC1();
                    Var2kUV_ESC2();
                    Var2kUV_ESC3();
                }
            } else {
                Var2kBGN_ESC1();
            }
        }
    } else {
        Var4k_ESC0();
        if (VUV==2 || VUV==3) {
            Var4kV_ESC1();
            Var4kV_ESC2();
            Var4kV_ESC3();
            Var4kV_ESC4();
        } else if (VUV==0) {
            Var4kUV_ESC1();
            Var4kUV_ESC2();
            Var4kUV_ESC3();
        } else {
            Var4kBGN_ESC1();
            if (UpdateFlag == 1) {
                Var4kBGN_ESC2();
                Var4kBGN_ESC3()
            }
        }
    }
}
```


Table 219: Syntax of Var2k_ESC0()

Syntax	No. of bits	Mnemonic
Var2k_ESC0()		
{		
VUV , 1-0;	**2**	**uimsbf**
}		

10.3.1.3.1 Voiced Frame of 2.0 kbit/s maximum variable bitrate mode

Table 220: Syntax of Var2kV_ESC1()

Syntax	No. of bits	Mnemonic
Var2kV_ESC1()		
{		
LSP4, 0;	**1**	**uimsbf**
SE_gain, 4-0;	**5**	**uimsbf**
LSP1, 4-0;	**5**	**uimsbf**
Pitch, 6-1;	**6**	**uimsbf**
LSP2, 6;	**1**	**uimsbf**
}		

Table 221: Syntax of Var2kV_ESC2()

Syntax	No. of bits	Mnemonic
Var2kV_ESC2()		
{		
LSP3, 4;	**1**	**uimsbf**
LSP2, 5;	**1**	**uimsbf**
}		

Table 222: Syntax of Var2kV_ESC3()

Syntax	No. of bits	Mnemonic
Var2kV_ESC3()		
{		
SE_shape1, 3-0;	**4**	**uimsbf**
SE_shape2, 3-0;	**4**	**uimsbf**
}		

Table 223: Syntax of Var2kV_ESC4()

Syntax	No. of bits	Mnemonic
Var2kV_ESC4()		
{		
LSP2, 4-0;	**5**	**uimsbf**
LSP3, 3-0;	**4**	**uimsbf**
Pitch, 0;	**1**	**uimsbf**
}		

10.3.1.3.2 Unvoiced frame of 2.0 kbit/s maximum variable bitrate mode

Table 224: Syntax of Var2kUV_ESC1()

Syntax	No. of bits	Mnemonic
Var2kUV_ESC1()		
{		
LSP4, 0;	**1**	**uimsbf**
VX_gain1[0], 3-0;	**4**	**uimsbf**
VX_gain1[1], 3-0;	**4**	**uimsbf**
LSP1, 4-0;	**5**	**uimsbf**
LSP2, 6-3;	**4**	**uimsbf**
}		

Table 225: Syntax of Var2kUV_ESC2()

Syntax	No. of bits	Mnemonic
Var2kUV_ESC2()		
{		
LSP3, 4-3;	**2**	**uimsbf**
}		

Table 226: Syntax of Var2kUV_ESC3()

Syntax	No. of bits	Mnemonic
Var2kUV_ESC3()		
{		
LSP2, 2-0;	**3**	**uimsbf**
LSP3, 2-0;	**3**	**uimsbf**
}		

Table 227: Syntax of Var2kBGN_ESC1()

Syntax	No. of bits	Mnemonic
Var2kBGN_ESC1()		
{		
}		

Table 228: Syntax of Var4kV_ESC0()

Syntax	No. of bits	Mnemonic
Var4kV_ESC0()		
{		
VUV,1-0;	**2**	**uimsbf**
}		

10.3.1.3.3 Voiced frame of 4.0 kbit/s maximum variable bitrate mode

Table 229: Syntax of Var4kV_ESC1()

Syntax	No. of bits	Mnemonic
Var4kV_ESC1()		
{		
LSP4,0;	**1**	**uimsbf**
SE_gain,4-0;	**5**	**uimsbf**
LSP1, 4-0;	**5**	**uimsbf**
Pitch, 6-1;	**6**	**uimsbf**
LSP2, 6-3;	**4**	**uimsbf**
SE_shape3, 6-2;	**5**	**uimsbf**
LSP3, 4;	**1**	**uimsbf**
LSP5, 7;	**1**	**uimsbf**
SE_shape4, 9;	**1**	**uimsbf**
SE_shape5, 8;	**1**	**uimsbf**
SE_shape6, 5;	**1**	**uimsbf**
}		

Table 230: Syntax of Var4kV_ESC2()

Syntax	No. of bits	Mnemonic
Var4kV_ESC2()		
{		
SE_shape4, 8-0;	**9**	**uimsbf**
SE_shape5, 7-0;	**8**	**uimsbf**
SE_shape6, 4-0;	**5**	**uimsbf**
}		

Table 231: Syntax of Var4kV_ESC3()

Syntax	No. of bits	Mnemonic
Var4kV_ESC3()		
{		
SE_shape1, 3-0;	**4**	**uimsbf**
SE_shape2, 3-0;	**4**	**uimsbf**
}		

Table 232: Syntax of Var4kV_ESC4()

Syntax	No. of bits	Mnemonic
Var4kV_ESC4()		
{		
LSP2, 2-0;	**3**	**uimsbf**
LSP3, 3-0;	**4**	**uimsbf**
LSP5, 6-0;	**7**	**uimsbf**
Pitch, 0;	**1**	**uimsbf**
SE_shape3, 1-0;	**2**	**uimsbf**
}		

10.3.1.3.4 Unvoiced frame of 4.0 kbit/s maximum variable bitrate mode

Table 233: Syntax of Var4kUV_ESC1()

Syntax	No. of bits	Mnemonic
Var4kUV_ESC1()		
{		
LSP4, 0;	**1**	**uimsbf**
VX_gain1[0], 3-0;	**4**	**uimsbf**
VX_gain1[1], 3-0;	**4**	**uimsbf**
LSP1, 4-0;	**5**	**uimsbf**
LSP2, 6-3;	**4**	**uimsbf**
}		

Table 234: Syntax of Var4kUV_ESC2()

Syntax	No. of bits	Mnemonic
Var4kUV_ESC2()		
{		
LSP3, 4-3;	**2**	**uimsbf**
}		

Table 235: Syntax of Var4kUV_ESC3()

Syntax	No. of bits	Mnemonic
Var4kUV_ESC3()		
{		
LSP2, 2-0;	**3**	**uimsbf**
LSP3, 2-0;	**3**	**uimsbf**
VX_shape1[0], 5-0;	**6**	**uimsbf**
VX_shape1[1], 5-0;	**6**	**uimsbf**
}		

10.3.1.3.5 Back Ground Noise (BGN) frame of 4.0 kbit/s maximum variable bitrate mode

Table 236: Syntax of Var4kBGN_ESC1()

Syntax	No. of bits	Mnemonic
Var4kBGN_ESC1()		
{		
UpdateFlag, 0;	**1**	**uimsbf**
}		

Table 237: Syntax of Var4kBGN_ESC2()

Syntax	No. of bits	Mnemonic
Var4kBGN_ESC2()		
{		
LSP4, 0;	**1**	**uimsbf**
VX_gain1[0], 3-0;	**4**	**uimsbf**
LSP1, 4-0;	**5**	**uimsbf**
LSP2, 6-3;	**4**	**uimsbf**
}		

Table 238: Syntax of Var4kBGN_ESC3()

Syntax	No. of bits	Mnemonic
Var4kBGN_ESC3()		
{		
LSP3, 4-3;	**2**	**uimsbf**
LSP2, 2-0;	**3**	**uimsbf**
LSP3, 2-0;	**3**	**uimsbf**
}		

10.3.1.3.6 Base Layer of the scalable mode of the variable bitrate mode

Table 239: Syntax of BaseVar_ESC0()

Syntax	No. of bits	Mnemonic
BaseVar_ESC0()		
{		
VUV, 1-0;	**2**	**uimsbf**
}		

10.3.1.3.7 Base Layer of the scalable mode of the variable bitrate mode - Voiced frame

Table 240: Syntax of BaseVarV_ESC1()

Syntax	No. of bits	Mnemonic
BaseVarV_ESC1()		
{		
LSP4, 0;	**1**	**uimsbf**
SE_gain, 4-0;	**5**	**uimsbf**
LSP1, 4-0;	**5**	**uimsbf**
Pitch, 6-1;	**6**	**uimsbf**
LSP2, 6;	**1**	**uimsbf**
}		

Table 241: Syntax of BaseVarV_ESC2()

Syntax	No. of bits	Mnemonic
BaseVarV_ESC2()		
{		
LSP3, 4;	**1**	**uimsbf**
LSP2, 5;	**1**	**uimsbf**
}		

Table 242: Syntax of BaseVarV_ESC3()

Syntax	No. of bits	Mnemonic
BaseVarV_ESC3()		
{		
SE_shape1, 3-0;	**4**	**uimsbf**
SE_shape2, 3-0;	**4**	**uimsbf**
}		

Table 243: Syntax of BaseVarV_ESC4()

Syntax	No. of bits	Mnemonic
BaseVarV_ESC4()		
{		
LSP2, 4-0;	**5**	**uimsbf**
LSP3, 3-0;	**4**	**uimsbf**
Pitch, 0;	**1**	**uimsbf**
}		

10.3.1.3.8 Base Layer of the scalable mode of the variable bitrate mode - Unvoiced frame

Table 244: Syntax of BaseVarUV_ESC1()

Syntax	No. of bits	Mnemonic
BaseVarUV_ESC1()		
{		
LSP4, 0;	**1**	**uimsbf**
VX_gain1[0], 3-0;	**4**	**uimsbf**
VX_gain1[1], 3-0;	**4**	**uimsbf**
LSP1, 4-0;	**5**	**uimsbf**
LSP2, 6-3;	**4**	**uimsbf**
}		

Table 245: Syntax of BaseVarUV_ESC2()

Syntax	No. of bits	Mnemonic
BaseVarUV_ESC2()		
{		
LSP3, 4-3;	**2**	**uimsbf**
}		

Table 246: Syntax of BaseVarUV_ESC3()

Syntax	No. of bits	Mnemonic
BaseVarUV_ESC3()		
{		
LSP2, 2-0;	**3**	**uimsbf**
LSP3, 2-0;	**3**	**uimsbf**
VX_shape1[0], 5-0;	**6**	**uimsbf**
VX_shape1[1], 5-0;	**6**	**uimsbf**
}		

10.3.1.3.9 Base layer of the scalable mode of the variable bitrate mode - Back Ground Noise (BGN) frame

Table 247: Syntax of BaseVarBGN_ESC1()

Syntax	No. of bits	Mnemonic
BaseVarBGN_ESC1()		
{		
UpdateFlag, 0;	**1**	**uimsbf**
}		

Table 248: Syntax of BaseVarBGN_ESC2()

Syntax	No. of bits	Mnemonic
BaseVarBGN_ESC2()		
{		
LSP4, 0;	**1**	**uimsbf**
VX_gain1[0], 3-0;	**4**	**uimsbf**
LSP1, 4-0;	**5**	**uimsbf**
LSP2, 6-3;	**4**	**uimsbf**
}		

Table 249: Syntax of BaseVarBGN_ESC3()

Syntax	No. of bits	Mnemonic
BaseVarBGN_ESC3()		
{		
LSP3, 4-3;	**2**	**uimsbf**
LSP2, 2-0;	**3**	**uimsbf**
LSP3, 2-0;	**3**	**uimsbf**
}		

10.3.1.4 Enhancement Layer of the scalable mode of the variable bitrate mode (only for voiced frame)

Table 250: Syntax of ErHVXCenh_varframe()

Syntax	No. of bits	Mnemonic
ErHVXCenh_varframe()		
{		
if (VUV==2 || VUV==3) {		
EnhVarV_ESC1();		
EnhVarV_ESC2();		
EnhVarV_ESC3();		
}		
}		

Table 251: Syntax of EnhVarV_SC1()

Syntax	No. of bits	Mnemonic
EnhVarV_ESC1()		
{		
SE_shape3, 6-2;	**5**	**uimsbf**
LSP5, 7;	**1**	**uimsbf**
SE_shape4, 9;	**1**	**uimsbf**
SE_shape5, 8;	**1**	**uimsbf**
SE_shape6, 5;	**1**	**uimsbf**
}		

Table 252: Syntax of EnhVarV_SC2()

Syntax	No. of bits	Mnemonic
EnhVarV_ESC2()		
{		
SE_shape4, 8-0;	**9**	**uimsbf**
SE_shape5, 7-0;	**8**	**uimsbf**
SE_shape6, 4-0;	**5**	**uimsbf**
}		

Table 253: Syntax of EnhVarV_SC3()

Syntax	No. of bits	Mnemonic
EnhVarV_ESC3()		
{		
LSP5, 6-0;	**7**	**uimsbf**
SE_shape3, 1-0;	**2**	**uimsbf**
}		

10.3.2 General information

See ISO/IEC 14496-3 subpart 2 speech coding – HVXC.

10.3.3 Tool description

See ISO/IEC 14496-3 subpart 2 speech coding – HVXC.

10.4 TwinVQ

10.4.1 Syntax

Table 254: Syntax of TVQ_frame()

Syntax	No. of bits	Mnemonic
TVQ_frame()		
{		
Error_Sensitivity_Category1();		
Error_Sensitivity_Category2();		
Error_Sensitivity_Category3();		
Error_Sensitivity_Category4();		
}		

Table 255: Syntax of Error_Sensitivity_Category1()

Syntax	No. of bits	Mnemonic
Error_Sensitivity_Category1()		
{		
window_sequence;	**2**	**bslbf**
window_shape;	**1**	**bslbf**
if(this_layer_stereo) {		
ms_mask_present;	**2**	**bslbf**
if(ms_mask_present == 1) {		
if (window_sequence == EIGHT_SHORT_SEQUENCE)		
scale_factor_grouping;	**7**	**bslbf**
ms_data();		
}		
}		
for(ch=0; ch< (this_layer_stereo ? 2:1); ch++) {		
ltp_data_present;	**1**	**bsblf**
if (ltp_data_present)		
ltp_data ();		
tns_data_present;	**1**	**bslbf**
if(tns_data_present)		
tns_data();		
}		
bandlimit_present;	**1**	**uimsbf**
if (window_sequence != EIGHT_SHORT_SEQUENCE){		
ppc_present;	**1**	**uimsbf**
postprocess_present;	**1**	**uimsbf**
}		
if (bandlimit_present){		
for (i_ch=0; i_ch<n_ch; i_ch++){		

index_blim_h[i_ch];	**2**	**uimsbf**
index_blim_l[i_ch];	**1**	**uimsbf**
}		
}		
if (ppc_present){		
for (idiv=0; idiv<N_DIV_P; idiv++){		
index_shape0_p[idiv];	**7**	**uimsbf**
index_shape1_p[idiv];	**7**	**uimsbf**
}		
for (i_ch=0; i_ch<n_ch; i_ch++){		
index_pit[i_ch];	**8**	**uimsbf**
index_pgain[i_ch];	**7**	**uimsbf**
}		
}		
for (i_ch=0; i_ch<n_ch; i_ch++){		
index_gain[i_ch];	**9**	**uimsbf**
if (N_SF[b_type]>1){		
for (isbm=0; isbm<N_SF[b_type]; isbm++){		
index_gain_sb[i_ch][isbm];	**4**	**uimsbf**
}		
}		
}		
for (i_ch=0; i_ch<n_ch; i_ch++){		
index_lsp0[i_ch];	**1**	**uimsbf**
index_lsp1[i_ch];	**6**	**uimsbf**
for (isplt=0; isplt<LSP_SPLIT; isplit++){		
index_lsp2[i_ch][isplt];	**4**	**uimsbf**
}		
}		
for (i_ch=0; i_ch<n_ch; i_ch++){		
for (isb=0; isb<N_SF; isb++){		
for (ifdiv=0; ifdiv<FW_N_DIV; ifdiv++){		
index_env[i_ch][isb][ifdiv];	**0,6**	**uimsbf**
}		
}		
}		
for (i_ch=0; i_ch<n_ch; i_ch++){		
for (isbm=0; isbm<N_SF; isbm++){		
index_fw_alf[i_ch][isbm];	**0,1**	**uimsbf**
}		
}		
}		

Table 256: Syntax of Error_Sensitivity_Category2()

Syntax	No. of bits	Mnemonic
Error_Sensitivity_Category2()		
{		
for (idiv=0; idiv<N_DIV; idiv++){		
index_shape0[idiv];	**5/6**	**uimsbf**
index_shape1[idiv];	**5/6**	**uimsbf**
}		
}		

Table 257: Syntax of Error_Sensitivity_Category3()

Syntax	No. of bits	Mnemonic
Error_Sensitivity_Category3()		
{		
if(this_layer_stereo) {		
ms_mask_present;	**2**	**bslbf**
if(ms_mask_present == 1) {		
ms_data();		
}		
}		
for (i_ch=0; i_ch<n_ch; i_ch++){		
fb_shift[i_ch];	**2**	**uimsbf**
}		
for (i_ch=0; i_ch<n_ch; i_ch++){		
index_gain[i_ch];	**8**	**uimsbf**
if (N_SF[b_type]>1){		
for (isbm=0; isbm<N_SF[b_type]; isbm++){		
index_gain_sb[i_ch][isbm];	**4**	**uimsbf**
}		
}		
}		
for (i_ch=0; i_ch<n_ch; i_ch++){		
index_lsp0[i_ch];	**1**	**uimsbf**
index_lsp1[i_ch]	**6**	**uimsbf**
for (isplt=0; isplt<LSP_SPLIT; isplit++){		
index_lsp2[i_ch][isplt];	**4**	**uimsbf**
}		
}		
for (i_ch=0; i_ch<n_ch; i_ch++){		
for (isb=0; isb<N_SF; isb++){		
for (ifdiv=0; ifdiv<FW_N_DIV; ifdiv++){		
index_env[i_ch][isb][ifdiv];	**0,6**	**uimsbf**
}		
}		
}		
for (i_ch=0; i_ch<n_ch; i_ch++){		
for (isbm=0; isbm<N_SF; isbm++){		
index_fw_alf[i_ch][isbm];	**0,1**	**uimsbf**
}		
}		
}		

Table 258: Syntax of Error_Sensitivity_Category4()

Syntax	No. of bits	Mnemonic
Error_Sensitivity_Category4()		
{		
for (idiv=0; idiv<N_DIV; idiv++){		
index_shape0[idiv];	**5/6**	**uimsbf**
index_shape1[idiv];	**5/6**	**uimsbf**
}		
}		

Table 259: Syntax of ltp_data()

Syntax	No. of bits	Mnemonic
ltp_data()		
{		
ltp_lag;	**11**	**uimsbf**
ltp_coef;	**3**	**uimsbf**
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<max_sfb; sfb++) {		
ltp_long_used[sfb];	**1**	**uimsbf**
}		
}		
}		

Table 260: Syntax of tns_data()

Syntax	No. of bits	Mnemonic
tns_data()		
{		
for (w=0; w<num_windows; w++) {		
n_filt[w];	**1..2**	**uimsbf**
if (n_filt[w])		
coef_res[w];	**1**	**uimsbf**
for (filt=0; filt<n_filt[w]; filt++) {		
length[w][filt];	**{4;6}**	**uimsbf**
order[w][filt];	**{3;5}**	**uimsbf**
if (order[w][filt]) {		
direction[w][filt];	**1**	**uimsbf**
coef_compress[w][filt];	**1**	**uimsbf**
for (i=0; i<order[w][filt]; i++)		
coef[w][filt][i];	**2..4**	**uimsbf**
}		
}		
}		
}		

Table 261: Syntax of ms_data()

Syntax	No. of bits	Mnemonic
ms_data()		
{		
for(g=0; g<num_window_groups; g++) {		
for(sfb=last_max_sfb_ms; sfb<max_sfb; sfb++) {		
ms_used[g][sfb];	**1**	**bslbf**
}		
}		
}		

10.4.2 General information

There are no differences to ISO/IEC 14496-3 except of the bitstream element order.

10.4.3 Tool description

There are no differences to ISO/IEC 14496-3 except of the bitstream element order.

10.5 AAC

10.5.1 Syntax

Table 262: Syntax of top-level payload for audio object types ER AAC LC, ER AAC LTP and ER AAC LD

Syntax	No. of bits	Mnemonic

```
er_raw_data_block()
{
    if ( channelConfiguration == 0 ) {
        /* reserved */
    }
    if (  channelConfiguration == 1 ) {
        single_channel_element ();
    }
    if (  channelConfiguration == 2 ) {
        channel_pair_element ();
    }
    if ( channelConfiguration == 3 ) {
        single_channel_element ();
        channel_pair_element ();
    }
    if (  channelConfiguration == 4 ) {
        single_channel_element ();
        channel_pair_element ();
        single_channel_element ();
    }
    if (  channelConfiguration == 5 ) {
        single_channel_element ();
        channel_pair_element ();
        channel_pair_element ();
    }
    if (  channelConfiguration == 6 ) {
        single_channel_element ();
        channel_pair_element ();
        channel_pair_element ();
        lfe_channel_element ();
    }
    if (  channelConfiguration == 7 ) {
        single_channel_element ();
        channel_pair_element ();
        channel_pair_element ();
        channel_pair_element ();
        lfe_channel_element ();
    }
    if (  channelConfiguration >= 8 ) {
        /* reserved */
    }
```

```
    cnt = bits_to_decode() / 8;
    while ( cnt >= 1 ) {
        cnt -= extension_payload(cnt);
    }
}
```

10.5.2 General information

For AAC, two kinds of bitstream payload syntax are available: scalable and multichannel. The following changes have to be applied to them:

- **Multichannel AAC**: The syntax of the top-level payload has been modified. raw_data_block() is replaced by er_raw_data_block() as described in Table 262. Please note, that due to this modification coupling_channel_element (), data_stream_element (), program_config_element (), and fill_element () are not supported within the error resilient bitstream syntax.

- **Scalable AAC:** The syntax of aac_scalable_main_element () is not changed for error resilience.

No other changes regarding syntax occur.

Data elements are subdivided into different categories depending on its error sensitivity and collected in instances of these categories.

Depending on epConfig, there are three ways to obtain these instances on decoder site as described in section 6.3.5.

Both, the order of these instances within an error resilient AAC frame and the order of data elements inside these instances, are described within the next section. If separate access units are used, the dependency structure between elementary streams has to be set up according to the order of instances.

10.5.2.1 Error sensitivity category assignment

The following table gives an overview about the error sensitive categories used for AAC (channel_pair_element() = CPE, individual_channel_stream() = ICS, extension_payload() = EPL):

Table 263 : Error sensitivity category assignment

category	payload	mandatory	leads / may lead to one instance per	description
0	main	yes	CPE	commonly used side information
1	main	yes	ICS	channel dependent side information
2	main	no	ICS	error resilient scale factor data
3	main	no	ICS	TNS data
4	main	yes	ICS	spectral data
5	extended	no	EPL	extension type
6	extended	no	EPL	DRC data
7	extended	no	EPL	bit stuffing

Table 265 shows the category assignment for the main payload (supported elements are SCE, LFE, and CPE). Within this table "-" means that this data element does not occur within this configuration.

Table 266 shows the category assignment for the extended payload.

10.5.2.2 Category instances and its dependency structure

The subdivision into instances is done on a frame basis, in case of scalable syntax in addition on a layer basis.

The order of instances within the error resilient AAC frame/layer as well as the dependency structure in case of several elementary streams is assigned according to the following rules:

Table 264: Dependency structure within the ER AAC payload

hierarchy level	error resilient multi-channel syntax	error resilience scalable syntax
frame / layer	base payload followed by extension payload	
base payload	order of syntactic elements follows order stated in Table 262	commonly used side information followed by ICSs
extended payload	no rule regarding the order of multiple EPLs is given, the kind of extension payload can be identified by extension_type	
syntactic element in base payload	commonly used side information followed by ICSs	-
ICS order in case of stereo	left channel followed by right channel	
ICS / EPL	dependency structure according to category numbers	

Figure 31 shows an example for the error resilient multi-channel syntax. The order of data elements inside each instance is based on the syntax description, i. e. the logical order is kept.

Note: If the reversible variable length coding (RVLC) tool is not used (aacScalefactorDataResilienceFlag == 0), the decoder does not expect instances of error sensitivity category 3, i. e. no access unit shall contain those instances.

10.5.3 Tables

Table 265: AAC error sensitivity category assignment for main payload

SCE, LFE	CPE, common_window == 0	CPE, common_window == 1	data_element	function
1	-	0	max_sfb	aac_scalable_extension_header()
-	-	0	ms_mask_present	aac_scalable_extension_header()
1	-	1	tns_data_present	aac_scalable_extension_header()
1	-	0	ics_reserved_bit	aac_scalable_main_header()
1	-	1	ltp_data_present	aac_scalable_main_header()
1	-	0	max_sfb	aac_scalable_main_header()
-	-	0	ms_mask_present	aac_scalable_main_header()
1	-	0	scale_factor_grouping	aac_scalable_main_header()
1	-	0	tns_channel_mono_layer	aac_scalable_main_header()
1	-	1	tns_data_present	aac_scalable_main_header()
1	-	0	window_sequence	aac_scalable_main_header()
1	-	0	window_shape	aac_scalable_main_header()
-	0	0	common_window	channel_pair_element()
-	0	0	element_instance_tag	channel_pair_element()
-	0	0	ms_mask_present	channel_pair_element()
-	0	0	ms_used	channel_pair_element()
1	1	1	diff_control	diff_control_data()
1	1	1	diff_control_lr	diff_control_data_lr()
1	1	0	ics_reserved_bit	ics_info()
1	1	1	ltp_data_present	ics_info()
1	1	0	max_sfb	ics_info()
1	1	0	predictor_data_present	ics_info()
1	1	0	scale_factor_grouping	ics_info()
1	1	0	window_sequence	ics_info()
1	1	0	window_shape	ics_info()
1	1	1	gain_control_data_present	individual_channel_stream()
1	1	1	global_gain	individual_channel_stream()
1	1	1	length_of_longest_codeword	individual_channel_stream()
1	1	1	length_of_reordered_spectral_data	individual_channel_stream()
1	1	1	pulse_data_present	individual_channel_stream()
1	1	1	tns_data_present	individual_channel_stream()
1	-	-	element_instance_tag	lfe_channel_element()
1	1	1	ltp_coef	ltp_data()
1	1	1	ltp_lag	ltp_data()
1	1	1	ltp_lag_update	ltp_data()
1	1	1	ltp_long_used	ltp_data()
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()
-	-	0	ms_used	ms_data()
1	1	1	number_pulse	pulse_data()

SCE, LFE	CPE, common_window == 0	CPE, common_window == 1	data_element	function
1	1	1	pulse_amp	pulse_data()
1	1	1	pulse_offset	pulse_data()
1	1	1	pulse_start_sfb	pulse_data()
4	4	4	reordered_spectral_data	reordered_spectral_data()
1	1	1	dpcm_noise_last_position	scale_factor_data()
1	1	1	dpcm_noise_nrg	scale_factor_data()
1	1	1	hcod_sf	scale_factor_data()
1	1	1	length_of_rvlc_escapes	scale_factor_data()
1	1	1	length_of_rvlc_sf	scale_factor_data()
1	1	1	rev_global_gain	scale_factor_data()
2	2	2	rvlc_cod_sf	scale_factor_data()
2	2	2	rvlc_esc_sf	scale_factor_data()
1	1	1	sf_concealment	scale_factor_data()
1	1	1	sf_escapes_present	scale_factor_data()
1	1	1	sect_cb	section_data()
1	1	1	sect_len_incr	section_data()
1	-	-	element_instance_tag	single_channel_element()
4	4	4	hcod	spectral_data()
4	4	4	hcod_esc_y	spectral_data()
4	4	4	hcod_esc_z	spectral_data()
4	4	4	pair_sign_bits	spectral_data()
4	4	4	quad_sign_bits	spectral_data()
3	3	3	coef	tns_data()
3	3	3	coef_compress	tns_data()
3	3	3	coef_res	tns_data()
3	3	3	direction	tns_data()
3	3	3	length	tns_data()
3	3	3	n_filt	tns_data()
3	3	3	order	tns_data()

Table 266: AAC error sensitivity category assignment for extended payload

extension_payload	data_element	function
6	drc_band_top	dynamic_range_info()
6	drc_bands_incr	dynamic_range_info()
6	drc_bands_present	dynamic_range_info()
6	drc_bands_reserved_bits	dynamic_range_info()
6	drc_tag_reserved_bits	dynamic_range_info()
6	dyn_rng_ct	dynamic_range_info()
6	dyn_rng_sgn	dynamic_range_info()
6	excluded_chns_present	dynamic_range_info()
6	pce_instance_tag	dynamic_range_info()
6	pce_tag_present	dynamic_range_info()
6	prog_ref_level	dynamic_range_info()
6	prog_ref_level_present	dynamic_range_info()
6	prog_ref_level_reserved_bits	dynamic_range_info()
6	additional_excluded_chns	excluded_channels()
6	exclude_mask	excluded_channels()
5	extension_type	extension_payload()
7	fill_byte	extension_payload()
7	fill_nibble	extension_payload()
7	other_bits	extension_payload()

10.5.4 Figures

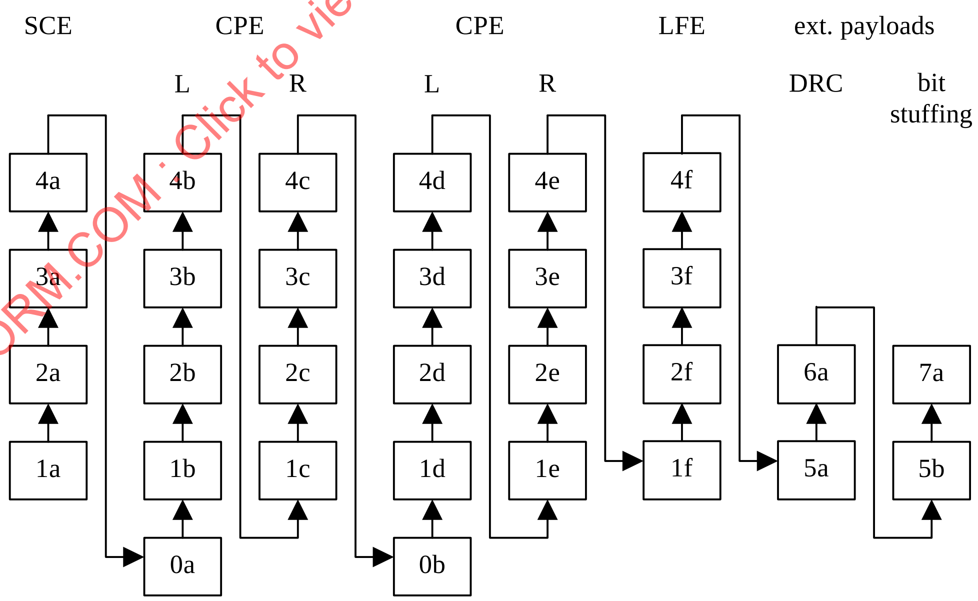


Figure 31: Dependency structure in case of error resilient multichannel AAC syntax (channelConfiguration == 6)

10.6 ER HILN

Since HILN is not included in Version 1, error sensitivity categories (ESC) are already defined in the parametric bitstream as described in Subclause 7.3 and thus no additional bitstream reordering is required. Below, the ordering of the different ESCx is described for the four different modes PARAmode == 0, 1, 2, 3.

- **PARAmode == 0 (HVXC only)**
  HVXC: ESC0 ESC1 ESC2 ESC3 ESC4
- **PARAmode == 1 (HILN only)**
  PARA/HILN: ESC0 ESC1 ESC2 ESC3 ESC4
- **PARAmode == 2 (switched HVXC / HILN)**
  PARA/HILN: ESC0 [ESC1 ESC2 ESC3 ESC4]
  HVXC 1/double: [ESC0 ESC1 ESC2 ESC3 ESC4]
  HVXC 2/double: [ESC0 ESC1 ESC2 ESC3 ESC4]
- **PARAmode == 3 (mixed HVXC / HILN)**
  PARA/HILN: ESC0 [ESC1 ESC2 ESC3 ESC4]
  HVXC 1/double: [ESC0 ESC1 ESC2 ESC3 ESC4]
  HVXC 2/double: [ESC0 ESC1 ESC2 ESC3 ESC4]

In PARAmode 3 for example, the ordering is PARA/HILN ESC0, ESC1, …, ESC4, followed by HVXC 1/double ESC0, ESC1, …, ESC4 and HVXC 2/double ESC0, ESC1, …, ESC4.

The ESC0 of “PARA/HILN” consists of the PARAswitchMode or PARAmixMode bitstream element in PARAframe() followed by the bitstream elements in HILNbasicFrameESC0(). The actual presence of these bitstream elements can depend on the current values of PARAmode, PARAswitchMode, and PARAmixMode. “HVXC 1/double” and “HVXC 2/double” denote the first and second ErHVXCfixframe() within an ErHVXCdoubleframe() respectively. The presence of ESCs in squared brackets depends on the value of PARAswitchMode or PARAmixMode in the current frame.

For the HILN enhancement and extension layer Elementary Streams, no error sensitivity classes are defined and all bitstream elements of HILNenexFrame() are handled as “ESC0”.

10.7 ER BSAC

BSAC has the layered structure where the syntax is arranged in order of importance in order to support the fine grain scalability and error resilience. Therefore the BSAC syntax can be channel coded effectively without the bitstream reordering for advanced channel coding techniques like unequal error protection (UEP) since the error resilient syntax are included in the subclause 8.3.2. However, error sensitivity categories (ESC) of the bitstream elements should be defined for advanced channel coding. The bitstream element can be classfied into the error sensitivity categories depending upon its error sensitivity as follows:

Table 267: BSAC error sensitivity category assignment

category	bitstream elements	description
0	frame_length, bsac_header() and general_header()	commonly used side information
1	bsac_layer_element(0)	BSAC base layer except common side
2	bsac_layer_element(1) –	1st quarternary enhancement layers
3	bsac_layer_element(top_layer/4+1) –	2nd quarternary enhancement layers
4	bsac_layer_element(top_layer/2+1) ~	3rd quarternary enhancement layers
5	basc_layer_element(top_layer*3/4+1) ~	4th quarternary enhancement layers

The lower category indicates the class with the higher error sensitivity, whereas the higher category indicates the class with the lower sensitivity.

11 Silence Compression Tool

11.1 Overview of the silence compression tool

The silence compression tool comprises a Voice Activity Detection (VAD) module, a Discontinuous Transmission (DTX) unit and a Comfort Noise Generator (CNG) module. The tool encodes/decodes the input signal at a lower bitrates during the non-active (silent) frames. During the active-voice (speech) frames, MPEG-4 CELP encoding and decoding are used. A block diagram of the CELP codec system with the silence compression tool is depicted in Figure 32.

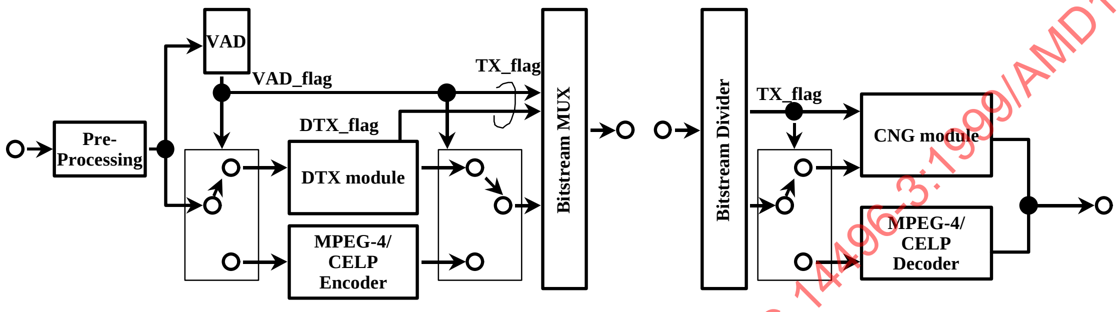


Figure 32: Block diagram of the codec system with the silence compression tool

At the transmission side, the DTX module encodes the input speech during the non-active frames. During the active-voice frames, the MPEG-4 CELP encoder is used. The voice activity flag (VAD_flag) indicating a non-active frame (VAD_flag=0) or an active-voice frame (VAD_flag=1) is determined from the input speech by the VAD module. During the non-active frames, the DTX module detects frames where the input characteristics change (DTX_flag=1 and 2: Change, DTX_flag=0: No Change). When a change is detected, the DTX module encodes the input speech to generate a SID (Silence Insertion Descriptor) information. The VAD_flag and the DTX_flag are sent together as a TX_flag to the decoder to keep synchronization between the encoder and the decoder.

At the receive side, the CNG module generates comfort noise based on the SID information during the non-active frames. During the active-voice frames, the MPEG-4 CELP decoder is used instead. Either the CNG module or the MPEG-4 CELP decoder is selected according to the TX_flag.

The SID information and the TX_flag are transmitted only when a change of the input characteristics is detected. Otherwise only the TX_flag is transmitted during non-active frames.

11.2 Definitions

CNG: Comfort Noise Generation

Coding mode: “I” for the RPE and “II” for the MPE (see ISO/IEC 14496-3:1999, Table 3.1)

DTX: Discontinuous Transmission

LP: Linear Prediction

LPCs: LP Coefficients

MPE: Multi-Pulse Excitation

MPE_Configuration: see ISO/IEC 14496-3:1999, subclause 3.4

RMS: root mean square

RPE: Regular-Pulse Excitation

RPE_Configuration: see ISO/IEC 14496-3:1999, subclause 3.4

SID: Silence Insertion Descriptor

SID frame: frame where the SID information is sent/received

signal mode: mode determined based on the average pitch prediction gain (see ISO/IEC 14496-3:1999, subclause 3.4)

VAD: Voice Activity Detection

11.3 Specifications of the silence compression tool

11.3.1 Transmission payload

There are four types of transmission payloads depending on the VAD/DTX decision. A TX_flag indicates the type of transmission payload and is determined by the VAD_flag and the DTX_flag as shown in Table 268. When the TX_flag indicates a active-voice frame (TX_flag=1), information generated by the MPEG-4 CELP encoder and the TX_flag are transmitted. When the TX_flag indicates a transition frame between an active-voice frame and a non-active frame, or a non-active frame in which the spectral characteristics of the input signal changes (TX_flag=2), High-Rate (HR) SID information and the TX_flag are transmitted to update the CNG parameters. When the TX_flag indicates a non-active frame in which the frame power of the input signal changes (TX_flag=3), Low-Rate (LR) SID information and the TX_flag are transmitted. Other non-active frames are categorized into the fourth type of the TX_flag (TX_flag=0). In this case, only the TX_flag is transmitted. Examples of the TX_flag change according to the VAD_flag and the DTX_flag are shown in Table 269.

Table 268: Relation among flags for the silence compression tool

Flags	Active-voice	Non-active		
VAD_flag	1	0		
DTX_flag	*[1]	0	1	2
TX_flag	1	0	2	3

Table 269: Examples of the TX_flag change according to the VAD_flag and the DTX_flag

Frame #	...k-5	k-4	k-3	k-2	k-1	k	k+1	k+2	k+3	k+4	k+5	k+6	k+7
VAD_flag	...1	1	1	1	0	0	0	0	0	0	0	0	0...
DTX_flag	...-*[1]	-	-	-	1	0	0	0	1	0	0	0	2...
TX_flag	...1	1	1	1	2	0	0	0	2	0	0	0	3...

	active-voice period	non-active period

11.3.2 Bitrates of the silence compression tool

During non-active frames, the silence compression tool operates at bitrates shown in Table 270. The bitrate depends on the coding mode defined in ISO/IEC 14496-3:1999, Table 3.1, the sampling rate and the frame length.

[1] The DTX_flag is not determined during the active-voice frames (VAD_flag = 1).

Table 270: Bitrates for the silence compression tool

Coding mode	Sampling rate [kHz]	Band width scalability	Frame length [ms]	Bitrate [bit/s]		
				TX_flag	HR-SID	LR-SID
I (RPE)	16	-	15	133	2533	400
			10	200	3800	600
II (MPE)	8	On*[2], Off	40	50	525	150
			30	67	700	200
			20	100	1050	300
			10	200	2100	600
	16	Off	20	100	1900	300
			10	200	3800	600
		On	40	50	1050	150
			30	67	1400	200
			20	100	2100	300
			10	200	4200	600

11.3.3 Algorithmic delay of the silence compression tool

The algorithmic delay is the same as that of MPEG-4 version 1 CELP, since the same frame length and the same additional look ahead length are used.

11.4 Syntax

This section describes the bitstream syntax and the bitstream semantics for the silence compression tool.

The decoder configuration information for the ER CELP object is transmitted in the DecoderConfigDescriptor() of the base layer and the optional enhancement layer Elementary Stream.

Error Resilient CELP Base Layer -- Configuration

The CELP core in the unscalable mode or as the base layer in the scalable mode requires the following ErrorResilientCelpSpecificConfig():

```
ErrorResilientCelpSpecificConfig () {
   ER_SC_CelpHeader (samplingFrequencyIndex);
}
```

Error Resilient CELP Enhancement Layer -- Configuration

The CELP core is used for both bitrate and bandwidth scalable modes. In the bitrate scalable mode, the enhancement layer requires no ErrorResilientCelpSpecificConfig(). In the bandwidth scalable mode, the enhancement layer has the following ErrorResilientCelpSpecificConfig():

```
ErrorResilientCelpSpecificConfig() {
   CelpBWSenhHeader(); /* Defined in ISO/IEC 14496-3 subpart 3.*/
}
```

The payload data for the ER CELP object is transmitted as AL-PDU payload in the base layer and the optional enhancement layer Elementary Stream.

[2] This occurs when decoding from BWS bitstreams is performed.

Error Resilient CELP Base Layer -- Access Unit payload

```
alPduPayload
{
    ER_SC_CelpBaseFrame();
}
```

Error Resilient CELP Enhancement Layer -- Access Unit payload

To parse and decode the Error Resilient CELP enhancement layers, information decoded from the Error Resilient CELP base layer is required. For the bitrate scalable mode, the following data for the Error Resilient CELP enhancement layers has to be included:

```
alPduPayload
{
    SC_CelpBRSenhFrame();
}
```

For the bandwidth scalable mode, the following data for the Error Resilient CELP enhancement layer has to be included:

```
alPduPayload
{
    SC_CelpBWSenhFrame();
}
```

11.4.1 Bitstream syntax

The bitstream syntax of ER_SC_CelpHeader(), ER_SC_CelpBaseFrame(), SID_LSP_VQ(), ER_SC_CelpBRSenhFrame(), ER_SC_CelpBWSenhFrame(), SID_NarrowBand_LSP(), SID_BandScalable_LSP(), SID_WideBand_LSP() and SID_MPE_frame() are shown in Table 271 through Table 279.

Table 271: Syntax of ER_SC_CelpHeader()

Syntax	No. of bits	Mnemonic
ER_SC_CelpHeader (samplingFrequencyIndex)		
{		
ExcitationMode;	**1**	**uimsbf**
SampleRateMode;	**1**	**uimsbf**
FineRateControl;	**1**	**uimsbf**
SilenceCompression;	**1**	**uimsbf**
if (ExcitationMode == RPE) {		
RPE_Configuration;	**3**	**uimsbf**
}		
if (ExcitationMode == MPE) {		
MPE_Configuration;	**5**	**uimsbf**
NumEnhLayers;	**2**	**uimsbf**
BandwidthScalabilityMode	**1**	**uimsbf**
}		
}		

Table 272: Syntax of ER_SC_CelpBaseFrame()

Syntax	No. of bits	Mnemonic
ER_SC_CelpBaseFrame()		
{		
if (SilenceCompression == OFF) {		
ER_CelpBaseFrame();		
} else {		
TX_flag;	**2**	**uimsbf**
if (TX_flag == 1) {		
ER_CelpBaseFrame();		
} else if (TX_flag == 2) {		
SID_LSP_VQ();		
SID_Frame();		
} else if (TX_flag == 3) {		
SID_Frame();		
}		
}		
}		

Table 273: Syntax of SID_LSP_VQ()

Syntax	No. of bits	Mnemonic
SID_LSP_VQ()		
{		
if (SampleRateMode == 8kHz) {		
SID_NarrowBand_LSP();		
} else {		
SID_WideBand_LSP();		
}		
}		

Table 274: Syntax of ER_SC_CelpBRSenhFrame()

Syntax	No. of bits	Mnemonic
ER_SC_CelpBRSenhFrame()		
{		
if (SilenceCompression == OFF) {		
CelpBRSenhFrame();		
} else if (TX_flag == 1) {		
CelpBRSenhFrame();		
}		
}		

Table 275: Syntax of ER_SC_CelpBWSenhFrame()

Syntax	No. of bits	Mnemonic
ER_SC_CelpBWSenhFrame()		
{		
if (SilenceCompression == OFF) {		
CelpBWSenhFrame();		
} else {		
if (TX_flag == 1) {		
CelpBWSenhFrame();		
}		
if (TX_flag == 2) {		
SID_BandScalable_LSP();		
}		
}		
}		

Table 276: Syntax of SID_NarrowBand_LSP()

Syntax	No. of bits	Mnemonic
SID_NarrowBand_LSP()		
{		
SID_lpc_indices [0];	**4**	**uimsbf**
SID_lpc_indices [1];	**4**	**uimsbf**
SID_lpc_indices [2];	**7**	**uimsbf**
}		

Table 277: Syntax of SID_BandScalable_LSP()

Syntax	No. of bits	Mnemonic
SID_BandScalable_LSP()		
{		
SID_lpc_indices [3];	**4**	**uimsbf**
SID_lpc_indices [4];	**7**	**uimsbf**
SID_lpc_indices [5];	**4**	**uimsbf**
SID_lpc_indices [6];	**6**	**uimsbf**
}		

Table 278: Syntax of SID_WideBand_LSP()

Syntax	No. of bits	Mnemonic
SID_WideBand_LSP()		
{		
SID_lpc_indices [0];	**5**	**uimsbf**
SID_lpc_indices [1];	**5**	**uimsbf**
SID_lpc_indices [2];	**7**	**uimsbf**
SID_lpc_indices [3];	**7**	**uimsbf**
SID_lpc_indices [4];	**4**	**uimsbf**
SID_lpc_indices [5];	**4**	**uimsbf**
}		

Table 279: Syntax of SID_MPE_frame()

Syntax	No. of bits	Mnemonic
SID_frame()		
{		
SID_rms_index;	**6**	**uimsbf**
}		

11.4.2 Bitstream semantics

SilenceCompression A one bit identifier indicating whether Silence Compression is used or not.

SilenceCompression	SilenceCompressionID	Description
0	SC_OFF	SilenceCompression is disabled
1	SC_ON	SilenceCompression is enabled

Bitstream semantics for the silence compression tool are shown in Table 280.

Table 280: Bitstream semantics for the silence compression tool

HR/LR-SID	Coding mode	Sampling rate [kHz]	Band width scalability	Parameter	Description
HR-SID	I	16	Off	TX_flag SID_rms_index SID_lpc_indices[0] SID_lpc_indices[1] SID_lpc_indices[2] SID_lpc_indices[3] SID_lpc_indices[4] SID_lpc_indices[5]	Two bits indicating transmission mode Frame energy 0-4th LSPs of the 1st stage VQ 5-9th LSPs of the 1st stage VQ 10-14th LSPs of the 1st stage VQ 15-19th LSPs of the 1st stage VQ 0-4th LSPs of the 2nd stage VQ 5-9th LSPs of the 2nd stage VQ
	II	8	On, Off	TX_flag SID_rms_index SID_lpc_indices[0] SID_lpc_indices[1] SID_lpc_indices[2]	Two bits indicating transmission mode Frame energy 0-4th LSPs of the 1st stage VQ 5-9th LSPs of the 1st stage VQ 0-4th LSPs of the 2nd stage VQ
		16	On	TX_flag SID_rms_index SID_lpc_indices[3] SID_lpc_indices[4] SID_lpc_indices[5] SID_lpc_indices[6]	Two bits indicating transmission mode Frame energy 0-9th LSPs of 1st stage VQ 10-19th LSPs of 1st stage VQ 0-4th LSPs of 2nd stage VQ 5-9th LSPs of 2nd stage VQ
			Off	TX_flag SID_rms_index SID_lpc_indices[0] SID_lpc_indices[1] SID_lpc_indices[2] SID_lpc_indices[3] SID_lpc_indices[4] SID_lpc_indices[5]	Two bits indicating transmission mode Frame energy 0-4th LSPs of 1st stage VQ 5-9th LSPs of 1st stage VQ 10-14th LSPs of 1st stage VQ 15-19th LSPs of 1st stage VQ 0-4th LSPs of 2nd stage VQ 5-9th LSPs of 2nd stage VQ
LR-SID	I, II	8, 16	On, Off	TX_flag SID_rms_index	Two bits indicating transmission mode Frame energy

11.4.3 LSP transmission

In case the silence compression tool is used in combination with FineRate Control enabled, a CELP frame with LPC_present = 1 and interpolation_flag = 0 must be transmitted in the first voice-active frame following a non-active frame. Voice-active frames are signaled by TX_flag = 1, non-active frames are signaled by TX_flag = 0, 2 or 3.

11.5 CNG module

Figure 33 depicts a structure of the CNG module that generates the comfort noise. The comfort noise is generated by filtering an excitation with an LP synthesis filter in a similar manner as with voice-active speech signals. The post filter may be used to improve the coding quality. The excitation is given by adding a multi-pulse excitation or a regular pulse excitation and a random excitation, scaled by their corresponding gains. The excitations are generated based on a random sequence independent of the SID information. The coefficients for the LP synthesis filter and gains are calculated from LSPs and a RMS value (frame energy) respectively, which are received as the SID information. The LSPs and the RMS are smoothed in order to improve the coding quality for the noisy input speech. The CNG module uses the same frame and subframe sizes as those in active speech frames. Processing in each part is described in the following sub-clauses.

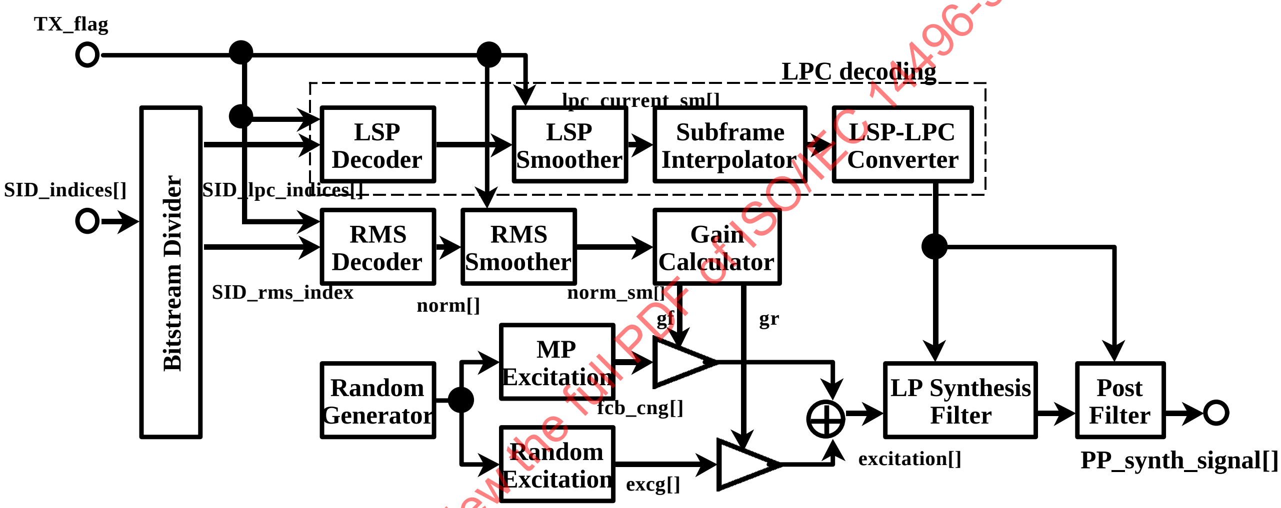


Figure 33: CNG module

11.5.1 Definitions

Input

TX_flag This field contains the transmission mode.
SID_lpc_indices[] This array contains the packed LP indices. The dimension is 3, 5 or 6 (see Table 280).
SID_rms_index This field contains the RMS index.

Output

PP_synth_signal[] This array contains the post-filtered (enhanced) speech signal. The dimension is *sbfrm_size*.

The following are help elements used in the CNG module:

lpc_order: the order of LP

sbfrm_size: the number of samples in a subframe

n_subframe: the number of subframes in a frame *int_Q_lpc_coefficients*[]: interpolated LPCs (see ISO/IEC 14496-3:1999, subclause 3.5.8.2).

11.5.2 LSP decoder

LSP *lpc_current*[] is decoded from the LSP indices *SID_lpc_indices*[]. The decoding process is identical to that described in ISO/IEC 14496-3:1999, subclause 3.5.6 with the following exceptions:

(1) A sub-set of the *lpc_indices*[] is transmitted to the decoder. A relation between the transmitted LSP indices *SID_lpc_indices*[] and the LSP indices for MPEG-4 CELP, *lpc_indices*[] is shown in Table 281.

(2) The decoding process is not carried out for the untransmitted indices.

Table 281: LSP index relation between the silence compression tool and MPEG-4 CELP

Coding mode	Sampling rate [kHz]	Band width scalability	Silence compression tool	MPEG-4 CELP
I (RPE)	16	Off	SID_lpc_indices[0]	lpc_indices[0]
			SID_lpc_indices[1]	lpc_indices[1]
			SID_lpc_indices[2]	lpc_indices[2]
			SID_lpc_indices[3]	lpc_indices[3]
			SID_lpc_indices[4]	lpc_indices[5]
			SID_lpc_indices[5]	lpc_indices[6]
II (MPE)	8	Off	SID_lpc_indices[0]	lpc_indices[0]
			SID_lpc_indices[1]	lpc_indices[1]
			SID_lpc_indices[2]	lpc_indices[2]
	16	On	SID_lpc_indices[3]	lpc_indices[5]
			SID_lpc_indices[4]	lpc_indices[6]
			SID_lpc_indices[5]	lpc_indices[7]
			SID_lpc_indices[6]	lpc_indices[8]
		Off	SID_lpc_indices[0]	lpc_indices[0]
			SID_lpc_indices[1]	lpc_indices[1]
			SID_lpc_indices[2]	lpc_indices[2]
			SID_lpc_indices[3]	lpc_indices[3]
			SID_lpc_indices[4]	lpc_indices[5]
			SID_lpc_indices[5]	lpc_indices[6]

11.5.3 LSP smoother

Smoothed LSPs *lsp_current_sm*[] are updated using the decoded LSPs *lsp_current*[] in each frame as:

$$lsp\_current\_sm[i] = \begin{cases} 0.875\ lsp\_current\_sm[i] + 0.125\ lsp\_current[i], & TX\_flag = 2 \\ 0.875\ lsp\_current\_sm[i] + 0.125\ lsp\_current\_sid[i], & TX\_flag = 0 \text{ or } 3 \end{cases}$$

where i=0,...,*lpc_order-1* and *lsp_current_sid*[] is *lsp_current*[] in the last SID frame. At the beginning of every non-active period, *lsp_current_sm*[] is initialized with *lsp_current*[] at the end of the previous active-voice period.

11.5.4 LSP interpolation and LSP-LPC conversion

LPCs for the LP synthesis, *int_Q_lpc_coefficients*[] are calculated from the smoothed LSPs *lpc_current_sm*[] using the LSP interpolation with the stabilization and the LSP-to-LPC conversion. These processes are described in ISO/IEC 14496-3:1999, subpart 3, subclause 3.5.6. A common buffer for the previous frame *lsp_previous*[] is used in both the silence compression tool and in CELP coding for active speech frames.

11.5.5 RMS decoder

The RMS of the input speech *qxnorm* in each subframe is reconstructed using *SID_rms_index* in the same process as described in ISO/IEC 14496-3:1999, subpart 3, subclause 3.5.7.2.3.2 with the exception that the m -law parameters are independent of the signal mode and set as *rms_max=7932* and *mu_law=1024*.

The reconstructed RMS of the input speech is converted into the RMS of the excitation signal (*norm*) using the reflection coefficients *par*[] as follows and used for the gain calculation:

```
norm = (qxnorm*subfrm_size)*(qxnorm*subfrm_size);
for (i = 0; i < lpc_order; i++)
{
   norm *= (1 - par[i] * par[i])* αs;
}
```

where *par*[] is calculated from the LPCs *int_Qlpc_coefficients*[] and a scaling factor α_s is 0.8.

11.5.6 RMS smoother

A smoothed RMS *norm_sm* is updated using *norm* in each subframe as follows:

$$norm\_sm = \begin{cases} 0.875\ norm\_sm + 0.125\ norm[subnum] & for\ TX\_flag = 2\ or\ 3 \\ 0.875\ norm\_sm + 0.125\ norm\_sid & for\ TX\_flag = 0 \end{cases}$$

where *subnum* is the current subframe number ranging from 0 to *n_subframe-1* and *norm_sid* is *norm*[*n_subframe-1*] in the last SID frame. In the first frame of every non-active period, *norm_sm* is set to *norm.* During the first 40 msec of the non-active period, *norm_sm* is initialized with *nrom* [*subnum*], *w*hen TX_flag = 2 or 3 and

$$\left| 20\log_{10} norm\_sid - 20\log_{10} norm[n\_subframe-1] \right| > 6\ dB.$$

11.5.7 CNG excitation generation

The CNG excitation signal *excitation*[] is computed from a multi-pulse excitation signal and a random excitation signal as follows:

```
for (i = 0; i < sbfrm_size; i++)
{
   excitation[i] = gf * fcb_cng[i] + gr * excg[i];
}
```

where fcb_cng[] and excg[] are respectively the multi-pulse (MP) or regular pulse (RP) excitation signal and a random excitation signal. gf and gr are their corresponding gains. The random excitation signal, positions and signs of pulses for the MP/RP excitation are sequentially produced from a random sequence in each subframe. In order to synchronise the random number CNG generator in the encoder and the decoder, the random excitation signal excg[] for a given subframe must be computed prior to the MP/RP Excitation generation fcb_cng[] for that subframe.

11.5.7.1 Random sequence

Random sequence are generated by the following function and are used for generation of the multi-pulse and the random excitation signals:

```
short Random (*seed)
{
   *seed = (short) ((int)(*seed * 31821 + 13849));
   return (*seed);
```

```
}
```

with an initial seed value of 21845 and commonly used for both excitations. This generator has a periodic cycle of 16 bits. The seed is initialized with 21845 at the beginning of every non-active period.

11.5.7.2 Generation of the random excitation

The random excitation signal of each subframe is a Gaussian random sequence, which is generated as follows:

```
for (i = 0; i < sbfrm_size; i++)
{
   excg[i] = Gauss (seed);
}
```

where

```
float Gauss (short *seed)
{
   temp = 0;
   for (i = 0; i < 12; i++)
   {
      temp += (float) Random (seed);
   }
   temp /= (2 * 32768);
   return (temp);
}
```


11.5.7.3 Multi-pulse excitation generation

In case MPE is used for coding of the voice-active frames, a multi-pulse excitation signal is generated for each subframe by randomly selecting the positions and signs of pulses. Multi-pulse structures of MPEG-4 version 1 CELP with MPE_Configuration = 24 and 31 are used for the sampling rate of 8 and 16 kbit/s, respectively. Positions and signs of 10 pulses are generated in a 40-sample vector. For subframe size of 80 samples, it is carried out twice to generate 20 pulses in an 80-sample vector. Indices of the positions and signs, mp_pos_idx and mp_sign_idx, are generated in each subframe as follows:

```
if (subframe size is 40 samples)
{
   setRandomBits (&mp_pos_idx, 20, seed);
   setRandomBits (&mp_sgn_idx, 10, seed);
}

if (subframe size is 80 samples)
{
   setRandomBits (&mp_pos_idx_1st_half, 20, seed);
   setRandomBits (&mp_sgn_idx_1st_half, 10, seed);
   setRandomBits (&mp_pos_idx_2nd_half, 20, seed);
   setRandomBits (&mp_sgn_idx_2nd_half, 10, seed);
}
```

where mp_pos_idx_1st_half and mp_sgn_idx_1st_half are indices of the positions and signs of the first half of the subframe and mp_pos_idx_2nd_half and 20 mp_sgn_idx_2nd_half are indices for the second half. The function setRandomBits() is defined in section 11.5.7.5.

11.5.7.4 Regular Pulse Excitation

In case RPE is used for coding of the voice-active frames, a regular pulse excitation signal is generated for each subframe. In case of non-active frames, the content of the adaptive codebook is initialised with zero. For the non-active frames only the fixed codebook is used. The fixed codebook excitation signal is generated by using a random shape_index as input for the RPE decoding process.

```
setRandomBits (&shape_index, n_bits, seed);

rpe_index = shape_index;
rpe_phase = rpe_index % D;
rpe_index = rpe_index / D;

for (n = Np - 1; n >= 0; n--)
{
    rpe_amps [n] = (rpe_index % 3) - 1;
    rpe_index = rpe_index / 3;
}

for (n = 0; n < sbfrm_size; n++)
{
    fcb_cng[n] = 0.0F;
}

for (n = 0; n < Np; n++)
{
    fcb_cng[rpe_phase + D*n] = gn[RPE_Configuration] * (float)(rpe_amps [n]);
}
```

nbits is set to 11 for RPE_Configurations 0 and 1 and set to 12 for RPE_configurations 2 and 3. D is the decimation factor, Np is the number of pulses per subframe and sbfrm_size is the number of samples per subframe as defined in MPEG-4 CELP version 1. The normalising factor gn is defined in Table 282.

Table 282: Normalizing factor gn for RPE

RPE_Configuration	gn [ ]
0	56756 / 32768
1	56756 / 32738
2	44869 / 32768
3	40132 / 32768

11.5.7.5 Random index generator function

A random index generator function setRandomBits() is defined for MPE and RPE as follows:

```
void setRandomBits (long *l, int n, short *seed)
{
    *l = 0xffff & Random(seed);
    if (n > 16)
    {
        *l |= (0xffff & Random(seed)) << 16;
    }

    if (n < 32)
    {
        *l &= ((unsigned long)1 << n) - 1;
    }
}
```

11.5.7.6 Gain calculation

Gains *gf* and *gr* are calculated from the smoothed RMS of the excitation, *norm_sm* as follows:

$$gf = a \cdot norm\_sm \Big/ \sqrt{\sum_{i=0}^{sbfrm\_size-1} fcb\_cng(i)^2 \Big/ sbfrm\_size}\ .$$

$$gr = [-a \cdot A_3 + \sqrt{a^2 \cdot A_3^2 - (a^2 - 1)A_1 A_2}\,] \Big/ A_2$$

where $a = 0.6$ and

$$A_1 = \sum_{i=0}^{sbfrm\_size-1} gf^2\, fcb\_cng[i]^2$$

$$A_2 = \sum_{i=0}^{sbfrm\_size-1} excg[i]^2$$

$$A_3 = \sum_{i=0}^{sbfrm\_size-1} gf \cdot fcb\_cng[i] \cdot excg[i]\ .$$

11.5.8 LP Synthesis filter

The synthesis filter is identical to LP synthesis filter in MPEG-4 CELP described in ISO/IEC 14496-3:1999, subpart 3, subclause 3.5.8.

11.5.9 Memory update

Since the encoder and decoder need to be kept synchronized during non-active periods, the excitation generation is performed on both encoder and decoder sides to update the corresponding buffers for the LP synthesis. The adaptive codebook is not used and is initialized with zero during non-active frames.

12 Extension of HVXC variable rate mode

12.1 Overview

This sub-clause describes the syntax and semantics of ER HVXC object including the operation of 4.0 kbit/s variable rate coding mode of HVXC. In version-1, variable bitrate mode based on 2 kbit/s mode is already supported. Here the operation of the variable bitrate mode of 4.0 kbit/s maximum is described.

In the fixed bitrate mode, we have 2 bit V/UV decision that is:

VUV=3 : full voiced, VUV=2 : mixed voiced, VUV=1 : mixed voiced, VUV=0 : unvoiced.

When the operating mode is variable bitrate mode, VUV=1 indicates “Background noise” status instead of “mixed voiced”. The current operating mode is defined by “HVXCconfig()” and decoder knows whether it’s variable or fixed rate mode and can understand the meaning of VUV=1. In the “variable rate coding”, bit assignment is varied depending on Voiced/unvoiced decision and bitrate saving is obtained mostly by reducing the bit assignment for Unvoiced speech (VUV=0) segment. When VUV=0 is selected, then it is checked whether the segment is real “unvoiced speech” or “background noise” segments. If it is declared to be “background noise”, then VUV is changed to 1 and bit assignment to the frame is further reduced. During the “background noise” mode, only the mode bits or noise update frame is transmitted according to the change of the background noise characteristics. Using this variable rate mode, average bitrate is reduced to 56-85% of the fixed bitrate mode depending on the source items.

12.2 Definitions

VUV:	V/UV(Voiced/UnVoiced) flag
VX_Shape1[0,1]:	Stochastic codebook index
VX_Gain1[0,1]:	Stochastic codebook gain index
HVXCvarMode:	variable rate mode
HVXCrateMode:	transmission rate mode
Var_ScalableFlag:	scalable flag for variable rate mode
NUM_SUBF1:	the number of subframes in one frame
NUM_SHAPE_L0:	the number of codebook index
UpdateFlag:	background noise update flag

12.3 Syntax

This section describes the bitstream syntax and the bitstream semantics for ER HVXC object type including the extension of HVXC variable rate mode.

An MPEG-4 Natural Audio Object ER HVXC object type is transmitted in one or two Elementary Streams: The base layer stream and an optional enhancement layer stream.

The bitstream syntax is described in pseudo-C code.

12.3.1 Decoder configuration (ER HvxcSpecificConfig)

The decoder configuration information for ER HVXC object type is transmitted in the DecoderConfigDescriptor() of the base layer and the optional enhancement layer Elementary Stream.

12.3.1.1 ER HVXC base layer -- configuration

For ER HVXC object type in unscalable mode or as base layer in scalable mode the following ErrorResilientHvxcSpecificConfig() is required:

```
ErrorResilientHvxcSpecificConfig() {
    ErHVXCconfig();
}
```

12.3.1.2 ER HVXC enhancement Layer -- configuration

ER HVXC object type provides a 2 kbit/s base layer plus a 2 kbit/s enhancement layer scalable mode. In this scalable mode the basic layer configuration must be as follows:

```
HVXCrateMode = 0 HVXC 2 kbit/s;
```

For the enhancement layer, there is no ErrorResilientHvxcSpecificConfig() required:

```
ErrorResilientHvxcSpecificConfig() {
}
```

Table 283: Syntax of ErHVXCconfig()

Syntax	No. of bits	Mnemonic
ErHVXCconfig()		
{		
HVXCvarMode;	**1**	**uimsbf**
HVXCrateMode;	**2**	**uimsbf**
extensionFlag;	**1**	**uimsbf**
if(extensionFlag) {		
var_ScalableFlag;	**1**	**uimsbf**
}		
}		

Table 284: HVXCvarMode

HVXCvarMode	Description
0	HVXC fixed bitrate
1	HVXC variable bitrate

Table 285: HVXCrateMode

HVXCrateMode	HVXCrate	Description
0	2000	HVXC 2 kbit/s
1	4000	HVXC 4 kbit/s
2	3700	HVXC 3.7 kbit/s
3 (reserved)		

Table 286: var ScalableFlag

Var ScalableFlag	Description
0	HVXC variable rate non-scalable mode
1	HVXC variable rate scalable mode

Table 287: HVXC constants

NUM_SUBF1	NUM_SHAPE_L0
2	64

12.3.2 Bitstream frame (alPduPayload)

The dynamic data for ER HVXC object type is transmitted as AL-PDU payload in the base layer and the optional enhancement layer Elementary Stream.

ER HVXC Base Layer -- Access Unit payload

```
alPduPayload {
    ErHVXCframe();
}
```

ER HVXC Enhancement Layer -- Access Unit payload

To parse and decode the ER HVXC enhancement layer, information decoded from the ER HVXC base layer is required.

```
alPduPayload {
```

```
        ErHVXCenhaFrame();
    }
```

Table 288: Syntax of ErHVXCframe()

Syntax	No. of bits	Mnemonic

ErHVXCframe()
{
    if(HVXCvarMode == 0) {
        ErHVXCfixframe(HVXCrate);
    } else {
        ErHVXCvarframe(HVXCrate);
    }
}		

Table 289: Syntax of ErHVXCenhaframe

Syntax	No. of bits	Mnemonic

ErHVXCenhaframe()
{
    if(HVXCvarMode == 0) {
        ErHVXCenh_fixframe();
    } else {
        ErHVXCenh_varframe();
    }
}		

The syntax of the ErHVXCfixframe(), ErHVXCvarframe(), ErHVXCenh_fixframe(), and ErHVXCenh_varframe() are described in the subcluase 10.3.

12.4 Specification of variable rate mode

Specification for the variable rate mode is described below.

12.4.1 Transmission payload

Transmission payloads with four different bitrates are used depending on V/UV decision and the result of background noise detection. VUV flag and UpdateFlag indicate the type of transmission payloads.

VUV is a parameter that has the result of V/UV decision and defined as;

$$VUV = \begin{cases} 0 & \text{Unvoiced speech} \\ 1 & \text{Background noise interval} \\ 2 & \text{Voiced speech 1} \\ 3 & \text{Voiced speech 2} \end{cases}$$

To indicate whether or not the frame marked “VUV=1” is noise update frame, a parameter “UpdateFlag” is introduced. UpdateFlag is used only when VUV=1.

$$UpdateFlag = \begin{cases} 0 & \text{not noise update frame} \\ 1 & \text{noise update frame} \end{cases}$$

If UpdateFlag is 0, the frame is not noise update frame, and if UpdateFlag is 1, the frame is noise update frame. The first frame of the “Background noise” mode is always classified as the noise update frame. In addition, if the gain or spectral envelope of the background noise frame is changed, a noise update frame is inserted.

At the noise update frame, the average of LSP parameters over the last 3 frames is computed and coded as LSP indices in the encoder. In the same manner, the average of Celp gain over the last 4 frames (8 subframes) is computed and coded as Celp gain index.

During the background noise interval (VUV=1), LSP parameters and excitation parameters are sent only when noise update frame is selected (UpdateFlag=1). Decoder output signals for background noise interval are generated using the LSP and excitation parameters transmitted at noise update frames.

If the current frame or the previous frame is “Background noise” mode, differential mode in LSP quantization is inhibited in the encoder, because LSP parameters are not sent during “Background noise” mode and inter frame coding is not possible.

12.4.2 Bitrates of variable rate mode

Using the background noise detection method described above, variable rate coding is carried out based on fixed bitrate 4 kbit/s HVXC. The bitrate at each mode is shown below.

Mode(VUV)	Back Ground Noise(1)		UV(0)	V(2,3)
	UpdateFlag=0	UpdateFlag=1		
V/UV UpdateFlag LSP Excitation	2bit/20msec 1bit/20msec 0bit/20msec	2bit/20msec 1bit/20msec 18bit/20msec 4bit/20msec (gain only)	2bit/20msec 0bit/20msec 18bit/20msec 20bit/20msec	2bit/20msec 0bit/20msec 26bit/20msec 52bit/20msec
Total	3bit/20msec 0.15 kbit/s	25bit/20msec 1.25 kbit/s	40bit/20msec 2.0 kbit/s	80bit/20msec 4.0 kbit/s

12.5 Decoding process of variable rate mode

Decoding process for the variable rate mode is described below.

12.5.1 Decoding process

In the decoder, voiced frame (VUV=2,3) is processed in the same manner as 4 kbit/s fixed bitrate mode, and unvoiced frame (VUV=0) is processed in the same manner as 2 kbit/s fixed bitrate mode. When the background noise mode is selected (VUV=1), decoder output signal is generated in the same manner as unvoiced speech at 2 kbit/s fixed bitrate mode. The decoder parameters for back ground noise interval are generated by using the parameters transmitted at noise update frames (VUV=1, UpdateFlag=1) and sometimes at preceding unvoiced frames (VUV=0). The subclauses below show how to generate the decoder parameters for back ground noise interval.

12.5.1.1 LSP decoding

In the decoder, two sets of previously transmitted LSP parameters, prevLSP1 and prevLSP2, are held.

prevLSP1: transmitted LSP parameters

prevLSP2: transmitted LSP parameters before prevLSP1

“Background noise” mode occurs only after “unvoiced” or “background noise” mode. When the “background noise” mode is selected, LSP parameters are transmitted only when the frame is “noise update frame” (UpdateFlag=1). If new LSP parameters are transmitted, prevLSP1 is copied to prevLSP2 and newly transmitted LSPs are copied to prevLSP1 regardless of VUV decision.

LSP parameters for each frame during the "background noise" mode are generated by the interpolation between prevLSP1 and prevLSP2 using the equation:

$$qLsp(i) = ratio \cdot prevLsp1(i) + (1 - ratio) \cdot prevLsp2(i) \cdots i = 1..10 \quad (12.1)$$

where

$$ratio = \frac{2 \cdot (bgnIntval + rnd) + 1}{2 \cdot BGN\_INTVL} \quad (12.2)$$

qLsp(i) is the i-th LSP to be used for decoding operation of the current frame, prevLsp1(i) is the i-th LSP of prevLSP1, prevLsp2(i) is the i-th LSP of prevLSP2 ($1 \le i \le 10$). In this equation, bgnIntval is a counter which counts the number of consecutive background noise frames, and is reset to 0 at the receipt of background noise update frame. BGN_INTVL(=12) is a constant, and rnd is a randomly generated integer value between –3 and 3. If counter bgnIntval reaches BGN_INTVL, bgnIntvl is set to BGN_INTVL-1, and if the ratio obtained by the equation (12.2) is smaller than 0 or greater than 1, the value of rnd is set to 0 and ratio is recomputed.

12.5.1.2 Excitation generation

During the period of "background noise" mode, the Gain index (VX_gain[0]) transmitted in the noise update frame is used for all the subframes, the values of Shape index (VX_Shape1[0,1]) are randomly generated between 0 and NUM_SHAPE_L0-1. These excitation parameters are used with the interpolated LSP parameters as described above to generate the signals of background noise mode.

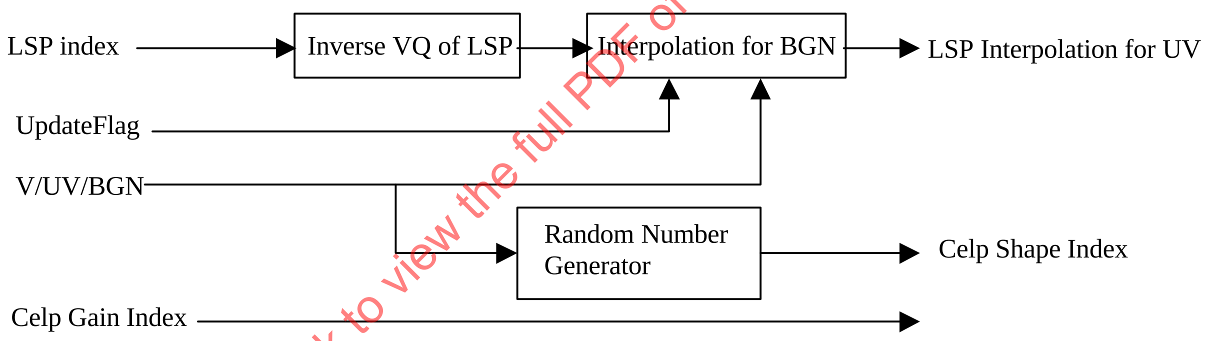


Figure 34: Additional diagram for variable rate decoder

Annex A
(informative)

Parametric audio encoder

A.1 Overview of the encoder tools

The following Figure A.1 shows a general block diagram of a parametric encoder. First the input signal is separated into the two parts which are coded by HVXC and by HILN tools. This can be done manually or automatically. Automatic switching between speech and music signals is supported (see Subclause A.3.1), allowing the use of HVXC for speech and HILN for music. For both HVXC and HILN parameter estimation and parameter encoding can be performed. A common bitstream formatter allows operation either in HVXC only, HILN only, or also in combined modes, i.e. switched or mixed mode.

The following description of an HILN parametric encoder is informative and also alternative techniques for signal separation and parameter estimation can be used in an encoder.

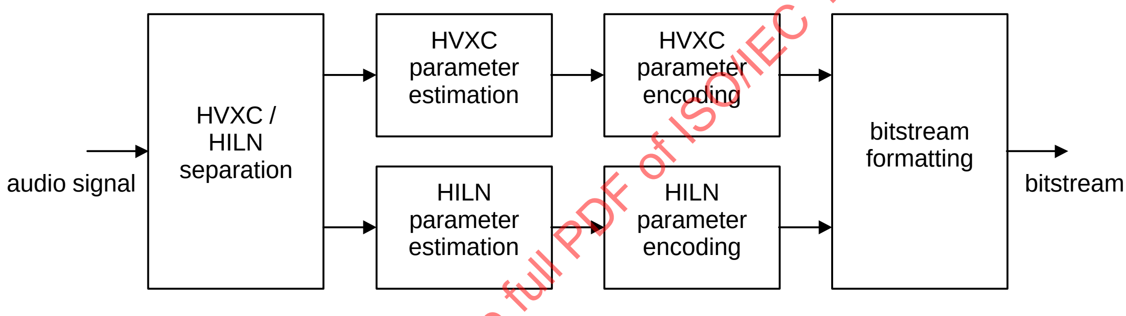


Figure A.1: General block diagram of the integrated parametric encoder

A.2 HILN encoder tools

The basic principle of the "Harmonic and Individual Lines plus Noise" (HILN) encoder is to analyze the input signal in order to extract parameters describing the signal. These parameters are coded and transmitted as a bitstream. In the decoder the output signal is synthesized based on the parameters extracted and transmitted by the encoder.

The encoder consists of two main parts: "Parameter Extraction" and "Parameter Coding". In the encoder, the input signal is divided into consecutive frames and for each frame a set of parameters describing the signal in this frame is extracted and coded. Due to this parametric description, a wide range of bitrates, sampling rates and frame lengths are possible. Typically a frame length of 32 ms is used. For input signals with 8 to 16 kHz sampling rate typically a bitrate of 6 to 16 kbit/s is used.

The "Parameter Extraction" and "Parameter Coding" is described in detail in the following subclauses.

A.2.1 HILN parameter extraction

Since different parameter sets and different synthesis techniques can be applied, the input signal of the encoder has to be split up in an appropriate way. This is performed by the *Separation* unit. Depending on the most appropriate synthesis technique, a parameter set is derived for each part of the input signal in the *Model Based Parameter Estimation* unit. The two units *Separation* and *Model Based Parameter Estimation* can be regarded as the analysis stage which produces a parametric description of the input signal. The separation of the input signal is enhanced by feeding back the signals which are generated in the *Synthesis* unit from all the parameters of previously separated parts. The *Separation* and the *Model Based Parameter Estimation* additionally receive data from a synthesis model independent *Pre-Analysis*. Prior to transmission, the parameters are fed through the *Quantization and Coding* unit, which is controlled by a *Psychoacoustic Model*. This *Psychoacoustic Model* processes the input signal in order to derive information about the relevancy of synthesis parameters. In addition, the synthesized signal is fed into the *Psychoacoustic Model*, which thus is allowed to assist the *Model Based Parameter Estimation*.

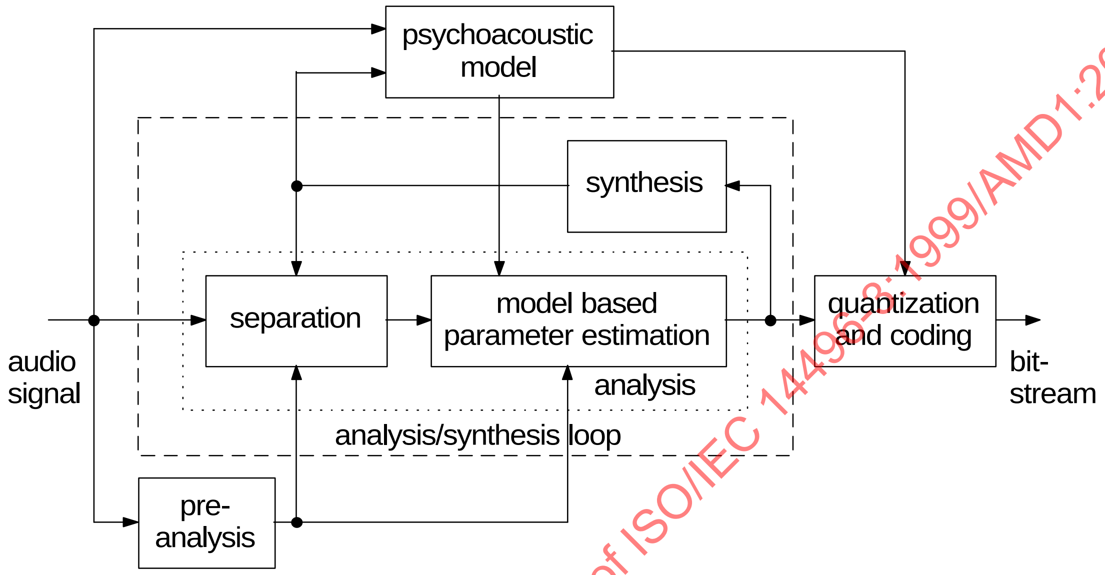


Figure A.2: Block diagram of the HILN encoder

In the parameter extraction, the input signal is separated into three different parts: "harmonic lines", "individual lines" and "noise".

For each of these parts parameters describing the signal are extracted. These are basically:

- harmonic lines: fundamental frequency and amplitudes of the harmonic components
- individual lines: frequency and amplitude of each individual line
- noise: spectral shape of the noise

Additionally parameters for amplitude envelopes and for continuation of spectral lines from one frame to the next can be determined.

The signal separation and parameter estimation is implemented in three steps: First the fundamental frequency of the harmonic part of the signal is estimated. Then the parameters of the relevant spectral lines are estimated and these lines are classified as "individual lines" or "harmonic lines" depending on the frequency with respect to the fundamental frequency. After all relevant spectral lines are extracted, the remaining residual signal is assumed to be noise-like and its spectral shape is described by a set of parameters.

The harmonic line extraction of the HILN tools can also be utilized in an integrated parametric coder utilizing both the HVXC speech coding tools as well as the HILN coding tools simultaneously. If the input signal is e.g. a speech signal mixed with background music, the HILN encoder can be used to extract only those individual spectral lines that do not belong to the harmonic part of the signal. These individual lines are encoded by the HILN tools and the remaining signal - consisting of the harmonic signal part and noise - then is encoded by the HVXC parametric speech codec tools. In the decoder, the audio signal is reconstructed by adding the output of the "individual line" synthesizer and the HVXC decoder.