
**Information technology — Coding of
audio-visual objects —**

**Part 3:
Audio**

**AMENDMENT 3: Scalable Lossless Coding
(SLS)**

Technologies de l'information — Codage des objets audiovisuels —

Partie 3: Codage audio

AMENDEMENT 3: Codage extensible sans perte (SLS)

PDF disclaimer

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.

© ISO/IEC 2006

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

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.org
Web www.iso.org

Published in Switzerland

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. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. 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 document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Amendment 3 to ISO/IEC 14496-3:2005/Amd.3:2005 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

This Amendment specifies Audio Scalable Lossless Coding (SLS).

Information technology — Coding of audio-visual objects —

Part 3: Audio

AMENDMENT 3: Scalable Lossless Coding (SLS)

In ISO/IEC 14496-3, Introduction, add the following to the end of the subclause "MPEG-4 general audio coding tools":

MPEG-4 SLS (Scalable Lossless Coding) is a tool used in combination with optional MPEG-4 General Audio coding tools to provide fine-grain scalable to numerical lossless coding of digital audio waveform.

In Part 3: Audio, Subpart 1, in subclause 1.3 Terms and Definitions, add:

SLS: Audio Scalable to Lossless Coding

and increase the index-number of subsequent entries.

In Part 3: Audio, Subpart 1, in subclause 1.5.1.1 Audio object type definition, amend table 1.1 with the updates in the table below:

Tools/ Modules							
<u>Audio Object Type</u>	Error Mapping (*)	Integer TNS (*)	Integer M/S (*)	IntMDCT (*)	BPGC/CBAC/LEMC (*)	Remark	Object Type ID
...							
(escape)						X	31
...							
SLS	X	X	X	X	X		37
SLS non-core				X	X		38
...							

Note: (*) marks new columns

In Part 3: Audio, Subpart 1, subclause 1.4 (Symbols and Abbreviations) add the following subclause:

1.4.9 Arithmetic data types

INT32 32 bit signed integer using two's complement

INT64 64 bit signed integer using two's complement

In Part 3: Audio, Subpart 1, subclause 1.5 add the following subclauses:

1.5.1.2.31 SLS object type

The SLS object is supported by the scalable to lossless tool which provides fine-grain scalable to lossless enhancement of MPEG perceptual audio codecs, such as AAC, allowing multiple enhancement steps from the audio quality of the core codec up to near-lossless and lossless signal representation. It also provides stand-alone lossless audio coding when the core audio codec is omitted.

1.5.1.2.32 SLS Non-Core object type

The SLS non-core object is supported by the scalable to lossless tool. It is similar to the SLS object type but the core audio codec is omitted.

In Part 3: Audio, Subpart 1, in subclause 1.6.2.1 AudioSpecificConfig, amend table 1.8 with the updates in the table below:

Syntax	No. of bits	Mnemonic
AudioSpecificConfig () { ... switch (audioObjectType) { case 37: case 38: SLSSpecificConfig(); break; ... } ... }		

In Part 3: Audio, Subpart 1, in subclause 1.6.2.1 add the following subclause:

1.6.2.1.13 SLSSpecificConfig

Defined in ISO/IEC 14496-3 subpart 12.

In Part 3: Audio, Subpart 1, in subclause 1.6.2.2.1 Overview, add the following to table 1.14:

Audio Object Type	Object Type ID	Definition of elementary stream payloads and detailed syntax	Mapping of audio payloads to access units and elementary streams
...			
SLS	37	ISO/IEC 14496-3 subpart 12	
SLS non_core	38	ISO/IEC 14496-3 subpart 12	

Create Part 3: Audio, Subpart 12:

Subpart 12: Technical description of scalable lossless coding

12.1 Scope

This subpart of ISO/IEC 14496-3 describes the MPEG-4 scalable lossless coding algorithm for audio signals. This description partially relies on the specification as given in subpart 4.

12.2 Terms and definitions

12.2.1 Definitions

The following definitions are used in this subpart.

Core Layer	The MPEG-4 GA T/F coder used as the first layer in SLS . The audio object types AAC LC, AAC Scalable (without LTP), ER AAC LC, ER AAC Scalable and ER BSAC are supported.
LLE Layer	Lossless enhancement layer used in SLS to enhance the quality of the core layer towards lossless coding.
Bit-Plane	Position of specific bit in binary data word, starting with 0 as the position of the least significant bit (LSB). For example, the binary bit-plane symbols from bit-plane 0, 1, 2, and 3 of data word 0x0011 1101 (0x3d) are 1, 0, 1, and 1 respectively.
BPGC	Bit-Plane Golomb Code
CBAC	Context Based Arithmetic Code
LEMC	Low Energy Mode Code
Implicit Band	A scale factor band for which the quantized spectral data presented in the core layer bit-stream will be used in determining part of the necessary side information for the LLE layer.
Explicit Band	A scale factor band for which the quantized spectral data presented in the core layer bit-stream will not be used in determining the necessary side information for the LLE layer. All the side information will be coded explicitly in the LLE payload.
Oversampling Factor (osf)	Ratio between sampling rates of LLE Layer and Core Layer, possible values are 1, 2 and 4.
Oversampling Range	High frequency range covered only by the LLE Layer, comprises $(osf-1)*1024$ resp. $(osf-1)*128$ frequency values per window.
Reserved	All fields labelled <i>Reserved</i> are reserved for future standardization. All Reserved fields must be set to zero.

12.2.2 Notations

In order to make the description stringent, the following notations are used in this subpart:

- Vectors are indicated by bold lower-case names, e.g. **vector**.
- Matrices (and vectors of vectors) are indicated by bold upper-case single letter names, e.g. **M**.
- Variables are indicated by italics, e.g. *variable*.
- Functions are indicated as *func(x)*

12.2.3 Definitions

DIV(m,n) Integer division with truncation of the result of m/n to an integer value towards $-\infty$.

[•] The floor operation. Returns the largest integer that is less than or equal to the real-valued argument.

12.3 Payloads for the audio object

Table 12.1 – Syntax of SLSSpecificConfig

Syntax	No. of bits	Mnemonics
SLSSpecificConfig(samplingFrequencyIndex, channelConfiguration, audioObjectType)		
{		
pcmWordLength;	3	uimbsf
aac_core_present;	1	uimbsf
lle_main_stream;	1	uimbsf
reserved_bit;	1	uimbsf
frameLength;	3	uimbsf
if (!channelConfiguration){		
program_config_element();		
}		
}		

Table 12.2 – Top layer payload for lle stream

Syntax	No. of bits	Mnemonics
lle_element()		
{		
for (ch=0;ch<channel_number;){		
if (is_channel_pair(ch)) {		
lle_channel_pair_element();		
ch += 2;		
} else {		
lle_single_channel_element();		
ch++;		
}		
}		
}		

Table 12.3 – Syntax of `Ile_single_channel_element`

Syntax	No. of bits	Mnemonics
<pre> Ile_single_channel_element() { Ile_individual_channel_stream(1); } </pre>		

Table 12.4 – Syntax of `Ile_channel_pair_element`

Syntax	No. of bits	Mnemonics
<pre> Ile_channel_pair_element() { Ile_individual_channel_stream(1); Ile_individual_channel_stream(0); } </pre>		

Table 12.5 – Syntax of `Ile_individual_channel_stream`

Syntax	No. of bits	Mnemonics
<pre> Ile_individual_channel_stream(is_first_channel) { Ile_ics_length; if (is_first_channel) { element_instance_tag; } Ile_reserved_bit; if (Ile_main_stream) { Ile_header(is_first_channel); Ile_side_info(); } Ile_data(); byte_align(); } </pre>	 16 4 1	 uimsbf uimsbf uimsbf

Table 12.6 – Syntax of `Ile_header()`

Syntax	No. of bits	Mnemonics
<code>Ile_header(is_first_channel)</code>		
{		
if (<code>Ile_channel_pair_element</code> && <code>common_window</code> && <code>is_first_channel</code>) {		
<code>use_stereo_intmdct</code> ;	1	uimsbf
}		
if (<code>aac_core_present</code>) {		
<code>band_type_signaling</code> ;	2	uimsbf
if (<code>band_type_signaling</code> ==1) {		
for(<code>g</code> =0; <code>g</code> < <code>num_window_groups</code> ; <code>g</code> ++) {		
for(<code>sfb</code> =0; <code>sfb</code> < <code>max_sfb</code> ; <code>sfb</code> ++) {		
<code>band_type</code> [<code>g</code>][<code>sfb</code>];	1	uimsbf
}		
}		
}		
} else {		
if (<code>is_first channel</code>) {		
<code>windows_sequence</code> ;	2	uimsbf
}		
}		
}		

Table 12.7 – Syntax of `Ile_side_info`

Syntax	No. of bits	Mnemonics
<code>Ile_side_info()</code>		
{		
For(<code>g</code> =0; <code>g</code> < <code>num_window_groups</code> ; <code>g</code> ++) {		
for(<code>sfb</code> =0; <code>sfb</code> < <code>num_sfb</code> + <code>num_osf_sfb</code> ; <code>sfb</code> ++) {		
if (<code>band_type</code> [<code>g</code>][<code>sfb</code>]== <code>Explicit_Band</code>) {		
<code>vcod_dpcm_max_bp</code> [<code>g</code>][<code>sfb</code>];	1...17	bslbf
}		
if (<code>max_bp</code> [<code>g</code>][<code>sfb</code>] != -1) {		
<code>vcod_lazy_bp</code> [<code>g</code>][<code>sfb</code>];	1... 2	bslbf
}		
}		
}		
<code>cb_cbac</code> ;	1	uimsbf
}		

Table 12.8 – Syntax of `Ile_data`

Syntax	No. of bits	Mnemonics
<code>Ile_data()</code>		
{		
BPGC/CBAC data;	varies	bslbf
LEMC data;	varies	bslbf
}		

12.4 Semantics

Data elements:

aac_core_present	Indicates, whether the lossless enhancement operates on top of an MPEG-4 GA T/F core (<i>aac_core_present</i> =1) or in non-core mode (<i>aac_core_present</i> =0).
lle_main_stream	Indicates, whether the current stream represents an LLE main stream including all the necessary side information or an LLE extension stream that extends the previous LLE stream.
pcmWordlength	Quantization word length of the original PCM waveform.

Table 12.9 – Word length of original PCM waveform

pcmWordlength	Word length of original PCM waveform
0	8
1	16
2	20
3	24
4 – 7	Reserved

frameLength	Length of the IntMDCT frame in the LLE layer.
--------------------	---

Table 12.10 – Length of the IntMDCT frame

frameLength	Length of the IntMDCT frame	Oversampling factor of the IntMDCT filterbank (osf)
0	1024	1
1	2048	2
2	4096	4
3-7	Reserved	Reserved

element_instance_tag	Unique instance tag for syntactic elements. All syntactic elements containing instance tags may occur more than once, but must have a unique <i>element_instance_tag</i> in each audio frame. When the MPEG-4 GA T/F core is present, syntactic elements of SLS and MPEG-4 GA T/F from the same audio channel use the same <i>element_instance_tag</i> .
lle_ics_length	Length of LLE individual channel stream (LLE_ICS) for the current frame; in bytes.
band_type_signaling	By default, the band type for a scale factor band is defined as follows: A scale factor band that is in a section coded with the zero codebook (ZERO_HCB), Intensity Stereo (IS) coded, or Perceptual Noise Substitution (PNS) coded is an <i>Explicit_Band</i>. Otherwise it is an <i>Implicit_Band</i>. Scale factor bands above <i>max_sfb</i> and in the oversampling range are always <i>Explicit_Band</i>. This default band type can be overwritten by <i>band_type_signaling</i> in the following way:

Table 12.11 – Band type signaling

Value of band_type_signaling	band type
00	Use default
01	Band type signaling for each sfb follows
10	All sfb are Explicit_Band
11	Reserved

band_type[g][sfb] Band type signaling for each scale factor band when band_type_signaling==01. A scale factor band is set to Explicit_Band if band_type[g][sfb] is 0.

Table 12.12 –Band type

Value	Band type
0	Explicit_Band
1	Default

vcod_dpcm_max_bp[g][sfb] The variable length coded maximum bit-plane for scale factor band *sfb* and group *g*.

vcod_lazy_bp[g][sfb] The variable length coded lazy bit-plane for non-zero scale factor band *sfb* and group *g*.

cb_cbac Indication of frequency table that will be used in the LLE decoding process.

Table 12.13 – cb_cbac table

cb_cbac	Frequency table
0	BPGC
1	CBAC

bpgc/cbac_data The binary bit-stream of the bpgc/cbac coded residual spectrum data

low_energy_mode_data The binary bit-stream of the LEMC mode coded residual spectrum data

12.5 SLS decoder tool

12.5.1 Overview

The block diagram of the scalable lossless (SLS) decoder is given in Figure 12.1. The core layer MPEG-4 GA stream is decoded by a deterministic Core Layer decoder. Its output, which is a deterministic spectrum in the MDCT domain, is sent to the inverse error mapping process. Meanwhile, the residual IntMDCT spectrum, carried in the LLE layer streams, is decoded and sent to the inverse error mapping process to reconstruct the IntMDCT spectrum. An inverse integer Mid/Side (M/S) and an inverse integer TNS process are then invoked and performed on the IntMDCT coefficients if necessary. Finally, its output is inversely transformed by using the inverse IntMDCT process to produce the PCM audio samples. A detailed description of each process is given in the subsequent sections.

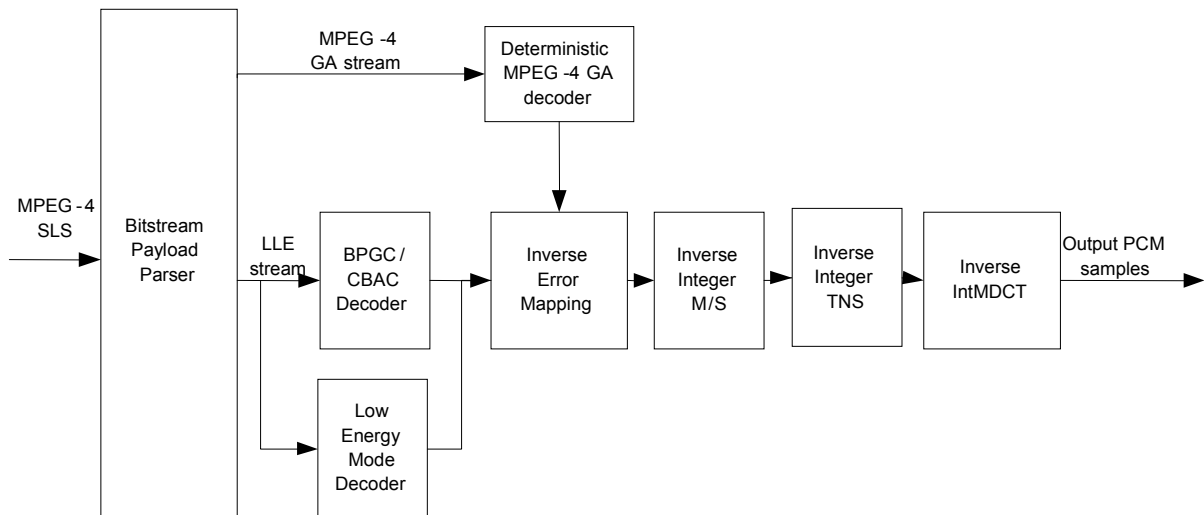


Figure 12.1 – SLS decoder block diagram

12.5.1.1 Non-core Mode

In the non-core mode SLS works as a stand-alone codec without AAC core. In case of the SLS audio object type this is signalled by **aac_core_present=0** for the non-core mode and **aac_core_present=1** for the core-based mode. In case of the SLS non-core audio object type it is always **aac_core_present=0**.

In the non-core mode the following default values are used:

- window_shape = 0 (sine window)
- if (lle_channel_pair_element) common_window = 1 (on)
- if (use_stereo_intmdct) all M/S flags are on, else all M/S flags are off
- if (window_sequence == EIGHT_SHORT_SEQUENCE) grouping = {2,2,2,2}

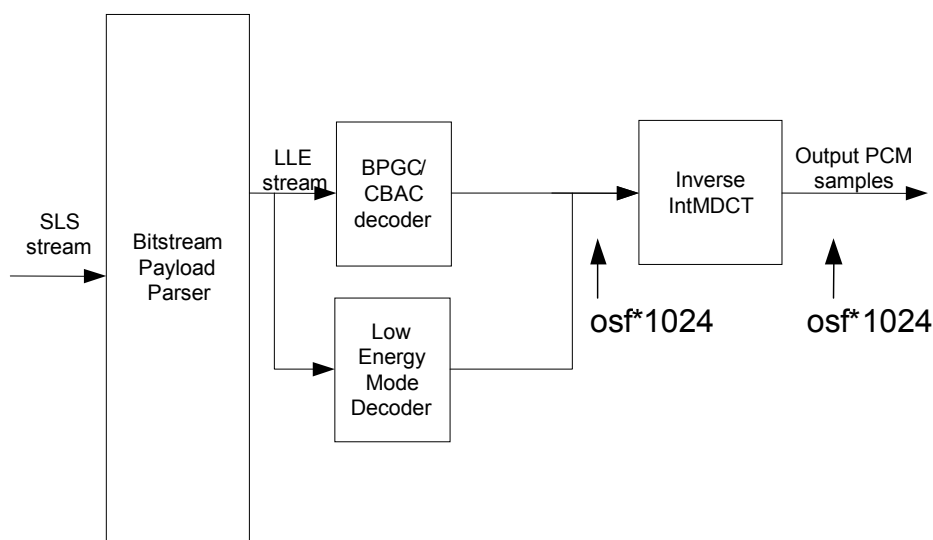


Figure 12.2 – SLS non-core decoder block diagram

12.5.2 Oversampling technique

The core layer is allowed to operate at a lower sampling rate than the LLE layers. The following table shows some possible sampling rate combinations.

Table 12.14 – Example combinations of sampling rates for Core and LLE layers

	Core@ 48 kHz	Core@ 96 kHz	Core@ 192 kHz
LLE@ 48 kHz	X (osf = 1)		
LLE@ 96 kHz	X (osf = 2)	X (osf = 1)	
LLE@ 192 kHz	X (osf = 4)	X (osf = 2)	X (osf = 1)

This technique is referred to as “Oversampling” in the following.

The scalability of the codec using different sampling rates is achieved by changing the length of the inverse IntMDCT in the decoder accordingly. While the AAC core processes 1024 values in each frame, the SLS codec needs to process $osf \times 1024$ values per frame. This is achieved by extending the length of the inverse IntMDCT in the decoder to $osf \times 1024$ spectral lines. The 1024 inverse quantized spectral values from the AAC core are added to the 1024 low-frequency values of the SLS residual spectrum. This is illustrated in **Figure 12.3**.

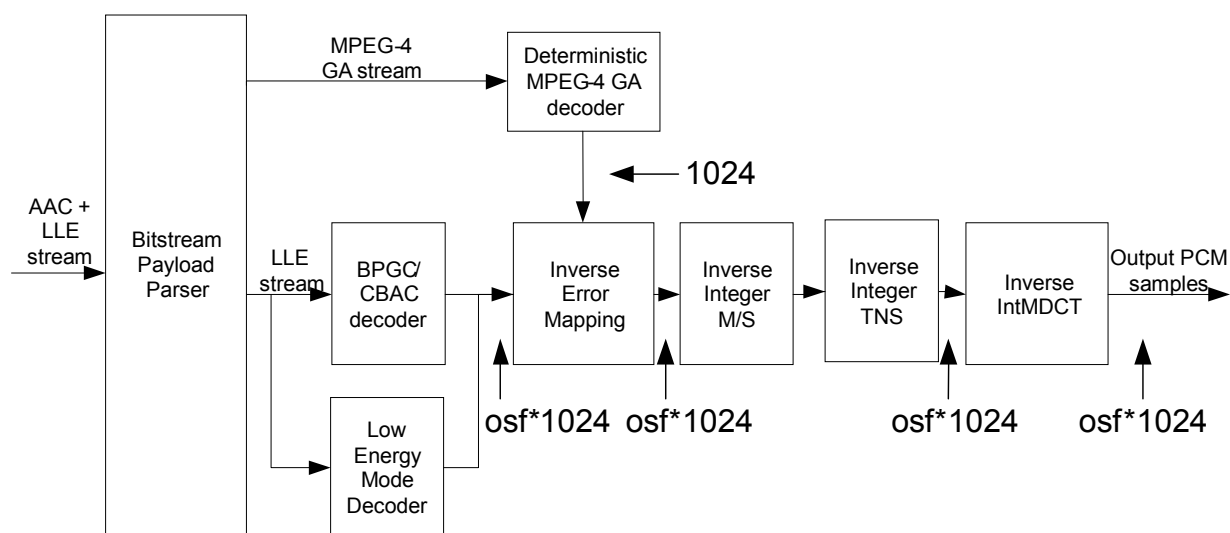


Figure 12.3 – Structure of SLS decoder with oversampling

12.5.3 SLS with Scalable AAC Core

If the core layer is AAC Scalable, the spectral data decoded from the SLS layers are added to the spectral data decoded from the AAC Scalable streams with a deterministic inverse AAC quantizer. The resulting spectral data is then processed with inverse integer M/S and inverse integer TNS process if necessary. Finally, the output is transformed by the inverse IntMDCT to produce the PCM audio samples. The decoding process is illustrated in Figure 12.4.

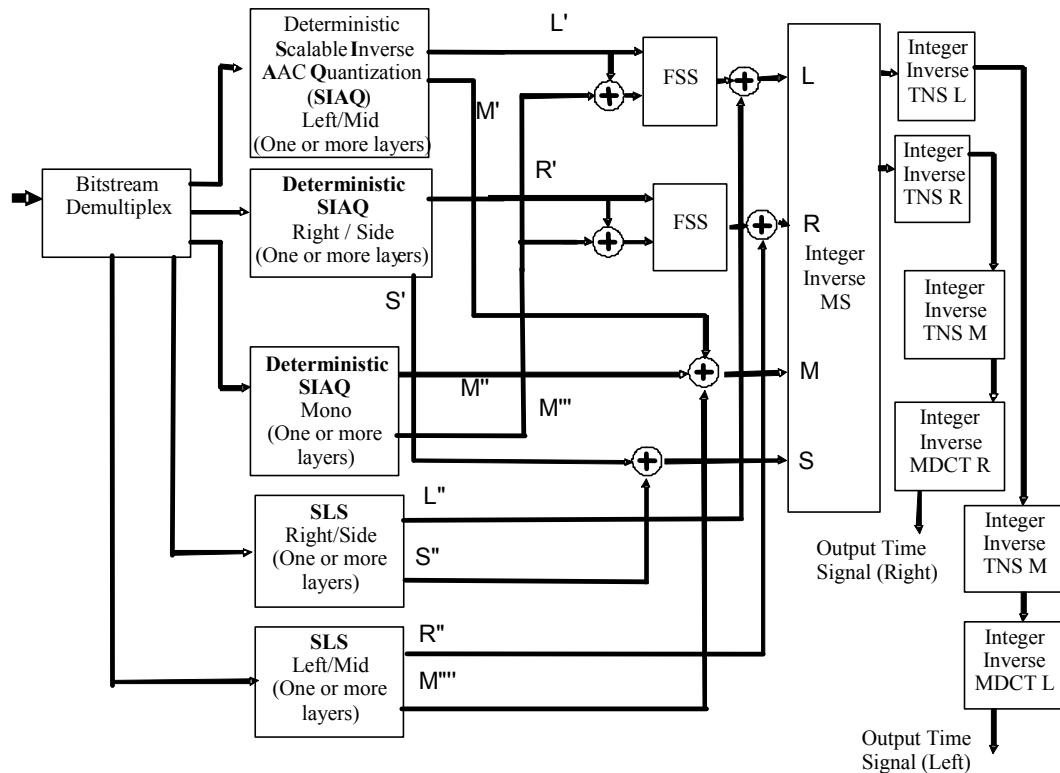


Figure 12.4 – Structure of SLS decoder with Scalable AAC core layer streams

12.5.4 Decoding of lle_single_channel_element (LLE_SCE) and lle_channel_pair_element (LLE_CPE)

12.5.4.1 Definitions

lle_ics_length	Length of LLE individual channel stream (LLE_ICS) in bytes.
vcod_dpcm_max_bp[g][sfb]	The variable length coded maximum bit-plane for scale factor band <i>sfb</i> and group <i>g</i> . This element is only present for insignificant scale factor bands.
vcod_lazy_bp[g][sfb]	The variable length coded lazy bit-plane for non-zero scale factor band <i>sfb</i> and group <i>g</i> .
g	Group index.
sfb	Scale factor band within group.
win	Window index.
bin	Frequency bin index.
num_window_groups	Number of groups of windows which share one set of scale factors.

num_sfb	Number of scale factor bands per short window in case of EIGHT_SHORT_SEQUENCE, number of scale factor bands for long windows otherwise.
num_osf_sfb	Number of scale factor bands per window in the oversampling range. The oversampling range is covered by $(osf-1)*16$ bands with a width of 64 in case of long windows resp. $(osf-1)*4$ bands with a width of 32 in case of short windows.
max_bp[g][sfb]	The maximum bit-plane for group <i>g</i> and scale factor band <i>sfb</i> .
lazy_bp[g][sfb]	The lazy bit-plane for group <i>g</i> and scale factor band <i>sfb</i> .
read_bits(n)	Read <i>n</i> consecutive bits from the inputs bit-stream in the order of bsbf.
quant[g][win][sfb][bin]	AAC quantized spectral data.
interval[g][win][sfb][k]	Quantization intervals in the core AAC encoder.

12.5.4.2 Decoding process

12.5.4.2.1 LLE_SCE and LLE_CPE

An LLE_SCE is composed of an lle_individual_channel_stream (LLE_ICS) while an LLE_CPE has two lle_individual_channel_streams (LLE_ICS).

12.5.4.2.2 Decoding an LLE_ICS

In the LLE_ICS, the order of the decoding process is given in the following flowchart:

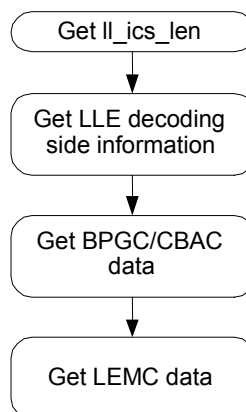


Figure 12.5 – Process of decoding LLE_ICS

For SLS bit-stream composed of an lle_main stream (lle_main_stream = 1) and multiple (≥ 1) lle_extension stream (lle_main_stream = 0), for each LLE_ICS, the lle_data() is constructed by concatenating the lle_data() elements from the lle_main stream, and all the available lle_extension streams in sequences as shown in the following figure:

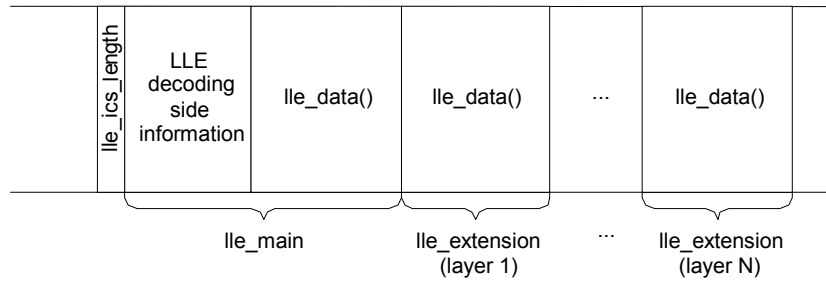


Figure 12.6 – Construction of LLE_ICS for from multiple LLE streams

If there is an intermediate LLE_extension stream missing, the data in lle_data() of the subsequent streams can not be used.

12.5.4.2.3 Recovering BPGC/CBAC side information

For each scale factor band of band type Explicit_Band, a maximum bit-plane (*max_bp*) is transmitted. In addition, for each scale factor band, a lazy bit-plane (*lazy_bp*) is transmitted unless the residual spectral data is all zero for this scale factor band (which is signalled by maximum bit-plane = -1). The *max_bp* is coded using variable length coded DPCM relative to the previously transmitted maximum bit-plane. The first value in each window group is coded using 5 bits PCM. The *max_bp* value is coded in unary representation. The following table gives some examples of how the DPCM value of *max_bp* is coded.

Table 12.15 – Codeword for decoding the DPCM value of *max_bp*

DPCM <i>max_bp</i>	codeword	codeword length
0	1	1
(s)1	01(s)	3
(s)2	001(s)	4
...	...	...
(s)10	0000000001(s)	12
...	...	...

The difference between *max_bp* and *lazy_bp*, whose value is within the range {1, 2, 3} is decoded as follows:

Table 12.16 – Codeword for decoding the difference between *max_bp* and *lazy_bp*

<i>max_bp</i> - <i>lazy_bp</i>	codeword	codeword length
1	10	2
2	0	1
3	11	2

The following pseudo code illustrates the decoding process for *max_bp* and *lazy_bp*.

```

for (g = 0; g < num_window_groups; g++)
    init = 0;
    for (sfb = 0; sfb < num_sfb+num_osf_sfb; sfb++){
        if (band_type[g][sfb]== Explicit_Band) {
            if (!init){
                max_bp[g][sfb] = read_bits(5) - 1; init ++;
            }
            else {
                m = 0;
                while (read_bits(1) == 0) m++;
                if (m) {
                    if (read_bits(1)) m = -m;
                }
                max_bp[g][sfb] = m0 - m;
            }
            m0 = max_bp[g][sfb];
        }
        if (max_bp[g][sfb]>=0) {
            if (read_bits(1)==0)
                lazy_bp[g][sfb] = max_bp[g][sfb] - 2;
            else {
                if (read_bits(1)==0) lazy_bp[g][sfb] = max_bp[g][sfb] - 1;
                else lazy_bp[g][sfb] = max_bp[g][sfb] - 3;
            }
        }
    }
}

```

For Implicit_Bands, *max_bp[g][sfb]* is calculated from the quantization thresholds of the core layer quantizer as follows:

As the first step, the maximum bit-plane *M* for each residual spectral bin for significant scale factor bands can be calculated from

$$M[g][win][sfb][bin] = INT \{ \log_2 [interval[g][win][sfb][bin]] \}$$

where *interval[g][win][sfb][bin]* is the quantization interval that is given by:

$$interval[g][win][sfb][bin] = thr(quant[g][win][sfb][bin] + 1) - thr(quant[g][win][sfb][bin]) + 1$$

Here *thr(x)* and *inv_quant(x)* are, respectively, the deterministic quantization threshold and the corresponding deterministic inverse quantization for AAC quantizer. They are calculated as in the following pseudo code:

```

If (x==0)
    thr(x)=0;
else
    thr(x) = (thrMantissa(|x|-1, scale_res)) << (12+scale_int);

inv_quant(x) = (invQuantMantissa(|x|, scale_res)) << (12+scale_int);

```

where

$$scale_int = DIV(scale, 4)$$

$$scale_res = scale - scale_int * 4, \text{ and}$$

$$scale = scale_factor(sfb) + core_scaling_factor + scale_osf - 118.$$

The value of *core_scaling_factor* is given in Table 12.17.

Table 12.17 – Table for *core_scaling_factor*

sfb Type \ Word Length	16	20	24
Long Window (2048), M/S	0	16	32
Long Window (2048), non M/S	2	18	34
Short Window (256), M/S	6	22	38
Short Window (256), non M/S	8	24	40

Table 12.18 – Table for *scale_osf*

<i>osf</i>	1	2	4
<i>scale_osf</i>	0	2	4

The functions *thrMantissa()* and *invQuantMantissa()* are defined in 12.5.11.

For scalefactor bands coded with IS or PNS the value of *inv_quant(x)* is set to 0.

The maximum bit-plane *max_bp* for each *sfb* is the maximum value of *M* for spectral data that belongs to that *sfb*:

$$\text{max_bp}[g][sfb] = \max (M[g][win][sfb][bin])$$

12.5.5 Decoding of *lle_data*

12.5.5.1 Definitions

<i>lle_data()</i>	Part of the bit-stream which contains the coded residual spectrum data.
<i>window_group_len[g]</i>	Number of windows in each group.
<i>is_lle_ics_eof()</i>	An auxiliary function to detect the end of LLE_ICS.
<i>read_bits(n)</i>	Read <i>n</i> consecutive bits from the input bit-stream in the order of bslbf. If there exists no bit to be fed in the bitstream, it returns '0' by default.
<i>cur_bp[g][sfb]</i>	The current decoded bit-plane.
<i>res[g][win][sfb][k]</i>	The reconstructed residual integer spectral data vector.
<i>amp[g][win][sfb][k]</i>	The amplitude of the reconstructed residual integer spectral data vector.
<i>sign[g][win][sfb][k]</i>	The sign of the reconstructed residual integer spectral data vector.
<i>determine_frequency()</i>	The function to determine the probability of the symbol '1' according to either the CBAC or the BPGC frequency table.
<i>ambiguity_check(f)</i>	The function to detect ambiguity for the arithmetic decoding. The argument <i>f</i> indicates the probability of the symbol '1'.
<i>terminate_decoding()</i>	The function to terminate decoding of the LLE data when ambiguity occurs.

`smart_decoding_cbac_bpgc()` The function to decode additional symbols in the absence of incoming bits in the cbac/bpgc mode decoding. This decoding continues up to the point where there exists no ambiguity. It includes *ambiguity_check(f)* and *terminate_decoding()*.

`smart_decoding_low_energy()` The function to decode additional symbols in the absence of incoming bits in the low energy mode decoding. It also includes *ambiguity_check(f)* and *terminate_decoding()*.

12.5.5.2 Decoding process

12.5.5.2.1 Overview

The residual integer spectral data vector is decoded from the LLE data stream *lle_data()*. Firstly, all scale factor bands with *lazy_bp* > 0 are BPGC/CBAC decoded, where the amplitude of the residual spectral data *res* is bit-plane decoded starting from the maximum bit-plane *max_bp* and progressing to lower bit-planes until bit-plane 0 for each scale factor band. Subsequently, the low energy mode decoding is invoked to decode the remaining scale factor bands with *lazy_bp* ≤ 0.

The SLS decoder can provide the functionality of fine-grain scalability (FGS) by truncating the LLE bitstream. Moreover, it allows to decode additional symbols beyond the point of truncation by exploiting the properties of arithmetic coding.

12.5.5.2.2 BPGC/CBAC decoding process

The BPGC decoding or CBAC decoding process is performed on scale factor bands for which *lazy_bp* > 0. The BPGC/CBAC bit-plane decoding process is used to decode the bit-plane symbols for reconstructing the residual integer spectral data *res*. The bit-plane decoding process is started from *max_bp* for each sfb, and progressively proceeds to lower bit-planes. For the first NUM_BP bit-plane scans the bit-plane symbols are arithmetic decoded as illustrated in the following pseudo code:

```
/* preparing the help element */
for (g=0;g<num_window_groups;g++){
    for (sfb = 0;sfb<num_sfb;sfb++){
        width = swb_offset[g][sfb+1] - swb_offset[g][sfb];
        for (win = 0;win <window_group_len[g];win++) {
            for (bin=0;bin<width;bin++){
                is_sig[g][win][sfb][bin] =
                    (quant[g][sfb][win][bin]) && (band_type[g][sfb]==ImplicitBand)?1:0;
                /* sign will be determined implicitly if is_sig == 1 */
                res[g][win][sfb][bin] = 0;
            }
            cur_bp[g][sfb] = max_bp[g][sfb];
        }
    }
}

/* BPGC/CBAC decoding */
while ((max_bp[g][sfb] - cur_bp[g][sfb]<LAZY_BP) && (cur_bp[g][sfb] >= 0)){
    for (g=0;g<num_window_groups;g++){
        for (sfb = 0;sfb<num_sfb;sfb++){
            if ((cur_bp[g][sfb]>=0) && (lazy_bp[g][sfb] > 0)){
                width = swb_offset[g][sfb+1] - swb_offset[g][sfb];
                for (win=0;win<window_group_len[g];win++){
                    for (bin=0;bin<width;bin++){
                        if (!is_lle_ics_eof()){
                            if (interval[g][win][sfb][bin] >
                                res[g][win][sfb][bin] + (1<<cur_bp[g][sfb]))
                            {
                                freq = determine_frequency();
                                res[g][win][sfb][bin] += decode(freq) << cur_bp[g][sfb];
                                /* decode bit-plane cur_bp*/
                            }
                        }
                    }
                }
            }
        }
    }
}
```

```

        if ((!is_sig[g][win][sfb][bin]) && (res[g][win][sfb][bin] )) {
            /* decode sign bit of res if necessary */
            res[g][win][sfb][bin] *= (decode(freq_sign)) ? 1:-1;
            is_sig[g][win][sfb][bin] = 1;
        }
    }
} else {
    smart_decoding_cbac_bpgc();
}
}
cur_bp[g][sfb]--; /* progress to next bit-plane */
}
}
}

```

After that, BPGC/CBAC enters the lazy decoding mode after skipping the 2 bit terminating string, where the bit-plane symbols are directly read from the input bit-stream:

```

/* BPGC/CBAC lazy decoding */
read_bits(1);read_bits(1); /* skip the 2 AC termination string before lazy coding
while (cur_bp[g][sfb] >= 0){
    for (g=0;g<num_window_groups;g++){
        for (sfb = 0;sfb<num_sfb+num_osf_sfb;sfb++){
            if ((cur_bp[g][sfb]>=0) && (lazy_bp[g][sfb] > 0)){
                width = swb_offset[g][sfb+1] - swb_offset[g][sfb];
                for (win=0;win<window_group_len[g];win++){
                    for (bin=0;bin<width;bin++){
                        if (!is_lle_ics_eof()){
                            if (interval[g][win][sfb][bin] >
                                res[g][win][sfb][bin] + (1<<cur_bp[g][sfb]))
                            {
                                res[g][win][sfb][bin] += read_bit() << cur_bp[g][sfb];
                                /* decode bit-plane cur_bp */
                                if (((!is_sig[g][win][sfb][bin]) && (res[g][win][sfb][bin] )) ) {
                                    /* decode sign bit of res if necessary */
                                    res[g][win][sfb][bin] *= (read_bit())? 1:-1;
                                    is_sig[g][win][sfb][bin] = 1;
                                }
                            }
                        }
                    }
                }
            }
        }
        cur_bp[g][sfb]--;
    }
}
}

```

the value of NUM BP is determined in the following table.

Table 12.19 – Value of NUM_BP

cb_cbac	NUM_BP
0 (BPGC)	4
1 (CBAC)	6

The probability assignment *freq* in the above BPGC/CBAC decoding process is either the BPGC frequency *freq_bpgc* or the CBAC frequency *freq_cbac* depending on whether the current LLE_ICS is decoded with the BPGC or the CBAC frequency table. *freq_bpgc* is determined by the relationship of the *cur_bp* to the *lazy_bp*

parameter as given in the following table. The sign bits in the above decoding process are decoded with frequency 8192, i.e., $freq_sign = 8192$.

Table 12.20 – $freq_bpgc$ table

cur_bp	BPGC frequency
$lazy_bp+3$	64
$lazy_bp+2$	964
$lazy_bp+1$	3277
$lazy_bp$	5461
$<lazy_bp$	8192

The value $freq_cbac$ is determined by the context of the bit-plane symbol which is currently being decoded. There are three types of context used in CBAC which are listed in the following.

- **Context 1: frequency band (fb)**

The fb context is determined by the index of the interleaved residual IntMDCT spectral data $c[i]$, $i=0,\dots,1024*osf-1$, and the sampling rate of the current LLE layer as shown in the following table.

Table 12.21 – Frequency band (fb) context [frequency bin]

Sampling Rate	44100	48000	96000	192000	Other
Context no					
0 (Low Band)	0 - 185	0 - 169	0 - 84	0 - 42	0 - 338
1 (Mid Band)	186 - 511	170 - 469	85 - 234	43 - 117	338 - 938
2 (High Band)	>511	>469	>234	>117	> 938

- **Context 2: significant state (ss)**

For interleaved residual IntMDCT spectral data $c[i]$, $i=0,\dots,1024*osf-1$ that is insignificant (i.e., the bit-plane symbols of $c[i]$ decoded so far are all zeroes) the ss context is determined by the significance of its adjacent spectral data:

$$sig_cx(i, bp) = \{sig_state(i-2, bp), sig_state(i-1, bp), sig_state(i+1, bp), sig_state(i+2, bp)\}$$

where $sig_state(i, bp)$ is defined as:

$$sig_state(i, bp) = \begin{cases} 0 & c[i] \text{ is insignificant before bitplane } bp \\ 1 & c[i] \text{ is significant before bitplane } bp \end{cases}$$

and $sig_state(i, bp)$ is defined as 0 if i is smaller than 0 or larger than the IntMDCT length.

For $c[i]$ that is already significant, the ss context is determined by the band type of the scalefactor band that it is from:

$$sig_core(i) = \begin{cases} 0 & c[i] \text{ is from an Explicit_Band} \\ 1 & c[i] \text{ is from an Implicit_Band} \end{cases}$$

Furthermore, for the latter case, the *ss* context is further determined according to the value of *quant_interval(i, bp)* defined as:

$$\text{quant_interval}(i, bp) = \begin{cases} 0 & \text{rec_spectrum}[i] + 2^{bp+1} \leq \text{interval}[i] \\ 1 & \text{rec_spectrum}[i] + 2^{bp} \leq \text{interval}[i] < \text{rec_spectrum}[i] + 2^{bp+1} \end{cases}$$

The detailed context assignment of the *ss* context is summarized in the following table:

Table 12.22 – Significance state (SS) context

Context no	<i>sig_state(i, cur_bp)</i>	<i>sig_cx(i, cur_bp)</i>	<i>sig_core(i)</i>	<i>quant_interval(i)</i>
0	0	{0,0,0,0}	x	x
1	0	{0,0,0,1} {1,0,0,0}	x	x
2	0	{0,0,1,0} {0,1,0,0}	x	x
3	0	{0,0,1,1} {1,1,0,0}	x	x
4	0	{0,1,0,1} {1,0,1,0}	x	x
5	0	{0,1,1,0}	x	x
6	0	{0,1,1,1} {1,1,1,0}	x	x
7	0	{1,0,0,1}	x	x
8	0	{1,0,1,1} {1,1,0,1}	x	x
9	0	{1,1,1,1}	x	x
10	1	x	0	x
11	1	x	1	0
12	1	x	1	1

- **Context 3: distance to lazy (*d2l*)**

The *d2l* context is determined by the distance of *cur_bp* to the *lazy_bp* parameter of the currently decoded bit-plane symbol. The detailed assignment is listed in the following table.

Table 12.23 – Distance to lazy (D2L) context

Context no	<i>cur_bp – lazy_bp</i>
0	<-2
1	-2
2	-1
3	0
4	1
5	2
6	3

The frequencies *freq_cbac* for each context are given in the following table.

Table 12.24 – *freq_cbac* table

<i>d2l</i> <i>fb*13+ss</i>	0	1	2	3	4	5	6
0	8192	7823	7826	6506	4817	2186	1053
1	8192	8344	7983	6440	4202	1362	64
2	8192	8399	8382	7016	4202	1234	64
3	8192	8305	7960	6365	3963	1285	64
4	8192	8335	8146	6655	3746	825	64
5	8192	8473	8244	6726	3929	927	64
6	8192	8398	7919	6098	3581	875	64
7	8192	8359	8028	6382	3459	631	64
8	8192	8192	8192	5461	3277	964	64
9	8192	8333	7481	5288	3076	732	64
10	8192	7658	6898	5145	1424	1636	64
11	8192	5471	5732	6264	4890	1279	93
12	8192	8180	8136	7897	5715	1553	64
13	8192	7242	6876	6083	3604	1214	950
14	8192	7897	7570	6583	3733	1067	900
15	8192	8071	7928	7069	4294	1406	1200
16	8192	8197	7952	6906	4050	1457	1101
17	8192	8278	8039	7094	4160	1381	64
18	8192	8307	8139	7263	4407	1555	64
19	8192	8339	8124	7065	4074	1636	64
20	8192	8213	7918	6827	3787	1161	64
21	8192	8286	8067	6902	3855	1387	64
22	8192	8336	8072	6705	3731	1558	64
23	8192	7636	6962	5036	1985	1037	64
24	8192	5519	5270	5238	4778	1588	219
25	8192	7884	7528	6743	4848	1970	64
26	8192	6084	6323	5929	3321	900	385
27	8192	7862	7618	6728	4409	1431	1302
28	8192	8078	7871	7081	5119	2371	1670
29	8192	8294	8046	7239	5218	2032	967
30	8192	8378	8119	7351	5413	1947	64
31	8192	8378	8207	7491	5624	2444	64
32	8192	8484	8302	7626	5514	2021	64
33	8192	8302	8006	7192	4941	1561	64
34	8192	8464	8246	7510	5217	1780	64
35	8192	8544	8442	7742	4944	2010	64
36	8192	7556	6771	4859	2638	2155	64
37	8192	5916	4780	4713	4239	1240	182
38	8192	7658	7095	5986	3886	1394	64

12.5.5.2.3 Low Energy Mode Code (LEMC) decoding

The following pseudo code illustrates the LEMC decoding process that is performed on scale factor bands for which *lazy_bp* ≤ 0.

```

/* low energy mode decoding */
for (g = 0; g < num_window_groups; g++){
    for (sfb = 0; sfb < num_sfb+num_osf_sfb; sfb++){
        if ((cur_bp[g][sfb] >= 0) && (lazy_bp[g][sfb] <= 0))
        {
            width = swb_offset[g][sfb+1] - swb_offset[g][sfb];

```