INTERNATIONAL STANDARD

ISO/IEC 14496-3

Third edition
2005-12-01

AMENDMENT 3
2006-06-01

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)

Reference number
ISO/IEC 14496-3:2005/Amd.3:2006(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.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 Audio Object Type	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 *Reserved* 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 $-\infty$.

$\lfloor \bullet \rfloor$ 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	uimsbf
aac_core_present;	1	uimsbf
lle_main_stream;	1	uimsbf
reserved_bit;	1	uimsbf
frameLength;	3	uimsbf
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 lle_single_channel_element

Syntax	No. of bits	Mnemonics

lle_single_channel_element()
{
  lle_individual_channel_stream(1);
}		

Table 12.4 – Syntax of lle_channel_pair_element

Syntax	No. of bits	Mnemonics

lle_channel_pair_element()
{
  lle_individual_channel_stream(1);
  lle_individual_channel_stream(0);
}		

Table 12.5 – Syntax of lle_individual_channel_stream

Syntax	No. of bits	Mnemonics
lle_individual_channel_stream(is_first_channel) {		
lle_ics_length;	16	uimsbf
if (is_first_channel) {		
element_instance_tag;	4	uimsbf
}		
lle_reserved_bit;	1	uimsbf
if (lle_main_stream) {		
lle_header(is_first_channel);		
lle_side_info();		
}		
lle_data();		
byte_align();		
}		

Table 12.6 – Syntax of lle_header()

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

Table 12.7 – Syntax of lle_side_info

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

Table 12.8 – Syntax of lle_data

Syntax	No. of bits	Mnemonics
lle_data()		
{		
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 (*aac_core_present*=1) or in non-core mode (*aac_core_present*=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 element_instance_tag 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 element_instance_tag.

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 Explicit_Band. Otherwise it is an Implicit_Band.

Scale factor bands above max_sfb and in the oversampling range are always Explicit_Band.

This default band type can by overwritten by band_type_signaling 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.

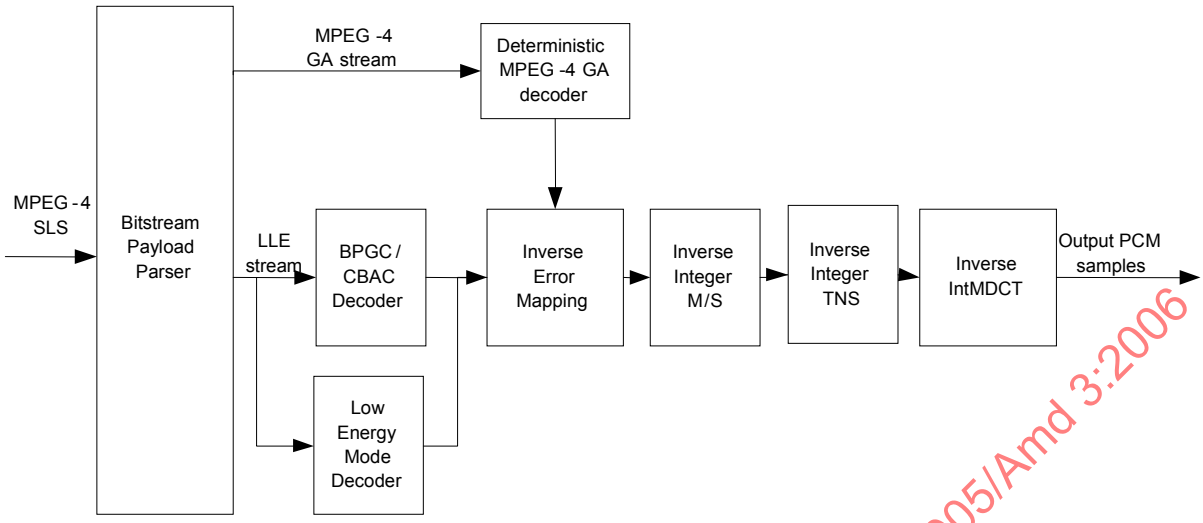

Figure 12.1 – SLS decoder block diagram

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}

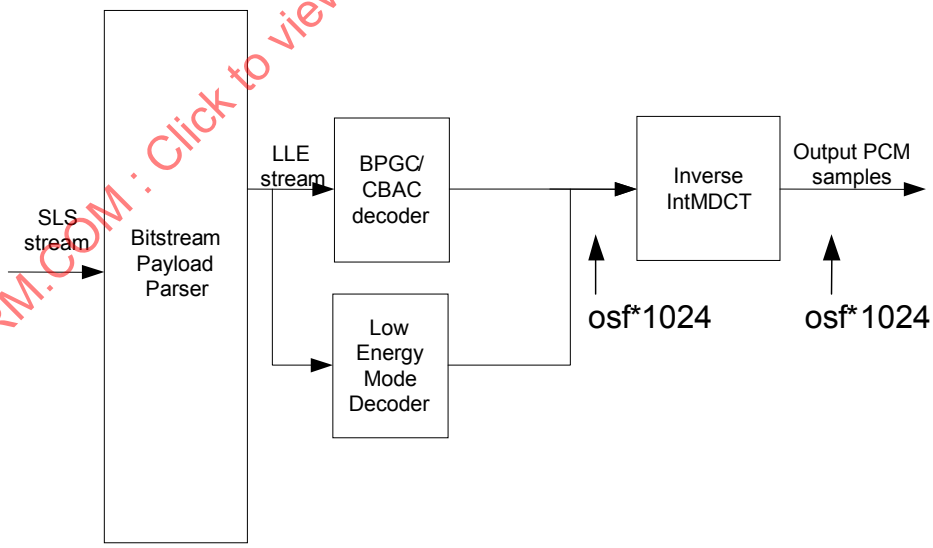

Figure 12.2 – SLS non-core decoder block diagram

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*1024 values per frame. This is achieved by extending the length of the inverse IntMDCT in the decoder to osf*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**.

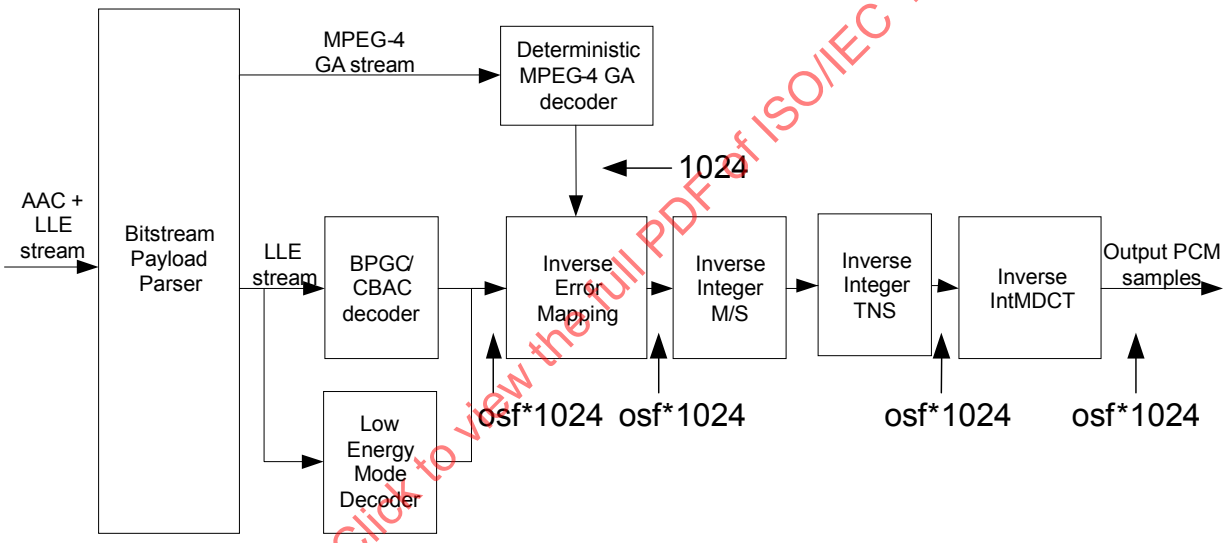

Figure 12.3 – Structure of SLS decoder with oversampling

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.

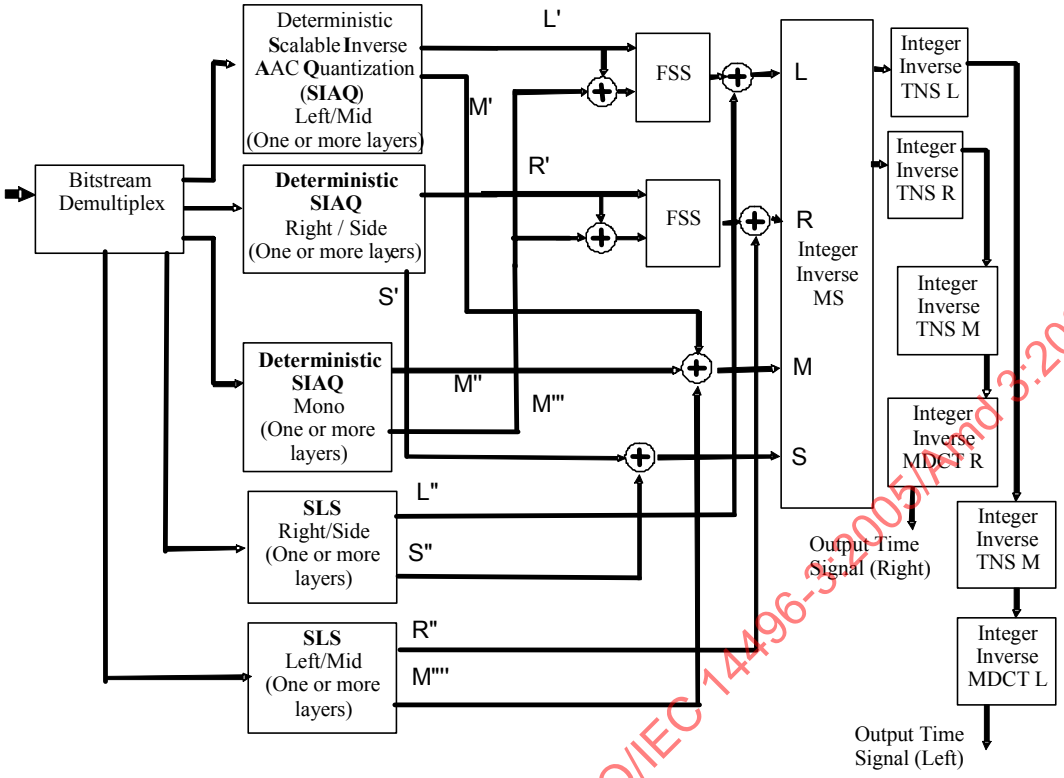

Figure 12.4 – Structure of SLS decoder with Scalable AAC core layer streams

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 *sfb* and group *g*. 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 *sfb* and group *g*.
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_SQEUENCE, 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 *g* and scale factor band *sfb.*
lazy_bp[g][sfb]	The lazy bit-plane for group *g* and scale factor band *sfb.*
read_bits(n)	Read n consecutive bits from the inputs bit-stream in the order of bslbf.
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:

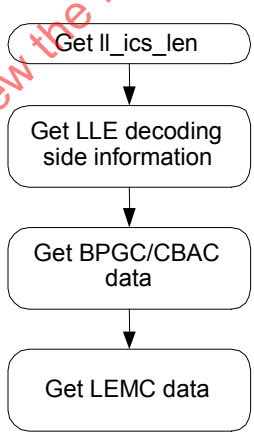

Figure 12.5 – Process of decoding LLE_ICS

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:

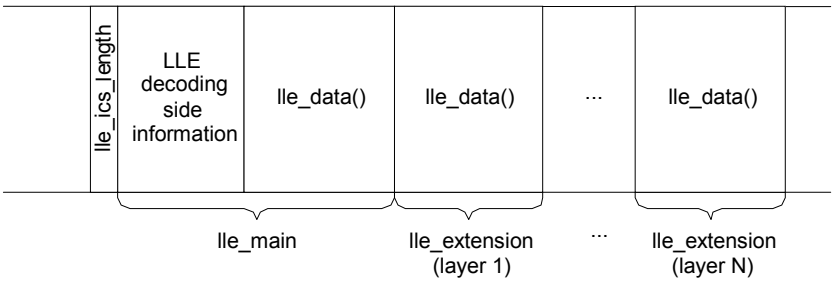

Figure 12.6 – Construction of LLE_ICS for from multiple LLE streams

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 *max_bp*	codeword	codeword length
0	1	1
(s)1	01(s)	3
(s)2	001(s)	4
…	…	…
(s)10	00000000001(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*

max_bp - lazy_bp	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

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

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

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

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, 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*

Word Length / sfb Type	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*

osf	1	2	4
scale_osf	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*:

$$max\_bp[g][sfb] = \max\left(M[g][win][sfb][bin]\right)$$

12.5.5 Decoding of lle_data

12.5.5.1 Definitions

lle_data()	Part of the bit-stream which contains the coded residual spectrum data.
window_group_len[g]	Number of windows in each group.
is_lle_ics_eof()	An auxiliary function to detect the end of LLE_ICS.
read_bits(n)	Read *n* 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.
cur_bp[g][sfb]	The current decoded bit-plane.
res[g][win][sfb][k]	The reconstructed residual integer spectral data vector.
amp[g][win][sfb][k]	The amplitude of the reconstructed residual integer spectral data vector.
sign[g][win][sfb][k]	The sign of the reconstructed residual integer spectral data vector.
determine_frequency()	The function to determine the probability of the symbol '1' according to either the CBAC or the BPGC frequency table.
ambiguity_check(f)	The function to detect ambiguity for the arithmetic decoding. The argument *f* indicates the probability of the symbol '1'.
terminate_decoding()	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,…,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 Context no	44100	48000	96000	192000	Other
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,…,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:

$$quant\_interval(i,bp) = \begin{cases} 0 & rec\_spectrum[i] + 2^{bp+1} \le interval[i] \\ 1 & rec\_spectrum[i] + 2^{bp} \le interval[i] < rec\_spectrum[i] + 2^{bp+1} \end{cases},$$

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

Table 12.22 – Significance state (SS) context

Context no	*sig_state(i,cur_bp)*	*sig_cx(i,cur_bp)*	*sig_core(i)*	*quant_interval(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	*cur_bp – lazy_bp*
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

d2l *fb**13+*ss*	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];
```

```
            for (win=0;win<window_group_len[g];win++){
                res[g][sfb][win][bin] = 0;
                pos = 0;
                for (bin=0;bin<width;bin++){
                    if (!is_lle_ics_eof ()){
                        /* decoding of binary string and reconstructing res */
                        while (decode(freq_silence[pos])==1) {
                            res[g][sfb][win][bin] ++;
                            pos++;
                            if (pos>2) pos = 2;
                            if (res[g][sfb][win][bin]==(1<<(max_bp[g][sfb]+1))-1) break;
                        }
                        /* decoding of sign of res */
                        if (!is_sig[g][win][sfb][bin]) && res[g][sfb][win][bin]){
                            res[g][sfb][win][bin] *= (decode(freq_sign))? -1:1;
                            is_sig[g][win][sfb][bin] = 1;
                        }
                    }
                    else smart_decoding_low_energy();
                }
            }
        }
    }
}
```

The probability assignments for the low energy mode decoding, *freq_bpgc* and *freq_silence* are given in the following tables. The sign bits in the above decoding process are decoded with frequency 8192, i.e. *freq_sign* = 8192.

Table 12.25 – *freq_silence* table

lazy_bp pos	0	-1	-2	-3
0	12603	9638	6554	3810
1	7447	3344	1820	X
>1	6302	745	552	X

The following table defines the mapping between the binary string decoded in case of the low energy mode and the residual spectral data *res*. The sign bit of *res* is decoded after the first non-zero bit-plane symbol has been decoded.

Table 12.26 – Binarization of *res* in low energy mode coding

Amplitude of *res*[g][win][sfb][bin]	Binary string
0	0
1	1 0
2	1 1 0
3	1 1 1 0
4	1 1 1 1 0
…	…
2^(*max_bp*[g][sfb]+1)-2	1 1 … … … 1 0
2^(*max_bp*[g][sfb]+1)-1	1 1 … … … 1 1
pos	0 1 2 3 …

12.5.5.2.4 Arithmetic decoding

The following pseudo code illustrates the integer arithmetic decoding process used in the BPGC/CBAC and the low energy mode decoding process.

```
Definitions:
#define CODE_WL    16
#define PRE_SHT    14
#define TOP_VALUE  (((long)1<<CODE_WL)-1)
#define QTR_VALUE  (TOP_VALUE/4+1)
#define HALF_VALUE (2*QTR_VALUE)
#define TRDQTR_VALUE   (3*QTR_VALUE)

Initialization:
low = 0;
high = TOP_VALUE;
value = 0;

The decoding subroutine

int decode(int freq)
{
    range = (long)((high-low)+1);
    cumu = ((long)((value-low)+1)<<PRE_SHT);
    if (cumu<range*freq)  {
        sym = 1;
        high = low + (range*freq>>PRE_SHT)-1;
    }
    else  {
        sym = 0;
            low = low + (range*freq>>PRE_SHT);
    }
        for (;;)  {
            if (high<HALF_VALUE)  {
        }  else if (low>=HALF_VALUE)  {
                value -= HALF_VALUE;
                low -= HALF_VALUE;
                high -= HALF_VALUE;
            }  else if (low>=QTR_VALUE && high<TRDQTR_VALUE)  {
              value -= QTR_VALUE;
                low -= QTR_VALUE;
                high -= QTR_VALUE;
            }  else
                break;
            low = 2*low;
            high = 2*high+1;
            value = 2*value + read_bits(1);  /*input next bit from bit-stream */
    }
    return sym;
}
```

12.5.5.2.5 Smart arithmetic decoding of truncated SLS bitstreams

The smart arithmetic decoding provides an efficient way to decode an intermediate layer corresponding to a given target bitrate. This algorithm exploits the fact that a decoding buffer still contains meaningful information for arithmetic decoding even if there is no bit left to be fed into the decoding buffer. The decoding process continues as long as there exists no ambiguity in determining a symbol.

The following pseudo code illustrates the algorithm to detect the ambiguity in the arithmetic decoding module. The variable *num_dummy_bits* represents the number of calls to evoke the function of *read_bits(1)* in the arithmetic decoding process just after the truncation point.

```
int ambiguity_check(int freq)
{
    /* if there is no ambiguity, returns 1 */
    /* otherwise, returns 0                */
```

```
    upper = 1<<num_dummy_bits;
    decisionVal = ((high-low)*freq>>PRE_SHT)-value+low-1;
    if(decisionVal>upper || decisionVal<0) return 0;
    else return 1;
}
```

Either *smart_decoding_cbac_bpgc()* or *smart_decoding_low_energy()* is executed when *num_dummy_bits* is greater than 0. In order to prevent sign bit errors, the spectral value of the current spectral line should be set to zero when an ambiguity can occur while decoding a sign bit. Notice that all index variables in the smart decoding process should be carried over from the previous arithmetic decoding process.

```
smart_decoding_cbac_bpgc()
{
    /* BPGC/CBAC normal decoding with ambiguity detection */
    while ((max_bp[g][sfb] - cur_bp[g][sfb]<LAZY_BP) && (cur_bp[g][sfb] >= 0)){
        for (;g<num_window_groups;g++){
            for (;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<window_group_len[g];win++){
                        for (;bin<width;bin++){
                            if (interval[g][win][sfb][bin] >
                               res[g][win][sfb][bin] + (1<<cur_bp[g][sfb])
                            {
                               freq = determine_frequency();
                               if (ambiguity_check(freq)) {
                                   /* no ambiguity for arithmetic decoding */
                                   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 */
                                       if (ambiguity_check(freq)) {
                                           res[g][win][sfb][bin] *= (decode(freq_sign))? 1:-1;
                                           is_sig[g][win][sfb][bin] = 1;
                                       }
                                       else {
                                           /* discard the decoded symbol prior to sign symbol */
                                           res[g][win][sfb][bin] = 0;
                                           terminate_decoding();
                                       }
                                   }
                               }
                               else terminate_decoding();
                            }
                        }
                    }
                    cur_bp[g][sfb]--; /* progress to next bit-plane */
                }
            }
        }
    }
}

smart_decoding_low_energy()
{
    /* low energy mode decoding */
    for (;g < num_window_groups; g++){
        for (; 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<window_group_len[g];win++){
                    res[g][sfb][win][bin] = 0;
                    pos = 0;
                    for (;bin<width;bin++){
                        while (1) {
```

```
                        /* if ambiguity check is false, discard the spectrum is set to be 0 */
                        if(!ambiguity_check(freq)) res[g][sfb][win][bin] = 0, terminate_decoding();
                        tmp = decode(freq_silence[pos]);
                        if(tmp==0) break;
                        res[g][sfb][win][bin] ++;
                        pos++;
                        if (pos>2) pos = 2;
                        if (res[g][sfb][win][bin]==(1<<(max_bp[g][sfb]+1))-1) break;
                    }
                    /* decoding of sign of res */
                    if (!is_sig[g][win][sfb][bin]) && res[g][sfb][win][bin]){
                        /* if ambiguity check is false,the current spectrum value is set to be 0 */
                        if(!ambiguity_check(freq)) res[g][sfb][win][bin] = 0, terminate_decoding();
                        res[g][sfb][win][bin] *= (decode(freq_sign))? -1:1;
                        is_sig[g][win][sfb][bin] = 1;
                    }
                }
            }
        }
    }
}
```

12.5.6 Compensation for IntMDCT residual for early terminating BPGC/CBAC decoding

If the BPGC/CBAC decoding process is terminated early due to a truncation of the LLE_ICS, *res* is compensated with a *res_fill* term as follows:

```
for (g=0;g<num_window_groups;g++){
    for (sfb = 0;sfb<num_sfb+num_osf_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++){
                if (is_sig[g][win][sfb][bin]){
                    if (res[g][win][sfb][bin] >= 0) res[g][win][sfb][bin] += (res_fill >> (23-
stop_bp[g][win][sfb][bin]));
                    else if (res[g][win][sfb][bin] < 0) res[g][win][sfb][bin] -= (res_fill >> (23-
stop_bp[g][win][sfb][bin]));
                }
            }
        }
    }
}
```

Here *stop_bp*[g][win][sfb][bin] is the highest bit-plane for which the bit-plane symbol is not decoded due to the early termination of LLE_ICS. The value of *res_fill* is given in the following table.

Table 12.27 – Value of *res_fill*

stop_bp	***res_fill***
lazy_bp+3	1572608
lazy_bp+2	3079935
lazy_bp+1	5172975
lazy_bp	6990507
<*lazy_bp*	8388607

12.5.7 Inverse error mapping

12.5.7.1 Principle

The inverse error mapping process is used to reconstruct the IntMDCT spectral data from the IntMDCT residual data from the LLE layer and the quantized MDCT spectral data from the core layer. This process is only applied in the non-oversampling range. The input to the inverse error mapping tool is the residual spectral data *res* and the quantized spectral data *quant*. Its output is the reconstructed IntMDCT spectral data *c*. The inverse error mapping procedure is described in the following:

```
if (quant[g][win][sfb][bin]==0)
    c[g][win][sfb][bin]=res[g][win][sfb][bin];
else
    c[g][win][sfb][bin] = sign(quant[g][win][sfb][bin]) *
                          (res[g][win][sfb][bin]+ref(quant[g][win][sfb][bin]);
```

To ensure lossless coding, in the SLS encoder the following error mapping procedure should be employed for the same spectral range:

```
if (quant[g][win][sfb][bin]==0)
    res[g][win][sfb][bin]=c[g][win][sfb][bin];
else
    res[g][win][sfb][bin]=sign(quant[g][win][sfb][bin])* c[g][win][sfb][bin]-
                          ref(quant[g][win][sfb][bin]);
```

The function *ref(x)* in the above process is deterministically calculated as follows

```
if ((sfb is Implicit_Band) then
    ref(x) = thr(abs(x))
else if  (sfb is Explicit_Band)
    ref(x) = inv_quant(abs(x))
```

Here the calculation of thr() and inv_quant() follows subclause 12.5.4.2.3

12.5.8 Integer Mid/Side process

If the Mono IntMDCT is used for the left and the right channel (common_window == 0 or use_stereo_intmdct == 0), the inverse integer M/S processing has to be applied to the scale factor bands where the M/S flag is switched on.

The Mid/Side (M/S) decoding is performed on the integer spectral values by a lossless $-\pi/4$ Givens rotation using the lifting scheme as follows:

Step 1:

$$S = S - NINT\left(c_1 \cdot M\right);$$

Step 2:

$$M = M - NINT\left(c_2 \cdot S\right);$$

Step 3:

$R = M$;

$L = S - NINT\left(c_1 \cdot R\right)$;

where M,S,R,L denotes the spectral data of Mid, Side, Left, and Right channels and $c_1 = (\cos\frac{\pi}{4} - 1) \Big/ \sin\frac{\pi}{4}$,

and $c_2 = \sin\frac{\pi}{4}$.

These three multiplications are performed in a fixed-point fashion by using integer coefficients and bit shifts. The detailed fixed-point arithmetic is described in subclause 12.5.10.3.

The inverse Stereo IntMDCT expects an M/S spectrum by default. Hence M/S has to be applied to the scale factor bands where the M/S flag is switched off.

The Mid/Side (M/S) coding is performed on the integer spectral values by a lossless $\pi/4$ Givens rotation using the lifting scheme as follows:

Step 1

$S = R$;

$M = L + NINT\left(c_1 \cdot R\right)$;

Step 2:

$S = S + NINT\left(c_2 \cdot M\right)$;

Step 3:

$M = M + NINT\left(c_1 \cdot S\right)$;

where M,S,L,R denote the spectral data of Mid, Side, Left, and Right channels and $c_1 = (\cos\frac{\pi}{4} - 1) \Big/ \sin\frac{\pi}{4}$

, and $c_2 = \sin\frac{\pi}{4}$.

These three multiplications are performed in a fixed-point fashion by using integer coefficients and bit shifts. The detailed fixed-point arithmetic is described in subclause 12.5.10.3.

12.5.9 Integer Temporal Noise Shaping (IntTNS)

When Temporal Noise Shaping (TNS) is used in the AAC core, the same TNS filter is applied to the integer spectral values in SLS. In order to convert this filter to a deterministic invertible integer filter, the following changes to the TNS tool description in Subpart 4 are required:

For determining the LPC coefficients, instead of the function `tns_decode_coef()` in subclause 4.6.9.3, the function `int_tns_decode_coef()` is used, as described in the following pseude-code:

```
define SHIFT_INTTNS 21

INT32 tnsInvQuantCoefFixedPoint(coef_res, coef)
```

```
{
  INT32 intTnsCoef_res3[8] = {-2065292, -1816187, -1348023, -717268,
                              0, 909920, 1639620, 2044572};
  INT32 intTnsCoef_res4[16] = {-2088206, -2017095, -1877294, -1673563,
                               -1412842, -1104008, -757579, -385351,
                               0, 436022, 852989, 1232675,
                               1558488, 1816187, 1994510, 2085664};
  if (coef_res == 3) {
    return intTnsCoef_res3[4+coef];
  }
  if (coef_res == 4) {
    return intTnsCoef_res4[8+coef];
  }
}

/* Decoder transmitted coefficients for one TNS filter */
int_tns_decode_coef( order, coef_res, *coef, INT32 *a )
{
  INT32 tmp[TNS_MAX_ORDER+1], b[TNS_MAX_ORDER+1];

  /* Inverse quantization */
  for (i=0; i<order; i++)  {
    tmp[i+1] = tnsInvQuantCoefFixedPoint(coef_res, coef[i]);
  }
  /* Conversion to LPC coefficients */
  /* worst case for order == 12 and all coefficients == 1:
     6th coefficient raised by 12!/(6!*6!) = 924
     -> 10 bits headroom required -> SHIFT_INTTNS == 21 */

  a[0] = 1<<SHIFT_INTTNS;
  for (m=1; m<=order; m++)  {
    b[0] = a[0];
    for (i=1; i<m; i++)  {
         b[i] = a[i] + ((((((INT64)tmp[m])*a[m-i])>>(SHIFT_INTTNS-1))+1)>>1);
    }
    b[m] = tmp[m];
    for (i=0; i<=m; i++)  {
         a[i] = b[i];
    }
  }
}
```

Based on the resulting fixed-point LPC coefficients, a deterministic integer version of the TNS filter is applied to the integer spectrum in the decoder. This is done by replacing the function `tns_ar_filter()` in subclause 4.6.9.3 by the function `int_tns_ar_filter()`, described by the following pseudo-code:

```
int_tns_ar_filter(INT32 *spec, size, inc, INT32 *lpc, order )
{
  INT32 y, state[TNS_MAX_ORDER];
  INT64 temp_accu;

  for (i=0; i<order; i++)
    state[i] = 0;

  if (inc == -1)
    spec += size-1;

  for (i=0; i<size; i++) {
    y = *spec;
    temp_accu = 0;
    for (j=0; j<order; j++) {
      temp_accu += ((INT64)lpc[j+1]) * state[j];
    }
    y -= (INT32)( ( ( temp_accu >> (SHIFT_INTTNS-1) ) + 1) >> 1 );
    for (j=order-1; j>0; j--)
         state[j] = state[j-1];
    state[0] = y;
    *spec = y;
    spec += inc;
  }
}
```

If the StereoIntMDCT is used, the integer spectral values represent the M/S spectrum instead of the L/R spectrum. In this case, the inverse Integer M/S has to be applied before the IntTNS and the forward Integer M/S has to be applied afterwards.

IntTNS in the encoder

In order to ensure lossless reconstruction, the corresponding forward LPC prediction has to be applied to the integer spectrum in the encoder. This is achieved by applying the function `int_tns_decode_coef()` and the corresponding forward filter to the integer spectrum, as described in the following pseudo-code:

```
int_tns_filter_encode(length, order, direction, INT32* spec, INT32 *lpc)
{
  INT64 temp_accu;
  if (direction) {
    /* Startup, initial state is zero */
    temp[length-1]=spec[length-1];
    for (i=length-2;i>(length-1-order);i--) {
      temp[i]=spec[i];
      temp_accu = 0;
      k++;
      for (j=1;j<=k;j++) {
        temp_accu += ((INT64)temp[i+j]) * a[j];
      }
      spec[i] += (INT32)( ( ( temp_accu >> (SHIFT_INTTNS-1) ) + 1) >> 1);
    }

    /* Now filter the rest */
    for   (i=length-1-order;i>=0;i--) {
      temp[i]=spec[i];
      temp_accu = 0;
      for (j=1;j<=order;j++) {
        temp_accu += ((INT64)temp[i+j]) * a[j];
      }
      spec[i] += (INT32)( ( ( temp_accu >> (SHIFT_INTTNS-1) ) + 1) >> 1 );
    }


  } else {
    /* Startup, initial state is zero */
    temp[0]=spec[0];
    for (i=1;i<order;i++) {
      temp[i]=spec[i];
      temp_accu = 0;
      for (j=1;j<=i;j++) {
        temp_accu += ((INT64)temp[i-j]) * a[j];
      }
      spec[i] += (INT32)( ( ( temp_accu >> (SHIFT_INTTNS-1) ) + 1) >> 1 );
    }

    /* Now filter the rest */
    for   (i=order;i<length;i++) {
      temp[i]=spec[i];
      temp_accu = 0;
      for (j=1;j<=order;j++) {
        temp_accu += ((INT64)temp[i-j])*a[j];
      }
      spec[i] += (INT32)( ( ( temp_accu >> (SHIFT_INTTNS-1) ) + 1) >> 1 );
    }
  }
}
```

In case the StereoIntMDCT is used, the integer spectral values represent the M/S spectrum instead of the L/R spectrum. In this case the inverse Integer M/S has to be applied before the IntTNS and the forward Integer M/S has to be applied afterwards.

12.5.10 IntMDCT and Inverse IntMDCT

12.5.10.1 Description

The IntMDCT is an invertible integer approximation of the MDCT. The following section describes the structural implementation of the MDCT and IMDCT used for the forward and inverse IntMDCT.

In the following, the frame length *N* always denotes the number of new input samples in each block, which is equal to the number of frequency values, so *N* is either osf*1024 or osf*128.

12.5.10.2 MDCT and Inverse MDCT (IMDCT)

The MDCT is defined by

$$X(m) = \sqrt{\frac{2}{N}} \sum_{k=0}^{2N-1} w(k)x(k)\cos\frac{(2k+1+N)(2m+1)\pi}{4N}$$
$$m = 0,\ldots,N-1$$

N: Frame length (osf*1024 or osf*128)

X(m): Values of MDCT spectrum

x(k): Input samples

w(k): Window function (Sine or KBD)

The IMDCT is defined by

$$x(k) = w(k)\sqrt{\frac{2}{N}} \sum_{m=0}^{N-1} X(m)\cos\frac{(2k+1+N)(2m+1)\pi}{4N}$$
$$k = 0,\ldots,2N-1$$

The input of the MDCT and the output of the IMDCT have a 50% overlap, i.e. *N* samples. In the IMDCT the output of two succeeding blocks is added in the overlapping region.

12.5.10.2.1 MDCT and IMDCT by DCT-IV

For the IntMDCT the MDCT and the IMDCT are divided into two blocks:

- Windowing and Time Domain Aliasing (TDA)

- Discrete Cosine Transform of Type IV (DCT-IV)

These two blocks are illustrated in Figure 12.7 for the MDCT and the IMDCT

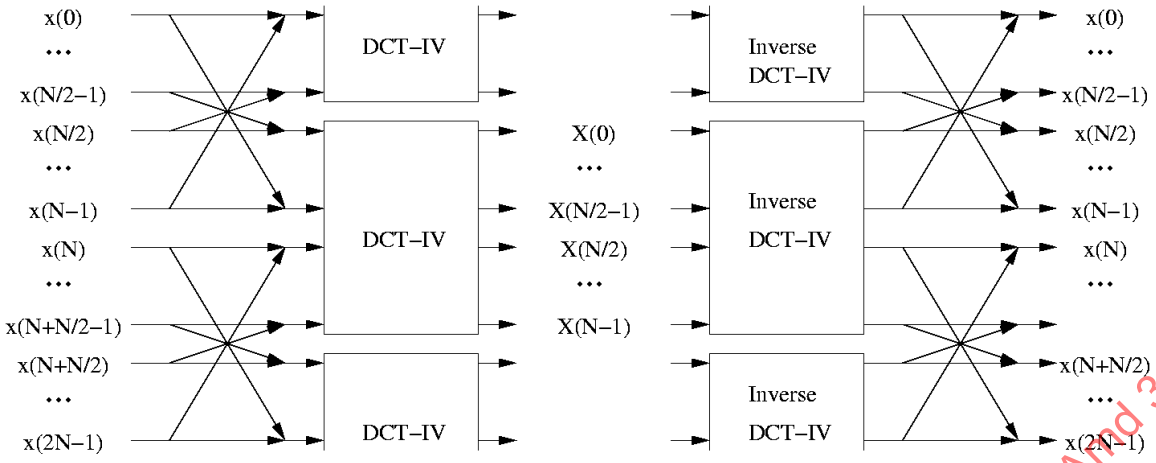

Figure 12.7 – MDCT and IMDCT by Windowing/TDA and DCT-IV

12.5.10.2.2 Calculation of windowing/TDA block

12.5.10.2.3 Structure of MDCT and IMDCT for different window sequences

In the MDCT the Windowing/TDA block is calculated by

$$\begin{pmatrix} x(k) \\ x(N-1-k) \end{pmatrix} \mapsto \begin{pmatrix} w(N-1-k) & w(k) \\ -w(k) & w(N-1-k) \end{pmatrix} \begin{pmatrix} x(k) \\ x(N-1-k) \end{pmatrix}$$
$$k = 0,\ldots,N/2-1$$

In the IMDCT this block is inverted by

$$\begin{pmatrix} x(k) \\ x(N-1-k) \end{pmatrix} \mapsto \begin{pmatrix} w(N-1-k) & -w(k) \\ w(k) & w(N-1-k) \end{pmatrix} \begin{pmatrix} x(k) \\ x(N-1-k) \end{pmatrix}$$
$$k = 0,\ldots,N/2-1$$

Note that the overlap/add operation is already contained in this calculation.

The DCT-IV of length *N* is defined by:

$$X(m) = \sqrt{\frac{2}{N}} \sum_{k=0}^{N-1} x(k) \cos \frac{(2k+1)(2m+1)\pi}{4N}$$
$$m = 0,\ldots,N-1$$

The inverse DCT-IV of length *N* has the same coefficients, it is defined by:

$$x(k) = \sqrt{\frac{2}{N}} \sum_{m=0}^{N-1} X(m) \cos \frac{(2k+1)(2m+1)\pi}{4N}$$
$$k = 0,\ldots,N-1$$

For the calculation of the MDCT the output of two succeeding Windowing/TDA stages is considered. Let $x'(0),\ldots,x'(N-1)$ be the output of the Windowing/TDA stage of the previous block and $x'(N),\ldots,x'(2N-1)$ be the output of the Windowing/TDA stage of the current block. Then the DCT-IV is applied to the N values

$$-x'(N+N/2-1), -x'(N+N/2-2), \ldots, -x'(N), -x'(N-1), -x'(N-2), \ldots, -x'(N/2)$$

i.e. the second half of the previous block and the first half of the current block are used. The order of the values is reverted and values are multiplied with -1 before applying the DCT-IV of length N.

The second half of current block of Windowing/TDA output values have to be stored for the next block.

This structure is illustrated in Figure 12.8:

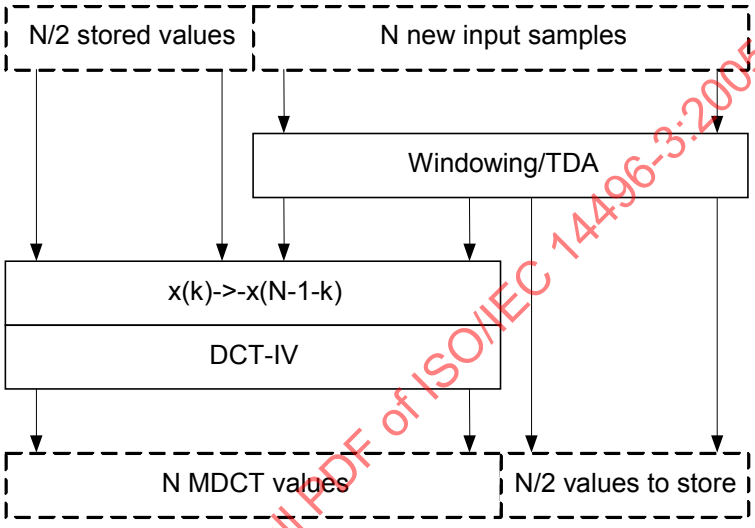

Figure 12.8 – Structure for MDCT by DCT-IV and Windowing/TDA

For the calculation of the IMDCT, the MDCT spectral values are transformed by the inverse DCT-IV, the output is multiplied with -1 and the order is reversed. Then the second half is stored for the next block, the first half is processed by the inverse Windowing/TDA block together with the values stored from the previous block.

This structure is illustrated in Figure 12.9:

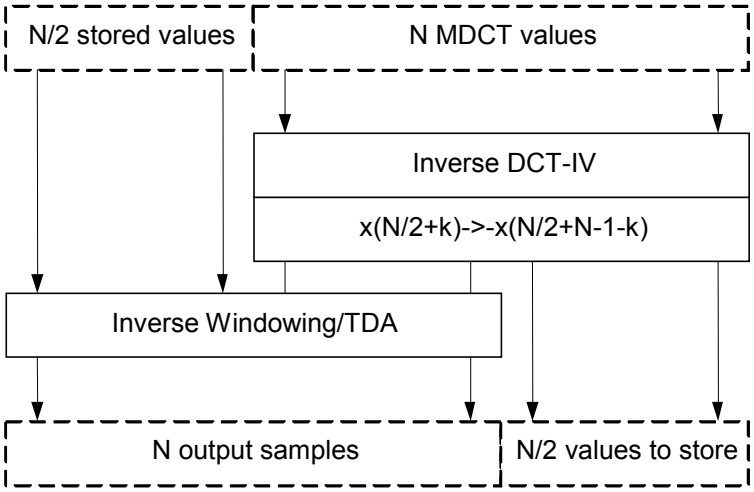

Figure 12.9 – Structure for IMDCT by DCT-IV and Windowing/TDA

12.5.10.2.4 Structure of MDCT and IMDCT for different window sequences

The structure in Figure 12.8 resp. Figure 12.9 is slightly modified for different window sequences. In the following this structure is illustrated for typical window sequences.

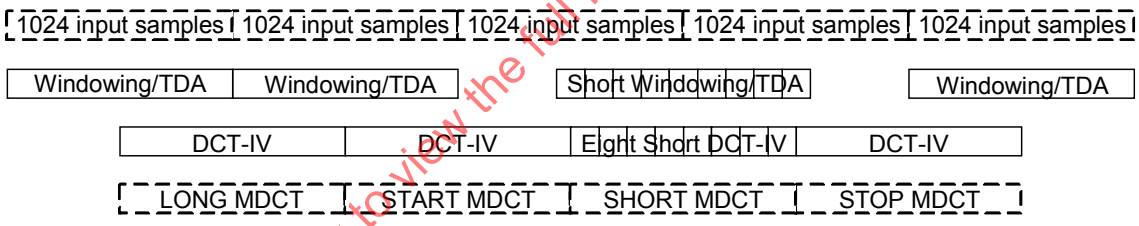

Figure 12.10 – Structure of MDCT for LONG, START, SHORT, STOP sequence

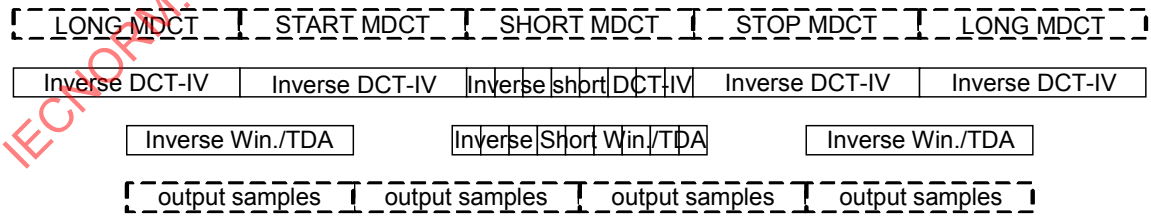

Figure 12.11 – Structure of IMDCT for LONG, START, SHORT, STOP, LONG sequence

12.5.10.2.5 The IntMDCT

The IntMDCT is an invertible integer approximation of the MDCT. Two versions of this transform are used here, relying on the same algorithm:

Mono-IntMDCT: This version provides the IntMDCT spectrum of one channel.

Stereo-IntMDCT: In case of a channel pair element with common_window and use_stereo_intmdct switched on, this version is used. It provides the mid/side IntMDCT spectrum of the left and right channel simultaneously.

Decomposition of MDCT into lifting steps

For the IntMDCT, all calculations are decomposed into so-called lifting steps, allowing to introduce a rounding operation without losing the perfect reconstruction property.

In the forward IntMDCT the Windowing/TDA block is calculated by 3N/2 lifting steps:

$$\begin{pmatrix} x(k) \\ x(N-1-k) \end{pmatrix} \mapsto \begin{pmatrix} 1 & -\frac{w(N-1-k)-1}{w(k)} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ -w(k) & 1 \end{pmatrix} \begin{pmatrix} 1 & -\frac{w(N-1-k)-1}{w(k)} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x(k) \\ x(N-1-k) \end{pmatrix}$$

$$k = 0,\ldots,N/2-1$$

In the inverse IntMDCT the Windowing/TDA block is calculated by:

$$\begin{pmatrix} x(k) \\ x(N-1-k) \end{pmatrix} \mapsto \begin{pmatrix} 1 & \frac{w(N-1-k)-1}{w(k)} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ w(k) & 1 \end{pmatrix} \begin{pmatrix} 1 & \frac{w(N-1-k)-1}{w(k)} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x(k) \\ x(N-1-k) \end{pmatrix}$$

$$k = 0,\ldots,N/2-1$$

These calculations are mathematically equivalent to the one described above, because the window function *w(k)* fulfils the TDAC condition

$$w(k)^2 + w(N-1-k)^2 = 1, \quad k = 0,\ldots,N/2-1$$

After each lifting step, a rounding operation is applied to stay in the integer domain.

Calculation of Int-DCT-IV

For the IntMDCT, the DCT-IV is calculated in an invertible integer fashion, called the Int-DCT-IV. The Int-DCT-IV of length *N* is implemented by so-called multi-dimensional lifting steps. They have the following general structure:

$$\begin{pmatrix} I_{N/2} & 0 \\ A & I_{N/2} \end{pmatrix}$$

with the identity matrix $I_{N/2}$ of size *N*/2 and an arbitrary (*N*/2)x(*N*/2) matrix *A*.

Applying this block matrix means that the first half of the input values are processed by the matrix *A* and then added to the second half of the input values.

For an integer approximation, the output values of the matrix *A* are rounded to integer before adding them. Figure 12.12 illustrates this process.

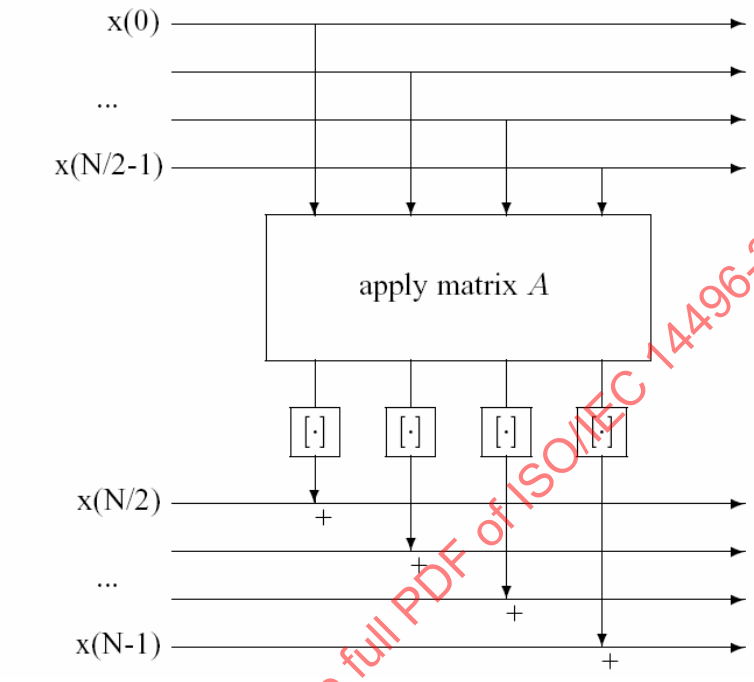

Figure 12.12 – Forward step for multi-dimensional lifting including rounding

This process can be inverted by

$$\begin{pmatrix} I_{N/2} & 0 \\ -A & I_{N/2} \end{pmatrix}$$

i.e. the same matrix *A* is applied to the first half of the values and the resulting values are subtracted from the second half of the input values.

For the invertible integer approximation, the output values of *A* are rounded to integer before subtracting them.

To apply this structure to the IntMDCT, the DCT-IV of length N is decomposed in the following way:

$$DCTIV_N = \begin{pmatrix} I & 0 \\ 0 & -I \end{pmatrix} \begin{pmatrix} 1 & & & & & cs(0) \\ & \ddots & & & \iddots & \\ & & 1 & cs(N/2-1) & & \\ & & & 1 & & \\ & & & & \ddots & \\ & & & & & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & & s(N/2-1) & 1 & \\ & \iddots & & & & \ddots \\ s(0) & & & & & 1 \end{pmatrix} \begin{pmatrix} 1 & & & & & cs(0) \\ & \ddots & & & \iddots & \\ & & 1 & cs(N/2-1) & & \\ & & & 1 & & \\ & & & & \ddots & \\ & & & & & 1 \end{pmatrix}$$

$$\begin{pmatrix} I & \frac{1}{2}\sqrt{2}DCTIV_{N/2} \\ 0 & I \end{pmatrix} \begin{pmatrix} I & 0 \\ -\sqrt{2}DCTIV_{N/2} & I \end{pmatrix} \begin{pmatrix} I & \frac{1}{2}\sqrt{2}DCTIV_{N/2} \\ 0 & I \end{pmatrix} \begin{pmatrix} I & -\frac{1}{2}I \\ 0 & I \end{pmatrix} \begin{pmatrix} I & 0 \\ I & I \end{pmatrix} QP$$

with the values

$$s(k) = \sin\frac{(2k+1)\pi}{4N}, \quad cs(k) = \frac{\cos\frac{(2k+1)\pi}{4N} - 1}{\sin\frac{(2k+1)\pi}{4N}}, \quad k = 0,\ldots,N/2-1$$

and the permutation matrices P and Q with

$$P_{4k,4k} = P_{4k+1,4k+1} = P_{4k+2,4k+3} = P_{4k+3,4k+2} = 1, \quad k = 0,\ldots,N/4-1$$
$$P_{k,l} = 0 \quad else$$

i.e. every second pair of values is swapped, and

$$Q_{k,2k} = Q_{N/2+k,2k+1} = 1, \quad k = 0,\ldots,N/2-1$$
$$Q_{k,l} = 0 \quad else$$

i.e. the even indices are arranged in the first half, the odd indices are arranged in the second half.

Thus the DCT-IV is basically decomposed into 8 multi-dimensional lifting steps.

The corresponding inverse lifting decomposition for the inverse DCT-IV is given by:

$$DCTIV_N^{-1} = P^{-1}Q^{-1}\begin{pmatrix} I & 0 \\ -I & I \end{pmatrix}\begin{pmatrix} I & \frac{1}{2}I \\ 0 & I \end{pmatrix}\begin{pmatrix} I & -\frac{1}{2}\sqrt{2}DCTIV_{N/2} \\ 0 & I \end{pmatrix}\begin{pmatrix} I & 0 \\ \sqrt{2}DCTIV_{N/2} & I \end{pmatrix}$$

$$\begin{pmatrix} I & -\frac{1}{2}\sqrt{2}DCTIV_{N/2} \\ 0 & I \end{pmatrix}\begin{pmatrix} 1 & & & & -cs(0) \\ & \ddots & & \iddots & \\ & & 1 & -cs(N/2-1) & \\ & & & 1 & \\ & & & & \ddots \\ & & & & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & & & & \\ & \ddots & & & \\ & & 1 & & \\ & & -s(N/2-1) & 1 & \\ & \iddots & & & \ddots \\ -s(0) & & & & 1 \end{pmatrix}\begin{pmatrix} 1 & & & & -cs(0) \\ & \ddots & & \iddots & \\ & & 1 & -cs(N/2-1) & \\ & & & 1 & \\ & & & & \ddots \\ & & & & 1 \end{pmatrix}\begin{pmatrix} I & 0 \\ 0 & -I \end{pmatrix}$$

Calculation of Stereo-Int-DCT-IV used for Stereo-IntMDCT

In case of a stereo signal coded as a channel_pair_element with common_window and use_stereo_intmdct switched on, the DCT-IV is calculated for both channels in one step, including the M/S calculation. This is achieved by using the decomposition of the Int-DCT-IV described above, and omitting the three stages of one-dimensional lifting steps and the two permutations, resulting in:

$$\begin{pmatrix} I & 0 \\ 0 & -I \end{pmatrix}\begin{pmatrix} I & \frac{1}{2}\sqrt{2}DCTIV_N \\ 0 & I \end{pmatrix}\begin{pmatrix} I & 0 \\ -\sqrt{2}DCTIV_N & I \end{pmatrix}\begin{pmatrix} I & \frac{1}{2}\sqrt{2}DCTIV_N \\ 0 & I \end{pmatrix}\begin{pmatrix} I & -\frac{1}{2}I \\ 0 & I \end{pmatrix}\begin{pmatrix} I & 0 \\ I & I \end{pmatrix}$$

$$= \begin{pmatrix} \frac{1}{2}\sqrt{2}DCTIV_N & \frac{1}{2}\sqrt{2}DCTIV_N \\ \frac{1}{2}\sqrt{2}DCTIV_N & -\frac{1}{2}\sqrt{2}DCTIV_N \end{pmatrix}$$

$$= \begin{pmatrix} \frac{1}{2}\sqrt{2} * I & \frac{1}{2}\sqrt{2} * I \\ \frac{1}{2}\sqrt{2} * I & -\frac{1}{2}\sqrt{2} * I \end{pmatrix}\begin{pmatrix} DCTIV_N & 0 \\ 0 & DCTIV_N \end{pmatrix}$$

Hence, this simplification of the DCT-IV algorithm results in an integrated calculation of the M/S matrix and the DCT-IV for the left and the right channel. In this mode of the IntMDCT the DCT-IV operates at a length of *N* instead of *N*/2.

The corresponding inverse lifting decomposition for the inverse Stereo-Int-DCT-IV is given by:

$$\begin{pmatrix} I & 0 \\ -I & I \end{pmatrix}\begin{pmatrix} I & \frac{1}{2}I \\ 0 & I \end{pmatrix}\begin{pmatrix} I & -\frac{1}{2}\sqrt{2}DCTIV_N \\ 0 & I \end{pmatrix}\begin{pmatrix} I & 0 \\ \sqrt{2}DCTIV_N & I \end{pmatrix}\begin{pmatrix} I & -\frac{1}{2}\sqrt{2}DCTIV_N \\ 0 & I \end{pmatrix}\begin{pmatrix} I & 0 \\ 0 & -I \end{pmatrix}$$

$$= \begin{pmatrix} \frac{1}{2}\sqrt{2}DCTIV_N & \frac{1}{2}\sqrt{2}DCTIV_N \\ \frac{1}{2}\sqrt{2}DCTIV_N & -\frac{1}{2}\sqrt{2}DCTIV_N \end{pmatrix}$$

$$= \begin{pmatrix} DCTIV_N^{-1} & 0 \\ 0 & DCTIV_N^{-1} \end{pmatrix}\begin{pmatrix} \frac{1}{2}\sqrt{2}*I & \frac{1}{2}\sqrt{2}*I \\ \frac{1}{2}\sqrt{2}*I & -\frac{1}{2}\sqrt{2}*I \end{pmatrix}$$

Noise shaping

In the lifting steps where time-domain signals are processed, the rounding operations are connected to an error feedback to provide a spectral shaping of the approximation noise.

This approximation noise affects the lossless coding efficiency mainly in the high frequency region where audio signals usually contain a very small amount of energy, especially at sampling rates of 96 kHz and higher. Hence, a low-pass characteristic of the approximation noise improves the lossless coding efficiency.

A first-order noise shaping filter is used, as illustrated in Figure 12.13.

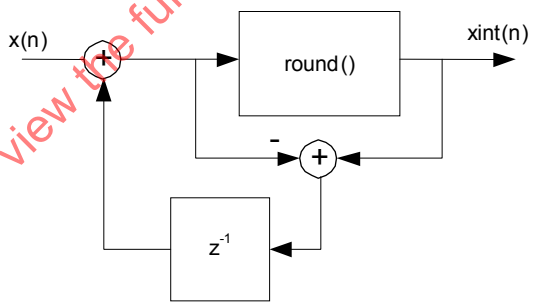

Figure 12.13 – Noise shaping filter for IntMDCT

For the IntMDCT this filter is applied to the three stages of lifting steps in the Windowing/TDA processing and to the first rounding stage of the Int-DCT-IV processing.

For the inverse IntMDCT the same filter is applied to the three stages of lifting steps in the Windowing/TDA processing and to the last rounding stage of the inverse Int-DCT-IV processing.

12.5.10.3 Algorithm for IntMDCT and inverse IntMDCT

Arithmetics

All operations are based on integer arithmetics. The following formats are used:

- INT32 for input, output and intermediate values, pre-defined fixed-point coefficients

- INT64 for multiplications with fixed-point coefficients, results are shifted and stored in INT32 immediately after each multiplication

Basic definitions for IntMDCT:

```
SINE_DATA_SIZE = 8192
SHIFT = 30
SHIFT_FOR_ERROR_FEEDBACK = 6
```

All floating point operations in the algorithm are performed in a fixed-point fashion. The number of fractional bits is given by *SHIFT*.

The necessary floating point coefficients for the multiplications in the lifting steps and for the intermediate fixed-point calculations are stored as fixed-point values in INT32:

```
INT32_coeff = nearestint((1<<SHIFT)*FLOAT_coeff);
```

The following float coefficients are stored in this way:

```
sineData[k] = sin(k*pi/(2*SINE_DATA_SIZE)), k=0,...,SINE_DATA_SIZE/2
```

defined in *sineData[SINE_DATA_SIZE/2+1]* (see 12.5.11).

```
sineData_cs[k] =
(1-cos(k*pi/(2*SINE_DATA_SIZE)))/sin(k*pi/(2*SINE_DATA_SIZE)),
        k=0,...,SINE_DATA_SIZE/2
```

defined in *sineData_cs[SINE_DATA_SIZE/2+1]* (see 12.5.11).

The corresponding values for the KBD window are pre-defined in *KBDWindow[SINE_DATA_SIZE/2]* resp. *KBDWindow_cs[SINE_DATA_SIZE/2]* (see Annex B).

Basic functions for IntMDCT:

```
INT32 multShiftINT32(INT32 x, INT32 y, int shift) {
      return ( (INT32)((INT64)x*y)>>shift) );
}

INT32 multShiftRoundINT32(INT32 x, INT32 y, int shift) {
      return ( (multShiftINT32(x,y,shift-1) + 1 ) >>1 );
}

INT32 shiftRoundINT32(INT32 y, INT32 shift) {
      return (((y>>(shift-1))+1)>>1);
}

INT32 shiftRoundINT32withErrorFeedback(INT32 y, INT32* errorFeedback, int shift) {
       y += *errorFeedback;
       result = shiftRoundINT32(y,shift);
       *errorFeedback = (result << shift) - y;
       return (result);
}

void rotateINT32(int index, INT32 xin, INT32 yin, INT32* xout, INT32* yout) {
       xin += multShiftINT32(-sineData_cs[index], yin, SHIFT);
       yin += multShiftINT32(sineData[index], xin, SHIFT);
       xin += multShiftINT32(-sineData_cs[index], yin, SHIFT);
```

```
        *xout = xin;
        *yout = yin;
}

void multHalfSqrt2(INT32* x) {
     *x = multShiftINT32(sineData[SINE_DATA_SIZE/2], *x, SHIFT);
}

void rotatePlusMinusINT32(INT32 xin, INT32 yin, INT32* xout, INT32* yout) {
     xtmp = xin;
     ytmp = yin;
     *xout = xtmp + ytmp;
     *yout = xtmp - ytmp;
}

void rotatePlusMinusNormINT32(INT32 xin, INT32 yin, INT32* xout, INT32* yout) {
     rotatePlusMinusINT32(xin, yin, xout, yout);
     multHalfSqrt2(xout);
     multHalfSqrt2(yout);
}

void addINT32(INT32* xin, INT32* xout, int N) {
  for (i=0; i<N; i++)
    xout[i] += xin[i];
}

void diffINT32(INT32* xin, INT32* xout, int N) {
  for (i=0; i<N; i++)
    xout[i] -= xin[i];
}

void copyINT32(INT32* xin, INT32* xout, int N) {
  for (i=0; i<N; i++)
    xout[i] = xin[i];
}

void shiftLeftINT32(INT32* x, int N, int shift) {
  for (i=0; i<N; i++)
    x[i] <<= shift;
}

void shiftRightINT32(INT32* x, int N, int shift) {
  for (i=0; i<N; i++)
    x[i] >>= shift;
}
```

Definitions for windowing/TDA algorithm:

For forward Windowing/TDA:

```
direction = 1;
```

For inverse Windowing/TDA:

```
direction = -1;
```

Algorithm for forward resp. inverse windowing/TDA:

```
  if (windowShape == 0) {
    window = sineData;
    window_cs = sineData_cs;
```

```
  } else {
    window = KBDWindow;
    window_cs = KBDWindow_cs;
  }
  errorFeedback = 0;
  for (i=0; i<N/2; i++) {
    tmp = multShiftINT32(-window_cs[(2*i+1)*SINE_DATA_SIZE/(2*N)],
            signal[N-1-i],
            SHIFT - SHIFT_FOR_ERROR_FEEDBACK);
    signal[i] -= direction *
      shiftRoundINT32withErrorFeedback(tmp,
                    &errorFeedback,
                    SHIFT_FOR_ERROR_FEEDBACK);
  }
  errorFeedback = 0;
  for (i=0; i<N/2; i++) {
    tmp = multShiftINT32(window[(2*i+1)*SINE_DATA_SIZE/(2*N)],
            signal[i],
            SHIFT - SHIFT_FOR_ERROR_FEEDBACK);
    signal[N-1-i] -= direction *
      shiftRoundINT32withErrorFeedback(tmp,
                    &errorFeedback,
                    SHIFT_FOR_ERROR_FEEDBACK);
  }
  errorFeedback = 0;
  for (i=0; i<N/2; i++) {
    tmp = multShiftINT32(-window_cs[(2*i+1)*SINE_DATA_SIZE/(2*N)],
            signal[N-1-i],
            SHIFT - SHIFT_FOR_ERROR_FEEDBACK);
    signal[i] -= direction *
      shiftRoundINT32withErrorFeedback(tmp,
                    &errorFeedback,
                    SHIFT_FOR_ERROR_FEEDBACK);
  }
```

Algorithm for the forward resp. inverse Int-DCT-IV:

The input values are transformed to the forward resp. inverse Int-DCT-IV values in-place.

In case of the Mono IntMDCT, *signal0[0,...,N-1]* represents the input values of one channel, *signal1[0,...,N/2-1]* corresponds to the upper half values *signal0[N/2,...,N-1]*.

If the Stereo IntMDCT is used, the length N is twice the frame length in the Int-DCT-IV algorithm; *signal0[0,...,N/2-1]* represents the left channel input values and *signal1[0,...,N/2-1]* represents the right channel input values.

A temporary buffer *liftBuffer[k]* ,k=0,...,N/2-1 of INT32 values is used.

Algorithm for forward Int-DCT-IV:

Permutation P:

```
if (Mono_IntMDCT) {
   for (i=0; i<N; i+=4) {
       (signal0[i+2], signal0[i+3]) = (signal0[i+3], signal0[i+2]);
   }
}
```

Permutation Q:

```
if (Mono_IntMDCT) {
   for (i=0; i<N; i++) {
       temp[i] = signal[i];
   }
   for (i=0; i<N/2; i++) {
       signal[i] = temp[2*i];
       signal[N/2+i] = temp[2*i+1];
   }
}
```

Apply lifting steps:

```
  addINT32(signal0, signal1, N/2);

  liftingStep2and3(signal1, liftBuffer, N);
  addINT32(liftBuffer, signal0, N/2);

  liftingStep4(signal0, liftBuffer, N);
  addINT32(liftBuffer, signal1, N/2);

  liftingStep5and6(signal1, liftBuffer, N, Mono_IntMDCT);
  addINT32(liftBuffer, signal0, N/2);

  if (Mono_IntMDCT) {
    liftingStep7(signal0, liftBuffer, N);
    addINT32(liftBuffer, signal1, N/2);

    liftingStep8(signal1, liftBuffer, N);
    addINT32(liftBuffer, signal0, N/2);
  }
```

Multiply with -1:

```
  for (k=0; k<N/2; k++) {
    signal1[k] *= -1;
  }
```

Algorithm for inverse Int-DCT-IV:

Multiply with -1:

```
  for (k=0; k<N/2; k++) {
    signal1[k] *= -1;
  }
```

Apply inverse lifting steps:

```
  if (Mono_IntMDCT) {
    liftingStep8(signal1, liftBuffer, N);
    diffINT32(liftBuffer, signal0, N/2);

    liftingStep7(signal0, liftBuffer, N);
    diffINT32(liftBuffer, signal1, N/2);
  }
```

```
  liftingStep5and6(signal1, liftBuffer, N, Mono_IntMDCT);
  diffINT32(liftBuffer, signal0, N/2);

  liftingStep4(signal0, liftBuffer, N);
  diffINT32(liftBuffer, signal1, N/2);

  liftingStep2and3(signal1, liftBuffer, N);
  diffINT32(liftBuffer, signal0, N/2);

  diffINT32(signal0, signal1, N/2);
```

Inverse permutation Q:

```
  if (Mono_IntMDCT) {
    for (i=0; i<N; i++) {
      temp[i] = signal[i];
    }
    for (i=0; i<N/2; i++) {
      signal[2*i] = temp[i];
      signal[2*i+1] = temp[n/2+i];
    }
  }
```

Inverse permutation P:

```
  if (Mono_IntMDCT) {
    for (i=0; i<N; i+=4) {
      (signal0[i+2], signal0[i+3]) = (signal0[i+3], signal0[i+2]);
    }
  }
```

Lifting steps for forward and inverse Int-DCT-IV:

```
void liftingStep2and3(INT32* signal1, INT32* liftBuffer, int N) {
  copyINT32(signal1, liftBuffer, N/2);
  shiftNormalize = DCTIVsqrt2_fixpt(liftBuffer, N/2) + 1;
  if (shiftNormalize > SHIFT_FOR_ERROR_FEEDBACK) {
    shiftRightINT32(liftBuffer, N/2,
shiftNormalize - SHIFT_FOR_ERROR_FEEDBACK);
    shiftNormalize = SHIFT_FOR_ERROR_FEEDBACK;
  }
  for (k=0; k<N/2; k++) {
    liftBuffer[k] -= signal1[k] << (shiftNormalize - 1);
  }
  errorFeedback = 0;
  for (k=0; k<N/2; k++) {
    liftBuffer[k] = shiftRoundINT32withErrorFeedback(liftBuffer[k],
&errorFeedback, shiftNormalize);
  }
}

void liftingStep4(INT32* signal0, INT32* liftBuffer, int N) {
  copyINT32(signal0, liftBuffer, N/2);
  shiftNormalize = DCTIVsqrt2_fixpt(liftBuffer, N/2);
  for (k=0; k<N/2; k++) {
    liftBuffer[k] = -shiftRoundINT32(liftBuffer[k],shiftNormalize);
  }
}

void liftingStep5and6(INT32* signal1, INT32* liftBuffer, int N, int mono) {
  copyINT32(signal1, liftBuffer, N/2);
  shiftNormalize = DCTIVsqrt2_fixpt(liftBuffer, N/2);
  shiftRightINT32(liftBuffer, N/2, shiftNormalize);
  if (mono) {
    for (k=0; k<N/2; k++)
```

```
      liftBuffer[k] += multShiftINT32(-sineData_cs[step*(2*k+1)],
signal1[N/2-1-k],(SHIFT-1));   }
  for (k=0; k<N/2; k++)
    liftBuffer[k] = ((liftBuffer[k]+1)>>1);
}

void liftingStep7(INT32* signal0, INT32* liftBuffer, int N) {
  for (k=0; k<N/2; k++)
    liftBuffer[N/2-1-k] =
multShiftRoundINT32(sineData[(2*k+1)*SINE_DATA_SIZE/(2*N)],
signal0[k],SHIFT);
}

void liftingStep8(INT32* signal1, INT32* liftBuffer, int N) {
  for (k=0; k<N/2; k++)
    liftBuffer[k] =
multShiftRoundINT32(-sineData_cs[(2*k+1)*SINE_DATA_SIZE/(2*N)],
signal1[N/2-1-k],SHIFT);
}
```

Algorithm for SQRT(2)*DCT-IV:

Both in the forward and the inverse Int-DCT-IV the calculation of SQRT(2)*DCT-IV is required. This calculation is performed in a deterministic fixed-point fashion:

```
int DCTIVsqrt2_fixpt(INT32 *data, int N) {
  preShift = msbHeadroomINT32(data, N) - 1;
  if (preShift > 15) preShift = 15;
  if (preShift < 0) preShift = 0;
  shiftLeftINT32(data, N, preShift);
  preModulationDCT_fixpt(data, xr, xi, N);
  fftShift = srfft_fixpt(xr, xi, N/2);
  postModulationDCT_fixpt(xr, xi, data, N);
  shiftNormalize = (log2int(N) - 2) / 2 + preShift - fftShift;
  sqrt2Normalize = (log2int(N) - 2) & 2;
  if (sqrt2Normalize) {
    for (i=0; i<N; i++)
      multHalfSqrt2(&data[i]);
  }
  return shiftNormalize;
}
```

Pre-modulation for DCT-IV:

```
void preModulationDCT_fixpt(INT32 *x, INT32 *xr, INT32 *xi, int N) {
  for(i=0;i<N/4;i++) {
    rotateINT32((4*i+1)*SINE_DATA_SIZE/(2*N),
        x[N-1-2*i],x[2*i],
        &xi[i],&xr[i]);
    rotateINT32((4*i+3)*SINE_DATA_SIZE/(2*N),
        x[2*i+1],-x[N-2-2*i],
        &xr[N/2-1-i],&xi[N/2-1-i]);
  }
}
```

Post-modulation for DCT-IV:

```
void postModulationDCT_fixpt(INT32 *xr, INT32 *xi, INT32 *x, int N) {
  x[0] = xr[0];
```

```
  x[N-1] = -xi[0];
  for(i=1;i<N/4;i++){
    rotateINT32(2*i*SINE_DATA_SIZE/N,
        xr[i],-xi[i],
        &x[2*i],&x[N-2*i-1]);
    rotateINT32(2*i*SINE_DATA_SIZE/N,
        xr[N/2-i],xi[N/2-i],
        &x[2*i-1],&x[N-2*i]);
  }
  rotatePlusMinusNormINT32(xr[N/4],xi[N/4],
              &x[N/2],&x[N/2-1]);
}
```

Split-Radix FFT:

```
int srfft_fixpt(INT32 *xr, INT32 *xi, int N) {
  numShifts = 0;
  /* L = 1,2,4,...,N/2 */
  for (L=1; L<N; L*=2) {
    M = N/L; /* M = N, N/2,...,2 */
    M2 = M/2;
    M4 = M2/2;
    /* L: number of sub-blocks
       M: length of sub-block */
    numShifts += shiftIfRequired(xr, xi, N);
    for (l=0; l<L; l++) {
      /* butterfly on (x[k],x[M2+k]), k = 0,...,N2-1 on each sub-block */
      for (k=0; k<M2; k++) {
    rotatePlusMinusINT32(xr[l*M+k],xr[l*M+M2+k],
                &xr[l*M+k],&xr[l*M+M2+k]);
    rotatePlusMinusINT32(xi[l*M+k],xi[l*M+M2+k],
                &xi[l*M+k],&xi[l*M+M2+k]);
      }
    }
    numShifts += shiftIfRequired(xr, xi, N);
    if (M > 2) {
      for (l=0; l<L; l++) {
    if (srfftIndex(l) == 0) {
      /* x[N2+N4+k] -> -j*x[N2+N4+k] , k = 0,...,N4-1 on each sub-block
*/
      for (k=0; k<M4; k++) {
        swap(&xr[l*M+M2+M4+k],&xi[l*M+M2+M4+k]);
        xi[l*M+M2+M4+k] *= -1;
      }
    } else {
      /* complex multiplications */
      for (k = 1; k < M4; k++) {
        rotateINT32(4*k*SINE_DATA_SIZE/(2*M),
            xi[l*M+k],xr[l*M+k],
            &xi[l*M+k],&xr[l*M+k]);
        rotateINT32(4*k*SINE_DATA_SIZE/(2*M),
            xi[l*M+M2-k],-xr[l*M+M2-k],
            &xr[l*M+M2-k],&xi[l*M+M2-k]);
      }
      for (k = 1; 3*k < M4; k++) {
        rotateINT32(4*3*k*SINE_DATA_SIZE/(2*M),
            xi[l*M+M2+k],xr[l*M+M2+k],
            &xi[l*M+M2+k],&xr[l*M+M2+k]);
        rotateINT32(4*3*k*SINE_DATA_SIZE/(2*M),
            -xi[l*M+M-k],xr[l*M+M-k],
            &xr[l*M+M-k],&xi[l*M+M-k]);
      }
      for (; 3*k < 2*M4; k++) {
        rotateINT32(4*(M2-3*k)*SINE_DATA_SIZE/(2*M),
            xi[l*M+M2+k],-xr[l*M+M2+k],
            &xr[l*M+M2+k],&xi[l*M+M2+k]);
```

```
        rotateINT32(4*(M2-3*k)*SINE_DATA_SIZE/(2*M),
            -xi[l*M+M-k],-xr[l*M+M-k],
            &xi[l*M+M-k],&xr[l*M+M-k]);
      }
      for (; 3*k < 3*M4; k++) {
        rotateINT32(4*(3*k-M2)*SINE_DATA_SIZE/(2*M),
            -xr[l*M+M2+k],xi[l*M+M2+k],
            &xi[l*M+M2+k],&xr[l*M+M2+k]);
        rotateINT32(4*(3*k-M2)*SINE_DATA_SIZE/(2*M),
            -xr[l*M+M-k],-xi[l*M+M-k],
            &xr[l*M+M-k],&xi[l*M+M-k]);
      }
      rotatePlusMinusNormINT32(xi[l*M+M4],xr[l*M+M4],
                  &xr[l*M+M4],&xi[l*M+M4]);
      rotatePlusMinusNormINT32(-xr[l*M+M-M4],xi[l*M+M-M4],
                  &xr[l*M+M-M4],&xi[l*M+M-M4]);
    }
      }
    }
  }
  bit_reverse_fixpt(xr,N);
  bit_reverse_fixpt(xi,N);
  return numShifts;
}
```

Basic functions for FFT:

```
int msbHeadroomINT32(INT32 *x, int N) {
  max = 0;
  for (i=0; i<N; i++) {
    max |= ABS(x[i]);
  }
  return (30-log2int(max));
}

int shiftIfRequired(INT32 *xr, INT32 *xi, int N) {
  shiftRequired = 0;
  if ((!msbHeadroomINT32(xr,N))||(!msbHeadroomINT32(xi,N))) {
    shiftRequired = 1;
    shiftRightINT32(xr, N, 1);
    shiftRightINT32(xi, N, 1);
  }
  return shiftRequired;
}

void bit_reverse_fixpt(INT32 *x, int N) {
  for (m=1,j=0; m<N-1; m++) {
    for(k=N>>1; k>(j^=k)&k); k>>=1);
    if (j>m) swap(&x[m],&x[j]);
  }
}

srfftIndexTable[32] = {0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0,
                       0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1};

int srfftIndex(int l) {
  return srfftIndexTable[(srfftIndexTable[l>>4]<<4)|(l&0xF)];
}
```

Forward and inverse Integer Mid/Side processing:

```
void IntMidSideINT32(INT32* l, INT32* r) /* L/R -> M/S */
{
  m = *l;
  s = *r;
  m += multShiftRoundINT32(-sineData_cs[SINE_DATA_SIZE/2], s, SHIFT);
```

```
  s += multShiftRoundINT32( sineData[SINE_DATA_SIZE/2],    m, SHIFT);
  m += multShiftRoundINT32(-sineData_cs[SINE_DATA_SIZE/2], s, SHIFT);
  *l = s;
  *r = m;
}


void InverseIntMidSideINT32(INT32* l, INT32* r) /* M/S -> L/R */
{
  m = *l;
  s = *r;
  s -= multShiftRoundINT32(-sineData_cs[SINE_DATA_SIZE/2], m, SHIFT);
  m -= multShiftRoundINT32( sineData[SINE_DATA_SIZE/2],    s, SHIFT);
  s -= multShiftRoundINT32(-sineData_cs[SINE_DATA_SIZE/2], m, SHIFT);
  *l = s;
  *r = m;
}
```

12.5.11 Computation of table values based on compact tables

The values of the tables *sineData*, *sineData_cs*, *thrMantissa()*, and *invQuantMantissa()* are computed from the compact tables in Annex B. This is described in the following pseudo code:

```
/* interpolate value between v0 = data[0] and v8 = data[8],
   using additionally vm8 = data[-8] and v16 = data[16] */
INT32 interpolateValue1to7(INT32 vm8,
                           INT32 v0,
                           INT32 v8,
                           INT32 v16,
                           INT32 l)
{
  INT32 value;
  INT32 d1, d2, d3;

  d1 = 2*(v8-v0); /* 1 add, 1 shift */
  d2 = v8-vm8; /* 1 add */
  d3 = v16-v0; /* 1 add */

  if (l==1) {
    value = v0 + ( ( 8*d2 - d2 + d1 + 64 ) >> 7 ); /* 4 adds, 2 shifts */
  } else if (l==2) {
    value = v0 + ( ( 2*d2 + d2 + d1 + 16 ) >> 5 ); /* 4 adds, 2 shifts */
  } else if (l==3) {
    value = v0 + ( ( 16*d2 - d2 + 8*d1 + d1 + 64 ) >> 7 ); /* 5 adds, 3 shifts */
  } else if (l==4) {
    value = v0 + ( ( d2 + d1 + 4 ) >> 3 ); /* 3 adds, 1 shifts */
  } else if (l==5) {
    value = v8 - ( ( 16*d3 - d3 + 8*d1 + d1 + 64 ) >> 7 ); /* 5 adds, 3 shifts */
  } else if (l==6) {
    value = v8 - ( ( 2*d3 + d3 + d1 + 16 ) >> 5 ); /* 4 adds, 2 shifts */
  } else if (l==7) {
    value = v8 - ( ( 8*d3 - d3 + d1 + 64 ) >> 7 ); /* 4 adds, 2 shifts */
  }
  return value;
}


INT32 interpolateFromCompactTable(int index, INT32* compactTable)
{
  INT32 value;

  j = index%8;
  k = index/8;

  if (j == 0) {
    value = compactTable[k+1];
    return value;
  }
  value = interpolateValue1to7(compactTable[k],
                               compactTable[k+1],
                               compactTable[k+2],
                               compactTable[k+3],
                               j);
  return value;
}
```

The values for the sineData and the sineData_cs tables are computed by applying

```
  sineData[index] = interpolateFromCompactTable(index, sineDataCompact);
```

resp.

```
  sineData_cs[index] = interpolateFromCompactTable(index, sineDataCompact_cs);
```

The values for the function invQuantMantissa() are computed by

```
INT32 invQuantMantissa(int quant, int res)
{
  INT32 value;
  INT32 pow_2_quat[4] = {0, 1276901417, 1518500250, 1805811301};
   /* (int)(pow(2.0,res/4.0)*(1<<SHIFT)+0.5) */

  if (quant < MAX_INV_QUANT_TABLE) {
    value = invQuantCompact[quant];
    if (res > 0) {
      value = multShiftRoundINT32(value,
                                  pow_2_quat[res],
                                  SHIFT);
    }

  } else {

    l = quant%8;
    k = quant/8;

    if (l == 0) {
      value = invQuantMantissa(k, res)<<4; /* 8^(4/3) = 16 = 2^4 */
    } else {
      value = interpolateValue1to7(invQuantMantissa(k-1, res)<<4,
                                   invQuantMantissa(k, res)<<4,
                                   invQuantMantissa(k+1, res)<<4,
                                   invQuantMantissa(k+2, res)<<4,
                                   l);
    }
  }
  return value;
}
```

The value for the function thrMantissa() are computed by

```
INT32 thrMantissa(quant, res)
{
  INT32 value;
  INT32 invQuant0;
  INT32 invQuant1;

  if (quant < MAX_THR_TABLE) {

    value = thrCompact[res][quant];

  } else {

    invQuant0 = invQuantMantissa(quant,res);
    invQuant1 = invQuantMantissa(quant+1,res);
    value = invQuant1+(((invQuant0-invQuant1)*13)>>5);

  }
  return value;
}
```

Annex 12.A
(informative)

Encoder description

12.A.1 Overview

The SLS encoder generates, for a given PCM audio input, a lossless bit-stream that can be decoded to a bit-exact reproduction of the given PCM audio by using an SLS decoder. Furthermore, the lossless stream generated by the SLS encoder can be truncated to lower bit-rates down to the bit-rate of the core MPEG-4 GA encoder. This way, the resulting bit-stream can be decoded by the SLS decoder to produce a lossy reproduction of the original audio in such a way that better signal fidelity is always achieved with higher rates in the LLE layer.

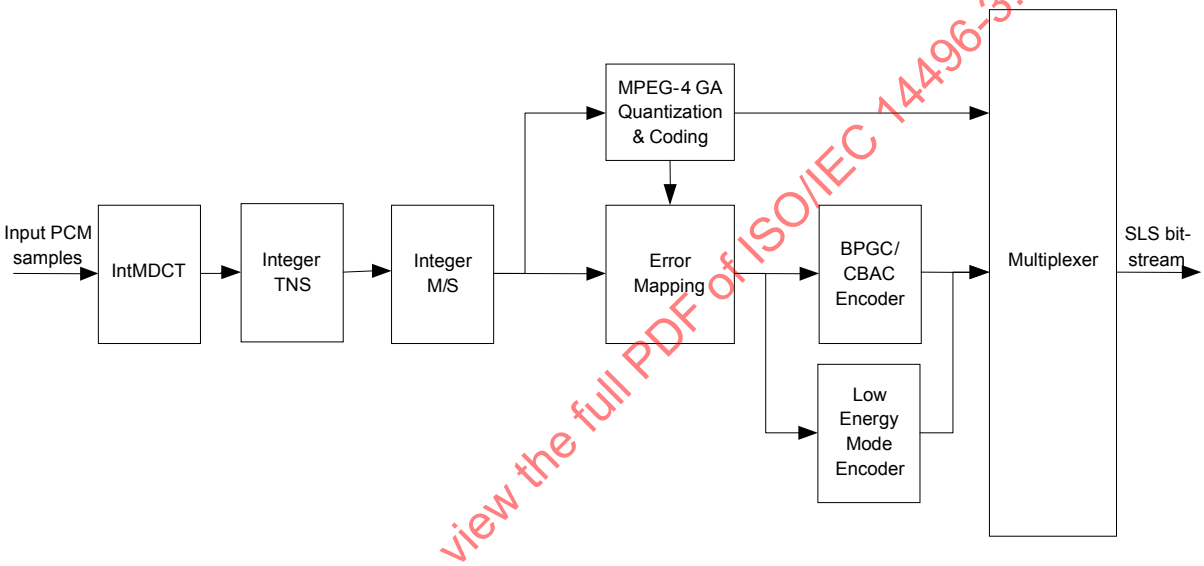

Figure 12.A.1 – Block diagram of SLS encoder

12.A.1.1 Encoding with oversampling

For the encoding process incorporating the oversampling technique, two approaches are possible: Downsampling in the MDCT domain and downsampling in the time domain.

Downsampling in the MDCT domain is illustrated in Figure 12.A.2. The input signal is processed by an IntMDCT of length osf*1024 and the first 1024 spectral values are fed into the MPEG-4 GA encoder.

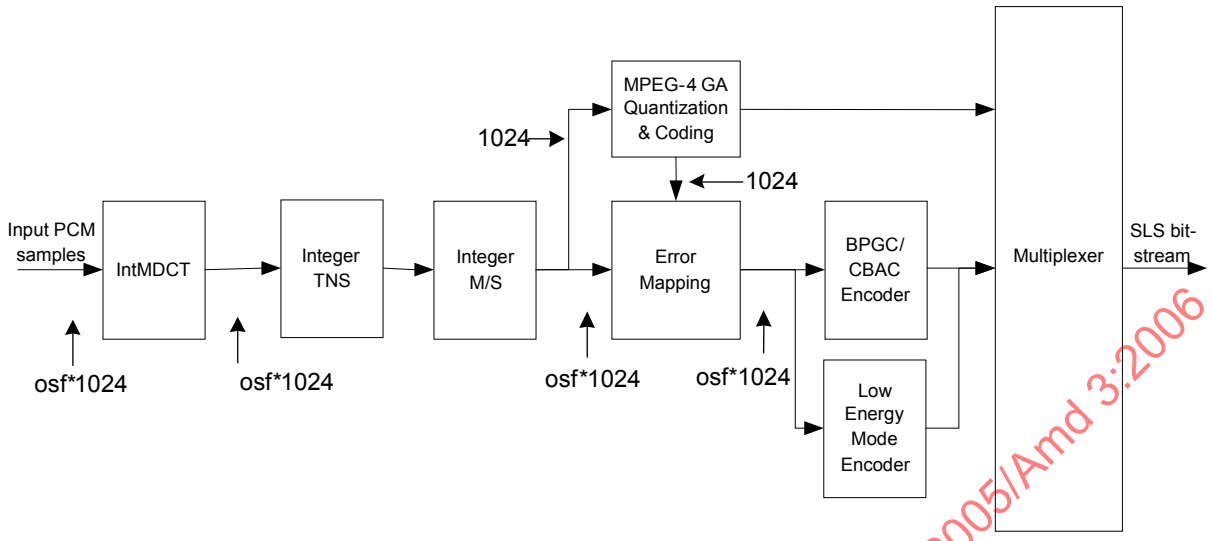

Figure 12.A.2 – Block diagram of SLS encoder with downsampling in the MDCT domain

For the second possible encoding approach, the input signal is downsampled in the time domain and the complete AAC encoding part is performed in parallel. This is illustrated in Figure 12.A.3.

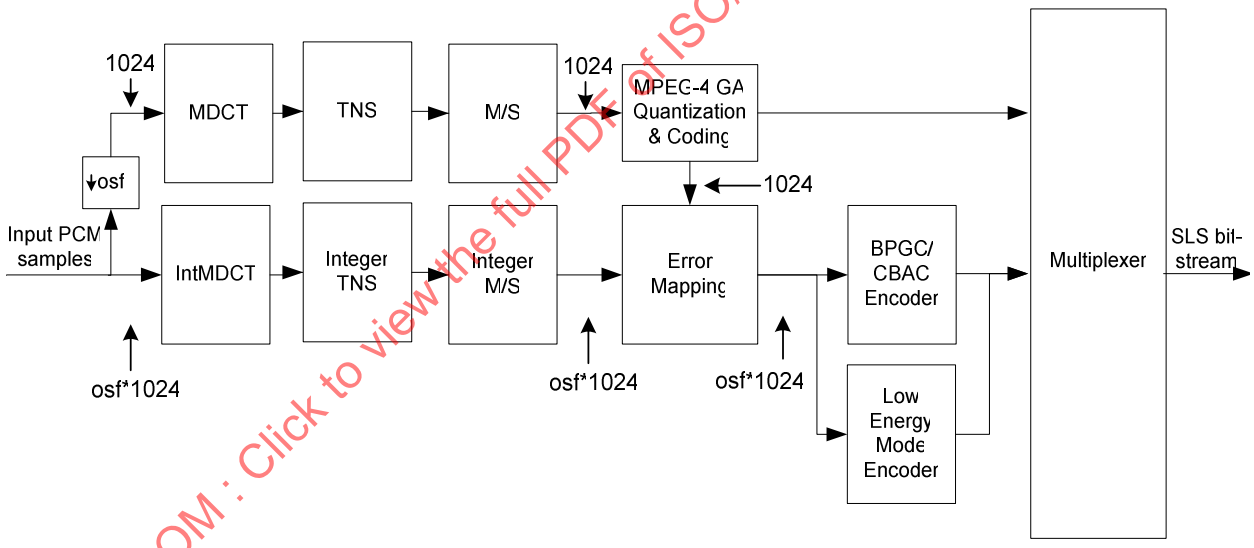

Figure 12.A.3 – Block diagram of SLS encoder with downsampling in the time domain

12.A.2 Integer MDCT

The IntMDCT is already described in the normative part.

12.A.3 Grouping and interleaving

There are two types of window used in the SLS implementation. They are the same as in the AAC windowing scheme. One is a long window with *osf**1024 IntMDCT coefficients, the other is a short window with *osf**128 IntMDCT coefficients. An integer transform is applied to a windowed audio sequence. When the short window

is employed, the set of *osf**1024 IntMDCT coefficients is handled as a matrix of 8 by *osf**128 frequency coefficients representing the time-frequency evolution of the signal over the duration of eight short windows. The same *grouping and interleaving* process adopted in AAC is followed here. To be specific, assume that before interleaving the set of *osf**1024 IntMDCT coefficients *c* are indexed as

c[*g*][*w*][*b*][*k*]

where

g is the index on groups

w is the index on windows within a group

b is the index on scale factor bands within a window

k is the index on coefficients within a scale factor band

and the right-most index varies most rapidly.

After interleaving the coefficients are indexed as

c[*g*][*b*][*w*][*k*]

In the subsequent sections, when a short window is used we assume that the signal process is performed on the interleaved spectrum for eight short window frames unless otherwise specified.

12.A.4 Integer mid/side

If the Mono IntMDCT is used for the left and the right channel, the integer M/S processing has to be applied to the scale factor bands where the M/S flag is set to '1'.

The Stereo IntMDCT delivers by default an M/S spectrum. Hence M/S has to be turned off for the scale factor bands where it is not desired.

The algorithms for both the forward and the inverse Integer Mid/Side processing are described in the normative part.

12.A.5 Normalize before AAC coding

After IntMDCT, a scale factor *core_scaling* is used to normalize the IntMDCT coefficients $c(k)$ for each scale factor band *sfb* in order to provide the AAC core layer input spectrum $c'(k)$.

$$c' = core\_scaling \cdot c,$$

where the value of the scale factor *core_scaling* is jointly determined by the type of the corresponding scale factor band and the word length of the input audio given in the following table:

Table 12.A.1 – Value of core_scaling

Word Length / sfb Type	16	20	24
Long Window (2048), M/S	32	2	$1/8$
Long Window (2048), non M/S	$32\sqrt{2}$	$2\sqrt{2}$	$\sqrt{2}/8$
Short Window (256), M/S	$8\sqrt{2}$	$\sqrt{2}/2$	$\sqrt{2}/32$
Short Window (256), non M/S	16	1	$1/16$

After normalization, the normalized c' is quantized with the core AAC quantizer, whose output quantization index i is then Huffman coded. It is then multiplexed with the necessary side information, e.g. scale factor *scale_factor(sfb)* used in the quantizer for each scale factor band *sfb*, according to the AAC bit-stream syntax, to generate the core AAC bit-stream.

12.A.6 Error mapping

In the LLE layer, an error mapping procedure is employed to remove the information that has been already coded in the core layer. The input to the error mapping module is the 1024 IntMDCT coefficients *c[sfb][k]* in the non-oversampling range and its corresponding quantized value in the core encoder *quant[sfb][k].* Its output is the IntMDCT residual spectrum *res*. The detailed error mapping procedure is already given in the normative part of this document (12.5.7).

12.A.7 BPGC/CBAC encoder

In SLS, the IntMDCT residual spectrum *res* is coded by the BPGC/CBAC coding process that consists of the following steps:

- BPGC/CBAC parameter determination
- Bit-plane coding of residual integer spectral data
- Low energy mode coding of residual integer spectral data

12.A.7.1 BPGC/CBAC parameter determination

As a first step, the maximum bit-planes *max_bp* for each scale factor band are identified. For Implicit_Band, the maximum bit-plane *M* for each residual spectral data and can be calculated from

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

where $interval[g][win][sfb][bin]$ is the AAC quantization interval that is calculated as shown in the normative part of this document.

For Explicit_Band, *M* is given as:

$$M[g][win][sfb][bin] = INT\left(\log_2\left|res[g][win][sfb][bin]\right|\right),$$

and we further define $\log_2 0 = -1$ for the above log calculation. The maximum bit-plane *max_bp* for each scalefactor band is the maximum value of *M* for spectral data that belongs to *sfb*:

$$max\_bp[g][sfb] = \max\left(M[g][win][sfb][bin]\right)$$

After finding *max_bp* for each scale factor band, the lazy plane *lazy_bp* is selected from the three possible values *max_bp*-1, *max_bp*-2, and *max_bp*-3.

12.A.7.2 Bit-plane coding of residual integer spectral data

The following pseudo code illustrates how the sign and the amplitude of the IntMDCT residual spectral data *res* are coded into the BPGC/CBAC data stream. The help element *M* for an insignificant scalefactor band is set to the value of *max_bp* in order to be compatible with the decoding process. The BPGC/CBAC coding process is performed on scale factor bands for which *lazy_bp*>0.

```
/* preparing of help elements */
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]==Implicit_band))?1:0;
        }
    cur_bp[g][sfb] = max_bp[g][sfb];
    }
}
/* BPGC/CBAC normal coding process */
while ((there exists max_bp[g][sfb]-i >= 0) && (i<LAZY_BP)){
    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++){
                            sym = (abs(res[g][win][sfb][bin])&(1<<cur_bp[g][sfb]))?1:0;
                            sgn = (sign(res[g][win][sfb][bin])+1)/2;
                            if (interval[g][win][sfb][bin]>res[g][win][sfb][bin]+(1<<cur_bp[g][sfb]) {
                               encode(sym,freq);  /* encode bit-plane cur_bp*/
                               if ((!is_sig[g][win][sfb][bin])&&(sym)){
                                   encode(sgn,freq_sign);   /* encode sign bit if necessary *    /
                                   is_sig[g][win][sfb][bin] = 1;
                               }
                            }
                        }
                    }
                }
            }
            cur_bp[g][sfb]--;
        }
    }
}

The BPGC/CBAC lazy coding mode is started after the first NUM_BP bit-planes have been coded.

/* BPGC/CBAC lazy coding process */
flush_encode(); /* flush the AC encoder before lazy coding */
while (there exists max_bp[g][sfb]-i >= 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++){
                        sym = (abs(res[g][win][sfb][bin])&(1<<cur_bp[g][sfb]))?1:0;
                        sgn = (sign(res[g][win][sfb][bin])+1)/2;
                        if (interval[g][win][sfb][bin]>res[g][win][sfb][bin]+(1<<cur_bp[g][sfb]) {
                            write_bit(sym);  /* encode bit-plane cur_bp*/
                            if ((!is_sig[g][win][sfb][bin])&&(sym)){
                               write_bit(sgn);   /* encode sign bit if necessary */
                               is_sig[g][win][sfb][bin] = 1;
                            }
                        }
                    }
                }
            }
            cur_bp[g][sfb]--;
        }
    }
}
```

The value of NUM_BP is listed in Table 12.19

12.A.7.3 Low energy mode code (LEMC) encoding

For scale factor bands with *lazy_bp* = 0, -1, -2, and -3, the residual spectral data *res* is not coded with the BPGC/CBAC. Instead, it is coded with the LEMC coding process. In the low energy mode, the amplitude of the residual spectral data *res* is first converted into binary format as listed in Table 12.26. The resulting binary string is then coded arithmetically. Note that the low energy mode coding process is performed directly after the BPGC coding process is completed.

The low energy mode coding process is illustrated with the following pseudo code:

```
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++){
                pos = 0;
                for (bin=0;bin<width;bin++){
                    if (!is_lle_ics_eof ()){
                        amp = abs(res[g][sfb][win][bin]);
                        sgn = (sign(res[g][sfb][win][bin])  + 1)/2;
                        dumb = amp;
                        while (dumb > 0) { /* binarize and encoding for non-zero res*/
                            encode(1,freq_silence[position]);
                            position++;
                            if (position>2) position = 2;
                            dumb --;
                        }
                        if (amp != (1<<(max_bp[g][win][sfb] + 1)) - 1)
                            encode(0,freq_silence[position]);  /* encode of terminating 0 */
                        if (amp)
                            encode(sgn,freq_sgn);   /* encode of sign symbol*/
                        }
                    }
                }
            }
        }
    }
}
```

The frequency assignment *freq* for BPGC/CBAC encoding and low energy mode encoding has already been given in the normative part.

The following pseudo code explains how the binary symbol is arithmetically coded in the BPGC/CBAC and the low energy mode coding processes.

Definitions:

```
#define CODE_WL    16
#define PRE_SHT            14
#define TOP_VALUE  (((long)1<<CODE_WL)-1)
#define QTR_VALUE          (TOP_VALUE/4+1)
#define HALF_VALUE (2*QTR_VALUE)
#define TRDQTR_VALUE   (3*QTR_VALUE)
```

Initialization:

```
low = 0;
high = TOP_VALUE;
fbits = 0;
```

The encoding subroutine:

```
void encode(int sym, int freq)
{
    range = (long)(high-low)+1;
    if (sym)
        high = low + (range*freq>>PRE_SHT)-1;
    else
        low = low + (range*freq>>PRE_SHT);
    for (;;)  {
            if (high<HALF_VALUE)  {
            output_bit(0);
        while (fbits > 0)  {
                output_bit (1);
                fbits --;
        }
        }  else if (low>=HALF_VALUE)  {
            output_bit(1);
        while (fbits > 0)  {
                output_bit (0);
                fbits --;
        }
            low -= HALF_VALUE;
            high -= HALF_VALUE;
        }  else if (ow>=QTR_VALUE && high<TRDQTR_VALUE)  {
            fbits += 1;
            low -= QTR_VALUE;
            high -= QTR_VALUE;
        }  else
        break;
        low = 2*low;
        high = 2*high+1;
    return;
}
```

flush the status of encode:

```
/* flush the state register of AC encoder*/
flush_encode()
{
    fbits += 1;
    if (low < QTR_VALUE)
        output_bit(0);
        while (fbits > 0)  {
                output_bit (1);
                fbits --;
        }
    else
        output_bit(1);
        while (fbits > 0)  {
                output_bit (0);
                fbits --;
}
```

12.A.8 Method of bitstream truncation by re-parsing the bitstream

The full SLS bitstream can be truncated at any given target bitrate in a simple way. The modification of the values of *lle_ics_length* does not affect the LLE decoding results before the truncation point, since *lle_ics_length* is independent from the LLE decoding procedure. The bitstream truncation can be performed as follows:

1. Read the *lle_ics_length* from the bitstream

2. Read the LLE bitstream

3. Calculate the available frame length at a given target bitrate. The simplest way to calculate the available frame length is as follows:

   target_bits = (int)(target_bitrate/2.*1024.*osf/sampling_rate+0.5)-16;

   target_bytes = (target_bits+7)/8;

   The variable *target_bitrate* represents the target bitrate in bits/sec. The variable *osf* represents the oversampling factor. The variable *sampling_rate* represents the sampling frequency of the input audio signal in Hz.

4. Update *lle_ics_length* by taking the minimum of the available frame length and the current frame length.

   lle_ics_length = min(lle_ics_length, target_bytes);

5. Generate the truncated bitstream with the updated *lle_ics_length.*

The resulting truncated bitstream is decoded with the smart arithmetic decoding method as described in 12.5.5.2.5.

Annex 12.B
(normative)

Tables

12.B.1 Tables for pre-defined fixed-point coefficients

```
define SINE_DATA_SIZE 8192

/* sin(0,...,pi/4) and +-1 */
INT32 sineDataCompact[515] = {
-1647099, 0, 1647099, 3294193, 4941281, 6588356, 8235416, 9882456,
11529474, 13176464, 14823423, 16470347, 18117233, 19764076, 21410872, 23057618,
24704310, 26350943, 27997515, 29644021, 31290457, 32936819, 34583104, 36229307,
37875426, 39521455, 41167391, 42813230, 44458968, 46104602, 47750128, 49395541,
51040837, 52686014, 54331067, 55975992, 57620785, 59265442, 60909960, 62554335,
64198563, 65842639, 67486561, 69130324, 70773924, 72417357, 74060620, 75703709,
77346620, 78989349, 80631892, 82274245, 83916404, 85558366, 87200127, 88841683,
90483029, 92124163, 93765079, 95405776, 97046247, 98686491, 100326502, 101966277,
103605812, 105245103, 106884147, 108522939, 110161476, 111799753, 113437768, 115075515,
116712992, 118350194, 119987118, 121623759, 123260114, 124896179, 126531950, 128167423,
129802595, 131437462, 133072019, 134706263, 136340190, 137973796, 139607077, 141240030,
142872651, 144504935, 146136880, 147768480, 149399733, 151030634, 152661180, 154291367,
155921191, 157550647, 159179733, 160808445, 162436778, 164064728, 165692293, 167319468,
168946249, 170572633, 172198615, 173824192, 175449360, 177074115, 178698453, 180322371,
181945865, 183568930, 185191564, 186813762, 188435520, 190056834, 191677702, 193298119,
194918080, 196537583, 198156624, 199775198, 201393302, 203010933, 204628085, 206244756,
207860942, 209476638, 211091842, 212706549, 214320755, 215934457, 217547651, 219160334,
220772500, 222384147, 223995270, 225605867, 227215933, 228825464, 230434456, 232042906,
233650811, 235258165, 236864966, 238471210, 240076892, 241682010, 243286558, 244890535,
246493935, 248096755, 249698991, 251300640, 252901697, 254502159, 256102022, 257701283,
259299937, 260897982, 262495412, 264092224, 265688415, 267283981, 268878918, 270473223,
272066891, 273659918, 275252302, 276844038, 278435122, 280025552, 281615322, 283204430,
284792871, 286380643, 287967740, 289554160, 291139898, 292724951, 294309316, 295892988,
297475964, 299058239, 300639811, 302220676, 303800829, 305380268, 306958988, 308536985,
310114257, 311690799, 313266607, 314841679, 316416009, 317989595, 319562433, 321134518,
322705848, 324276419, 325846226, 327415267, 328983538, 330551034, 332117752, 333683689,
335248841, 336813204, 338376774, 339939549, 341501523, 343062693, 344623057, 346182609,
347741347, 349299266, 350856364, 352412636, 353968079, 355522689, 357076462, 358629395,
360181484, 361732726, 363283116, 364832652, 366381329, 367929144, 369476093, 371022173,
372567379, 374111709, 375655159, 377197725, 378739403, 380280190, 381820082, 383359076,
384897167, 386434353, 387970630, 389505993, 391040440, 392573967, 394106570, 395638246,
397168991, 398698801, 400227673, 401755603, 403282588, 404808624, 406333708, 407857835,
409381002, 410903207, 412424444, 413944711, 415464004, 416982319, 418499653, 420016002,
421531363, 423045732, 424559105, 426071480, 427582852, 429093217, 430602573, 432110916,
433618242, 435124548, 436629829, 438134084, 439637307, 441139496, 442640647, 444140756,
445639820, 447137835, 448634799, 450130706, 451625555, 453119340, 454612060, 456103710,
457594286, 459083786, 460572205, 462059541, 463545789, 465030947, 466515010, 467997976,
469479840, 470960600, 472440251, 473918791, 475396216, 476872522, 478347705, 479821764,
481294693, 482766489, 484237150, 485706671, 487175049, 488642281, 490108363, 491573292,
493037064, 494499676, 495961124, 497421405, 498880516, 500338453, 501795212, 503250791,
504705185, 506158392, 507610408, 509061229, 510510853, 511959275, 513406493, 514852502,
516297300, 517740883, 519183248, 520624391, 522064309, 523502998, 524940456, 526376678,
527811662, 529245404, 530677900, 532109148, 533539144, 534967884, 536395365, 537821584,
539246538, 540670223, 542092635, 543513772, 544933630, 546352205, 547769495, 549185496,
550600205, 552013618, 553425732, 554836544, 556246051, 557654248, 559061133, 560466703,
561870954, 563273883, 564675486, 566075761, 567474703, 568872310, 570268579, 571663506,
573057087, 574449320, 575840202, 577229728, 578617896, 580004702, 581390144, 582774218,
584156920, 585538248, 586918198, 588296766, 589673951, 591049748, 592424154, 593797166,
595168781, 596538995, 597907806, 599275210, 600641203, 602005783, 603368947, 604730691,
606091012, 607449906, 608807372, 610163404, 611518001, 612871159, 614222875, 615573145,
616921967, 618269338, 619615253, 620959711, 622302707, 623644239, 624984303, 626322897,
627660017, 628995660, 630329823, 631662503, 632993696, 634323400, 635651611, 636978327,
638303543, 639627258, 640949467, 642270169, 643589359, 644907034, 646223192, 647537830,
648850943, 650162530, 651472587, 652781111, 654088099, 655393548, 656697454, 657999816,
659300629, 660599890, 661897597, 663193747, 664488336, 665781362, 667072820, 668362709,
669651026, 670937767, 672222928, 673506508, 674788504, 676068911, 677347728, 678624950,
679900576, 681174602, 682447025, 683717842, 684987051, 686254647, 687520629, 688784993,
690047736, 691308855, 692568348, 693826211, 695082441, 696337036, 697589992, 698841307,
700090977, 701339000, 702585372, 703830092, 705073155, 706314559, 707554301, 708792378,
```

```
710028787, 711263525, 712496590, 713727978, 714957687, 716185713, 717412054, 718636707,
719859669, 721080937, 722300508, 723518380, 724734549, 725949013, 727161768, 728372813,
729582143, 730789757, 731995651, 733199822, 734402269, 735602987, 736801974, 737999228,
739194745, 740388522, 741580558, 742770848, 743959390, 745146182, 746331221, 747514503,
748696026, 749875788, 751053785, 752230015, 753404474, 754577161, 755748072, 756917205,
758084557, 759250125, 760413906
};

/* (1-cos)/sin (0,...,pi/4) == tan(0,...,pi/8) */
INT32 sineDataCompact_cs[515] = {
-823550, 0, 823550, 1647101, 2470653, 3294209, 4117769, 4941333,
5764903, 6588480, 7412065, 8235658, 9059261, 9882875, 10706500, 11530138,
12353790, 13177456, 14001138, 14824836, 15648551, 16472285, 17296039, 18119812,
18943607, 19767425, 20591265, 21415130, 22239020, 23062936, 23886880, 24710851,
25534852, 26358882, 27182944, 28007038, 28831164, 29655325, 30479520, 31303752,
32128020, 32952326, 33776671, 34601055, 35425481, 36249948, 37074458, 37899011,
38723609, 39548253, 40372944, 41197681, 42022468, 42847304, 43672190, 44497128,
45322119, 46147163, 46972261, 47797414, 48622624, 49447892, 50273217, 51098602,
51924047, 52749553, 53575122, 54400754, 55226449, 56052210, 56878038, 57703932,
58529894, 59355926, 60182027, 61008200, 61834444, 62660762, 63487153, 64313620,
65140162, 65966781, 66793478, 67620255, 68447111, 69274047, 70101066, 70928168,
71755353, 72582623, 73409979, 74237422, 75064952, 75892571, 76720280, 77548080,
78375971, 79203955, 80032033, 80860205, 81688474, 82516838, 83345301, 84173862,
85002523, 85831284, 86660147, 87489113, 88318182, 89147356, 89976635, 90806021,
91635515, 92465117, 93294829, 94124652, 94954586, 95784632, 96614793, 97445068,
98275458, 99105965, 99936590, 100767333, 101598196, 102429179, 103260284, 104091512,
104922863, 105754339, 106585941, 107417669, 108249525, 109081509, 109913623, 110745868,
111578245, 112410754, 113243397, 114076174, 114909088, 115742138, 116575326, 117408652,
118242119, 119075726, 119909475, 120743367, 121577403, 122411583, 123245910, 124080383,
124915004, 125749775, 126584695, 127419766, 128254990, 129090366, 129925897, 130761582,
131597424, 132433423, 133269580, 134105896, 134942372, 135779010, 136615810, 137452774,
138289901, 139127195, 139964654, 140802281, 141640077, 142478042, 143316178, 144154485,
144992965, 145831619, 146670447, 147509452, 148348633, 149187992, 150027529, 150867247,
151707146, 152547227, 153387491, 154227939, 155068572, 155909392, 156750399, 157591594,
158432979, 159274554, 160116320, 160958280, 161800432, 162642780, 163485323, 164328063,
165171001, 166014138, 166857475, 167701013, 168544753, 169388696, 170232844, 171077197,
171921756, 172766522, 173611498, 174456682, 175302078, 176147685, 176993505, 177839539,
178685788, 179532253, 180378935, 181225835, 182072955, 182920295, 183767856, 184615640,
185463648, 186311880, 187160339, 188009024, 188857937, 189707079, 190556451, 191406055,
192255890, 193105960, 193956264, 194806803, 195657579, 196508594, 197359847, 198211340,
199063074, 199915050, 200767270, 201619735, 202472445, 203325401, 204178606, 205032059,
205885762, 206739717, 207593924, 208448384, 209303098, 210158069, 211013296, 211868780,
212724524, 213580528, 214436793, 215293321, 216150112, 217007167, 217864489, 218722077,
219579933, 220438059, 221296454, 222155121, 223014061, 223873274, 224732763, 225592527,
226452568, 227312888, 228173487, 229034367, 229895528, 230756972, 231618701, 232480714,
233343014, 234205601, 235068477, 235931643, 236795100, 237658849, 238522891, 239387228,
240251860, 241116790, 241982017, 242847543, 243713370, 244579498, 245445929, 246312664,
247179704, 248047050, 248914703, 249782666, 250650937, 251519520, 252388415, 253257624,
254127147, 254996985, 255867141, 256737615, 257608408, 258479521, 259350957, 260222715,
261094797, 261967205, 262839939, 263713002, 264586393, 265460114, 266334167, 267208552,
268083271, 268958326, 269833716, 270709444, 271585511, 272461918, 273338666, 274215756,
275093191, 275970970, 276849095, 277727567, 278606389, 279485560, 280365082, 281244956,
282125184, 283005767, 283886706, 284768003, 285649658, 286531673, 287414049, 288296787,
289179889, 290063356, 290947189, 291831390, 292715959, 293600899, 294486209, 295371892,
296257949, 297144381, 298031190, 298918376, 299805941, 300693886, 301582213, 302470922,
303360016, 304249495, 305139361, 306029615, 306920258, 307811292, 308702717, 309594536,
310486750, 311379359, 312272365, 313165770, 314059575, 314953781, 315848389, 316743401,
317638818, 318534642, 319430873, 320327513, 321224564, 322122027, 323019902, 323918192,
324816898, 325716021, 326615563, 327515524, 328415907, 329316712, 330217941, 331119595,
332021676, 332924185, 333827123, 334730492, 335634293, 336538528, 337443197, 338348303,
339253846, 340159828, 341066251, 341973115, 342880423, 343788175, 344696373, 345605018,
346514112, 347423657, 348333652, 349244101, 350155004, 351066363, 351978180, 352890454,
353803189, 354716385, 355630045, 356544168, 357458757, 358373814, 359289339, 360205334,
361121800, 362038740, 362956154, 363874044, 364792411, 365711256, 366630582, 367550390,
368470680, 369391456, 370312717, 371234466, 372156704, 373079432, 374002652, 374926366,
375850574, 376775279, 377700482, 378626184, 379552387, 380479093, 381406302, 382334016,
383262238, 384190968, 385120208, 386049959, 386980223, 387911001, 388842296, 389774108,
390706439, 391639290, 392572664, 393506561, 394440984, 395375933, 396311410, 397247417,
398183956, 399121027, 400058633, 400996775, 401935455, 402874673, 403814433, 404754734,
405695580, 406636971, 407578909, 408521396, 409464433, 410408022, 411352164, 412296861,
413242115, 414187927, 415134299, 416081232, 417028728, 417976789, 418925416, 419874612,
420824376, 421774712, 422725621, 423677104, 424629163, 425581800, 426535017, 427488814,
428443194, 429398159, 430353709, 431309847, 432266574, 433223893, 434181804, 435140309,
436099411, 437059110, 438019409, 438980309, 439941811, 440903918, 441866632, 442829953,
443793884, 444758426, 445723581
};
```

```
define MAX_INV_QUANT_TABLE 1025

/* (int)(0.5 + (1<<12) * pow((double)quant,(double)4/(double)3)) */
INT32 invQuantCompact[MAX_INV_QUANT_TABLE] = {
  0, 4096, 10321, 17722, 26008, 35020, 44658, 54848, 65536, 76680, 88246,
  100204, 112530, 125204, 138207, 151524, 165140, 179043, 193222, 207666,
  222365, 237312, 252497, 267915, 283558, 299419, 315494, 331776, 348260,
  364942, 381817, 398880, 416128, 433556, 451161, 468940, 486889, 505005,
  523285, 541726, 560325, 579080, 597988, 617046, 636253, 655607, 675104,
  694742, 714521, 734437, 754490, 774676, 794995, 815445, 836023, 856729,
  877561, 898517, 919596, 940797, 962118, 983557, 1005114, 1026788, 1048576,
  1070478, 1092493, 1114619, 1136855, 1159201, 1181655, 1204216, 1226883,
  1249656, 1272533, 1295513, 1318595, 1341779, 1365063, 1388447, 1411930,
  1435511, 1459190, 1482964, 1506835, 1530800, 1554860, 1579013, 1603258,
  1627596, 1652025, 1676545, 1701154, 1725853, 1750641, 1775517, 1800480,
  1825530, 1850666, 1875888, 1901195, 1926586, 1952062, 1977620, 2003262,
  2028986, 2054792, 2080679, 2106646, 2132694, 2158822, 2185029, 2211315,
  2237679, 2264122, 2290641, 2317238, 2343911, 2370660, 2397485, 2424385,
  2451360, 2478409, 2505533, 2532730, 2560000, 2587343, 2614758, 2642246,
  2669805, 2697436, 2725137, 2752909, 2780751, 2808663, 2836645, 2864696,
  2892815, 2921003, 2949260, 2977584, 3005975, 3034434, 3062960, 3091552,
  3120211, 3148935, 3177726, 3206581, 3235502, 3264487, 3293537, 3322651,
  3351829, 3381071, 3410376, 3439744, 3469175, 3498668, 3528224, 3557841,
  3587521, 3617262, 3647064, 3676928, 3706852, 3736836, 3766881, 3796986,
  3827151, 3857375, 3887658, 3918001, 3948403, 3978863, 4009381, 4039958,
  4070593, 4101285, 4132035, 4162842, 4193707, 4224628, 4255606, 4286640,
  4317731, 4348878, 4380080, 4411339, 4442653, 4474022, 4505446, 4536925,
  4568459, 4600047, 4631689, 4663386, 4695137, 4726941, 4758799,
  4790711, 4822675, 4854693, 4886764, 4918887, 4951063, 4983291, 5015571,
  5047904, 5080288, 5112724, 5145211, 5177750, 5210340, 5242981, 5275673,
  5308416, 5341209, 5374053, 5406947, 5439891, 5472885, 5505929, 5539022,
  5572165, 5605357, 5638599, 5671889, 5705229, 5738617, 5772054, 5805540,
  5839073, 5872655, 5906285, 5939963, 5973689, 6007463, 6041284, 6075152,
  6109068, 6143030, 6177040, 6211097, 6245200, 6279351, 6313547, 6347790,
  6382079, 6416414, 6450796, 6485223, 6519696, 6554214, 6588778, 6623388,
  6658043, 6692742, 6727487, 6762277, 6797112, 6831991, 6866915, 6901883,
  6936896, 6971953, 7007054, 7042199, 7077388, 7112621, 7147897, 7183217,
  7218581, 7253988, 7289438, 7324931, 7360467, 7396047, 7431669, 7467334,
  7503041, 7538791, 7574584, 7610418, 7646295, 7682214, 7718176, 7754179,
  7790224, 7826310, 7862439, 7898609, 7934820, 7971073, 8007367, 8043702,
  8080078, 8116495, 8152954, 8189452, 8225992, 8262572, 8299193, 8335854,
  8372556, 8409298, 8446080, 8482902, 8519764, 8556666, 8593608, 8630590,
  8667611, 8704672, 8741772, 8778912, 8816091, 8853309, 8890567, 8927863,
  8965199, 9002573, 9039986, 9077438, 9114929, 9152458, 9190026, 9227632,
  9265277, 9302960, 9340681, 9378440, 9416237, 9454072, 9491946, 9529856,
  9567805, 9605791, 9643815, 9681877, 9719976, 9758112, 9796285, 9834496,
  9872744, 9911029, 9949351, 9987710, 10026106, 10064538, 10103007, 10141513,
  10180056, 10218635, 10257251, 10295902, 10334591, 10373315, 10412076,
  10450872, 10489705, 10528574, 10567479, 10606419, 10645395, 10684407,
  10723455, 10762538, 10801657, 10840811, 10880000, 10919225, 10958485,
  10997781, 11037111, 11076477, 11115877, 11155313, 11194783, 11234288,
  11273828, 11313403, 11353012, 11392656, 11432334, 11472047, 11511794,
  11551576, 11591392, 11631242, 11671126, 11711044, 11750997, 11790983,
  11831003, 11871058, 11911145, 11951267, 11991423, 12031612,
  12071834, 12112091, 12152380, 12192703, 12233060, 12273450, 12313873,
  12354329, 12394818, 12435341, 12475896, 12516485, 12557106, 12597761,
  12638448, 12679168, 12719920, 12760706, 12801523, 12842374, 12883257,
  12924172, 12965120, 13006100, 13047113, 13088158, 13129235, 13170344,
  13211485, 13252658, 13293864, 13335101, 13376370, 13417671, 13459004,
  13500369, 13541765, 13583193, 13624652, 13666143, 13707666, 13749220,
  13790806, 13832423, 13874071, 13915750, 13957461, 13999203, 14040976,
  14082780, 14124615, 14166482, 14208379, 14250307, 14292266, 14334256,
  14376276, 14418327, 14460409, 14502522, 14544665, 14586839, 14629043,
  14671278, 14713543, 14755838, 14798164, 14840520, 14882906, 14925323,
  14967770, 15010246, 15052753, 15095290, 15137857, 15180454, 15223081,
  15265737, 15308424, 15351140, 15393886, 15436662, 15479467, 15522302,
  15565166, 15608060, 15650984, 15693937, 15736919, 15779931, 15822972,
  15866042, 15909142, 15952271, 15995429, 16038616, 16081832, 16125077,
  16168351, 16211655, 16254987, 16298348, 16341738, 16385157, 16428604,
  16472080, 16515585, 16559119, 16602681, 16646272, 16689892, 16733540,
  16777216, 16820921, 16864654, 16908416, 16952206, 16996024, 17039871,
  17083745, 17127648, 17171580, 17215539, 17259526, 17303541, 17347585,
  17391656, 17435755, 17479883, 17524038, 17568221, 17612431, 17656670,
  17700936, 17745230, 17789551, 17833900, 17878277, 17922681, 17967113,
  18011572, 18056059, 18100573, 18145115, 18189684, 18234280, 18278903,
  18323554, 18368232, 18412937, 18457670, 18502429, 18547216, 18592029,
```