Subpart 4 of FCD 14496-3 is split into several files:
r

w2203tfs	Syntax, semantics and decoder description
w2203tft	This file. T/F tool descriptions and normative Annex
w2203tfa	Informative annex (Transport streams, Encoder tools)
w2203tvq	Twin-VQ vector quantizer tables

3 T/F-Tool Descriptions

3.1 Quantization

(identical to ISO/IEC 13818-7)

3.1.1 Tool description

For quantization of the spectral coefficients in the encoder a non uniform quantizer is used. Therefore the decoder must perform the inverse non uniform quantization after the Huffman decoding of the scalefactors (see clause 9 and 11) and spectral data (see clause 9).

3.1.2 Definitions

Help elements:

x_quant[g][win][sfb][bin]	quantized spectral coefficient for group *g*, window *win*, scalefactor band *sfb*, coefficient *bin*.
x_invquant[g][win][sfb][bin]	spectral coefficient for group *g*, window *win*, scalefactor band *sfb*, coefficient *bin* after inverse quantization.

3.1.3 Decoding process

The inverse quantization is described by the following formula:

$$x\_invquant = Sign(x\_quant) \cdot \left| x\_quant \right|^{\frac{4}{3}} \quad \forall \quad k$$

The maximum allowed absolute amplitude for *x_quant* is 8191. The inverse quantization is applied as follows:

```
for( g=0; g<num_window_groups; g++ ) {
   for( sfb=0; sfb < max_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++ ) {
            x_invquant[g][win][sfb][bin] = sign(x_quant[g][win][sfb][bin]) *
                                    abs(x_quant[g][win][sfb][bin]) ^(4/3);
         }
      }
   }
}
```

3.2 Scalefactors

(identical to ISO/IEC 13818-7)

3.2.1 Tool description

The basic method to adjust the quantization noise in the frequency domain is the noise shaping using scalefactors. For this purpose the spectrum is divided in several groups of spectral coefficients called scalefactor bands which share one scalefactor (see clause 8.3.4). A scalefactor represents a gain value which is used to change the amplitude of all spectral coefficients in that scalefactor band. This mechanism is used to change the allocation of the quantization noise in the spectral domain generated by the non uniform quantizer.
For window_sequences which contain SHORT_WINDOWs grouping can be applied, i.e. a specified number of consecutive SHORT_WINDOWs may have only one set of scalefactors. Each scalefactor is then applied to a group of scalefactor bands corresponding in frequency (see clause 8.3.4).
In this tool the scalefactors are applied to the inverse quantized coefficients to reconstruct the spectral values.

3.2.2 Definitions

Bit stream elements:

global_gain	An 8-bit unsigned integer value representing the value of the first scalefactor. It is also the start value for the following differential coded scalefactors (see Table 6.12)
scale_factor_data()	Part of bit stream which contains the differential coded scalefactors (see Table 6.14)
hcod_sf[]	Huffman codeword from the Huffman code table used for coding of scalefactors, see Table 6.14 and clause 9.2

Help elements:

dpcm_sf[g][sfb]	Differential coded scalefactor of group g, scalefactor band sfb.
x_rescal[]	rescaled spectral coefficients

sf[g][sfb]	Array for scalefactors of each group
get_scale_factor_gain()	Function that returns the gain value corresponding to a scalefactor

3.2.3 Decoding process

3.2.3.1 Scalefactor bands

Scalefactors are used to shape the quantization noise in the spectral domain. For this purpose, the spectrum is divided into several scalefactor bands (see section 8.3.4). Each scalefactor band has a scalefactor, which represents a certain gain value which has to be applied to all spectral coefficients in this scalefactor band. In case of EIGHT_SHORT_SEQUENCE a scalefactor band may contain multiple scalefactor window bands of consecutive SHORT_WINDOWs (see clause 8.3.4 and 8.3.5).

3.2.3.2 Decoding of scalefactors

For all scalefactors the difference to the preceeding value is coded using the Huffman code book given in table A.1. See clause 9 for a detailed description of the Huffman decoding process. The start value is given explicitly as a 8 bit PCM in the bitstream element **global_gain**. A scalefactor is not transmitted for scalefactor bands which are coded with the Huffman codebook ZERO_HCB. If the Huffman codebook for a scalefactor band is coded with INTENSITY_HCB or INTENSITY_HCB2, the scalefactor is used for intensity stereo (see clause 9 and 12.2). In that case a normal scalefactor does not exist (but is initialized to zero to have an valid in the array).

The following pseudo code describes how to decode the scalefactors *sf[g][sfb]*:

```
last_sf = global_gain;
for( g=0; g < num_window_groups; g++ ) {
   for( sfb=0; sfb<max_sfb; sfb++ ) {
      if( sfb_cb[g][sfb] != ZERO_HCB && sfb_cb[g][sfb] != INTENSITY_HCB
          && sfb_cb[g][sfb] != INTENSITY_HCB2 ) {
         dpcm_sf = decode_huffman() - index_offset; /* see clause 4 */
         sf[g][sfb] = dpcm_sf + last_sf;
         last_sf = sf[g][sfb];
      }
      else {
         sf[g][sfb] = 0;
      }
   }
}
```

3.2.3.3 Applying scalefactors

The spectral coefficients of all scalefactor bands which correspond to a scalefactor have to be rescaled according to their scalefactor. In case of a window sequence that contains groups of short windows all coefficients in grouped scalefactor window bands have to be scaled using the same scalefactor.
In case of window_sequences with only one window, the scalefactor bands and their corresponding coefficients are in spectral ascending order. In case of EIGHT_SHORT_SEQUENCE and grouping the spectral coefficients of grouped short windows are interleaved by scalefactor window bands. See clause 8.3.5 for more detailed information.
The rescaling operation is done according to the following pseudo code:

```
for( g=0; g<num_window_groups; g++ ) {
   for( sfb=0; sfb < max_sfb; sfb++ ) {
      width = (swb_offset [sfb+1] - swb_offset [sfb] );
      for( win = 0; win < window_group_len[g]; win++ ) {;
         gain = get_scale_factor_gain( sf[g][sfb] );
         for( k=0; k<width; k++ ) {
            x_rescal[g][window][sfb][k] =
               x_invquant[g][window][sfb][k] * gain;
         }
      }
   }
}
```

The function *get_scale_factor_gain(sf[g][sfb])* returns the gain factor that corresponds to a scalefactor. The return value follows the equation:

$$gain = 2^{0.25\cdot(sf[g][sfb]-SF\_OFFSET)}$$

The constant SF_OFFSET must be set to 100.

The following pseudo code describes this operation:

```
get_scale_factor_gain( sf[g][sfb] )  {
   SF_OFFSET = 100;
   gain = 2^(0.25 * ( sf[g][sfb] - SF_OFFSET));
   return( gain );
}
```

3.3 Noiseless Coding

(similar to ISO/IEC 13818-7)

3.3.1 Tool description

Noiseless coding is used to further reduce the redundancy of the scalefactors and the quantized spectrum of each audio channel.

The global_gain is coded as an 8 bit unsigned integer. The first scalefactor associated with the quantized spectrum is differentially coded relative to the global_gain value and then Huffman coded using the scalefactor codebook. The remaining scalefactors are differentially coded relative to the previous scalefactor and then Huffman coded using the scalefactor codebook.

Noiseless coding of the quantized spectrum relies on two divisions of the spectral coefficients. The first is a division into scalefactor bands that contain a multiple of 4 quantized spectral coefficients. See clause 8.3.4 and 8.3.5.

The second division, which is dependent on the quantized spectral data, is a division by scalefactor bands to form sections. The significance of a section is that the quantized spectrum within the section is represented using a single Huffman codebook chosen from a set of 11 possible codebooks. The length of a section and its associated Huffman codebook must be transmitted as side information in addition to the section's Huffman coded spectrum. Note that the length of a section is given in scalefactor bands rather than scalefactor window bands (see clause 8.3.4). In order to maximize the match of the statistics of the quantized spectrum to that of the Huffman codebooks the number of sections is permitted to be as large as the number of scalefactor bands. The maximum size of a section is max_sfb scalefactor bands.

As indicated in Table 00.22, spectrum Huffman codebooks can represent signed or unsigned n-tuples of coefficients. For unsigned codebooks, sign bits for every non-zero coefficient in the n-tuple immediately follow the associated codeword.

The noiseless coding has two ways to represent large quantized spectra. One way is to send the escape flag from the escape (ESC) Huffman codebook, which signals that the bits immediately following that codeword plus optional sign bits are an escape sequence that encodes values larger than those represented by the ESC Huffman codebook. A second way is the pulse escape method, in which relatively large-amplitude coefficients can be replaced by coefficients with smaller amplitudes in order to enable the use of Huffman code tables with higher coding efficiency. This replacement is corrected by sending the position of the spectral coefficient and the differences in amplitude as side information. The frequency information is represented by the combination of the scalefactor band number to indicate a base frequency and an offset into that scalefactor band.

3.3.2 Definitions

sect_cb[g][i] spectrum Huffman codebook used for section i in group g (see 6.3, Table 6.13).

sect_len_incr used to compute the length of a section, measures number of scalefactor bands from start of section. The length of **sect_len_incr** is 3 bits if window_sequence is EIGHT_SHORT_SEQUENCE and 5 bits otherwise (see 6.3, Table 6.13).

global_gain global gain of the quantized spectrum, sent as unsigned integer value (see 6.3, Table 6.12).

hcod_sf[] Huffman codeword from the Huffman code table used for coding of scalefactors (see 6.3, Table 6.14).

hcod[sect_cb[g][i]][w][x][y][z] Huffman codeword from codebook **sect_cb[g][i]** that encodes the next 4-tuple (w, x, y, z) of spectral coefficients, where w, x, y, z are quantized spectral coefficients. Within an n-tuple, w, x, y, z are ordered as described in 8.3.5. so that x_quant[group][win][sfb][bin] = w, x_quant[group][win][sfb][bin+1] = x,

	x_quant[group][win][sfb][bin+2] = y and x_quant[group][win][sfb][bin+3] = z. N-tuples progress from low to high frequency within the current section (see 6.3, Table 6.16).
hcod[sect_cb[g][i]][y][z]	Huffman codeword from codebook **sect_cb[g][i]** that encodes the next 2-tuple (y, z) of spectral coefficients, where y, z are quantized spectral coefficients. Within an n-tuple, y, z are ordered as described in 8.3.5 so that x_quant[group][win][sfb][bin] = y and x_quant[group][win][sfb][bin+1] = z. N-tuples progress from low to high frequency within the current section (see 6.3, Table 6.16).
quad_sign_bits	sign bits for non-zero coefficients in the spectral 4-tuple. A '1' indicates a negative coefficient, a '0' a positive one. Bits associated with lower frequency coefficients are sent first (see 6.3, Table 6.16).
pair_sign_bits	sign bits for non-zero coefficients in the spectral 2-tuple. A '1' indicates a negative coefficient, a '0' a positive one. Bits associated with lower frequency coefficients are sent first (see 6.3, Table 6.16).
hcod_esc_y	escape sequence for quantized spectral coefficient y of 2-tuple (y,z) associated with the preceeding Huffman codeword (see 6.3, Table 6.16).
hcod_esc_z	escape sequence for quantized spectral coefficient z of 2-tuple (y,z) associated with the preceeding Huffman codeword (see 6.3, Table 6.16).
pulse_data_present	1 bit indicating whether the pulse escape is used (1) or not (0) (see 6.3, Table 6.17). Note that pulse_data_present must be 0 for an EIGHT_SHORT_SEQUENCE.
number_pulse	2 bits indicating how many pulse escapes are used. The number of pulse escapes is from 1 to 4 (see 6.3, Table 6.17).
pulse_start_sfb	6 bits indicating the index of the lowest scalefactor band where the pulse escape is achieved (see 6.3, Table 6.17).
pulse_offset[i]	5 bits indicating the offset (see 6.3, Table 6.17).
pulse_amp[i]	4 bits indicating the unsigned magnitude of the pulse (see 6.3, Table 6.17).

sect_start[g][i]	offset to first scalefactor band in section i of group g (see 6.3, Table 6.13).
sect_end[g][i]	offset to one higher than last scalefactor band in section i of group g (see 6.3, Table 6.13).
num_sec[g]	number of sections in group g (see 6.3, Table 6.13).
escape_flag	the value of 16 in the ESC Huffman codebook
escape_prefix	the bit sequence of N 1's
escape_separator	one 0 bit
escape_word	an N+4 bit unsigned integer word, msb first
escape_sequence	the sequence of *escape_prefix*, *escape_separator* and *escape_word*
escape_code	2^(N+4) + escape_word
x_quant[g][win][sfb][bin]	Huffman decoded value for group *g*, window *win*, scalefactor band *sfb*, coefficient *bin*
spec[w][k]	de-interleaved spectrum. *w* ranges from 0 to *num_windows*-1 and *k* ranges from 0 to *swb_offset[num_swb]*-1.

The noiseless coding tool requires these constants (see clause 6.3, spectral_data()).

ZERO_HCB	0
FIRST_PAIR_HCB	5
ESC_HCB	11
QUAD_LEN	4
PAIR_LEN	2
NOISE_HCB	13
INTENSITY_HCB2	14
INTENSITY_HCB	15
ESC_FLAG	16

3.3.3 Decoding Process

Four-tuples or 2-tuples of quantized spectral coefficients are Huffman coded and transmitted starting from the lowest-frequency coefficient and progressing to the highest-frequency coefficient. For the case of multiple windows per block (EIGHT_SHORT_SEQUENCE), the grouped and interleaved set of spectral coefficients is treated as a single set of coefficients that progress from low to high. The set of coefficients may need to be de-interleaved after they are decoded (see clause 8.3.5). Coefficients are stored in the array x_quant[g][win][sfb][bin], and the order of transmission of the Huffman codewords is such that when they are decoded in the order received and stored in the array, *bin* is the most rapidly incrementing index and *g* is the most slowly incrementing index. Within a codeword, for those associated with spectral four-tuples, the order of decoding is w, x, y, z; for codewords associated with spectral two-tuples, the order of decoding is y, z. The set of coefficients is divided into sections and the sectioning information is transmitted starting from the lowest frequency section and progressing to the highest frequency section. The spectral information for sections that are coded with the “zero” codebook is not sent as this spectral information is zero. Similarly, spectral information for sections coded with the “intensity” codebooks is not sent. The spectral information for all scalefactor bands at and above **max_sfb**, for which there is no section data, is zero.

There is a single differential scalefactor codebook which represents a range of values as shown in Table 00.11. The differential scalefactor codebook is shown in Table A.1. There are eleven Huffman codebooks for the spectral data, as shown in Table 00.22. The codebooks are shown in Tables A.2 through A.12. There are four other “codebooks” above and beyond the actual Huffman codebooks, specifically the “zero” codebook, indicating that neither scalefactors nor quantized data will be transmitted, and the “intensity” codebooks indicating that this individual channel is part of a channel pair, and that the data that would normally be scalefactors is instead steering data for intensity stereo. Similarly, the “noise substitution” codebook indicates that the spectral coefficients are derived from random numbers rather than quantized spectral values, and that the data that would normally be scalefactors is instead noise energy data. In these cases, no quantized spectral data are transmitted. Codebook index 12 is reserved.

The spectrum Huffman codebooks encode 2- or 4-tuples of signed or unsigned quantized spectral coefficients, as shown in Table 00.22. This table also indicates the largest absolute value (LAV) able to be encoded by each codebook and defines a boolean helper variable array, unsigned_cb[], that is 1 if the codebook is unsigned and 0 if signed.

The result of Huffman decoding each differential scalefactor codeword is the codeword index, listed in the first column of Table A.1. This is translated to the desired differential scalefactor by adding index_offset to the index. Index_offset has a value of −60, as shown in Table 9.1. Likewise, the result of Huffman decoding each spectrum n-tuple is the codeword index, listed in the first column of Tables A.2 through A.12. This index is translated to the n-tuple spectral values as specified in the following pseudo C-code:

`unsigned` = Boolean value unsigned_cb[i], listed in second column of Table 9.2.
`dim` = Dimension of codebook, listed in the third column of Table 9.2.
`lav` = LAV, listed in the fourth column of Table 9.2.
`idx` = codeword index

```
if (unsigned) {
   mod = lav + 1;
   off = 0;
}
else {
   mod = 2*lav + 1;
   off = lav;
}

if (dim == 4) {
   w = INT(idx/(mod*mod*mod)) - off;
   idx -= (w+off)*(mod*mod*mod)
   x = INT(idx/(mod*mod)) - off;
   idx -= (x+off)*(mod*mod)
   y = INT(idx/mod) - off;
   idx -= (y+off)*mod
   z = idx - off;
}
else {
```

```
   y = INT(idx/mod) - off;
   idx -= (y+off)*mod
   z = idx - off;
}
```

If the Huffman codebook represents signed values, the decoding of the quantized spectral n-tuple is complete after Huffman decoding and translation of codeword index to quantized spectral coefficients. If the codebook represents unsigned values then the sign bits associated with non-zero coefficients immediately follow the Huffman codeword, with a '1' indicating a negative coefficient and a '0' indicating a positive one. For example, if a Huffman codeword from codebook 7

hcod[7][y][z]

has been parsed, then immediately following this in the bitstream is

pair_sign_bits

which is a variable length field of 0 to 2 bits. It can be parsed directly from the bitstream as

```
   if (y != 0)
      if (one_sign_bit == 1)
         y = -y
   if (z != 0)
      if (one_sign_bit == 1)
         z = -z
```

where one_sign_bit is the next bit in the bitstream and **pair_sign_bits** is the concatenation of the one_sign_bit fields.

The ESC codebook is a special case. It represents values from 0 to 16 inclusive, but values from 0 to 15 encode actual data values, and the value16 is an *escape_flag* that signals the presence of **hcod_esc_y** or **hcod_esc_z**, either of which will be denoted as an *escape_sequence*. This *escape_sequence* permits quantized spectral elements of LAV>15 to be encoded. It consists of an *escape_prefix* of N 1's, followed by an *escape_separator* of one zero, followed by an *escape_word* of N+4 bits representing an unsigned integer value. The *escape_sequence* has a decoded value of 2^(N+4)+*escape_word*. The desired quantized spectral coefficient is then the sign indicated by the pair_sign_bits applied to the value of the *escape_sequence*. In other words, an *escape_sequence* of 00000 would decode as 16, an *escape_sequence* of 01111 as 31, an *escape_sequence* of 1000000 as 32, one of 1011111 as 63, and so on. Note that restrictions in clause 10.3 dictate that the length of the escape_sequence is always less than 24 bits. For escape Huffman codewords the ordering of bitstream elements is Huffman codeword followed by 0 to 2 sign bits followed by 0 to 2 escape sequences.

When **pulse_data_present** is 1 (the pulse escape is used), one or several quantized coefficients have been replaced by coefficients with smaller amplitudes in the encoder. The number of coefficients replaced is indicated by **number_pulse**. In reconstructing the quantized spectral coefficients *x_quant* this replacement is compensated by adding **pulse_amp** to or subtracting **pulse_amp** from the previously decoded coefficients whose frequency indices are indicated by **pulse_start_sfb** and **pulse_offset**. Note that the pulse escape method is illegal for a block whose **window_sequence** is EIGHT_SHORT_SEQUENCE. The decoding process is specified in the following pseudo-C code:

```
if (pulse_data_present) {
   g = 0;
   win = 0;
   k = swb_offset[pulse_start_sfb];
   for (j = 0; j<number_pulse+1; j++) {
      k += pulse_offset[j];

      /* translate_pulse_parameters(); */
      for( sfb = pulse_start_sfb; sfb<num_swb;sfb++) {
         if( k < swb_offset[sfb+1]) {
            bin = k - swb_offset[sfb] ;
            break;
         }
      }

      /* restore coefficients */
      if (x_quant[g][win][sfb][bin] > 0 )
         x_quant[g][win][sfb][bin] += pulse_amp[j];
      else
         x_quant[g][win][sfb][bin] -= pulse_amp[j];
   }
}
```

Several decoder tools (TNS, filterbank) access the spectral coefficients in a non-interleaved fashion, i.e. all spectral coefficients are ordered according to window number and frequency within a window. This is indicated by using the notation spec[w][k] rather than x_quant[g][w][sfb][bin].

The following pseudo C-code indicates the correspondence between the four-dimensional, or interleaved, structure of array x_quant[ ][ ][ ][ ] and the two-dimensional, or de-interleaved, structure of array spec[ ][ ]. In the latter array the first index increments over the individual windows in the window sequence, and the second index increments over the spectral coefficients that correspond to each window, where the coefficients progress linearly from low to high frequency.

```
quant_to_spec() {
   k=0;
   for( g=0; g<num_window_groups; g++ ) {
      j=0;
      for( sfb=0; sfb < num_swb; sfb ++ ) {
         width = swb_offset[sfb+1] - swb_offset[sfb];
         for( win=0; win<window_group_length[g]; win++ ) {
            for( bin=0; bin<width; bin++ ) {
               spec[win+k][bin+j] = x_quant[g][win][sfb][bin] ;
            }
         }
         j+=width;
      }
      k+=window_group_length[g];
   }
}
```

3.3.4 Tables

Table 0.1 – Scalefactor Huffman codebook parameters

Codebook Number	Dimension of Codebook	index_offset	Range of values	Codebook listed in Table
0	1	-60	-60 to +60	A.1

Table 0.2 – Spectrum Huffman codebooks parameters

Codebook Number, i	unsigned_cb[i]	Dimension of Codebook	LAV for codebook	Codebook listed in Table
0	-	-	0	-
1	0	4	1	A.2
2	0	4	1	A.3
3	1	4	2	A.4
4	1	4	2	A.5
5	0	2	4	A.6
6	0	2	4	A.7
7	1	2	7	A.8
8	1	2	7	A.9
9	1	2	12	A.10
10	1	2	12	A.11
11	1	2	(16) ESC	A.12
12	-	-	(reserved)	-
13	-	-	percept. noise subst.	-
14	-	-	intensity out-of-phase	-
15	-	-	intensity in-phase	-

3.4 Interleaved Vector Quantization

3.4.1 Tool description

This process generates flattened MDCT spectrum using vector quantization. This quantization tool provides high coding gain, even at lower bitrates. Bitstream for this quantizer has a simple fixed-length structure, thus it is robust against transmission channel errors.
The decoding process consists of vector quantization part and reconstruction part. In the vector quantization part, subvectors are specified by codevector index. Then, subvectors are interleaved and combined into one output vector (see fig.3.4.1).

3.4.2 Definitions

Inputs:

f b_shift[][]	Syntax element indicating base frequency of active frequency band of the adaptive bandwidth control.
index0[]:	bitstream element indicating the codevector number of codebook 0
index1[]:	bitstream element indicating the codevector number of codebook 1
window_sequence:	bitstream element indicating window sequence type
side_info_bits	number of bits for side information
bitrate:	system parameter indicating bitrate
used_bits:	number of bits used by variable bit-rate tool, such as long term prediction tool
lyr:	indicates enhancement layer number. Number 0 is assigned for the base layer.

Outputs:

x_flat[]:	reconstructed coefficients

Parameters:

FRAME_SIZE	frame length
MAXBIT	maximum bits for shape codebook index representation
N_CH	number of channels
N_DIV	number of subvectors
N_SF	number of subframes in a frame
sp_cv0[][]	shape codebook of conjugate channel 0
sp_cv1[][]	shape codebook of conjugate channel 1
SP_CB_SIZE	shape codebook size
shape_index0	points the selected codevector of shape codebook 0
shape_index1	points the selected codevector of shape codebook 1
pol0	negates the selected codevector of shape codebook 0
pol1	negates the selected codevector of shape codebook 1

3.4.3 Parameter settings

The assignment of the shape codebook vectors, sp_cv0[][] and sp_cv1[][] is dependent on the window block types as listed in Tables from C.1 to C.30.

Parameters are set initially as listed below:

```
MAXBIT_SHAPE = 6
MAXBIT = MAXBIT_SHAPE + 1
SP_CB_SIZE=(1<<MAXBIT_SHAPE)
FRAME_SIZE = N_FR_L * N_CH
N_SF=  N_FR_L / N_FR
```

3.4.4 Decoding process

3.4.4.1 Initializations

Number of available bits, bits_available_vq is calculated as follows:

```
bits_available_vq =
   (int)(FRAME_SIZE*bitrate/sampling_frequency) - side_info_bits - used_bits;
```

N_DIV represents the number of subvectors. The length of each subvector is calculated by

```
   N_DIV = ((int)((bits_available_vq + MAXBIT*2-1)/(MAXBIT*2)))

   for(idiv=0; idiv<ntt_N_DIV; idiv++){
      length[idiv] = (FRAME_SIZE + N_DIV - 1 - idiv) / N_DIV
   }
```

If codevector length, length[], exceeds the number of codevector elements which are described in tables from C.1 to C.55, undefined elements of sp_cv0[] and sp_cv1[] (i.e. elements beyond defined area) are set to zero.

3.4.4.2 Index unpacking

The quantization index consists of the polarity and shape code information. So in the first stage of the inverse quantization, input indices are unpacked, and polarities and shapes are extracted.

The extracting of polarities is described as follows:

```
for (idiv=0; idiv<N_DIV; idiv++){
   pol0[idiv] = 2 * (index0 [idiv] / SP_CB_SIZE) - 1
   pol1[idiv] = 2 * (index1 [idiv] / SP_CB_SIZE) - 1
}
```

where
pol0[]: polarity of conjugate channel 0
pol1[]: polarity of conjugate channel 1

The shape code extraction is described as follows:

```
for (idiv=0; idiv<N_DIV; idiv++){
   index_shape0[idiv] = index0 [idiv] % SP_CB_SIZE
   index_shape1[idiv] = index1 [idiv] % SP_CB_SIZE
}
```

3.4.4.3 Reconstruction

Output coefficients are reconstructed as follows:

```
for (idiv=0; idiv<N_DIV; idiv++){
   for (icv=0; icv<length[idiv]; icv++){
      if ((icv<length[0]-1) &&
          ((N_DIV%(N_SF*N_CH)==0 && (N_SF*N_CH)>1) || ((N_SF*N_CH)&0x1)==0)))
         itmp = ((idiv+icv)%N_DIV)+icv*N_DIV;
      else
         itmp = idiv + icv * N_DIV;
      ismp = itmp / (N_SF*N_CH) + ((itmp % (N_SF*N_CH)) * (FRAME_SIZE /
(N_SF*N_CH)));
      x_flat_tmp[ismp] =
         (pol0[idiv]*sp_cv0[index_shape0[idiv]][icv]
          + pol1[idiv]*sp_cv1[index_shape1[idiv]][icv]) / 2;
   }
```

}

where
icv: indicates sample number in the shape code vector
idiv: indicates interleaved-division subvector
ismp: indicates sample number in the subframe
itmp: an integer

3.4.4.4 Adaptive active band selection

This procedure, which is activated in scaleabl configuration modes, limits the active band (see Fig. 3.4.2). For the compression modes, this procedure is not activated and the output x_flat[] is simply copied from x_flat_tmp[].

If lyr=0 or lyr=1, the active band is fixed as listed below:

lyr	ac_btm	ac_top
0 (base)	0.0	1/3
1	0.0	2/3

where ac_btm and ac_top is the bottom and top frequency of the active band, respectively. Values are ranged from 0 to 1 (i.e. 1.0 means the highest frequency).

If lyr >1, active band is selected according to the syntax element fb_shift as follows:

fb_shift	ac_btm	ac_top
0	0.0	2/3
1	1/12	3/4
2	1/6	5/6
3	1/3	1.0

The lower and upper boundaries in MDCT domain are calculated as follows:

```
for (i_ch=0; i_ch<N_CH; i_ch++){
   LOWER_BOUNDARY[lyr][i_ch] = ac_btm[lyr][i_ch] * N_FR;
   UPPER_BOUNDARY[lyr][i_ch] = ac_top[lyr][i_ch] * N_FR;
}
```

Then, the output x_flat[] is copied from x_flat_tmp[] as follows:

```
for (i_ch=0; i_ch<N_CH; i_ch++){
   for (isf=0; isf<N_SF; isf++){
      for (ismp=0; ismp<LOWER_BOUNDARY[lyr][i_ch]; ismp++){
         x_flat[ismp+(isf+i_ch*N_SF)*N_FR] = 0.;
      }
      for (ismp=LOWER_BOUNDARY[lyr][i_ch]; ismp<UPPER_BOUNDARY[lyr][i_ch]; ismp++){
         ismp2 = ismp - LOWER_BOUNDARY[lyr][i_ch];
         x_flat[ismp+(isf+i_ch*N_SF)*N_FR] = x_flat_tmp[ismp2+(isf+i_ch*N_SF)*N_FR];
      }
      for (ismp=UPPER_BOUNDARY[lyr][i_ch]; ismp<N_FR; ismp++){
         x_flat[ismp+(isf+i_ch*N_SF)*N_FR] = 0.;
      }
   }
}
```

3.4.5 Diagrams

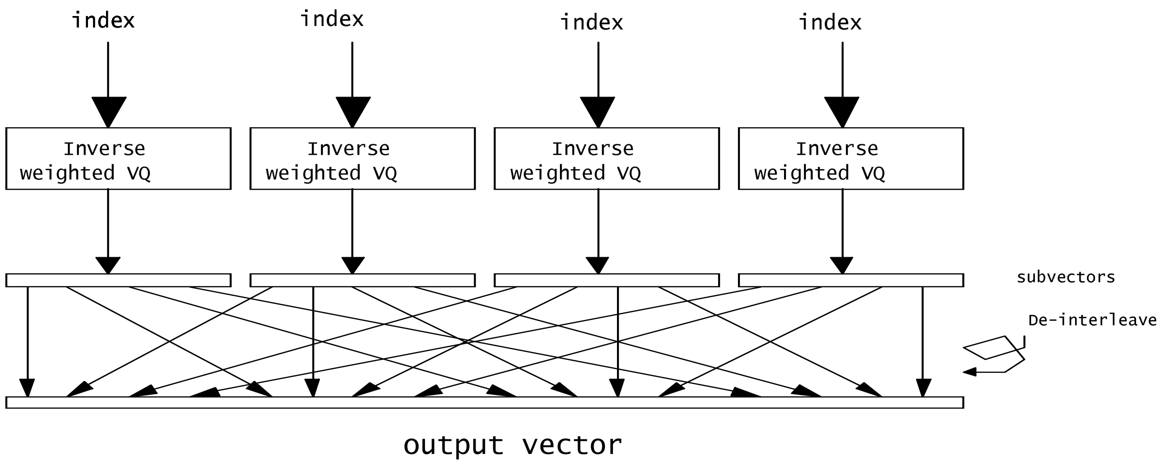


figure 3.4.1 : Decoding process of interleaved voector quantization tool.

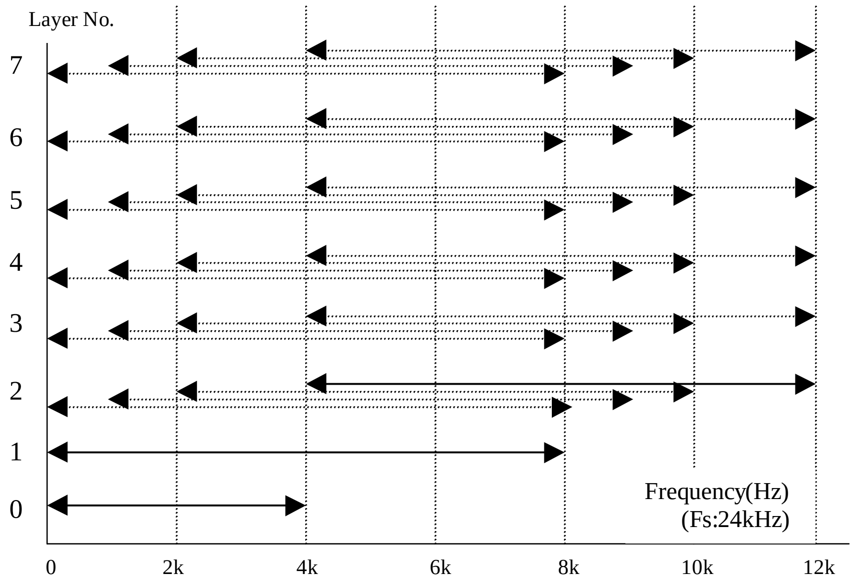


Fig. 3.4.2 Adaptive active band selection.(SAMPLING_FREQUENCY=24kHz)

3.5 Prediction

(non low complexity part identical to ISO/IEC 13818-7)

3.5.1 Tool description

Prediction is used for an improved redundancy reduction and is especially effective in case of more or less stationary parts of a signal which belong to the most demanding parts in terms of required bitrate. Prediction can be applied to every channel using an intra channel (or mono) predictor which exploits the auto-correlation between the spectral components of consecutive frames. Because a window_sequence of type EIGHT_SHORT_SEQUENCE indicates signal changes, i.e. non-stationary signal characteristics, prediction is only used if window_sequence is of type ONLY_LONG_SEQUENCE, LONG_START_SEQUENCE or LONG_STOP_SEQUENCE. The use of the prediction tool is profile dependent. See clause 7 for detailed information.

For each channel prediction is applied to the spectral components resulting from the spectral decomposition of the filterbank. For each spectral component up to limit specified by PRED_SFB_MAX, there is one corresponding predictor resulting in a bank of predictors, where each predictor exploits the auto-correlation between the spectral component values of consecutive frames.

The overall coding structure using a filterbank with high spectral resolution implies the use of backward adaptive predictors to achieve high coding efficiency. In this case, the predictor coefficients are calculated from preceding quantized spectral components in the encoder as well as in the decoder and no additional side information is needed for the transmission of predictor coefficients - as would be required for forward adaptive predictors. A second order backward-adaptive lattice structure predictor is used for each spectral component, so that each predictor is working on the spectral component values of the two preceding frames. The predictor parameters are adapted to the current signal statistics on a frame by frame base, using an LMS based adaptation algorithm. If prediction is activated, the quantizer is fed with a prediction error instead of the original spectral component, resulting in a coding gain.

In order to keep storage requirements to a minimum, predictor state variables are quantized prior to storage.

3.5.2 Definitions

predictor_data_present 1 bit indicating whether prediction is used in current frame (1) or not (0) (always present for ONLY_LONG_SEQUENCE, LONG_START_SEQUENCE and LONG_STOP_SEQUENCE, see 6.3, Table 6.11).

predictor_reset 1 bit indicating whether predictor reset is applied in current frame (1) or not (0) (only present if **predictor_data_present** flag is set, see 6.3, Table 6.11).

predictor_reset_group_number 5 bit number specifying the reset group to be reset in current frame if predictor reset is enabled (only present if **predictor_reset** flag is set, see 6.3, Table 6.11).

prediction_used 1 bit for each scalefactor band (sfb) where prediction can be used indicating whether prediction is switched on (1) / off (0) in that sfb. If **max_sfb** is less than PRED_SFB_MAX then for i greater than or equal to max_sfb, prediction_used[i] is not transmitted and therfore is set to off (0) (only present if **predictor_data_present** flag is set, see 6.3, Table 6.11).

The following table specifies the upper limit of scalefactor bands up to which prediction can be used:

Sampling Frequency (Hz)	Pred_SFB_MAX	Number of Predictors	Maximum Frequency using Prediction (Hz)
96000	33	512	24000.00
88200	33	512	22050.00
64000	38	664	20750.00
48000	40	672	15750.00
44100	40	672	14470.31
32000	40	672	10500.00
24000	41	652	7640.63
22050	41	652	7019.82
16000	37	664	5187.50
12000	37	664	3890.63
11025	37	664	3574.51
8000	34	664	2593.75

This means that at 48 kHz sampling rate prediction can be used in scalefactor bands 0 through 39. According to table 8.5 these 40 scalefactor bands include the MDCT lines 0 through 671, hence resulting in max. 672 predictors.

3.5.3 Decoding process

For each spectral component up to the limit specified by PRED_SFB_MAX of each channel there is one predictor. Prediction is controlled on a single_channel_element or channel_pair_element basis by the transmitted side information in a two step approach, first for the whole frame at all and then conditionally for each scalefactor band individually, see clause 0. The predictor coefficients for each predictor are calculated from preceding reconstructed values of the corresponding spectral component. The details of the required predictor processing are described in clause 0. At the start of the decoding process, all predictors are initialized. The initialization and a predictor reset mechanism are described in clause 0.

3.5.3.1 Predictor side information

The following description is valid for either one single_channel_element or one channel_pair_element and has to be applied to each such element. For each frame the predictor side information has to be extracted from the bitstream to control the further predictor processing in the decoder. In case of a single_channel_element the control information is valid for the predictor bank of the channel associated with that element. In case of a channel_pair_element there are the following two possibilities: If **common_window** = 1 then there is only one set of the control information which is valid for the two predictor banks of the two channels associated with that element. If **common_window** = 0 then there are two sets of control information, one for each of the two predictor banks of the two channels associated with that element.

If window_sequence is of type ONLY_LONG_SEQUENCE, LONG_START_SEQUENCE or LONG_STOP_SEQUENCE, the **predictor_data_present** bit is read. If this bit is not set (0) then prediction is switched off at all for the current frame and there is no further predictor side information present. In this case the **prediction_used** bit for each scalefactor band stored in the decoder has to be set to zero. If the **predictor_data_present** bit is set (1) then prediction is used for the current frame and the **predictor_reset** bit is read which determines whether predictor reset is applied in the current frame (1) or not (0). If **predictor_reset** is set then the next 5 bits are read giving a number specifying the group of predictors to be reset in the current frame, see also clause 0 for the details. If the **predictor_reset** is not set then there is no 5 bit number in the bitstream. Next, the **prediction_used** bits are read from the bitstream, which control the use of prediction in each scalefactor band individually, i.e. if the bit is set for a particular scalefactor band, then prediction is enabled for all spectral components of this scalefactor band and the quantized prediction error of each spectral component is transmitted instead of the quantized value of the spectral component. Otherwise, prediction is disabled for this scalefactor band and the quantized values of the spectral components are transmitted.

3.5.3.2 AAC predictor processing

3.5.3.2.1 General

The following description is valid for one single predictor and has to be applied to each predictor. A second order backward adaptive lattice structure predictor is used. Figure 13.1 shows the corresponding predictor flow graph on the decoder side. In principle, an estimate $x_{est}(n)$ of the current value of the spectral component $x(n)$ is calculated from preceding reconstructed values $x_{rec}(n-1)$ and $x_{rec}(n-2)$, stored in the register elements of the predictor structure, using the predictor coefficients $k_1(n)$ and $k_2(n)$. This estimate is then added to the quantized prediction error $e_q(n)$ reconstructed from the transmitted data resulting in the reconstructed value $x_{rec}(n)$ of the current spectral component $x(n)$. Figure 13.2 shows the block diagram of this reconstruction process for one single predictor.

Due to the realization in a lattice structure, the predictor consists of two so-called basic elements which are cascaded. In each element, the part $x_{est,m}(n)$, $m=1, 2$ of the estimate is calculated according to

$$x_{est,m}(n) = b \cdot k_m(n) \cdot a \cdot r_{q,m-1}(n-1) ,$$

where

$$r_{q,m}(n) = r_{q,m-1}(n-1) - b \cdot k_m(n) \cdot e_{q,m-1}(n)$$

and $$e_{q,m}(n) = e_{q,m-1}(n) - x_{est,m}(n) .$$

Hence, the overall estimate results to: $$x_{est}(n) = x_{est,1}(n) + x_{est,2}(n)$$

The constants

$$a \text{ and } b , \qquad 0 < a, b \le 1$$

are attenuation factors which are included in each signal path contributing to the recursivity of the structure for the purpose of stabilization. By this means, possible oscillations due to transmission errors or drift between predictor coefficients on the encoder and decoder side due to numerical inaccuracy can be faded out or even prevented.

In the case of stationary signals and with $a = b = 1$, the predictor coefficient of element m is calculated by

$$k_m = \frac{E\left[e_{q,m-1}(n) \cdot r_{q,m-1}(n-1)\right]}{\frac{1}{2} \cdot \left(E\left[e^2_{q,m-1}(n)\right] + E\left[r^2_{q,m-1}(n-1)\right]\right)}, \qquad m = 1, 2 \text{ and } e_{q,0}(n) = r_{q,0}(n) = x_{rec}(n)$$

In order to adapt the coefficients to the current signal properties, the expected values in the above equation are substituted by time average estimates measured over a limited past signal period. A compromise has to be chosen between a good convergence against the optimum predictor setting for signal periods with quasi stationary characteristic and the ability of fast adaptation in case of signal transitions. In this context algorithms with iterative improvement of the estimates, i.e. from sample to sample, are of special interest. Here, a "least mean square" (LMS) approach is used and the predictor coefficients are calculated as follows

$$k_m(n+1) = \frac{COR_m(n)}{VAR_m(n)}$$

with

$$COR_m(n) = \alpha \cdot COR_m(n-1) + r_{q,m-1}(n-1) \cdot e_{q,m-1}(n)$$

$$VAR_m(n) = \alpha \cdot VAR_m(n-1) + 0.5 \cdot \left(r^2_{q,m-1}(n-1) + e^2_{q,m-1}(n)\right)$$

where α is an adaptation time constant which determines the influence of the current sample on the estimate of the expected values. The value of α is chosen to

$$\alpha = 0.90625 \ .$$

The optimum values of the attenuation factors a and b have to be determined as a compromise between high prediction gain and small fade out time. The chosen values are

$$a = b = 0.953125 \ .$$

Independent of whether prediction is disabled - either at all or only for a particular scalefactor band - or not, all the predictors are run all the time in order to always adapt the coefficients to the current signal statistics.

If window_sequence is of type ONLY_LONG_SEQUENCE, LONG_START_SEQUENCE and LONG_STOP_SEQUENCE only the calculation of the reconstructed value of the quantized spectral components differs depending on the value of the **prediction_used** bit:

- If the bit is set (1), then the quantized prediction error reconstructed from the transmitted data is added to the estimate $x_{est}(n)$ calculated by the predictor resulting in the reconstructed value of the quantized spectral component, i.e. $x_{rec}(n) = x_{est}(n) + e_q(n)$
- If the bit is not set (0), then the quantized value of the spectral component is reconstructed directly from the transmitted data.

In case of short blocks, i.e. window_sequence is of type EIGHT_SHORT_SEQUENCE, prediction is always disabled and a reset is carried out for all predictors in all scalefactor bands, which is equivalent to a reinitialization, see clause 0.

For a single_channel_element, the predictor processing for one frame is done according to the following pseudo code:

(It is assumed that the reconstructed value y_rec(c) - which is either the reconstructed quantized prediction error or the reconstructed quantized spectral coefficient - is available from previous processing.)

```
if (ONLY_LONG_SEQUENCE || LONG_START_SEQUENCE || LONG_STOP_SEQUENCE) {
   for ( sfb=0; sfb<PRED_SFB_MAX; sfb++) {
      fc = swb_offset_long_window[fs_index][sfb];
      lc = swb_offset_long_window[fs_index][sfb+1];
      for (c= fc; c<lc; c++) {
```

```
        x_est[c] = predict();
        if (predictor_data_present && prediction_used[sfb] )
           x_rec[c] = x_est[c] + y_rec[c];
        else
           x_rec[c] = y_rec[c];
     }
   }
}
else {
   reset_all_predictors();
}
```

In case of channel_pair_elements with **common_window** = 1, the only difference is that the computation of x_est and x_rec in the inner for loop is done for both channels associated with the channel_pair_element. In case of channel_pair_elements with **common_window** = 0, each channel has prediction applied using that channel's prediction side information.

3.5.3.2.2 Quantization in Predictor Calculations

For a given predictor six state variables need to be saved: r_0, r_1, COR_1, COR_2, VAR_1 and VAR_2. These variables will be saved as truncated IEEE floating-point numbers (i.e. the 16 msb of a float storage word).

The predicted value x_{est} will be rounded to a 16-bit floating point representation (i.e. round to a 7-bit mantissa) prior to being used in any calculation. The exact rounding algorithm to be used is shown in pseudo-C function flt_round_inf(). Note that for complexity considerations, *round to nearest, infinity* is used instead of *round to nearest, even.*

The expressions (b / VAR_1) and (b / VAR_2) will be rounded to a 16-bit floating point representation (i.e. round to a 7-bit mantissa), which permits the ratio to be computed via a pair of small look-up tables. C-code for generating such tables is shown in pseudo-C function make_inv_tables().

All intermediate results in every floating point computation in the prediction algorithm will be represented in single precision floating point using rounding described below.

The IEEE Floating Point computational unit used in executing all arithmetic in the prediction tool will enable the following options:

- Round-to-Nearest, Even - Round to nearest representable value; round to the value with the least significant bit equal to zero (even) when the two nearest representable values are equally near.
- Overflow exception - Values whose magnitude is greater than the largest representable value will be set to the representation for infinty.
- Underflow exception - Gradual underflow (de-normalized numbers) will be supported; values whose magnitude is less than the smallest representable value will be set to zero.

3.5.3.2.3 Fast Algorithm for Rounding

```
/* this does not conform to IEEE conventions of round to
 * nearest, even, but it is fast
 */
static void
flt_round_inf(float *pf)
{
    int flg;
    ulong tmp, tmp1;
    float *pt = (float *)&tmp;
    *pt = *pf;                        /* write float to memory */
    tmp1 = tmp;                       /* save in tmp1 */
    flg = tmp & (ulong)0x00008000;    /* rounding position */
    tmp &= (ulong)0xffff0000;    /* truncated float */
    *pf = *pt;
    /* round 1/2 lsb toward infinity */
    if (flg) {
      tmp = tmp1 & (ulong)0xff810000;   /* 1.0 * 2^e + 1 lsb */
      *pf += *pt;                       /* add 1.0 * 2^e+ 1 lsb */
      tmp &= (ulong)0xff800000;         /* 1.0 * 2^e */
      *pf -= *pt;                       /* subtract 1.0 * 2^e */
   }
}
```

3.5.3.2.4 Generating Rounded b / Var

```
static float mnt_table[128];
static float exp_table[256];

/* function flt_round_even() only works for arguments in the range
 *           1.0 < *pf  < 2.0 - 2^-24
 */
static void
flt_round_even(float *pf)
{
   float f1, f2;

   f1 = 1.0;
   f2 = f1 + (*pf / (1<<15));
   f2 = f2 - f1;
   f2 = f2 * (1<<15);
   *pf = f2;
}

static void
make_inv_tables(void)
{
   int i;
   ulong tmp1, tmp;
   float *pf = (float *)&tmp;
   float ftmp;

   *pf = 1.0;
   tmp1 = tmp;                 /* float 1.0 */
   /* mantissa table */
   for (i=0; i<128; i++) {
      tmp = tmp1 + (i<<16);    /* float 1.m, 7 msb only */
      ftmp = b / *pf;          /* predictor constant b as in 8.3.2 */
      flt_round_even(&ftmp);   /* round to 16 bits */
      mnt_table[i] = ftmp;
   }

   /* exponent table */
   for (i=0; i<256; i++) {
      tmp = tmp1 + i<<23;      /* float 1.0 * 2^exp */
      ftmp = 1.0 / *pf;
      exp_table[i] = ftmp;
   }
}
```

3.5.3.3 Low complexity predictor processing

This Low Complexity (LC) mode of backward adaptive prediction delivers the same performance as the main mode (AAC predictor) described above, but with almost half complexity. The bitstream syntax of this prediction mode is exactly the same as in the AAC mode. The adaptation function is however different so that the LC mode is not functionally conformant to the AAC predictor. When compliance with MPEG-2 AAC is not necessary, this mode can be used to achieve lower complexity of decoding.

As the only difference between these two predictors is in the adaptive coefficient update part, this part is presented in this section. Other parts required by the LC prediction can be found in prediction section.

As the adaptive lattice predictor, the LC prediction also uses the second order predictor.

$$x_{est}(n) = a_1 x_{rec}(n-1) + a_2 x_{rec}(n-2)$$

The predictor coefficients are calculated according to the reconstructed spectral components. In order to reduce the complexity, we update (or estimate) the predictor every four samples. The covariance estimates of the reconstructed signal are computed by

$$r_{0,0} = 2\sum_{i=1}^{L-2} {x_{rec}}^2(n-i)\ ,\ r_{1,1} = \sum_{i=1}^{L-2} ({x_{rec}}^2(n-i-1) + {x_{rec}}^2(n-i+1))\ ,$$

$$r_{0,1} = r_{1,0} = r_1 = \sum_{i=1}^{L-2} (x_{rec}(n-i)x_{rec}(n-i-1) + x_{rec}(n-i+1)x_{rec}(n-i))\ ,$$

$$r_2 = 2\sum_{i=1}^{L-2} x_{rec}(n-i-1)x_{rec}(n-i+1)$$

For data length five, assume that we have the following data available

$$x_{rec}(n-4), x_{re}(n-3), x_{rec}(n-2), x_{rec}(n-1), x_{rec}(n)\ .$$

Then an obvious efficient algorithm is

$$r_{0,0} = 2*({x_{rec}}^2(n-1) + {x_{rec}}^2(n-2) + {x_{rec}}^2(n-3))\ ,$$

$$r_{1,1} = 2*{x_{rec}}^2(n-2) + {x_{rec}}^2(n) + {x_{rec}}^2(n-1) + {x_{rec}}^2(n-3) + {x_{rec}}^2(n-4)\ ,$$

$$r_{0,1} = r_{1,0} = r_1 = x_{rec}(n-4)x_{rec}(n-3) + 2*(x_{rec}(n-3) + x_{rec}(n-1))*x_{rec}(n-2) + x_{rec}(n-1)x_{rec}(n)$$

$$r_2 = ((x_{rec}(n-4) + x_{rec}(n))*x_{rec}(n-2) + x_{rec}(n-1)x_{rec}(n-3))*2\ .$$

With these covariances, the LP coefficients can be calculated by the following equation:

$$a_1 = \frac{(r_{1,1} - r_2)r_{0,1}}{r_{0,0}r_{1,1} - r_{0,1}^2}\ ,$$

$$a_2 = \frac{r_{0,0}r_2 - r_{0,1}r_1}{r_{0,0}r_{1,1} - r_{0,1}^2}\ .$$

If window_sequence is of type ONLY_LONG_SEQUENCE, LONG_START_SEQUENCE and LONG_STOP_SEQUENCE only the calculation of the reconstructed value of the quantized spectral components differs depending on the value of the **prediction_used** bit:

- If the bit is set (1), then the quantized prediction error reconstructed from the transmitted data is added to the estimate $x_{est}(n)$ calculated by the predictor resulting in the reconstructed value of the quantized spectral component, i.e. $x_{rec}(n) = x_{est}(n) + e_q(n)$
- If the bit is not set (0), then the quantized value of the spectral component is reconstructed directly from the transmitted data.

In case of short blocks, i.e. window_sequence is of type EIGHT_SHORT_SEQUENCE, prediction is always disabled and a reset is carried out for all predictors in all scalefactor bands, which is equivalent to a reinitialization, see clause 0.

For a single_channel_element, the predictor processing for one frame is done according to the following pseudo code:

(It is assumed that the reconstructed value y_rec(c) - which is either the reconstructed quantized prediction error or the reconstructed quantized spectral coefficient - is available from previous processing.)

```
if (ONLY_LONG_SEQUENCE || LONG_START_SEQUENCE || LONG_STOP_SEQUENCE) {
   for ( sfb=0; sfb<PRED_SFB_MAX; sfb++) {
      fc = swb_offset_long_window[fs_index][sfb];
      lc = swb_offset_long_window[fs_index][sfb+1];
      for (c= fc; c<lc; c++) {
         x_est[c] = predict();
         if (predictor_data_present && prediction_used[sfb] )
            x_rec[c] = x_est[c] + y_rec[c];
         else
            x_rec[c] = y_rec[c];
      }
   }
}
else {
   reset_all_predictors();
}
```

In case of channel_pair_elements with **common_window** = 1, the only difference is that the computation of x_est and x_rec in the inner for loop is done for both channels associated with the channel_pair_element. In case of channel_pair_elements with **common_window** = 0, each channel has prediction applied using that channel's prediction side information.

Quantization in Predictor Calculations

For a given predictor seven state variables need to be saved: $x_{rec}(n-4), x_{rec}(n-3), x_{rec}(n-2), x_{rec}(n-1), x_{rec}(n), a_1, a_2$. Because we update the predictor every four samples, not all variables are neccessary to be saved. Actually, for reconstructed spectral components, we only need to save ten variables every time. Including eight prediction coefficients, a total of eighteen variables needs to be saved. The ten reconstructed spectral compoenents will be saved as truncated IEEE floating-point numbers (i.e. the 16 msb of a float storage word) and other eight prediction coefficients will be uniformly quantized into eight bits.

The dequantized error spectral components are rounded toa 16-bit foating point representation(i.e. round to a 7-bit mantissa). The exact rounding algorithm to be used is shown in pseudo-C function flt_round_inf(). Note that for complexity considerations, *round to nearest, infinity* is used instead of *round to nearest, even* The recontructed spectral components (the predicted value x_{est} plus the dequantized spectral compoenent. error) will be truncated to a 16-bit floating point representation prior to being used in any calculation.

3.5.3.4 Predictor reset

Initialization of a predictor means that the predictor's state variables are set as follows: $r_0 = r_1 = 0$, $COR_1 = COR_2 = 0$, $VAR_1 = VAR_2 = 1$. When the decoding process is started, all predictors are initialized.

A cyclic reset mechanism is applied by the encoder and signaled to the decoder, in which all predictors are initialized again in a certain time interval in an interleaved way. On one hand this increases predictor stability by re-synchronizing the predictors of the encoder and the decoder and on the other hand it allows defined entry points in the bitstream.

The *whole set of predictors is subdivided into 30 so-called reset groups according to the following table:*

Reset group number	*Predictors of reset group*
1	*P0, P30, P60, P90,...*
2	*P1, P31, P61, P91,...*
3	*P2, P32, P62, P92,...*
...	
30	*P29, P59, P89, P119,...*

where P_i is the predictor which corresponds to the spectral coefficient indexed by i.

Whether or not a reset has to be applied in the current frame is determined by the **predictor_reset** bit. If this bit is set then the number of the predictor reset group to be reset in the current frame is specified in **predictor_reset_group_number**. All predictors belonging to that reset group are then initialized as described above. This initialization has to be done after the normal predictor processing for the current frame has been carried out. Note that **predictor_reset_group_number** cannot have the value 0 or 31.

A typical reset cycle starts with reset group number 1 and the reset group number is then incremented by 1 until it reaches 30, and then it starts with 1 again. Nevertheless, it may happen, e.g. due to switching between programs (bitstreams) or cutting and pasting, that there will be a discontinuity in the reset group numbering. If this is the case, these are the following three possibilities for decoder operation:

- Ignore the discontinuity and carry on the normal processing. This may result in a short audible distortion due to a mismatch (drift) between the predictors in the encoder and decoder. After one complete reset cycle (reset group n, n+1, ..., 30, 1, 2, ..., n-1) the predictors are re-synchronized again. Furthermore, a possible distortion is faded out because of the attenuation factors a and b.

- Detect the discontinuity, carry on the normal processing but mute the output until one complete reset cycle is performed and the predictors are re-synchronized again.
- Reset all predictors.

An encoder is required to signal the reset of a group at least once every 8 frames. Groups do not have to be reset in ascending order, but every group must be reset within the maximum reset interval of 8 x 30 = 240 frames. The bitstream syntax permits the encoder to signal the reset of a group at every frame, resulting in a minimum reset interval of 1 x 30 = 30 frames.

In case of a single_channel_element or a channel_pair_element with **common_window** = 0, the reset has to be applied to the predictor bank(s) of the channel(s) associated with that element. In case of a channel_pair_element with **common_window** = 1, the reset has to be applied to the two predictor banks of the two channels associated with that element.

In the case of a short block (i.e. window_sequence of type EIGHT_SHORT_SEQUENCE) all predictors in all scalefactor bands must be reset.

3.5.4 Diagrams

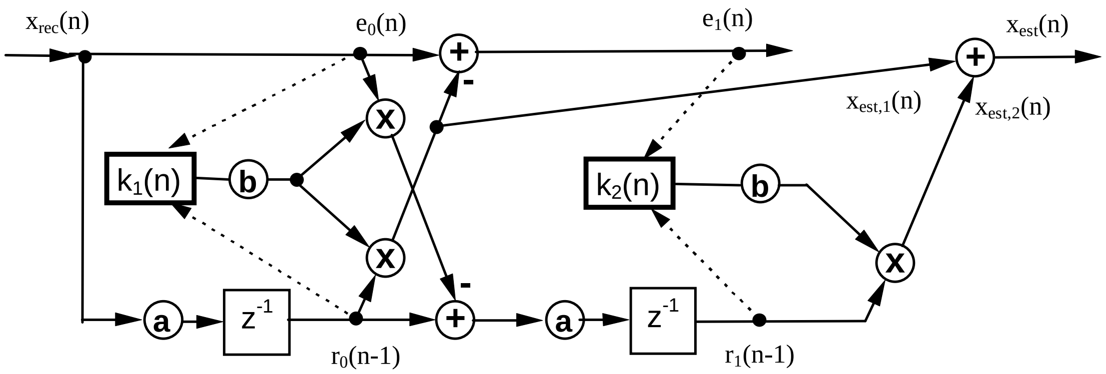


Figure 0.1 – Flow graph of AAC intra channel predictor for one spectral component in the decoder. The dotted lines indicate the signal flow for the adaptation of the predictor coefficients.

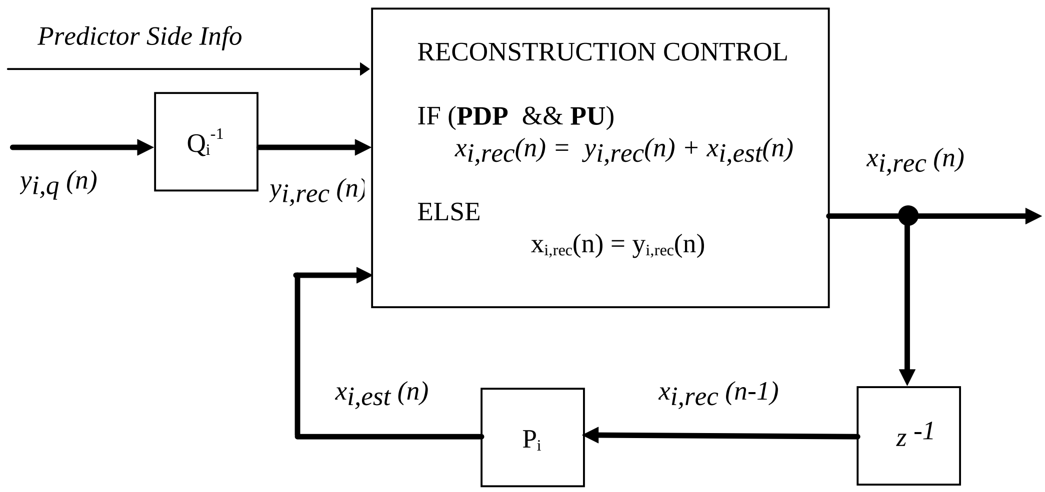


Figure 13.2 - Block diagram of decoder prediction unit for one single spectral component with predictor P_i and inverse quantizer Q_i^{-1}. The following abbreviations for the predictor side information:
PDP - predictor_data_present, PU - prediction_used.

3.6 Long Term Prediction

3.6.1 Tool description

Long term prediction (LTP) is an efficient tool for reducing the redundancy of signal between successive coding blocks. This tool is especially effective for the parts of a signal which have clear pitch property. The implementation complexity of LTP is significantly lower than the complexity of Backward Adaptive Prediction. Because the Long Term Predictor is a forward adaptive predictor (prediction coefficients are sent as side information), it is inherently less sensitive to round-off numerical errors in the decoder or bit errors in the transmitted spectral coefficients.
With MPEG-2 AAC based coding LTP can be used for any window type. With Interleaved Vector Quantisation (Twin-VQ) LTP can be only used for the long window type.

3.6.2 Definitions

ltp_data_present 1 bit indicating whether prediction is used in current frame (1) or not (0) (always present)

ltp_lag 11 bit number specifying the optimal delay from 0 to 2047

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

value of **ltp_coef**	value of LTP coefficient
000	0.570829
001	0.696616
010	0.813004
011	0.911304
100	0.984900
101	1.067894
110	1.194601
111	1.369533

ltp_short_used 1 bit indicating whether LTP is used for each short window (1) or not (0)

ltp_short_lag_present 1 bit indicating whether **ltp_short_lag** is actually transmitted (1), or omitted (0) from the bit-stream, which means that the value of **ltp_short_lag** is 0

ltp_short_lag 4 bit number specifying the relative delay for each short window to **ltp_lag** from -8 to 7

ltp_long_used 1 bit for each scalefactor band (sfb) where LTP can be used indicating whether LTP is switched on (1) or off (0) in that sfb.

3.6.3 Decoding process

The decoding process for LTP is carried out on each window of the current frame by applying 1-tap IIR filtering in the time domain to predict samples in the current frame by (quantised) samples in the previous frames. The processis controlled by the transmitted side information in a two step approach. The first control step defines whether LTP is used at all for the current frame. In the case of long window, the second control step defines, on which scalefactor bands LTP is used. In case of short windows the second control step defines which of the short windows in the coding block LTP is applied to. At the start of the decoding process, the reconstructed time samples are initialized by zeros.

For each frame, the LTP side information is extracted from the bitstream to control the further predictor processing at the decoder. In case of a single_channel_element the control information is valid for the channel with that element. In case of a channel_pair_element there are two sets of control data.

First, the **ltp_data_present** bit is read. If this bit is not set (0) then LTP is switched off for current frame and there is no further predictor side information present. In this case the **ltp_long_used** flag for each scalefactor band stored in the decoder has to be set to zero. If the **ltp_data_present** bit is set (1) then LTP is used for the current frame and the LTP parameters are read. The decoding process is different for long and short windows.

For long window, the LTP parameters are used to calculate the predicted time signals using the following formula:

$$x\_est(i) = \mathrm{ltp\_coef} * x\_rec(-N - 1 - \mathrm{ltp\_lag} + i),$$
$$i = 0,\dots,N$$

where $x\_est(i)$ are the predicted samples
$x\_rec(i)$ are reconstructed time domain samples
N is the length of transform window

Using the MDCT for long window, the predicted spectral components are obtained for current frame from the predicted time domain signal. Next, the **ltp_long_used** bits are read from the bitstream, which control the use of prediction in each scalefactor band individually, i.e. if the bit is set for a particular scalefactor band, all the predicted spectral components of this scalefactor band are used. Otherwise the predicted spectral components are set to zeros. That is, if the **ltp_long_used** bit is set, then the quantized prediction error reconstructed from the transmitted data is added to the predicted spectral component. If the bit is not set (0), then the quantized value of spectral component is reconstructed directly from the transmitted data.

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

The signal reconstruction part of decoding process for one channel can be described as following pseudo code. Here *x_est* is the predicted time domain signal, *X_est* is the corresponding frequency domain vector, *Y_rec* is the vector of decoded spectral coefficients and *X_rec* is the vector of reconstructed spectral coefficients.

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

3.7 Joint Coding

3.7.1 M/S Stereo

(similar to ISO/IEC 13818-7)

3.7.1.1 Tool description

The M/S joint channel coding operates on channel pairs. Channels are most often paired such that they have symmetric presentation relative to the listener, such as left/right or left surround/right surround. The first channel in the pair is denoted “left” and the second “right.” On a per-spectral-coefficient basis, the vector formed by the left and right channel signals is reconstructed or de-matrixed by either the identity matrix

$$\begin{bmatrix} l \\ r \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} l \\ r \end{bmatrix}$$

or the inverse M/S matrix

$$\begin{bmatrix} l \\ r \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} m \\ s \end{bmatrix}$$

The decision on which matrix to use is done on a scalefactor band by scalefactor band basis as indicated by the ms_used flags. M/S joint channel coding can only be used if common_window is ‘1’ (see clause 8.3.1).

3.7.1.2 Definitions

ms_mask_present	this two bit field indicates that the MS mask is 00 All zeros 01 A mask of max_sfb bands of ms_used follows this field 10 All ones 11 Reserved (see 6.3, Table 6.10)
ms_used[g][sfb]	one-bit flag per scalefactor band indicating that M/S coding is being used in windowgroup g and scalefactor band sfb (see 6.3, Table 6.10).
l_spec[]	Array containing the left channel spectrum of the respective channel pair.
r_spec[]	Array containing the right channel spectrum of the respective channel pair.
is_intensity(g,sfb)	function returning the intensity status, defined in 12.2.3
is_noise(g,sfb)	function returning the noise substitution status, defined in 2.12.1

3.7.1.3 Decoding Process

Reconstruct the spectral coefficients of the first (“left”) and second (“right”) channel as specified by the **mask_present** and the **ms_used[][]** flags as follows:

```
if (mask_present >= 1) {
   for (g=0; g<num_window_groups; g++) {
      for (b=0; b<window_group_length[g]; b++) {
         for(sfb=0; sfb<max_sfb; sfb++) {
            if ((ms_used[g][sfb] || mask_present == 2) &&
                  !is_intensity(g,sfb) && !is_noise(g,sfb)) {
               for (i=0; i< swb_offset[sfb+1]-swb_offset[sfb]; i++) {
                  tmp = l_spec[g][b][sfb][i] -
                      r_spec[g][b][sfb][i];
                  l_spec[g][b][sfb][i] =  l_spec[g][b][sfb][i] +
                                          r_spec[g][b][sfb][i];
                  r_spec[g][b][sfb][i] = tmp;
               }
            }
```

```
            }
        }
    }
}
```

Please note that ms_used[][] is also used in the context of intensity stereo coding and perceptual noise substitution. If intensity stereo coding or noise substitution is on for a particular scalefactor band, no M/S stereo decoding is carried out.

3.7.1.4 Diagrams

3.7.1.5 Tables

3.7.2 Intensity Stereo

(identical to ISO/IEC 13818-7)

3.7.2.1 Tool description

This tool is used to implement joint intensity stereo coding between both channels of a channel pair. Thus, both channel outputs are derived from a single set of spectral coefficients after the inverse quantization process. This is done selectively on a scalefactor band basis when intensity stereo is flagged as active.

3.7.2.2 Definitions

hcod_sf[]	Huffman codeword from the Huffman code table used for coding of scalefactors (see clause 9.2)
dpcm_is_position[][]	Differentially encoded intensity stereo position
is_position[group][sfb]	Intensity stereo position for each group and scalefactor band
l_spec[]	Array containing the left channel spectrum of the respective channel pair
r_spec[]	Array containing the right channel spectrum of the respective channel pair

3.7.2.3 Decoding Process

The use of intensity stereo coding is signaled by the use of the pseudo codebooks INTENSITY_HCB and INTENSITY_HCB2 (15 and 14) in the right channel (use of these codebooks in a left channel of a channel pair element is illegal). INTENSITY_HCB and INTENSITY_HCB2 signal in-phase and out-of-phase intensity stereo coding, respectively.
In addition, the phase relationship of the intensity stereo coding can be reversed by means of the ms_used field: Because M/S stereo coding and intensity stereo coding are mutually exclusive for a particular scalefactor band and group, the primary phase relationship indicated by the Huffman code tables is changed from in-phase to out-of-phase or vice versa if the corresponding ms_used bit is set for the respective band.
The directional information for the intensity stereo decoding is represented by an "intensity stereo position" value indicating the relation between left and right channel scaling. If intensity stereo coding is active for a particular group and scalefactor band, an intensity stereo position value is transmitted instead of the scalefactor of the right channel.
Intensity positions are coded just like scalefactors, i.e. by Huffman coding of differential values with two differences:

- there is no first value that is sent as PCM. Instead, the differential decoding is started assuming the last intensity stereo position value to be zero.
- Differential decoding is done separately between scalefactors and intensity stereo positions. In other words, the scalefactor decoder ignores interposed intensity stereo position values and vice versa (see clause 11.3.2)

The same codebook is used for coding intensity stereo positions as for scalefactors.

Two pseudo functions are defined for use in intensity stereo decoding:

```
function is_intensity(group,sfb) {
   +1   for window groups / scalefactor bands with right channel
           codebook sfb_cb[group][sfb] == INTENSITY_HCB
   -1   for window groups / scalefactor bands with right channel
           codebook sfb_cb[group][sfb] == INTENSITY_HCB2
```

```
   0    otherwise
   }

function invert_intensity(group,sfb)   {
   1-2*ms_used[group][sfb]   if (ms_mask_present == 1)
   +1                        otherwise
   }
```

The intensity stereo decoding for one channel pair is defined by the following pseudo code:

```
p = 0;
for (g=0; g<num_window_groups; g++)  {

   /* Decode intensity positions for this group */
   for (sfb=0; sfb<max_sfb; sfb++)
      if (is_intensity(g,sfb))
         is_position[g][sfb] = p += dpcm_is_position[g][sfb];

   /* Do intensity stereo decoding */
   for (b=0; b<window_group_length[g]; b++)  {
      for (sfb=0; sfb<max_sfb; sfb++)  {
         if (is_intensity(g,sfb))  {

            scale = is_intensity(g,sfb) * invert_intensity(g,sfb) *
                        0.5^(0.25*is_position[g][sfb]);
            /* Scale from left to right channel,
               do not touch left channel */
            for (i=0; i<swb_offset[sfb+1]-swb_offset[sfb]; i++)
               r_spec[g][b][sfb][i] = scale * l_spec[g][b][sfb][i];

         }

      }
   }

}
```

3.7.2.4 Integration with Intra Channel Prediction Tool

For scalefactor bands coded in intensity stereo the corresponding predictors in the right channel are switched to "off" thus effectively overriding the status specified by the prediction_used mask. The update of these predictors is done by feeding the intensity stereo decoded spectral values of the right channel as the "last quantized value" $x_{rec}(n-1)$. These values result from the scaling process from left to right channel as described in the pseudo code. The function of Long Term Prediction does not depend on Intensity Stereo.

3.7.3 Coupling Channel

(identical to ISO/IEC 13818-7)

3.7.3.1 Tool description

Coupling channel elements provide two functionalities: First, coupling channels may be used to implement generalized intensity stereo coding where channel spectra can be shared across channel boundaries. Second, coupling channels may be used to dynamically perform a downmix of one sound object into the stereo image. Note that this tool includes certain profile dependent parameters (see clause 7.1).

3.7.3.2 Definitions

ind_sw_cce_flag one bit indicating whether the coupled target syntax element is an independently switched (1) or a dependently switched (0) CCE (see 6.3, Table 6.18).

num_coupled_channels number of coupled target channels (see 6.3, Table 6.18)

cc_target_is_cpe one bit indicating if the coupled target syntax element is a CPE (1) or a SCE (0) (see 6.3, Table 6.18).

cc_target_tag_select four bit field specifying the element_instance_tag of the coupled target syntax element (see 6.3, Table 6.18).

cc_l one bit indicating that a list of gain_element values is applied to the left channel of a channel pair (see 6.3, Table 6.18).

cc_r one bit indicating that a list of gain_element values is applied to the right channel of a channel pair (see 6.3, Table 6.18).

cc_domain one bit indicating whether the coupling is performed before (0) or after (1) the TNS decoding of the coupled target channels (see 6.3, Table 6.18)

gain_element_sign one bit indicating if the transmitted gain_element values contain information about in-phase / out-of-phase coupling (1) or not (0) (see 6.3, Table 6.18)

gain_element_scale determines the amplitude resolution *cc_scale* of the scaling operation according to Table 00.11 (see 6.3, Table 6.18)

common_gain_element_present[c] one bit indicating whether Huffman coded common_gain_element values are transmitted (1) or whether Huffman coded differential gain_elements are sent (0) (see 6.3, Table 6.18)

dpcm_gain_element[][] Differentially encoded gain element

gain_element[group][sfb] Gain element for each group and scalefactor band

common_gain_element[] Gain element that is used for all window groups and scalefactor bands of one coupling target channel

spectrum_m(idx, domain) Pointer to the spectral data associated with the single_channel_element with index idx. Depending on the value of "domain", the spectral coefficients before (0) or after (1) TNS decoding are pointed to.

spectrum_l(idx, domain) Pointer to the spectral data associated with the left channel of the channel_pair_element with index idx. Depending on the value of "domain", the spectral coefficients before (0) or after (1) TNS decoding are pointed to.

spectrum_r(idx, domain) Pointer to the spectral data associated with the right channel of the channel_pair_element with index idx. Depending on the value of "domain", the spectral coefficients before (0) or after (1) TNS decoding are pointed to.

3.7.3.3 Decoding Process

The coupling channel is based on an embedded single_channel_element which is combined with some dedicated fields to accomodate its special purpose.

The coupled target syntax elements (SCEs or CPEs) are addressed using two syntax elements. First, the cc_target_is_cpe field selects whether a SCE or CPE is addressed. Second, a cc_target_tag_select filed selects the instance_tag of the SCE/CPE.

The scaling operation involved in channel coupling is defined by gain_element values which describe the applicable gain factor and sign. In accordance with the coding procedures for scalefactors and intensity stereo positions, gain_element values are differentially encoded using the Huffman table for scalefactors. Similarly, the decoded gain factors for coupling relate to window groups of spectral coefficients.

<u>Independently switched CCEs vs. dependently switched CCEs</u>
There are two kinds of CCEs. They are “independently switched” and “dependently switched” CCEs. An independently switched CCE is a CCE in which the window state (i.e. window_sequence and window_shape) of the CCE does not have to match that of any of the SCE or CPE channels that the CCE is coupled onto (target channels). This has several important implications:

- First, it is required that an independently switched CCE must only use the common_gain element, not a list of gain_elements.
- Second, the independently switched CCE must be decoded all the way to the time domain (i.e. including the synthesis filterbank) before it is scaled and added onto the various SCE and CPE channels that it is coupled to in the case that window state does not match.

A dependently switched CCE, on the other hand, must have a window state that matches all of the target SCE and CPE channels that it is coupled onto as determined by the list of cc_l and cc_r elements. In this case, the CCE only needs to be decoded as far as the frequency domain and then scaled as directed by the gain list before it is added to the target SCE or CPE channels.

The following pseudo code in function decode_ coupling_channel() defines the decoding operation for a dependently switched coupling channel element. First the spectral coefficients of the embedded single_channel_element are decoded into an internal buffer. Since the gain elements for the first coupled target (list_index == 0) are not transmitted, all gain_element values associated with this target are assumed to be 0, i.e.

the coupling channel is added to the coupled target channel in its natural scaling. Otherwise the spectral coefficients are scaled and added to the coefficients of the coupled target channels using the appropriate list of gain_element values.

An independently switched CCE is decoded like a dependently switched CCE having only common_gain_element's. However, the resulting scaled spectrum is transformed back into its time representation and then coupled in the time domain.

Please note that the gain_element lists may be shared between the left and the right channel of a target channel pair element. This is signalled by both cc_l and cc_r being zero as indicated in the table below:

cc_l,	cc_r	shared gain list present	left gain list present	right gain list present
0,	0	yes	no	no
0,	1	no	no	yes
1,	0	no	yes	no
1,	1	no	yes	yes

```
decode_coupling_channel()
{
   - decode spectral coefficients of embedded single_channel_element
     into buffer "cc_spectrum[]".

   /* Couple spectral coefficients onto target channels */
   list_index = 0;
   for (c=0; c<num_coupled_elements+1; c++)  {
      if (!cc_target_is_cpe[c])  {
         couple_channel( cc_spectrum,
                         spectrum_m( cc_target_tag_select[c], cc_domain ),
                         list_index++ );
      }
      if (cc_target_is_cpe[c])  {
         if (!cc_l[c]  &&  !cc_r[c])  {
            couple_channel( cc_spectrum,
                            spectrum_l( cc_target_tag_select[c], cc_domain ),
                            list_index );
            couple_channel( cc_spectrum,
                            spectrum_r( cc_target_tag_select[c], cc_domain ),
                            list_index++ );
         }
         if (cc_l[c])  {
            couple_channel( cc_spectrum,
                            spectrum_l( cc_target_tag_select[c], cc_domain ),
                            list_index++ ) );
         }
         if (cc_r[c])  {
            couple_channel( cc_spectrum,
                            spectrum_r( cc_target_tag_select[c], cc_domain ),
                            list_index++ ) );
         }
      }
   }
}
```

```
couple_channel( source_spectrum[], dest_spectrum[], gain_list_index )
{
   idx = gain_list_index;
   a = 0;
   cc_scale = cc_scale_table[gain_element_scale];
   for (g=0; g<num_window_groups; g++)  {

      /* Decode coupling gain elements for this group */
      if (common_gain_element_present[idx])  {

         for (sfb=0; sfb<max_sfb; sfb++)  {
            cc_sign[idx][g][sfb] = 1;
            gain_element[idx][g][sfb] = common_gain_element[idx];
```

```
        }

      }  else  {

         for (sfb=0; sfb<max_sfb; sfb++)  {
            if ( sfb_cb[g][sfb] == ZERO_HCB )
               continue;

            if (gain_element_sign)  {
               cc_sign[idx][g][sfb] =
                  1 - 2*(dpcm_gain_element[idx][g][sfb] & 0x1);
               gain_element[idx][g][sfb] =
                  a += (dpcm_gain_element[idx][g][sfb] >> 1);
            }
            else  {
               cc_sign[idx][g][sfb] = 1;
               gain_element[idx][g][sfb] =
                  a += dpcm_gain_element[idx][g][sfb];
            }
         }

      }


      /* Do coupling onto target channels */
      for (b=0; b<window_group_length[b]; b++)  {
         for (sfb=0; sfb<max_sfb; sfb++)  {

            if ( sfb_cb[g][sfb] != ZERO_HCB )  {
               cc_gain[idx][g][sfb] =
                  cc_sign[idx][g][sfb] * cc_scale^gain_element[idx][g][sfb];

               for (i=0; i<swb_offset[sfb+1]-swb_offset[sfb]; i++)
                  dest_spectrum[g][b][sfb][i] +=
                     cc_gain[idx][g][sfb] * source_spectrum[g][b][sfb][i];

            }

         }
      }

   }
}
```

Note: The array sfb_cb represents the codebook data respect to the CCE’s embedded single_channel_element (not the coupled target channel).

3.7.3.4 Tables

Table 0.1 – Scaling resolution for channel coupling (cc_scale_table)

Value of "gain_element_scale"	Amplitude Resolution "cc_scale"	Stepsize [dB]
0	2^(1/8)	0.75
1	2^(1/4)	1.50
2	2^(1/2)	3.00
3	2^1	6.00

3.8 Temporal Noise Shaping (TNS)

(similar to ISO/IEC 13818-7)

3.8.1 Tool description

Temporal Noise Shaping is used to control the temporal shape of the quantization noise within each window of the transform. This is done by applying a filtering process to parts of the spectral data of each channel.
Note that this tool includes certain profile dependent parameters (see clause 2.1).

3.8.2 Definitions

n_filt[w]	number of noise shaping filters used for window w (see 1.3, Table 6.15)
coef_res[w]	token indicating the resolution of the transmitted filter coefficients for window w, switching between a resolution of 3 bits (0) and 4 bits (1) (see 1.3, Table 6.15)
length[w][filt]	length of the region to which one filter is applied in window w (in units of scalefactor bands) (see 1.3, Table 6.15)
order[w][filt]	order of one noise shaping filter applied to window w (see 1.3, Table 6.15).
direction[w][filt]	1 bit indicating whether the filter is applied in upward (0) or downward (1) direction (see 1.3, Table 6.15)
coef_compress[w][filt]	1 bit indicating whether the most significant bit of the coefficients of the noise shaping filter filt in window w are omitted from transmission (1) or not (0) (see 1.3, Table 6.15)
coef[w][filt][i]	coefficients of one noise shaping filter applied to window w (see 1.3, Table 6.15)
spec[w][k]	Array containing the spectrum for the window w of the channel being processed

Note: Depending on the window_sequence the size of the following bitstream fields is switched for each transform window according to its window size:

Name	Window with 128 spectral lines	Other window size
’n_filt’	1	2
’length’	4	6
’order’	3	5

3.8.3 Decoding Process

The decoding process for Temporal Noise Shaping is carried out separately on each window of the current frame by applying all-pole filtering to selected regions of the spectral coefficients (see function tns_decode_frame).
The number of noise shaping filters applied to each window is specified by "n_filt". The target range of spectral coefficients is defined in units of scalefactor bands counting down "length" bands from the top band (or the bottom of the previous noise shaping band).
First the transmitted filter coefficients have to be decoded, i.e. conversion to signed numbers, inverse quantization, conversion to LPC coefficients as described in function tns_decode_coef().
Then the all-pole filters are applied to the target frequency regions of the channel’s spectral coefficients (see function tns_ar_filter()). The token "direction" is used to determine the direction the filter is slid across the coefficients (0=upward, 1=downward).
The constant TNS_MAX_BANDS defines the maximum number of scalefactor bands to which Temporal Noise Shaping is applied. The maximum possible filter order is defined by the constant TNS_MAX_ORDER. Both constants are profile dependent parameters.

The decoding process for one channel can be described as follows pseudo code:

```
/* TNS decoding for one channel and frame */
tns_decode_frame()
{
   for (w=0; w<num_windows; w++) {

      bottom = num_swb;
      for (f=0; f<n_filt[w]; f++)  {

         top = bottom;
         bottom = max( top - length[w][f], 0 );
         tns_order = min( order[w][f], TNS_MAX_ORDER );
         if (!tns_order)  continue;

         tns_decode_coef( tns_order, coef_res[w]+3, coef_compress[w][f],
                          coef[w][f], lpc[] );

         start = swb_offset[min(bottom,TNS_MAX_BANDS,max_sfb)];
         end =   swb_offset[min(top,TNS_MAX_BANDS,max_sfb)];
         if ((size = end - start) <= 0)  continue;

         if (direction[w][f])  {
            inc = -1;   start = end - 1;
         }  else  {
```

```
            inc =  1;
        }

        tns_ar_filter( &spec[w][start], size, inc, lpc[], tns_order );

      }

   }
}


/* Decoder transmitted coefficients for one TNS filter */
tns_decode_coef( order, coef_res_bits, coef_compress, coef[], a[] )
{

   /* Some internal tables */
   sgn_mask[]  = {  0x2,  0x4,  0x8 };
   neg_mask[]  = { ~0x3, ~0x7, ~0xf };

                                          /* size used for transmission */
   coef_res2 = coef_res_bits - coef_compress;
   s_mask = sgn_mask[ coef_res2 - 2 ];     /* mask for sign bit */
   n_mask = neg_mask[ coef_res2 - 2 ];     /* mask for padding neg. values */

   /* Conversion to signed integer */
   for (i=0; i<order; i++)
      tmp[i] = (coef[i] & s_mask) ? (coef[i] | n_mask) : coef[i];

   /* Inverse quantization */
   iqfac   = ((1 << (coef_res_bits-1)) - 0.5) / (π/2.0);
   iqfac_m = ((1 << (coef_res_bits-1)) + 0.5) / (π/2.0);
   for (i=0; i<order; i++)  {
      tmp2[i] = sin( tmp[i] / ((tmp[i] >= 0) ? iqfac : iqfac_m) );
   }

   /* Conversion to LPC coefficients */
   a[0] = 1;
   for (m=1; m<=order; m++)  {
      a[m] = tmp2[m-1];
      for (i=1; i<m; i++)  {
         b[i] = a[i] + tmp2[m-1] * a[m-i];
      }
      for (i=0; i<m; i++)  {
         a[i] = b[i];
      }
   }

}

tns_ar_filter( spectrum[], size, inc, lpc[], order )
{
   - Simple all-pole filter of order "order" defined by
     y(n) =  x(n) - lpc[1]*y(n-1) - ... - lpc[order]*y(n-order)

   - The state variables of the filter are initialized to zero every time

   - The output data is written over the input data ("in-place operation")

   - An input vector of "size" samples is processed and the index increment
     to the next data sample is given by "inc"
}
```

3.8.4 TNS in the scalable coder

Decoding of a scaleable core based AAC bitstream with one ore more Mono AAC Layer and one or more Stereo AAC Layer. Basically it is possible that either the core or the Mono AAC Layer(s) or the AAC stereo Layer(s) are omitted .

Case 1 Mono core + Mono AAC

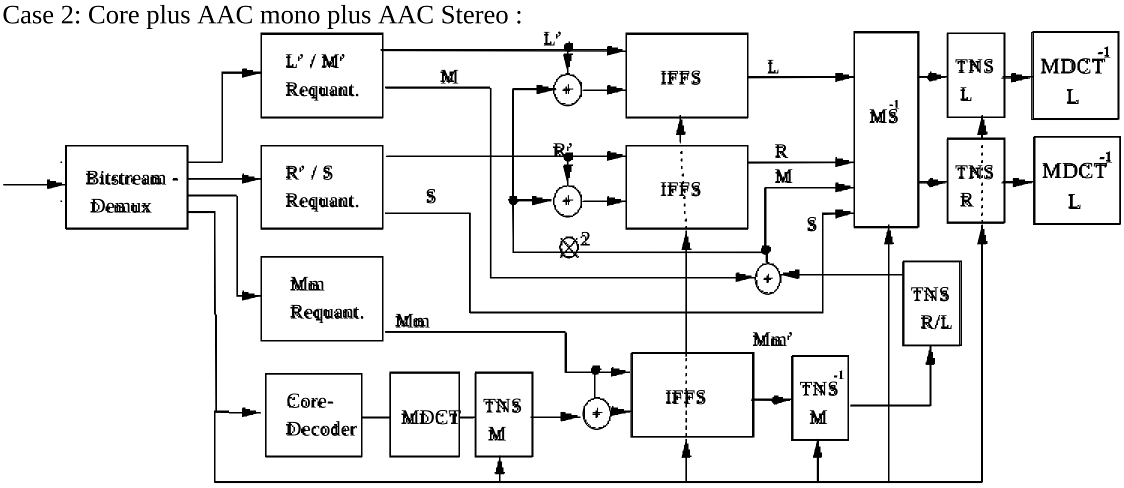
If TNS is used in a scalable coder with a core coder, the TNS encoder filters have to be applied to the output of the MDCT which is employed to generate the spectrum of the core coder. These encoder filters use the LPC coefficients already decoded for the corresponding TNS decoder filters. The filters are slid across the specified target frequency range exactly the way described for the decoder filter. The difference between decoder and encoder filtering is that each all-pole (auto-regressive) decoder filter used for TNS decoding is replaced by its inverse (all-zero, moving average) filter.

The filter equation is :

```
y[n] = x[n] + lpc[1] * x[n-1] + ... + lpc[order] * x[n-order]
```

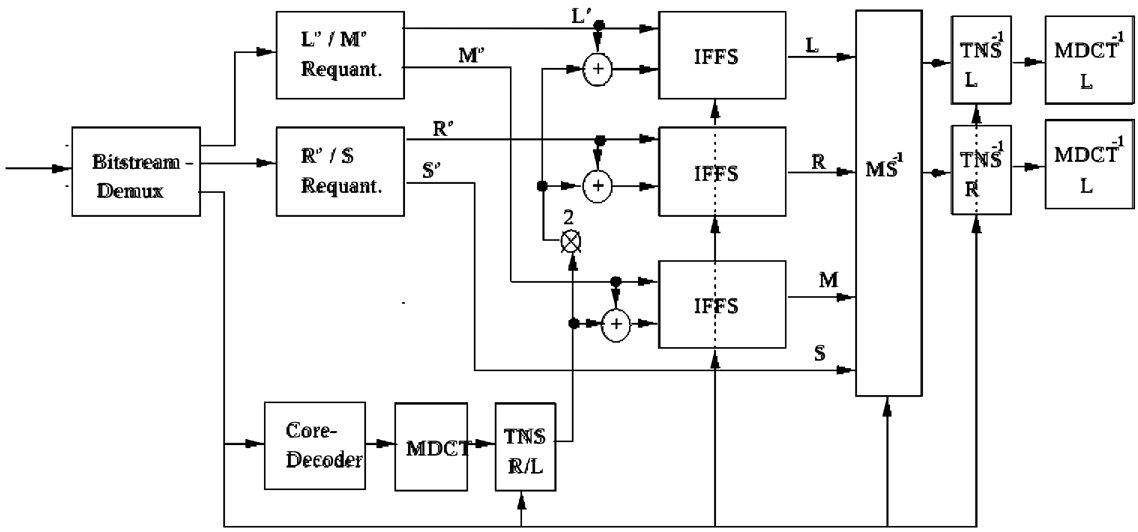
The number of filters, filtering direction etc. is controlled exactly like in the decoding process.

Case 2: Core plus AAC mono plus AAC Stereo :



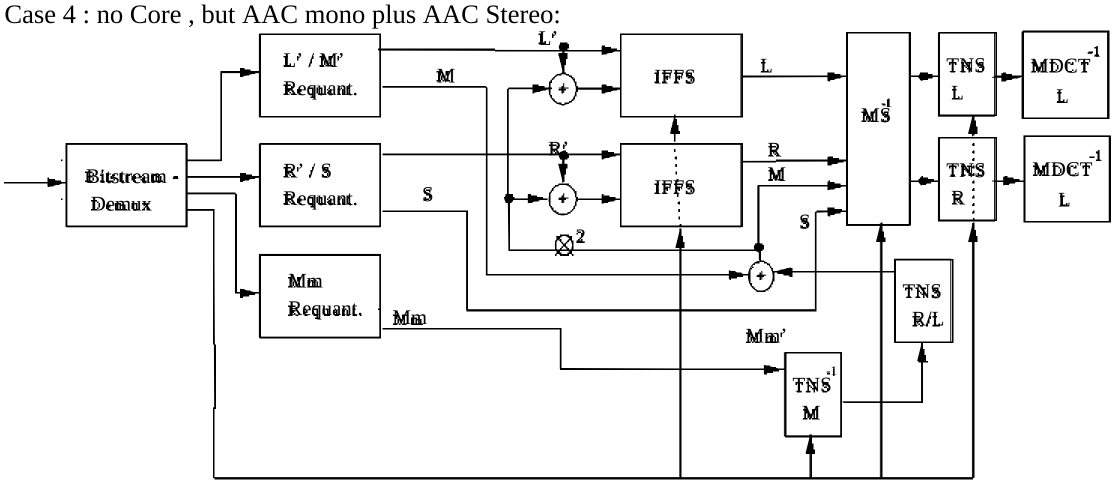
Here there are 3 sets of TNS coefficients transmitted : one set in the first AAC mono bitstream , and two (one for each channel) in the first AAC stereo Layer. The TNS filter coefficients of the mono layer are applied to the output of the MDCT for the Core, this spectrum is added to the requantized spectrum of the last Mono depending of the **diff_control** flags, then the inverse TNS filter is applied with the coefficients of the first Mono AAC Layer. After that a forward TNS filter is calculated with the TNS coefficients of ether the left or the right channel of the first stereo AAC layer depending on the bit **tns_channel_mono_layer.** The output is then added/omitted to either the requantized spectrum of the left or right channel of the last stereo AAC Layer depending on the **ms_used** bits and depending on the **diff_control_lr** bits. Then the invers MS matrix is calculated depending on the ms_usecd bits. The output is two reconstructed MDCT spectras, one for the left and one the right channel. Both are then feed into there inverse TNS filter one for each channel , the TNS coefficients are transmitted in the first AAC stereo layer.

Case 3 : Core plus AAC Stereo:



Here there are 2 sets of TNS coefficients transmitted : one for each channel in the first AAC stereo Layer. After decoding and upsampling of the Core layer a MDCT is calculated. Then a forward TNS filter is calculated with the TNS coefficients of ether the left or the right channel of the first stereo AAC layer depending on the bit **tns_channel_mono_layer .** After that a the output is then added/omitted to the requantized spectrum of either the left or right channel of the last stereo AAC Layer depending on the **ms_used** bits and depending on the **diff_control_lr** . Then the invers MS matrix is calculated depending on the ms_usecd bits. The output is two reconstructed MDCT spectras, one for the left and one the right channel. Both are then feed into there inverse TNS filter one for each channel , the TNS coefficients are transmitted in the first AAC stereo layer.

Case 4 : no Core , but AAC mono plus AAC Stereo:



Here there are 3 sets of TNS coefficients transmitted : one set in the first AAC mono bitstream , and two (one for each channel) in the first AAC stereo Layer. The inverse TNS filter is applied to the output spectrum of the last mono layer with the coefficients of the first Mono AAC Layer. After that a forward TNS filter is calculated with the TNS coefficients of ether the left or the right channel of the first stereo AAC layer depending on the bit **tns_channel_mono_layer .** The output is then added/omitted to either the requantized spectrum of the left or right channel of the last stereo AAC Layer depending on the **ms_used** bits and depending on the **diff_control_lr**. Then the invers MS matrix is calculated depending on the **ms_used** bits. The output is two reconstructed MDCT spectras, one for the left and one the right channel. Both are then feed into there inverse TNS filter one for each channel , the TNS coefficients are transmitted in the first AAC stereo layer.

3.9 Spectrum Normalization

3.9.1 Tool Description

In the TwinVQ decoder, spectral de-normalization is used in combination with inverse vector quantization of the MDCT coefficients, whose reproduction has globally flat shape. Using this tool, the spectral envelope is regenerated by decoding gain, Bark-scale envelope and an envelope specified with LPC parameters. Bark-scale envelope is reconstructed using a vector quantization decoder. LPC coefficients are quantized in LSP domain by means of 2-stage split vector quantization with moving average interframe prediction. Decoded LSP coefficients are directly used for generating an amplitude spectrum (square root of the power spectral envelope).

In a long MDCT block size mode, periodic peak components are optionally added to the flattened MDCT coefficients for low rate coder.

3.9.2 Definitions

α:	MA prediction coefficients used for LSP quantization
alfq[][] :	Predictive coefficients for Bark-envelope
AMP_MAX:	maximum value of mu-law quantizer for global gain
AMP_NM:	normalization factor for global gain
BAND_UPPER:	implicit bandwidth
BASF_STEP:	step size of base frequency quantizer for periodic peak components coding
bfreq[]:	base frequency of periodic peak components
blim_h[]:	bandwidth control factor (higher part)
blim_l[]:	bandwidth control factor (lower part)
BLIM_STEP_H:	number of steps of bandwidth control quantization (higher part)
BLIM_STEP_L:	number of steps of bandwidth control quantization (lower part)
CUT_M_H:	minimum bandwidth ratio (higher part)
CUT_M_L:	maximum bandwidth ratio (lower part)
cv_env[][]:	code vectors of envelope codebook
env[][][]:	Bark-scale envelope projected onto Bark-scale frequency axis
f b_shift[][]	Syntax element indicating the base frequency of active frequency band of the adaptive bandwidth control.
FW_ALF_STEP:	MA prediction coefficient for quantization of Bark-scale envelope
FW_CB_LEN:	length of code vector of Bark-scale envelop codebook
FW_N_DIV:	number of interleave division of Bark-scale envelope vector quantization
gain_p[][]:	gain factors of the periodic peak components
gain[][]:	gain factors of MDCT coefficients
global_gain[]:	global gain of MDCT coefficients normalized by AMP_NM
index_blim_h[]:	syntax element indicating higher part bandwidth control
index_blim_l[]:	syntax element indicating lower part of bandwidth control
index_env[][][]:	syntax elements indicating Bark-scale envelope elements
index_fw_alf[]:	syntax element indicating the MA prediction switch of Bark-scale envelope quantization
index_gain[][]:	syntax elements indicating global gain of MDCT coefficients
index_gain_sb[][][]:	syntax elements indicating subblock gain of MDCT coefficients
index_lsp0[][]:	syntax elements indicating MA prediction coefficients used for LSP quantization
index_lsp1[]:	syntax element indicating the first-stage LSP quantization
index_lsp2[][]:	syntax element indicating the second-stage LSP quantization
index_pgain[]:	syntax elements indicating gain of periodic peak components
index_pit[][]:	syntax elements indicating base frequency of periodic peak components
index_shape0_p[]:	syntax element indicating peak elements quantization index for shape vector of conjugate channel 0
index_shape1_p[]:	periodic peak elements quantization index for shape vector of conjugate channel 1
isp[]:	split point table for second-stage LSP quantization
lengthp[]:	lengths of code vectors for periodic peak components quantization
lnenv[][]	Bark-scale envelope projected onto linear-scale frequency axis
LOWER_BOUNDARY[][]:	lower boundary of active frequency band used in scaleable layers
lpenv[][]:	LPC spectral envelope

lsp[][]: LPC coefficients which range is set from zero to π .
lyr: indicates enhancement layer number. Number 0 is assigned for the base layer.
LSP_SPLIT: number of splits of 2nd-stage vector quantization for LSP coding
MU: mu factor for mu-law quantization for gain
N_CRB: number of subbands for Bark-scale envelope coding
N_DIV_P: number of interleave division for periodic peak components coding
N_FR: number of samples in a subframe
N_FR_P: number of elements of periodic peak components
N_SF: number of subframe in a frame. The value is set in section 3.4.3.
p_cv_env[][]: Bark-scale envelope vector reconstructed in the previous frame
pit[]: periodic peak components
pit_seq[][][]: periodic peak components projected into linear scale
PIT_CB_SIZE: size of codebook for periodic peak components quantization
PGAIN_MAX: maximum value of mu-law quantizer for gain of periodic peak components
PGAIN_MU: mu factor for mu-law quantization for periodic peak components
PGAIN_STEP: step size of mu-law quantizer for gain of periodic peak components
pol0_p: polarity of conjugate channel 0 for periodic peak components quantization
pol1_p polarity of conjugate channel 1 for periodic peak components quantization
sp_cv0[]: reconstructed shape of conjugate channel 0 for periodic peak components quantization
sp_cv1[]: reconstructed shape of conjugate channel 1 for periodic peak components quantization
STEP: step size of mu-law quantizer for global gain
SUB_AMP_MAX: maximum amplitude of mu-law quantizer for subframe gain ratio
SUB_AMP_NM: normalization factor for subframe gain ratio
SUB_STEP: step size of mu-law quantizer for subframe gain
subg_ratio[]: subframe gain ratio
UPPER_BOUNDARY[][]:
upper boundary of active frequency band used in scaleable layers
v[]: LSP coefficients
*v*1[]: reconstructed vector from 1st-stage VQ for LSP coding
*v*2[]: reconstructed vector from 2nd-stage VQ for LSP coding

x_flat[]: normalized MDCT coefficients (input)
spec[][][]: de-normalized MDCT coefficients (output)

3.9.3 Decoding process

The decoding process consists of five parts: gain decoding, Bark-scale envelope decoding, periodic peak components decoding, LPC spectrum decoding, and inverse normalization (see Fig. 3.9.1).

3.9.3.1 Initializations

Befor starting any process, prediction memories p_cv_env[][] and $\alpha_i^{(j)}$ are cleared.

3.9.3.2 Gain decoding

In the first step of gain decoding, the global gain is decoded using μ-law inverse quantizer described as follows:

```
for (i_ch=0; i_ch<N_CH; i_ch++){
   g_temp = index_gain * STEP + STEP / 2
   global_gain =
      (AMP_MAX * (exp10(g_temp * log10(1.+MU) / AMP_MAX) -1) / MU) / AMP_NM;
}
```

Next, subband gain ratios are decoded using mu-law inverse quantizer described as follows:

```
for (i_ch=0; i_ch<N_CH; i_ch++){
   if (N_SF > 1){
      for(isf=0; isf<N_SF; isf++){
         g_temp = index_gain_sb[i_ch][isf+1] * SUB_STEP +SUB_STEP/2.;
         subg_ratio[isf] =
           (SUB_AMP_MAX*(exp10(g_temp*log10(1.+MU)/SUB_AMP_MAX)-1)/MU)
            / SUB_AMP_NM;
      }
   }
   else{
      subg_ratio[i_ch][0] = 1
   }
}
```

Finally, gain factors are reconstructed as follows:

```
for (i_ch=0; i_ch<N_CH; i_ch++){
   for(isf=0; isf<N_SF; isf++){
      gain[i_ch][isf] = global_gain[i_ch] * subg_ratio[i_ch][isf] / SUB_AMP_NM;
   }
}
```

3.9.3.3 Decoding of periodic peak components

Periodic peak components are optionally added to the input coefficients. The periodic peak components are coded using vector quantization. This process is active when the parameter ppc_present is set to TRUE. Otherwise, all the elements of output array, pit_seq[] are set to zero and the process is skipped.

This process works when the block length type is LONG and the configuration mode is 16_16 of 08_06.

3.9.3.3.1 Decoding of polarity

```
for (idiv=0; idiv<N_DIV_P; idiv++){
   pol0[idiv] = 2*(index_shape0_p[idiv] / PIT_CB_SIZE) - 1
   pol1[idiv] = 2*(index_shape1_p[idiv] / PIT_CB_SIZE) - 1
}
```

3.9.3.3.2 Decoding of shape code

```
for (idiv=0; idiv<N_DIV_P; idiv++){
   index0[idiv] = index_shape0_p[idiv] % PIT_CB_SIZE
   index1[idiv] = index_shape1_p[idiv] % PIT_CB_SIZE
}
```

3.9.3.3.3 Decoding gain of periodic peak components

```
for (i_ch=0; i_ch<N_CH; i_ch++){
   temp = index_pgain[i_ch] * PGAIN_STEP + PGAIN_STEP / 2
   gain_p[i_ch] =
      (PGAIN_MAX*(exp10(temp*log10(1.+PGAIN_MU)/PGAIN_MAX)-1)/PGAIN_MU);
}
```

3.9.3.3.4 Reconstruction of periodic peak components

There are two steps of procedures. First the lengths of code vectors for periodic peak components, lengthp[], are calculated. Then, the periodic peak components pit[] are calculated.

```
for (idiv=0; idiv<N_DIV_P; idiv++){
   lengthp[idiv] = (N_FR_P*N_CH+N_DIV_P-1-idiv) / N_DIV_P;
}

for (idiv=0; idiv<N_DIV_P; idiv++){
   for (icv=0; icv<lengthp[idiv]; icv++){
      ismp = idiv + icv * N_DIV_P
      pit[ismp] = gain_p * (pol0[idiv]*pit_cv0[index0[idiv][icv]] \\
                     + pol1[idiv]*pit_cv1[index1[idiv]][icv]) / 2
   }
}
```

3.9.3.3.5 Projecting periodic peak components into linear scale

First, parameters are calculated as following:

```
fcmin = log2((N_FR/SAMPF)*0.2);
fcmax = log2((N_FR/SAMPF)*2.4);
```

If bitrate mode is 16 kbit/s, then
`bandwidth =2.`
If bitrate mode is 6 kbit/s (8kHz sampling), then
`bandwidth = 1.5`

where fcmin is the minimum frequency to be quantization expressed in log scale, fcmax is the maximum.

Next, base frequency of the periodic peak components is decoded according to the following procedure:

```
for (i_ch=0; i_ch<N_CH; i_ch++){
   dtmp = (double)index_pit[i_ch]/ (double) BASF_STEP;
   dtmp = dtmp * (fcmax-fcmin) + fcmin;
   bfreq[i_ch] = (double)pow2(dtmp);
}
```

Before projecting the perodic peak components into linear scale, all the elements of target array pit_seq[][] is set to zero:

```
for (i_ch=0; i_ch<N_CH; i_ch++){
   for (ismp=0; ismp<N_FR; ismp++){
      pit_seq[i_ch][ismp] = 0.;;
   }
}
```

Then, reconstructed pereodic peak components are projected to linear-scale as follows:

```
for (i_ch=0; i_ch<N_CH; i_ch++){
   npcount = (int)( N_FR_P*bandwidth/(N_FR/bfreq[i_ch]));
   iscount=0;
   for (jj=0; jj<npcount/2; jj++){
         pit_seq[i_ch][jj] = pit[jj+i_ch*N_FR_P];
         iscount ++;
   }
   for (ii=0; ii<(N_FR_P )&& (iscount<N_FR_P); ii++){
      i_smp = (int)(bfreq[i_ch]*(ii+1)+0.5);
      for (jj=-npcount/2; jj<(npcount-1)/2+1; jj++){
         pit_seq[i_ch][i_smp+jj] = pit[iscount+i_ch*N_FR_P];
         iscount ++;
         if(iscount >= N_FR_P) break;
      }
   }
}
```

In case of skipping the periodic peak components decoding process, all the elements of pitch component array pit_seq[][] are set to zero.

3.9.3.4 Decoding of Bark-scale envelope

The Bark-scale envelope is decoded in each subframe. There are two procedure stages: inverse quantizing of the envelope vectors env[][] and projecting the bark-scale envelopes env[][] onto the linear scale envelopes lnenv[][].

3.9.3.4.1 Inverse quantization of envelope vector

The inverse quantization part is illustrated in Fig. 3.9.2.

```
for (i_ch=0; i_ch<N_CH; i_ch++){
   for (isf=0; isf<N_SF; isf++){
      alfq[i_ch][isf] = index_fw_alf[i_ch][isf] * FW_ALF_STEP
      for (ifdiv=0; ifdiv<FW_N_DIV; ifdiv++){
         for (icv=0; icv<FW_CB_LEN; icv++){
            ienv = FW_N_DIV * icv + ifdiv
            dtmp = cv_env[index_env[ich][isf][ifdiv]][icv]
            env[i_ch][isf][ienv] = dtmp + alfq[i_ch][isf] * p_cv_env[i_ch][icv] + 1
            p_cv_env[i_ch][icv] = dtmp
         }
      }
   }
}
```

The cv_env[][] is the Bark-scale envelope codebook.

3.9.3.4.2 Projecting the Bark-scale envelope onto a linear scale

The envelopes env[][][] are expressed using the Bark scale on the frequency axis. The de-normalization procedure requires a linear-scale envelopes.

Befor the projecting process, the boundary table of the bark-scale subband, crb_tbl[], is determined. If the scaleable layer number lyr is equal or less than 1, values of the Bark-scale subband table is assigned dependent on the window type and configuration mode as listed in the tables from 3.9.6 to 3.9.23. If lyr>1, values of the Bark-scale subband table are stored in the temporal memory, crb_tbl_tmp.

After the crb_tbl[] is determined, the projecting process is done as follows:

```
for (i_ch=0; i_ch<N_CH; i_ch++){
   if (lyr>1){
      for (ienv=0; ienv<N_CRB; ienv++){
         crb_tbl[ienv] =
            crb_tbl_tmp[ienv]+LOWER_BOUNDARY[lyr][i_ch] - LOWER_BOUNDARY[2][i_ch];
         }
      }
   }
   for (isf=0; isf<N_SF; isf++){
      ismp=0
      for (ienv=0; ienv<N_CRB; ienv++){
         while (ismp<crb_tbl[ienv]){
            lnenv[i_ch][isf][ismp] = env[i_ch][isf][ienv]
            ismp++
         }
      }
   }
}
```

Values of the UPPER_BOUNDARY[][] and LOWER_BOUNDARY[][] are defined in section 3.4.4.4.

3.9.3.5 Decoding of LPC spectrum

The LPC spectrum is represented by LSP coefficients. In the decoding process, LSP coefficients are reconstructed first; then they are transformed into the LPC spectrum, which represents the square root of the power spectrum.

3.9.3.5.1 LSP coefficients decoding using the MA prediction

MA prediction coefficients are determined by referring to the coefficient table $\alpha\, t_i^{(j)}$. The rule is:

$$\alpha_i^{(j)}[i\_ch] = \alpha\, t_i^{(j)}[i\_ch](index\_lsp0[i\_ch]) \qquad \text{for } i = 1 \text{ to } N\_PR,\ j = 1 \text{ to } MA\_NP,\ i\_ch = 0 \text{ to } N\_CH - 1,$$

where i is LPC order and j is MA prediction order. The coefficient table $\alpha\, t_i^{(j)}$ is chosen according to the configuration mode among codebooks listed in tables C.48, C.51,
C.54, C.57, and C.60.

3.9.3.5.2 First-stage inverse quantization of LSP decoding

$$v1_i[i\_ch] = lspcode1_i(index\_lsp1[i\_ch]) \qquad \text{for } i = 1 \text{ to } N\_PR,\ i\_ch = 0 \text{ to } N\_CH - 1,$$

where lspcode1 is the first-stage LSP codebook chosen according to the configuration mode among codebooks listed in tables C.46, C.49, C.52 C.55, and C.58.

3.9.3.5.3 Second-stage inverse quantization of LSP decoding

$$v2_i[i\_ch] = lspcode2_i(index\_lsp2[i\_ch][k])$$
$$\text{for } k = 0 \text{ to } LSP\_SPLIT - 1,\ \ i = isp(k) + 1 \text{ to } isp(k+1) - 1,\ \ i\_ch = 0 \text{ to } N\_CH - 1,$$

where lspcode2 is the second-stage LSP codebook chosen according to the configuration mode among codebooks listed in tables C.47, C.50, C.53, C.56, and C.59. Values of isp(k) are assigned dependent on bitrate modes as listed in the tables from 3.9.24 to 3.9.26.

3.9.3.5.4 Reconstruction of LSP coefficients

LSP coefficients lsp[][] are calculated as follows:

$$\vec{v}[i\_ch] = \vec{v1}[i\_ch] + \vec{v2}[i\_ch], \quad \text{for } i\_ch = 0 \text{ to } N\_CH - 1$$

$$\alpha_i^{(0)}[i\_ch] = 1 - \sum_{j=1}^{MA\_NP} \alpha_i^{(j)}[i\_ch] \qquad \text{for } i = 1 \text{ to } N\_PR,\ i\_ch = 0 \text{ to } N\_CH - 1$$

$$lsp[i\_ch][i] = \sum_{j=0}^{MA\_NP} \alpha_i^{(j)}[i\_ch] \cdot v_i^{(-j)}[i\_ch] \quad \text{for } i = 1 \text{ to } N\_PR,\ i\_ch = 0 \text{ to } N\_CH - 1$$

$$\vec{v}^{(j-1)}[i\_ch] = \vec{v}^{(j)}[i\_ch] \qquad \text{for } j = -MA\_NP - 1 \text{ to } 0,\ i\_ch = 0 \text{ to } N\_CH - 1$$

3.9.3.5.5 Transforming LSP parameters into LPC spectrum

The LPC spectrum corresponding to ii-th MDCT coefficient, lpenv[][] is defind as follows:

```
for (i_ch=0; i_ch<N_CH; i_ch++){
   for (ii=1; ii<=N_FR-1; ii++){
         for (i=2, P[i_ch]=1.0; i<=N_PR; i+=2)
         P[i_ch] *= (cos(PI*ii/N_FR)-cos(lsp[i]))^2;
      for (i=1, Q[i_ch]=1.0; i<=N_PR; i+=2)
         Q[i_ch] *= (cos(PI*ii/N_FR)-cos(lsp[i]))^2;
      lpenv[ii] = 1/((1-cos(PI*ii/N_FR))*P[i_ch] + (1+cos(PI*ii/N_FR))*Q[i_ch]);
   }
}
```

If this tool is used as an element of scalable coder, LPC spectrum is squeezed into the active frequency band:

```
for (i_ch=0; i_ch<N_CH; i_ch++){
   nfr_lu = UPPER_BOUNDARY[lyr][i_ch] - LOWER_BOUNDARY[lyr][i_ch];
   for (ismp=0; ismp<LOWER_BOUNDARY[lyr][i_ch]; ismp++){
      lpenv_tmp[i_ch][ismp] = 0;
   }
   for(ismp=0; ismp<nfr_lu; ismp++){
      lpenv_tmp[i_ch][ismp+LOWER_BOUNDARY[lyr][i_ch]] =
         lpenv[i_ch][(int)(ismp*N_FR*/(ac_top - ac_btm))];
   }
   for (ismp=UPPER_BOUNDARY; ismp<N_FR; ismp++){
      lpenv_tmp[i_ch][ismp] = 0;
   }
   for(ismp=0; ismp<N_FR; ismp++){
      lpenv[i_ch][ismp] = lpenv_tmp[i_ch][ismp];
   }
}
```

The values of UPPER_BOUNDARY, LOWER_BOUNDARY, ac_top and ac_btm are defined in section 3.4.4.4.

3.9.3.6 Inverse normalization

Input coefficients x_flat[] are applied to inverse normalization according to the following procedure, and output coefficients spec[][][] are created.

```
for (i_ch=0; i_ch<N_CH; i_ch++){
   for (isf=0; isf<N_SF; isf++){
      for (ismp=0; ismp<N_FR; ismp++){
         spec[isf][ismp] =
          (x_flat[ismp+(isf+i_ch*N_SF)*N_FR]*lpenv[i_ch][ismp]*lnenv[i_ch][isf][ismp]
           + pit_seq[i_ch][ismp]) * gain[i_ch][isf];
      }
   }
}
```

3.9.3.7 Bandwidth control

This functionaliry is valid only in the compression configuration modes.

After the inverse normalization, upper and lower bands of output coefficients spec[][][] are set to zero.

In the bandwidth decoding modules, higher signal bandwidth ratio *blim_h*[] is decoded as follows:

```
for (i_ch=0; i_ch<N_CH; i_ch++){
   blim_h[i_ch] =
      (1. - (1.-CUT_M_H) * (double)index_blim_h[i_ch]/(double)BLIM_STEP_H))
      * BAND_UPPER;
}
```

The blim_l[] is decoded as follows:

```
for (i_ch=0; i_ch<N_CH; i_ch++){
   if (index_blim_l[i_ch] == 1) blim_l[i_ch] = CUT_M_L;
   else                         blim_l = 0;
}
```

In the bandwidth limitation module, higher and lower parts of MDCT coefficients are set to zero as follows:

```
for (i_ch=0; i_ch<N_CH; i_ch++){
   NbaseH = blim_h[i_ch] * N_FR
   NbaseL = blim_l[i_ch] * N_FR
   for (isf=0; isf<N_SF; isf++){
      for (ismp=NbaseH; ismp<N_FR; ismp++){
         spec[i_ch][isf][ismp] = 0
      }
      for (ismp=0; ismp<NbaseL; ismp++){
         spec[i_ch][isf][ismp] = 0
      }
   }
}
```

3.9.4 Diagrams

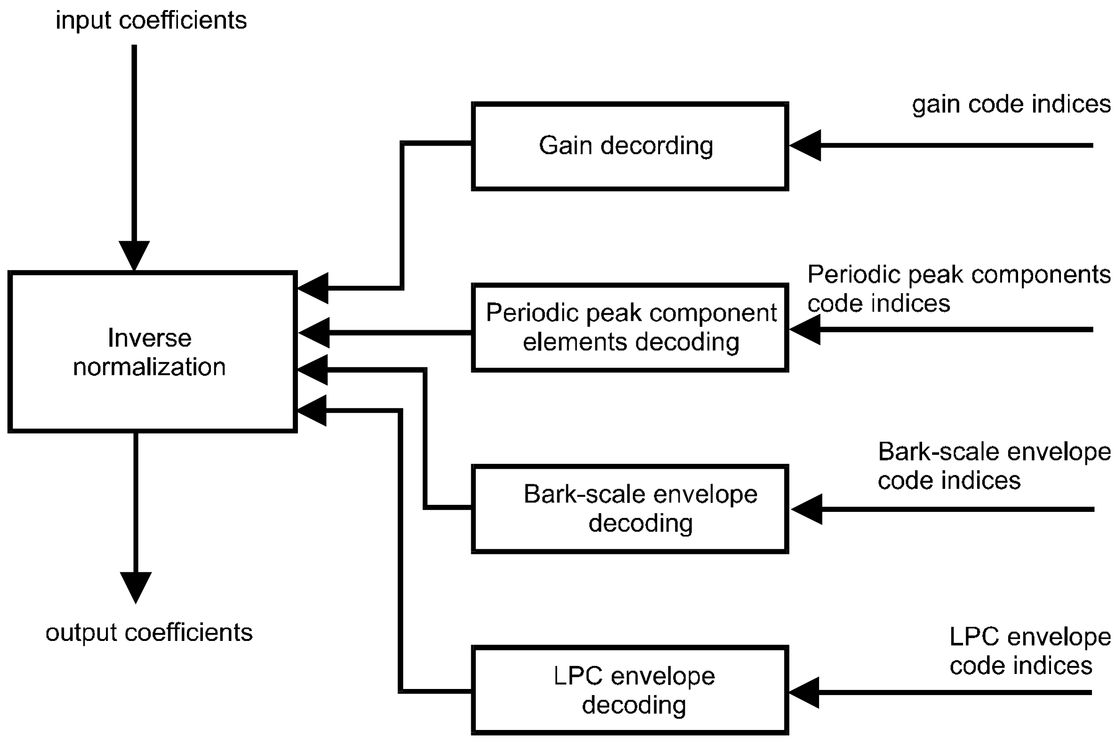


Figure 3.9. 1 — Decoding process of the TwinVQ

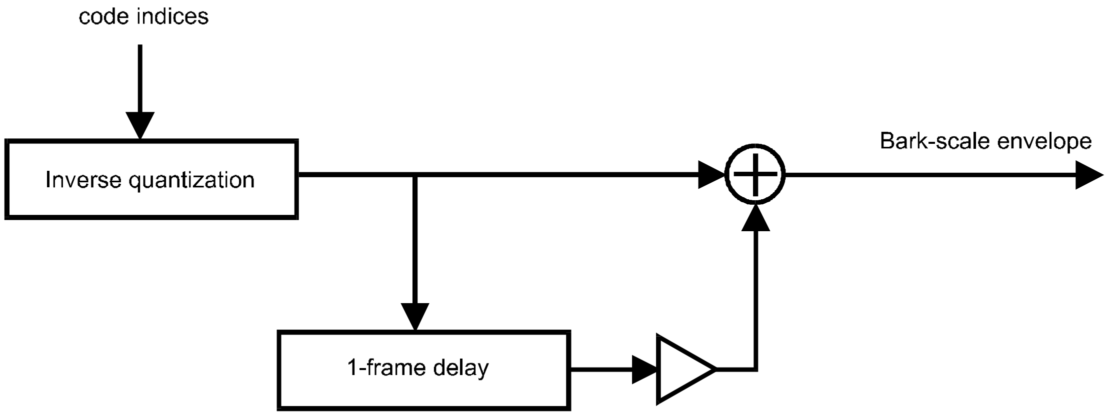


Figure 3.9. 2 — Reconstruction process of the Bark-scale envelope

3.9.5 Tables

Parameters used in the inverse spectrum normalization process are set as follows:

Table 3.9.1: Parameter table for the 24_06/24_06_960 configuration mode(Long-Short)

Parameter	Value	Meaning
Common:		
AMP_MAX	16000	Maximum amplitude in the mu-law coding of the frame gain
AMP_NM	1024	Normalized amplitude of flattened coefficients
BAND_UPPER	0.33333	Implicit bandwidth
BLIM_BITS_H	0	Bits for higher bandwidth control
BLIM_BITS_L	0	Bits for lower bandwidth control
BLIM_STEP_H	0	Number of steps of band limitation quantization
CUT_M_H	-	Minimum badwidth ratio (higher part)
CUT_M_L	-	Maximum bandwidth ratio (lower part)
FW_ALF_STEP	0.5	MA prediction coefficient for the envelope coding
GAIN_BIT	9	Number of bits for the frame gain coding
LSP_BIT0	1	Number of bits for prediction switch (LSP coding)
LSP_BIT1	6	Number of bits at the first stage (LSP coding)
LSP_BIT2	4	Number of bits per split VQ at the second stage (LSP coding)
LSP_SPLIT	3	Number of split at the second stage (LSP coding)
MA_NP	1	MA prediction order (LSP coding)
MU	100	Parameter mu in the mu-law coding of the frame power
N_PR	20	LPC order
NUM_STEP	512	Number of gain-quantization steps
STEP	AMP_MAX / (NUM_STEP-1) = 31.31	Step width of gain-quantization
SUB_AMP_MAX	4700	Maximum of subframe-gain to frame-gain ratio
SUB_AMP_NM	1024	Normalized subframe amplitude of flattened coefficients
SUB_GAIN_BIT	4	Number of bits for the subframe gain coding
SUB_NUM_STEP	16	Number of sub-gain-quantization steps
SUB_STEP	SUB_AMP_MAX/ (SUB_NUM_STEP-1)= 313.3	Step width of sub-gain-quantization
Long block:		
FW_CB_LEN	9	Envelope code vector length
FW_N_BIT	6	Envelope code bits
FW_N_DIV	7	Number of division in the envelope coding
N_CRB	64	Numbef of bark-scale subbands

N_FR	1024/960	MDCT block size
PIT_N_BIT	0	Available bits for periodic peak components quantization
Short block:		
FW_CB_LEN	-	Envelope code vector length
FW_N_BIT	0	Envelope code bits
FW_N_DIV	-	Number of division in the envelope coding
N_CRB	-	Numbef of bark-scale subbands
N_FR	128/120	MDCT block size

Table 3.9.2: Parameter table for the SCL_1/SCL_1_960 configuration mode

Parameter	Value	Meaning
Common:		
AMP_MAX	8000	Maximum amplitude in the mu-law coding of the frame gain
AMP_NM	1024	Normalized amplitude of flattened coefficients
FW_ALF_STEP	0.5	MA prediction coefficient for the envelope coding
GAIN_BIT	8	Number of bits for the frame gain coding
LSP_BIT0	1	Number of bits for prediction switch (LSP coding)
LSP_BIT1	6	Number of bits at the first stage (LSP coding)
LSP_BIT2	4	Number of bits per split VQ at the second stage (LSP coding)
LSP_SPLIT	3	Number of split at the second stage (LSP coding)
MA_NP	1	MA prediction order (LSP coding)
MU	100	Parameter mu in the mu-law coding of the frame power
N_PR	20	LPC order
NUM_STEP	256	Number of gain-quantization steps
STEP	AMP_MAX / (NUM_STEP-1) = 31.37	Step width of gain-quantization
SUB_AMP_MAX	6000	Maximum of subframe-gain to frame-gain ratio
SUB_AMP_NM	1024	Normalized subframe amplitude of flattened coefficients
SUB_GAIN_BIT	4	Number of bits for the subframe gain coding
SUB_NUM_STEP	16	Number of sub-gain-quantization steps
SUB_STEP	SUB_AMP_MAX/ (SUB_NUM_STEP-1)= 400.0	Step width of sub-gain-quantization
Long block:		
FW_CB_LEN	8	Envelope code vector length
FW_N_BIT	6	Envelope code bits
FW_N_DIV	8	Number of division in the envelope coding
N_CRB	64	Numbef of bark-scale subbands
N_FR	1024/960	MDCT block size
Short block:		
FW_CB_LEN	-	Envelope code vector length
FW_N_BIT	0	Envelope code bits
FW_N_DIV	-	Number of division in the envelope coding
N_CRB	-	Numbef of bark-scale subbands
N_FR	128/120	MDCT block size

Table 3.9.3: Parameter table for the SCL_2/SCL_2_960 configuration mode

Parameter	Value	Meaning
Common:		
AMP_MAX	8000	Maximum amplitude in the mu-law coding of the frame gain
AMP_NM	1024	Normalized amplitude of flattened coefficients
FW_ALF_STEP	0.5	MA prediction coefficient for the envelope coding
GAIN_BIT	7	Number of bits for the frame gain coding
LSP_BIT0	1	Number of bits for prediction switch (LSP coding)
LSP_BIT1	6	Number of bits at the first stage (LSP coding)
LSP_BIT2	4	Number of bits per split VQ at the second stage (LSP coding)

LSP_SPLIT	3	Number of split at the second stage (LSP coding)
MA_NP	1	MA prediction order (LSP coding)
MU	100	Parameter mu in the mu-law coding of the frame power
N_PR	20	LPC order
NUM_STEP	128	Number of gain-quantization steps
STEP	AMP_MAX / (NUM_STEP-1) = 62.99	Step width of gain-quantization
SUB_AMP_MAX	6000	Maximum of subframe-gain to frame-gain ratio
SUB_AMP_NM	1024	Normalized subframe amplitude of flattened coefficients
SUB_GAIN_BIT	4	Number of bits for the subframe gain coding
SUB_NUM_STEP	16	Number of sub-gain-quantization steps
SUB_STEP	SUB_AMP_MAX/ (SUB_NUM_STEP-1)= 400.0	Step width of sub-gain-quantization
Long block:		
FW_CB_LEN	8	Envelope code vector length
FW_N_BIT	6	Envelope code bits
FW_N_DIV	8	Number of division in the envelope coding
N_CRB	64	Numbef of bark-scale subbands
N_FR	1024/960	MDCT block size
Short block:		
FW_CB_LEN	-	Envelope code vector length
FW_N_BIT	0	Envelope code bits
FW_N_DIV	0	Number of division in the envelope coding
N_CRB	0	Numbef of bark-scale subbands
N_FR	128/120	MDCT block size

Table 3.9.4: Parameter table for the 16_16 configuration mode

Parameter	Value	Meaning
Common:		
AMP_MAX	10000	Maximum amplitude in the mu-law coding of the frame gain
AMP_NM	1024	Normalized amplitude of flattened coefficients
BAND_UPPER	1.0	Implicit bandwidth
BLIM_BITS_H	2	Bits for higher bandwidth control
BLIM_BITS_L	1	Bits for lower bandwidth control
BLIM_STEP_H	3	Number of steps of band limitation quantization
CUT_M_H	0.7	Minimum bandwidth ratio (higher part)
CUT_M_L	0.005	Maximum bandwidth ratio (lower part)
FW_ALF_STEP	0.5	MA prediction coefficient for the envelope coding
GAIN_BIT	8	Number of bits for the frame gain coding
LSP_BIT0	1	Number of bits for prediction switch (LSP coding)
LSP_BIT1	6	Number of bits at the first stage (LSP coding)
LSP_BIT2	4	Number of bits per split VQ at the second stage (LSP coding)
LSP_SPLIT	3	Number of split at the second stage (LSP coding)
MA_NP	1	MA prediction order (LSP coding)
MU	100	Parameter mu in the mu-law coding of the frame power
N_PR	16	LPC order
NUM_STEP	256	Number of gain-quantization steps
SAMPF	16000.0	Sampling frequency of input audio signal
STEP	AMP_MAX / (NUM_STEP-1) = 39.1	Step width of gain-quantization
SUB_AMP_MAX	4500	Maximum of subframe-gain to frame-gain ratio
SUB_AMP_NM	1024	Normalized subframe amplitude of flattened coefficients
SUB_GAIN_BIT	5	Number of bits for the subframe gain coding
SUB_NUM_STEP	32	Number of sub-gain-quantization steps
SUB_STEP	SUB_AMP_MAX/ (SUB_NUM_STEP-1)= 142.8	Step width of sub-gain-quantization
Long block:		

FW_CB_LEN	7	Envelope code vector length
FW_N_BIT	6	Envelope code bits
FW_N_DIV	3	Number of division in the envelope coding
N_CRB	21	Numbef of bark-scale subbands
N_FR	512	MDCT block size
PIT_N_BIT	28	Available bits for periodic peak components quantization
N_DIV_P	2	Number of division in the periodic peak components quantization
PIT_CB_SIZE	64	Size of codebook for periodic peak components quantization
PGAIN_MAX	20000.	maximum value of mu-law quantizer for gain of periodic peak components
PGAIN_MU	200	mu factor for mu-law quantization for periodic peak components
PGAIN_BITS	7	coding bits for gain of periodic peak components
PGAIN_STEP	PGAIN_MAX/ ((1<<PGAIN_BITS)-1)	step size of mu-law quantizer for gain of periodic peak components
BASF_BITS	9	bits for base frequncy coding in PPC coding
N_FR_P	30	number of elements of periodic peak components
Medium block:		
FW_CB_LEN	10	Envelope code vector length
FW_N_BIT	5	Envelope code bits
FW_N_DIV	2	Number of division in the envelope coding
N_CRB	20	Numbef of bark-scale subbands
N_FR	256	MDCT block size
Short block:		
FW_CB_LEN	10	Envelope code vector length
FW_N_BIT	6	Envelope code bits
FW_N_DIV	1	Number of division in the envelope coding
N_CRB	10	Numbef of bark-scale subbands
N_FR	64	MDCT block size

Table 3.9.5: Parameter table for the 08_06 configuration mode

Parameter	Value	Meaning
Common:		
AMP_MAX	10000	Maximum amplitude in the mu-law coding of the frame gain
AMP_NM	1024	Normalized amplitude of flattened coefficients
BAND_UPPER	1.0	Implicit bandwidth
BLIM_BITS_H	0	Bits for higher bandwidth control
BLIM_BITS_L	0	Bits for lower bandwidth control
BLIM_STEP_H	0	Number of steps of band limitation quantization
CUT_M_H	-	Minimum badwidth ratio (higher part)
CUT_M_L	-	Maximum bandwidth ratio (lower part)
FW_ALF_STEP	0.5	MA prediction coefficient for the envelope coding
GAIN_BIT	8	Number of bits for the frame gain coding
LSP_BIT0	1	Number of bits for prediction switch (LSP coding)
LSP_BIT1	5	Number of bits at the first stage (LSP coding)
LSP_BIT2	3	Number of bits per split VQ at the second stage (LSP coding)
LSP_SPLIT	3	Number of split at the second stage (LSP coding)
MA_NP	1	MA prediction order (LSP coding)
MU	100	Parameter mu in the mu-law coding of the frame power
N_PR	12	LPC order
NUM_STEP	256	Number of gain-quantization steps
STEP	AMP_MAX / (NUM_STEP-1) = 39.1	Step width of gain-quantization
SUB_AMP_MAX	4700	Maximum of subframe-gain to frame-gain ratio
SUB_AMP_NM	1024	Normalized subframe amplitude of flattened coefficients
SUB_GAIN_BIT	4	Number of bits for the subframe gain coding
SUB_NUM_STEP	16	Number of sub-gain-quantization steps
SUB_STEP	SUB_AMP_MAX/	Step width of sub-gain-quantization

	(SUB_NUM_STEP-1)= 313.3	
Long block:		
FW_CB_LEN	10	Envelope code vector length
FW_N_BIT	6	Envelope code bits
FW_N_DIV	3	Number of division in the envelope coding
N_CRB	30	Numbef of bark-scale subbands
N_FR	512	MDCT block size
PIT_N_BIT	28	Available bits for periodic peak components quantization
N_DIV_P	2	Number of division in the periodic peak components quantization
PIT_CB_SIZE	64	Size of codebook for periodic peak components quantization
PGAIN_MAX	20000.	maximum value of mu-law quantizer for gain of periodic peak components
PGAIN_MU	200	mu factor for mu-law quantization for periodic peak components
PGAIN_BITS	6	coding bits for gain of periodic peak components
PGAIN_STEP	PGAIN_MAX/ ((1<<PGAIN_BITS)-1	step size of mu-law quantizer for gain of periodic peak components
BASF_BITS	8	bits for base frequncy coding in PPC coding
N_FR_P	20	number of elements of periodic peak components
Medium block:		
FW_CB_LEN	10	Envelope code vector length
FW_N_BIT	6	Envelope code bits
FW_N_DIV	20	Number of division in the envelope coding
N_CRB	16	Numbef of bark-scale subbands
N_FR	256	MDCT block size
Short block:		
FW_CB_LEN	10	Envelope code vector length
FW_N_BIT	3	Envelope code bits
FW_N_DIV	1	Number of division in the envelope coding
N_CRB	12	Numbef of bark-scale subbands
N_FR	64	MDCT block size

Table 3.9.6: Bark-scale subband table for 24_06 configuration mode (LONG:1024)

isf	crb_tbl[isf]	isf	crb_tbl[isf]
0	3	11	83
1	8	12	99
2	13	13	119
3	18	14	145
4	24	15	180
5	30	16	228
6	36	17	298
7	43	18	406
8	51	19	596
9	60	20	1024
10	71		

Table 3.9.7: Bark-scale subband table for 24_06_960 configuration mode (LONG:960)

isf	crb_tbl[isf]	isf	crb_tbl[isf]
0	3	11	83
1	8	12	99
2	13	13	119
3	18	14	145
4	24	15	180
5	30	16	228
6	36	17	298
7	43	18	406
8	51	19	596

9	60	20	960
10	71		

Table 3.9.8: Bark-scale subband table for 24_06 configuration mode (MIDIUM:256)

isf	crb_tbl[isf]	isf	crb_tbl[isf]
0	2	8	21
1	3	9	26
2	5	10	33
3	7	11	43
4	9	12	58
5	11	13	83
6	13	14	131
7	17	15	256

Table 3.9.9: Bark-scale subband table for 24_06 configuration mode (MIDIUM:240)

isf	crb_tbl[isf]	isf	crb_tbl[isf]
0	2	8	21
1	3	9	26
2	5	10	33
3	7	11	43
4	9	12	58
5	11	13	83
6	13	14	131
7	17	15	240

Table 3.9.10: Bark-scale subband table for 24_06 configuration mode (SHORT:128)

isf	crb_tbl[isf]	isf	crb_tbl[isf]
0	2	6	14
1	4	7	16
2	6	8	22
3	8	9	36
4	10	10	60
5	12	11	128

Table 3.9.11: Bark-scale subband table for 24_06_960 configuration mode (SHORT:120)

isf	crb_tbl[isf]	isf	crb_tbl[isf]
0	2	6	14
1	4	7	16
2	6	8	22
3	8	9	36
4	10	10	60
5	12	11	120

Table 3.9.12: Bark-scale subband table for 24_06 configuration mode (SHORT:64)

isf	crb_tbl[isf]	isf	crb_tbl[isf]
0	1	6	7
1	2	7	8
2	3	8	11
3	4	9	18
4	5	10	30
5	6	11	64

Table 3.9.13: Bark-scale subband table for 24_06_960 configuration mode (SHORT:60)

isf	crb_tbl[isf]	isf	crb_tbl[isf]
0	1	6	7
1	2	7	8
2	3	8	11
3	4	9	18
4	5	10	30
5	6	11	60

Table 3.9.14: Bark-scale subband table for SCL_1 configuration mode (LONG:1024)

isf	crb_tbl[isf]	isf	crb_tbl[isf]	isf	crb_tbl[isf]
0	3	22	70	44	201
1	5	23	73	45	212
2	8	24	77	46	223
3	11	25	81	47	235
4	14	26	85	48	248
5	16	27	90	49	262
6	19	28	94	50	278
7	22	29	99	51	294
8	25	30	104	52	312
9	28	31	108	53	332
10	31	32	114	54	353
11	34	33	119	55	376
12	37	34	125	56	402
13	40	35	131	57	430
14	43	36	137	58	462
15	46	37	143	59	496
16	49	38	150	60	535
17	52	39	158	61	578
18	56	40	165	62	627
19	59	41	174	63	683
20	62	42	182		
21	66	43	191		

Table 3.9.15: Bark-scale subband table for SCL_1_960 configuration mode (LONG:960)

isf	crb_tbl[isf]	isf	crb_tbl[isf]	isf	crb_tbl[isf]
0	3	22	66	44	189
1	5	23	72	45	199
2	8	24	74	46	209
3	10	25	77	47	221
4	13	26	80	48	233
5	15	27	85	49	246
6	18	28	88	50	261
7	21	29	93	51	276
8	23	30	98	52	293
9	26	31	101	53	312
10	29	32	107	54	331
11	32	33	112	55	353

12	35	34	117	56	377
13	38	35	123	57	404
14	40	36	129	58	434
15	43	37	134	59	465
16	46	38	142	60	502
17	49	39	148	61	542
18	53	40	155	62	589
19	56	41	164	63	640
20	59	42	171		
21	62	43	179		

Table 3.9.16: Bark-scale subband table for SCL_2_960 configuration mode (LONG:960)

isf	crb_tbl[isf]	isf	crb_tbl[isf]	isf	crb_tbl[isf]
0	324	22	444	44	643
1	328	23	451	45	650
2	333	24	458	46	668
3	337	25	466	47	682
4	341	26	473	48	696
5	347	27	481	49	710
6	352	28	488	50	723
7	357	29	496	51	738
8	361	30	503	52	753
9	367	31	512	53	769
10	372	32	521	54	786
11	377	33	531	55	802
12	383	34	540	56	818
13	388	35	549	57	845
14	394	36	558	58	855
15	399	37	568	59	874
16	406	38	578	60	895
17	412	39	590	61	915
18	418	40	600	62	936
19	423	41	610	63	960
20	431	42	622		
21	437	43	632		

Table 3.9.17: Bark-scale subband table for SCL_2 configuration mode (LONG:1024)

isf	crb_tbl[isf]	isf	crb_tbl[isf]	isf	crb_tbl[isf]
0	346	22	474	44	686
1	350	23	481	45	698
2	355	24	489	46	713
3	360	25	497	47	727
4	364	26	504	48	742
5	370	27	513	49	757
6	375	28	521	50	771
7	381	29	529	51	787
8	385	30	537	52	803
9	391	31	546	53	820
10	397	32	556	54	838

11	402	33	566	55	855
12	408	34	576	56	872
13	414	35	586	57	901
14	420	36	596	58	911
15	426	37	606	59	932
16	433	38	617	60	955
17	439	39	629	61	976
18	446	40	640	62	998
19	453	41	651	63	1024
20	460	42	663		
21	466	43	674		

Table 3.9.18: Bark-scale subband table for 16_16 configuration modes (LONG)

isf	crb_tbl[isf]	isf	crb_tbl[isf]
0	6	11	96
1	12	12	110
2	18	13	127
3	25	14	147
4	31	15	171
5	38	16	201
6	46	17	239
7	54	18	288
8	63	19	360
9	73	20	512
10	83		

Table 3.9.19: Bark-scale subband table for 16_16 configuration mode (MEDIUM)

isf	crb_tbl[isf]	isf	crb_tbl[isf]
0	6	10	83
1	12	11	96
2	18	12	110
3	25	13	125
4	31	14	143
5	38	15	163
6	46	16	183
7	54	17	204
8	63	18	230
9	73	19	256

Table 3.9.20: Bark-scale subband table for 16_16 configuration mode (SHORT)

isf	crb_tbl[isf]	isf	crb_tbl[isf]
0	2	5	12
1	3	6	19
2	5	7	29
3	7	8	45
4	9	9	64

Table 3.9.21: Bark-scale subband table for 08_06 configuration mode (LONG)

isf	crb_tbl[isf]	isf	crb_tbl[isf]
0	7	15	142
1	15	16	154

2	22	17	168
3	30	18	183
4	38	19	199
5	46	20	217
6	54	21	236
7	62	22	257
8	70	23	381
9	79	24	308
10	88	25	338
11	98	26	373
12	108	27	413
13	119	28	459
14	130	29	512

Table 3.9.22: Bark-scale subband table for 08_06 configuration mode (MEDIUM)

isf	crb_tbl[isf]	isf	crb_tbl[isf]
0	6	10	75
1	11	11	84
2	17	12	95
3	23	13	108
4	29	14	123
5	35	15	141
6	42	16	161
7	49	17	186
8	57	18	217
9	65	19	256

Table 3.9.23: Bark-scale subband table for 08_06 configuration mode (SHORT)

isf	crb_tbl[isf]
0	3
1	6
2	9
3	12
4	16
5	21
6	27
7	35
8	47
9	64

Table 3.9.24: Values of isp[] for 24_06, SCL_1, SCL_1_960, SCL_2 and SCL_2_960 configuration modes

split_num	isp[split_num]
0	0
1	5
2	14
3	20

Table 3.9.25: Values of isp[] for 16_16 configuration mode

split_num	isp[split_num]
0	0
1	5
2	10
3	16

Table 3.9.26: Values of isp[] for 08_06 configuration mode

split_num	isp[split_num]
0	0
1	3
2	8
3	12

3.11 Filterbank and Block Switching

3.11.1 Tool Description

The time/frequency representation of the signal is mapped onto the time domain by feeding it into the filterbank module. This module consists of an inverse modified discrete cosine transform (IMDCT), and a window and an overlap-add function. In order to adapt the time/frequency resolution of the filterbank to the characteristics of the input signal, a block switching tool is also adopted. N represents the window length, where N is a function of the **window_sequence**, see 3.3.3. For each channel, the N/2 time-frequency values $X_{i,k}$ are transformed into the N time domain values $x_{i,n}$ via the IMDCT. After applying the window function, for each channel, the first half of the $z_{i,n}$ sequence is added to the second half of the previous block windowed sequence $z_{(i-1),n}$ to reconstruct the output samples for each channel $out_{i,n}$.

3.11.2 Definitions

The syntax elements for the filterbank are specified in the raw data stream for the **single_channel_element** (see 1.3, Table 6.9)**, channel_pair_element** (see 1.3, Table 6.10)**,** and the **coupling_channel** (see 1.3, Table 6.18). They consist of the control information **window_sequence** and **window_shape**.

window_sequence 2 bit indicating which window sequence (i.e. block size) is used (see 1.3, Table 6.11).

window_shape 1 bit indicating which window function is selected (see 1.3, Table 6.11).

Table 3.3 shows the four **window_sequences** (ONLY_LONG_SEQUENCE, LONG_START_SEQUENCE, EIGHT_SHORT_SEQUENCE, LONG_STOP_SEQUENCE).

3.11.3 Decoding Process

3.11.3.1 IMDCT

The analytical expression of the IMDCT is:

$$x_{i,n} = \frac{2}{N}\sum_{k=0}^{\frac{N}{2}-1} spec[i][k]\cos\left(\frac{2\pi}{N}\left(n+n_0\right)\left(k+\frac{1}{2}\right)\right) \text{ for } \quad 0 \le n < N$$

where:

n = sample index

i = window index

k = spectral coefficient index

N = window length based on the window_ sequence value

n_0 = (N / 2 + 1) / 2

The synthesis window length N for the inverse transform is a function of the syntax element **window_sequence** and the algorithmic context. It is defined as follows:

AAC-derived coder

$$N = \begin{cases} 2048, & \text{if ONLY\_LONG\_SEQUENCE (0x0)} \\ 2048, & \text{if LONG\_START\_SEQUENCE (0x1)} \\ 256, & \text{if EIGHT\_SHORT\_SEQUENCE (0x2), (8 times)} \\ 2048, & \text{if LONG\_STOP\_SEQUENCE (0x3)} \end{cases}$$

AAC-derived scalable coder with core coder frame sizes of multiples of 10ms

$$N = \begin{cases} 1920, & \text{if ONLY\_LONG\_SEQUENCE (0x0)} \\ 1920, & \text{if LONG\_START\_SEQUENCE (0x1)} \\ 240, & \text{if EIGHT\_SHORT\_SEQUENCE (0x2), (8 times)} \\ 1920, & \text{if LONG\_STOP\_SEQUENCE (0x3)} \end{cases}$$

The meaningful block transitions are as follows:

from ONLY_LONG_SEQUENCE to	{ ONLY_LONG_SEQUENCE LONG_START_SEQUENCE
from LONG_START_SEQUENCE to	{ EIGHT_SHORT_SEQUENCE LONG_STOP_SEQUENCE
from LONG_STOP_SEQUENCE to	{ ONLY_LONG_SEQUENCE LONG_START_SEQUENCE
from EIGHT_SHORT_SEQUENCE to	{ EIGHT_SHORT_SEQUENCE LONG_STOP_SEQUENCE

In addition to the meaningful block transitions the following transitions are possible:

from ONLY_LONG_SEQUENCE to	{ EIGHT_SHORT_SEQUENCE LONG_STOP_SEQUENCE
from LONG_START_SEQUENCE to	{ ONLY_LONG_SEQUENCE LONG_START_SEQUENCE
from LONG_STOP_SEQUENCE to	{ EIGHT_SHORT_SEQUENCE LONG_STOP_SEQUENCE
from EIGHT_SHORT_SEQUENCE to	{ ONLY_LONG_SEQUENCE LONG_START_SEQUENCE

This will still result in a reasonably smooth transition from one block to the next.

3.11.3.2 Windowing and block switching

Depending on the **window_sequence** and **window_shape** element different transform windows are used. A combination of the window halves described as follows offers all possible window_sequences.

For **window_shape** == 1, the window coefficients are given by the Kaiser - Bessel derived (KBD) window as follows:

$$W_{KBD\_LEFT,N}(n) = \sqrt{\frac{\sum_{p=0}^{n}\left[W'(n,\alpha)\right]}{\sum_{p=0}^{N/2}\left[W'(p,\alpha)\right]}} \quad \text{for} \quad 0 \le n < \frac{N}{2}$$

$$W_{KBD\_RIGHT,N}(n) = \sqrt{\frac{\sum_{p=0}^{N-n}\left[W'(n,\alpha)\right]}{\sum_{p=0}^{N/2}\left[W'(p,\alpha)\right]}} \quad \text{for} \quad \frac{N}{2} \le n < N$$

where:

W', Kaiser - Bessel kernel window function, see also [5], is defined as follows:

$$W'(n,\alpha) = \frac{I_0\left[\pi\alpha\sqrt{1.0 - \left(\frac{n - N/4}{N/4}\right)^2}\right]}{I_0\left[\pi\alpha\right]} \qquad \text{for } 0 \le n \le \frac{N}{2}$$

$$I_0\left[x\right] = \sum_{k=0}^{\infty}\left[\frac{\left(\frac{x}{2}\right)^k}{k!}\right]^2$$

$$\alpha = \text{kernel window alpha factor, } \alpha = \begin{cases} 4 \text{ for N} = 2048\ (1920) \\ 6 \text{ for N} = 256\ (240) \end{cases}$$

Otherwise, for **window_shape** == 0, a sine window is employed as follows:

$$W_{SIN\_LEFT,N}(n) = \sin\left(\frac{\pi}{N}\left(n + \frac{1}{2}\right)\right) \qquad \text{for} \quad 0 \le n < \frac{N}{2}$$

$$W_{SIN\_RIGHT,N}(n) = \sin\left(\frac{\pi}{N}\left(n + \frac{1}{2}\right)\right) \qquad \text{for} \quad \frac{N}{2} \le n < N$$

The window length N can be 2048 (1920) or 256 (240) for the KBD and the sine window. How to obtain the possible window sequences is explained in the parts a)-d) of this clause. All four window_sequences described below have a total length of 2048 samples.

For all kinds of window_sequences the window_shape of the left half of the first transform window is determined by the window shape of the previous block. The following formula expresses this fact:

$$W_{LEFT,N}(n) = \begin{cases} W_{KBD\_LEFT,N}(n), \text{ if } & window\_shape\_previous\_block == 1 \\ W_{SIN\_LEFT,N}(n), \text{ if } & window\_shape\_previous\_block == 0 \end{cases}$$

where:

window_shape_previous_block: **window_shape** of the previous block (i-1).

For the first block of the bitstream to be decoded the **window_shape** of the left and right half of the window are identical.

a) ONLY_LONG_SEQUENCE:

The **window_sequence** == ONLY_LONG_SEQUENCE is equal to one LONG_WINDOW (see Table 3.3) with a total window length N_l of 2048 (1920).

For **window_shape**==1 the window for ONLY_LONG_SEQUENCE is given as follows:

$$w(n)=\begin{cases} W_{LEFT,N\_l}(n), & \text{for } 0 \le n < N\_l/2 \\ W_{KBD\_RIGHT,N\_l}(n), & \text{for } N\_l/2 \le n < N\_l \end{cases}$$

If **window_shape**==0 the window for ONLY_LONG_SEQUENCE can be described as follows:

$$w(n)=\begin{cases} W_{LEFT,N\_l}(n), & \text{for } 0 \le n < N\_l/2 \\ W_{SIN\_RIGHT,N\_l}(n), & \text{for } N\_l/2 \le n < N\_l \end{cases}$$

After windowing, the time domain values ($z_{i,n}$) can be expressed as:

$$z_{i,n} = w(n) \cdot x_{i,n};$$

b) LONG_START_SEQUENCE:

The LONG_START_SEQUENCE is needed to obtain a correct overlap and add for a block transition from a ONLY_LONG_SEQUENCE to a EIGHT_SHORT_SEQUENCE.

Window length N_l and N_s is set to 2048 (1920) and 256 (240) respectively.

If **window_shape**==1 the window for LONG_START_SEQUENCE is given as follows:

$$w(n)=\begin{cases} W_{LEFT,N\_l}(n), & \text{for } 0 \le n < N\_l/2 \\ 1.0, & \text{for } N\_l/2 \le n < \frac{3N\_l-N\_s}{4} \\ W_{KBD\_RIGHT,N\_s}(n+\frac{N\_s}{2}-\frac{3N\_l-N\_s}{4}), & \text{for } \frac{3N\_l-N\_s}{4} \le n < \frac{3N\_l+N\_s}{4} \\ 0.0, & \text{for } \frac{3N\_l+N\_s}{4} \le n < N\_l \end{cases}$$

If **window_shape**==0 the window for LONG_START_SEQUENCE looks like:

$$w(n)=\begin{cases} W_{LEFT,N\_l}(n), & \text{for } 0 \le n < N\_l/2 \\ 1.0, & \text{for } N\_l/2 \le n < \frac{3N\_l-N\_s}{4} \\ W_{SIN\_RIGHT,N\_s}(n+\frac{N\_s}{2}-\frac{3N\_l-N\_s}{4}), & \text{for } \frac{3N\_l-N\_s}{4} \le n < \frac{3N\_l+N\_s}{4} \\ 0.0, & \text{for } \frac{3N\_l+N\_s}{4} \le n < N\_l \end{cases}$$

The windowed time-domain values can be calculated with the formula explained in a).

c) EIGHT_SHORT

The **window_sequence** == EIGHT_SHORT comprises eight overlapped and added SHORT_WINDOWs (see Table 3.3) with a length N_s of 256 (240) each. The total length of the window_sequence together with leading and following zeros is 2048 (1920). Each of the eight short blocks are windowed separately first. The short block number is indexed with the variable j = 0,…, M-1 (M=N_l/N_s).
The **window_shape** of the previous block influences the first of the eight short blocks ($W_0(n)$) only.
If **window_shape**==1 the window functions can be given as follows:

$$W_0(n) = \begin{cases} W_{LEFT,N\_s}(n), & \text{for } 0 \le n < N\_s/2 \\ W_{KBD\_RIGHT,N\_s}(n), & \text{for } N\_s/2 \le n < N\_s \end{cases}$$

$$W_{1-(M-1)}(n) = \begin{cases} W_{LEFT,N\_s}(n), & \text{for } 0 \le n < N\_s/2 \\ W_{KBD\_RIGHT,N\_s}(n), & \text{for } N\_s/2 \le n < N\_s \end{cases}$$

Otherwise, if **window_shape**==0, the window functions can be described as:

$$W_0(n) = \begin{cases} W_{LEFT,N\_s}(n), & \text{for } 0 \le n < N\_s/2 \\ W_{SIN\_RIGHT,N\_s}(n), & \text{for } N\_s/2 \le n < N\_s \end{cases}$$

$$W_{1-(M-1)}(n) = \begin{cases} W_{SIN\_LEFT,N\_s}(n), & \text{for } 0 \le n < N\_s/2 \\ W_{SIN\_RIGHT,N\_s}(n), & \text{for } N\_s/2 \le n < N\_s \end{cases}$$

The overlap and add between the EIGHT_SHORT **window_sequence** resulting in the windowed time domain values $z_{i,n}$ is described as follows:

$$z_{i,n} = \begin{cases} 0, & \text{for } 0 \le n < \frac{N\_l-N\_s}{4} \\ x_{0,n-\frac{N\_l-N\_s}{4}} \cdot W_0(n-\frac{N\_l-N\_s}{4}), & \text{for } \frac{N\_l-N\_s}{4} \le n < \frac{N\_l+N\_s}{4} \\ x_{j-1,n-\frac{N\_l+(2j-3)N\_s}{4}} \cdot W_{j-1}(n-\frac{N\_l+(2j-3)N\_s}{4}) + x_{j,n-\frac{N\_l+(2j-1)N\_s}{4}} \cdot W_j(n-\frac{N\_l+(2j-1)N\_s}{4}), & \\ & \text{for } 1 \le j < M,\ \frac{N\_l+(2j-1)N\_s}{4} \le n < \frac{N\_l+(2j+1)N\_s}{4} \\ x_{M-1,n-\frac{N\_l+(2M-3)N\_s}{4}} \cdot W_{M-1}(n-\frac{N\_l+(2M-3)N\_s}{4}), & \\ & \text{for } \frac{N\_l+(2M-1)N\_s}{4} \le n < \frac{N\_l+(2M+1)N\_s}{4} \\ 0, & \text{for } \frac{N\_l+(2M+1)N\_s}{4} \le n < N\_l \end{cases}$$

d) LONG_STOP_SEQUENCE

This window_sequence is needed to switch from a EIGHT_SHORT_SEQUENCE back to a ONLY_LONG_SEQUENCE.

If **window_shape**==1 the window for LONG_STOP_SEQUENCE is given as follows:

$$W(n) = \begin{cases} 0.0, & \text{for } 0 \le n < \frac{N\_l-N\_s}{4} \\ W_{LEFT,N\_s}(n-\frac{N\_l-N\_s}{4}), & \text{for } \frac{N\_l-N\_s}{4} \le n < \frac{N\_l+N\_s}{4} \\ 1.0, & \text{for } \frac{N\_l+N\_s}{4} \le n < N\_l/2 \\ W_{KBD\_RIGHT,N\_l}(n), & \text{for } N\_l/2 \le n < N\_l \end{cases}$$

If **window_shape**==0 the window for LONG_START_SEQUENCE is determined by:

$$w(n) = \begin{cases} 0.0, & \text{for } 0 \le n < \frac{N\_l - N\_s}{4} \\ W_{LEFT,N\_s}(n - \frac{N\_l - N\_s}{4}), & \text{for } \frac{N\_l - N\_s}{4} \le n < \frac{N\_l + N\_s}{4} \\ 1.0, & \text{for } \frac{N\_l + N\_s}{4} \le n < N\_l / 2 \\ W_{SIN\_RIGHT,N\_l}(n), & \text{for } N\_l / 2 \le n < N\_l \end{cases}$$

The windowed time domain values can be calculated with the formula explained in a).

3.11.3.3 Overlapping and adding with previous window sequence

Besides the overlap and add within the EIGHT_SHORT **window_sequence** the first (left) half of every **window_sequence** is overlapped and added with the second (right) half of the previous **window_sequence** resulting in the final time domain values $out_{i,n}$. The mathematic expression for this operation can be described as follows. It is valid for all four possible window_sequences.

$$out_{i,n} = z_{i,n} + z_{i-1,n+\frac{N}{2}}; \qquad \text{for } 0 \le n < \frac{N}{2}, \quad N = 2048\ (1920)$$

3.11.4 Three block type mode

Three block type mode is an extention of the filterbank and block switching tool. This mode is enabled when interleaved vector quantization and spectrum and normalization tools are used. In this extention, block types with medium length are used additionally.

3.11.4.1 Syntax elements

The syntax elements for extended filterbank are specified in the raw data stream element_stream_vq(). Differences from non-extended mode are:

- In this mode the syntax element **window_shape** is not transmitted but set to zero.
- The syntax element **window_sequence** is indicated by 4 bits.

3.11.4.2 Block sizes

Window length N of non-extended filterbank are set to 1024 for long blocks, and 128 for short blocks. However, for extended mode, N is set to 2*N_FR for each frame length (see tables from ? to ?). Window length for long, medium, and short block is represented as N_L, N_M, N_S respectively.

For extended mode, three block lengths, N_L, N_M, and N_S is necessary for filterbank settings. N_L, N_M, N_S is set to 2*N_FR for long, medium, short frame length respectively (see tables from ? to ?).

3.11.4.3 IMDCT

In extended mode, medium-length blocks are used. So nine window sequences are possible, and the synthesis window length N for the IMDCT as a function of **window_sequence** is defined as follows:

$$N = \begin{cases} N_L, & \text{if ONLY\_LONG\_SEQUENCE (0x0)} \\ N_L, & \text{if LONG\_SHORT\_SEQUENCE (0x1)} \\ N_S, & \text{if ONLY\_SHORT\_SEQUENCE (0x2),} \quad (N_L / N_S \text{ times}) \\ N_L, & \text{if SHORT\_LONG\_SEQUENCE (0x3)} \\ N_M, & \text{if SHORT\_MEDIUM\_SEQUENCE (0x4),} \; (N_L / N_M \text{ times}) \\ N_L, & \text{if MEDIUM\_LONG\_SEQUENCE (0x5)} \\ N_L, & \text{if LONG\_MEDIUM\_SEQUENCE (0x6)} \\ N_M, & \text{if MEDIUM\_SHORT\_SEQUENCE (0x7),} \; (N_L / N_M \text{ times}) \\ N_M, & \text{if ONLY\_MEDIUM\_SEQUENCE (0x8),} \quad (N_L / N_M \text{ times}) \end{cases}$$

3.11.4.4 Window sequences

The meaningful block length transitions out of all possible transitions are marked as ‘o’ in following diagram.

window sequence from	OL	LS	OS	SL	SM	ML	LM	MS	OM
ONLY_LONG_SEQUENCE	o	o					o		
LONG_SHORT_SEQUENCE			o	o	o				
ONLY_SHORT_SEQUENCE			o	o	o				
SHORT_LONG_SEQUENCE	o	o					o		
SHORT_MEDIUM_SEQUENCE						o		o	o
MEDIUM_LONG_SEQUENCE	o	o					o		
LONG_MEDIUM_SEQUENCE						o		o	o
MEDIUM_SHORT_SEQUENCE			o	o	o				
ONLY_MEDIUM_SEQUENCE						o		o	o

3.11.4.5 Windowing and block switching

a) If window_sequence == ONLY_LONG_SEQUENCE, windowing and block switching is the same as non-extended mode, total length N is set to N_L.

b) If window_sequence == LONG_SHORT_SEQUENCE, windowing and block switching is the same as LONG_START_SEQUENCE of non-extended mode.

c) If window_sequence == ONLY_SHORT_SEQUENCE, windowing and block switching is the same as EIGHT_SHORT_SEQUENCE of non-extended mode. Window length N_l is set to N_L, N_s is set to N_S.

d) If window_sequence == SHORT_LONG_SEQUENCE, windowing and block switching is the same as LONG_STOP_SEQUENCE, window length N_l is set to N_L, N_s is set to N_S.

e) If window_sequence == SHORT_MEDIUM_SEQUENCE, windowing and block switching is the same as ONLY_SHORT_SEQUENCE, window length N_l is set to N_L, N_s is set to N_M, except for the first window function W_0 is given from equation used in SHORT_LONG_WINDOW, N_l=N_M, N_s=N_S.

f) If window_sequence == MEDIUM_LONG_SEQUENCE, windowing and block switching is the same as SHORT_LONG_SEQUENCE, window length N_l is set to N_L, N_s is set to N_M.

g) If window_sequence == LONG_MEDIUM_SEQUENCE, windowing and block switching is the same as LONG_SHORT_SEQUENCE, window length N_l is set to N_L, N_s is set to N_M.

h) If window_sequence == MEDIUM_SHORT_SEQUENCE, windowing and block switching is the same as ONLY_SHORT_SEQUENCE, N_l = N_L, N_s = N_M, except for the last window function W_{M-1} ($M=N_L/N_M$) is given from equation used in LONG_SHORT_WINDOW, N_l = N_M, N_s=N_S.

I) If window_sequence == ONLY_MEDIUM_SEQUENCE, windowing and block switching is the same as ONLY_SHORT_SEQUENCE, window length N_l is set to NL, N_s is set to NM.

1.4.1 Overlapping and adding with previous window sequence

For this mode, window length for overlapping and adding is set to N_L, instead of 2048.

3.12 Gain Control

3.12.1 Tool description

The gain control tool is made up of several gain compensators and overlap/add processing stages, and an IPQF (Inverse Polyphase Quadrature Filter) stage. This tool receives non-overlapped signal sequences provided by the IMDCT stages, window_sequence and gain_control_data, and then reproduces the output PCM data. The block diagram for the gain control tool is shown in Figure 00.11.

Due to the characteristics of the PQF filterbank, the order of the MDCT coefficients in each even PQF band must be reversed. This is done by reversing the spectral order of the MDCT coefficients, i.e. exchanging the higher frequency MDCT coefficients with the lower frequency MDCT coefficients.

If the gain control tool is used, the configuration of the filter bank tool is changed as follows. In the case of an EIGHT_SHORT_SEQUENCE window_sequence, the number of coefficients for the IMDCT is 32 instead of 128 and eight IMDCTs are carried out. In the case of other window_sequence values, the number of coefficients for the IMDCT is 256 instead of 1024 and one IMDCT is performed. In all cases, the filter bank tool outputs a total of 2048 non-overlapped values per frame. These values are supplied to the gain control tool as $U_{W,B}(j)$ defined in 11.3.3.

The IPQF combines four uniform frequency bands and produces a decoded time domain output signal. The aliasing components introduced by the PQF in the encoder are cancelled by the IPQF.

The gain values for each band can be controlled independently except for the lowest frequency band. The step size of gain control is 2 ^ n where n is an integer.

The gain control tool outputs a time signal sequence which is $AS(n)$ defined in 11.3.4.

3.12.2 Definitions

gain control data	side information indicating the gain values and the positions used for the gain change.
IPQF band	each split band of IPQF.
adjust_num	3-bit field indicating the number of gain changes for each IPQF band. The maximum number of gain changes is seven (see 1.3, Table 6.23).
max_band	2-bit field indicating the number of IPQF bands which contain spectral data counting from the lowest IPQF band to higher IPQF bands. The number of IPQF bands which contain spectral data is **max_band** + 1 (see 1.3, Table 6.23).
alevcode	4-bit field indicating the gain value for one gain change (see 1.3, Table 6.23).
aloccode	2-, 4-, or 5-bit field indicating the position for one gain change. The length of this data varies depending on the window sequence (see 1.3, Table 6.23).

3.12.3 Decoding process

The following four processes are required for decoding.

(1) Gain control data decoding

(2) Gain control function setting
(3) Gain control windowing and overlapping
(4) Synthesis filter

3.12.3.1 Gain control data decoding

Gain control data are reconstructed as follows.

(1)

$$NAD_{W,B} = \textbf{adjust\_num}[B][W]$$

(2)

$$ALOC_{W,B}(m) = AdjLoc(\textbf{aloccode}[B][W][m-1]), \quad 1 \le m \le NAD_{W,B}$$

$$ALEV_{W,B}(m) = 2^{AdjLev(\textbf{alevcode}[B][W][m-1])}, \quad 1 \le m \le NAD_{W,B}$$

(3)

$$ALOC_{W,B}(0) = 0$$

$$ALEV_{W,B}(0) = \begin{cases} 1, & if\ NAD_{W,B} == 0 \\ ALEV_{W,B}(1), & otherwise \end{cases}$$

(4)

$$ALOC_{W,B}(NAD_{W,B}+1) = \begin{cases} 256, \quad W == 0 & if\ \text{ONLY\_LONG\_SEQUENCE} \\ \left.\begin{matrix} 112, \quad W == 0 \\ 32, \quad W == 1 \end{matrix}\right\} & if\ \text{LONG\_START\_SEQUENCE} \\ 32, \quad 0 \le W \le 7 & if\ \text{EIGHT\_SHORT\_SEQUENCE} \\ \left.\begin{matrix} 112, \quad W == 0 \\ 256, \quad W == 1 \end{matrix}\right\} & if\ \text{LONG\_STOP\_SEQUENCE} \end{cases}$$

$$ALEV_{W,B}(NAD_{W,B}+1) = 1$$

where

$NAD_{W,B}$:	Gain Control Information Number, an integer
$ALOC_{W,B}(m)$:	Gain Control Location, an integer
$ALEV_{W,B}(m)$:	Gain Control Level, an integer-valued real number
B:	Band ID, an integer from 1 to 3
W:	Window ID, an integer from 0 to 7
m:	an integer

aloccode[*B*][*W*][*m*] must be set so that $\{ALOC_{W,B}(m)\}$ satisfies the following conditions.

$$ALOC_{W,B}(m_1) < ALOC_{W,B}(m_2), \quad 1 \le m_1 < m_2 \le NAD_{W,B}+1$$

In cases of LONG_START_SEQUENCE and LONG_STOP_SEQUENCE, the values 14 and 15 of **aloccode**[*B*][*0*][*m*] are invalid. *AdjLoc*() is defined in Table 00.11. *AdjLev*() is defined in Table 00.22.

3.12.3.2 Gain control function setting

The Gain control function is obtained as follows.

(1)

$$M_{W,B,j} = Max\{m: ALOC_{W,B}(m) \le j\},$$

$$\begin{matrix} 0 \le j \le 255, \quad W == 0 & if\ \text{ONLY\_LONG\_SEQUENCE} \\ \left.\begin{matrix} 0 \le j \le 111, \quad W == 0 \\ 0 \le j \le 31, \quad W == 1 \end{matrix}\right\} & if\ \text{LONG\_START\_SEQUENCE} \\ 0 \le j \le 31, \quad 0 \le W \le 7 & if\ \text{EIGHT\_SHORT\_SEQUENCE} \end{matrix}$$

$$\left.\begin{array}{ll} 0 \le j \le 111, & W == 0 \\ 0 \le j \le 255, & W == 1 \end{array}\right\} \quad \textit{if}\ \text{LONG\_STOP\_SEQUENCE}$$

(2)

$$FMD_{W,B}(j) = \begin{cases} Inter\begin{pmatrix} ALEV_{W,B}\left(M_{W,B,j}\right), \\ ALEV_{W,B}\left(M_{W,B,j}+1\right), \\ j - ALOC_{W,B}\left(M_{W,B,j}\right) \end{pmatrix}, \\ \qquad\qquad \textit{if}\ ALOC_{W,B}\left(M_{W,B,j}\right) \le j \le ALOC_{W,B}\left(M_{W,B,j}\right) + 7 \\ ALEV_{W,B}\left(M_{W,B,j}+1\right), \quad otherwise \end{cases}$$

(3)

if ONLY_LONG_SEQUENCE

$$GMF_{0,B}(j) = \begin{cases} ALEV_{0,B}(0) \times PFMD_B(j), & 0 \le j \le 255 \\ FMD_{0,B}(j-256), & 256 \le j \le 511 \end{cases}$$

$$PFMD_B(j) = FMD_{0,B}(j), \quad 0 \le j \le 255$$

if LONG_START_SEQUENCE

$$GMF_{0,B}(j) = \begin{cases} ALEV_{0,B}(0) \times ALEV_{1,B}(0) \times PFMD_B(j), & 0 \le j \le 255 \\ ALEV_{1,B}(0) \times FMD_{0,B}(j-256), & 256 \le j \le 367 \\ FMD_{1,B}(j-368), & 368 \le j \le 399 \\ 1, & 400 \le j \le 511 \end{cases}$$

$$PFMD_B(j) = FMD_{1,B}(j), \quad 0 \le j \le 31$$

if EIGHT_SHORT_SEQUENCE

$$GMF_{W,B}(j) = \begin{cases} ALEV_{W,B}(0) \times PFMD_B(j), & W == 0, \quad 0 \le j \le 31 \\ ALEV_{W,B}(0) \times FMD_{W-1,B}(j), & 1 \le W \le 7, \quad 0 \le j \le 31 \\ FMD_{W,B}(j-32), & 0 \le W \le 7, \quad 32 \le j \le 63 \end{cases}$$

$$PFMD_B(j) = FMD_{7,B}(j), \quad 0 \le j \le 31$$

if LONG_STOP_SEQUENCE

$$GMF_{0,B}(j) = \begin{cases} 1, & 0 \le j \le 111 \\ ALEV_{0,B}(0) \times ALEV_{1,B}(0) \times PFMD_B(j-112), & 112 \le j \le 143 \\ ALEV_{1,B}(0) \times FMD_{0,B}(j-144), & 144 \le j \le 255 \\ FMD_{1,B}(j-256), & 256 \le j \le 511 \end{cases}$$

$$PFMD_B(j) = FMD_{1,B}(j), \quad 0 \le j \le 255$$

(4)

$$AD_{W,B}(j) = \frac{1}{GMF_{W,B}(j)},$$

$$\begin{array}{lll} 0 \le j \le 511, & W == 0 & \textit{if}\ \text{ONLY\_LONG\_SEQUENCE} \\ 0 \le j \le 511, & W == 0 & \textit{if}\ \text{LONG\_START\_SEQUENCE} \\ 0 \le j \le 63, & 0 \le W \le 7 & \textit{if}\ \text{EIGHT\_SHORT\_SEQUENCE} \\ 0 \le j \le 511, & W == 0 & \textit{if}\ \text{LONG\_STOP\_SEQUENCE} \end{array}$$

where

$FMD_{W,B}(j)$: Fragment Modification Function, a real number

$PFMD_B(j)$: Fragment Modification Function of previous frame, a real number

$GMF_{W,B}(j)$: Gain Modification Function, a real number

$AD_{W,B}(j)$: Gain Control Function, a real number

$ALOC_{W,B}(m)$: Gain Control Location defined in 0, an integer

$ALEV_{W,B}(m)$: Gain Control Level defined in 0, an integer-valued real number

B: Band ID, an integer from 1 to 3

W: Window ID, an integer from 0 to 7

$M_{W,B,j}$: an integer

m: an integer

and

$$Inter(a,b,\ j) = 2^{\frac{(8-j)\log_2(a)+j\log_2(b)}{8}}$$

Note that the initial value of $PFMD_B(j)$ must be set 1.0.

3.12.3.3 Gain control windowing and overlapping

Band Sample Data are obtained through the processes (1) to (2) shown below.

(1) Gain Control Windowing

if B=0

$$T_{W,B}(j) = U_{W,B}(j),$$

$$\begin{array}{lll} 0 \le j \le 511, & W == 0 & if \text{ ONLY\_LONG\_SEQUENCE} \\ 0 \le j \le 511, & W == 0 & if \text{ LONG\_START\_SEQUENCE} \\ 0 \le j \le 63, & 0 \le W \le 7 & if \text{ EIGHT\_SHORT\_SEQUENCE} \\ 0 \le j \le 511, & W == 0 & if \text{ LONG\_STOP\_SEQUENCE} \end{array}$$

else

$$T_{W,B}(j) = AD_{W,B}(j) \times U_{W,B}(j),$$

$$\begin{array}{lll} 0 \le j \le 511, & W == 0 & if \text{ ONLY\_LONG\_SEQUENCE} \\ 0 \le j \le 511, & W == 0 & if \text{ LONG\_START\_SEQUENCE} \\ 0 \le j \le 63, & 0 \le W \le 7 & if \text{ EIGHT\_SHORT\_SEQUENCE} \\ 0 \le j \le 511, & W == 0 & if \text{ LONG\_STOP\_SEQUENCE} \end{array}$$

(2) Overlapping

if ONLY_LONG_SEQUENCE

$$V_B(j) = PT_B(j) + T_{0,B}(j), \quad 0 \le j \le 255$$

$$PT_B(j) = T_{0,B}(j+256), \quad 0 \le j \le 255$$

if LONG_START_SEQUENCE

$$V_B(j) = PT_B(j) + T_{0,B}(j), \quad 0 \le j \le 255$$

$$V_B(j+256) = T_{0,B}(j+256), \quad 0 \le j \le 111$$

$$PT_B(j) = T_{0,B}(j+368), \quad 0 \le j \le 31$$

if EIGHT_SHORT_SEQUENCE

$$V_B(j) = PT_B(j) + T_{W,B}(j), \quad W == 0, \quad 0 \le j \le 31$$

$$V_B(32W+j) = T_{W-1,B}(j+32) + T_{W,B}(j), \quad 1 \le W \le 7, \quad 0 \le j \le 31$$

$$PT_B(j) = T_{w,B}(j+32), \quad W == 7, \quad 0 \le j \le 31$$

if LONG_STOP_SEQUENCE

$$V_B(j) = PT_B(j) + T_{0,B}(j+112), \quad 0 \le j \le 31$$

$$V_B(j+32) = T_{0,B}(j+144), \quad 0 \le j \le 111$$

$$PT_B(j) = T_{0,B}(j+256), \quad 0 \le j \le 255$$

where

$U_{W,B}(j)$: Band Spectrum Data, a real number

$T_{W,B}(j)$: Gain Controlled Block Sample Data, a real number

$PT_B(j)$: Gain Controlled Block Sample Data of previous frame, a real number

$V_B(j)$: Band Sample Data, a real number

$AD_{W,B}(j)$: Gain Control Function defined in 0, a real number

B: Band ID, an integer from 0 to 3
W: Window ID, an integer from 0 to 7
j: an integer

Note that the initial value of $PT_B(j)$ must be set 0.0.

3.12.3.4 Synthesis filter

Audio Sample Data are obtained from the following equations.

(1)

$$\tilde{V}_B(j) = \begin{cases} V_B(k), & if\ j == 4k, \\ 0, & else \end{cases} \quad 0 \le B \le 3$$

(2)

$$Q_B(j) = Q(j) \times \cos\left(\frac{(2B+1)(2j-3)\pi}{16}\right), \quad 0 \le j \le 95, \quad 0 \le B \le 3$$

(3)

$$AS(n) = \sum_{B=0}^{3} \sum_{j=0}^{95} Q_B(j) \times \tilde{V}_B(n-j)$$

where

$AS(n)$: Audio Sample Data

$V_B(n)$: Band Sample Data defined in 0, a real number

$\tilde{V}_B(j)$: Interpolated Band Sample Data, a real number

$Q_B(j)$: Synthesis Filter Coefficients, a real number

$Q(j)$: Prototype Coefficients given below, a real number

B: Band ID, an integer from 0 to 3
W: Window ID, an integer from 0 to 7
n: an integer
j: an integer
k: an integer

The values of $Q(0)$ to $Q(47)$ are shown in Table 00.33. The values of $Q(48)$ to $Q(95)$ are obtained from the following equation.

$$Q(j) = Q(95-j), \quad 48 \le j \le 95$$

3.12.4 Diagrams

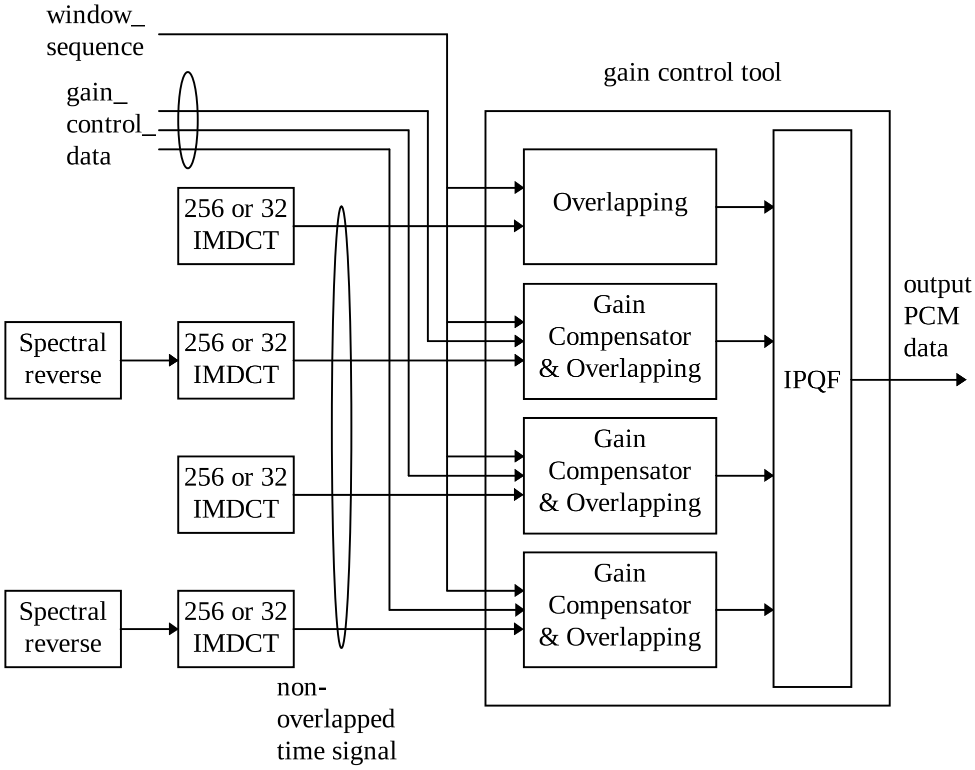


Figure 0.1 – Block diagram of gain control tool

3.12.5 Tables

Table 0.1 – *AdjLoc()*

AC	*AdjLoc(AC)*	*AC*	*AdjLoc(AC)*
0	0	16	128
1	8	17	136
2	16	18	144
3	24	19	152
4	32	20	160
5	40	21	168
6	48	22	176
7	56	23	184
8	64	24	192
9	72	25	200
10	80	26	208
11	88	27	216
12	96	28	224
13	104	29	232
14	112	30	240
15	120	31	248

Table 0.2 – *AdjLev()*

AV	*AdjLev(AV)*
0	-4
1	-3
2	-2
3	-1
4	0

5	1
6	2
7	3
8	4
9	5
10	6
11	7
12	8
13	9
14	10
15	11

Table 0.3 – *Q()*

j	*Q(j)*	*j*	*Q(j)*
0	9.7655291007575512E-05	24	-2.2656858741499447E-02
1	1.3809589379038567E-04	25	-6.8031113858963354E-03
2	9.8400749256623534E-05	26	1.5085400948280744E-02
3	-8.6671544782335723E-05	27	3.9750993388272739E-02
4	-4.6217998911921346E-04	28	6.2445363629436743E-02
5	-1.0211814095158174E-03	29	7.7622327748721326E-02
6	-1.6772149340010668E-03	30	7.9968338496132926E-02
7	-2.2533338951411081E-03	31	6.5615493068475583E-02
8	-2.4987888343213967E-03	32	3.3313658300882690E-02
9	-2.1390815966761882E-03	33	-1.4691563058190206E-02
10	-9.5595397454597772E-04	34	-7.2307890475334147E-02
11	1.1172111530118943E-03	35	-1.2993222541703875E-01
12	3.9091309127348584E-03	36	-1.7551641029040532E-01
13	6.9635703420118673E-03	37	-1.9626543957670528E-01
14	9.5595442159478339E-03	38	-1.8073330670215029E-01
15	1.0815766540021360E-02	39	-1.2097653136035738E-01
16	9.8770514991715300E-03	40	-1.4377370758549035E-02
17	6.1562567291327357E-03	41	1.3522730742860303E-01
18	-4.1793946063629710E-04	42	3.1737852699301633E-01
19	-9.2128743097707640E-03	43	5.1590021798482233E-01
20	-1.8830775873369020E-02	44	7.1080020379761377E-01
21	-2.7226498457701823E-02	45	8.8090632488444798E-01
22	-3.2022840857588906E-02	46	1.0068321641150089E+00
23	-3.0996332527754609E-02	47	1.0737914947736096E+00

3.13 Noiseless Coding for the Small Step Scalability : BSAC

BSAC stands for bit sliced arithmetic coding and is the name of a noiseless coding kernl that provides a fine granule scalability in the MPEG-4 audio T/F coder. The BSAC noiseless coding module is an alternative to the AAC coding module, with all other modules of the AAC-based coder remaining unchanged. The BSAC noiseless coding is used to make the bitstream scalable and further reduce the redundancy of the scalefactors and the quantized spectrum. The BSAC noiseless decoding process is split into 4 clauses. Clause 3.13.1 to 3.13.4 describe the detailed decoding process of the spectral data, the stereo or pns related data, the scalefactors and the arithmetic model index.

3.13.1 Arithmetic decoding of Bit-Sliced Spectral Data

3.13.1.1 Tool description

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

In BSAC scalable coding scheme, a quantized sequence is divided into coding bands, as shown in clause 2.3.11.4. And, a quantized sequence is mapped into a bit-sliced sequence within a coding band. Four-dimensional vectors are formed from the bit-sliced sequence of the quantized spectrum and each 4-dimensional vector is divided into two subvectors. The noiseless coding of the subvectors relies on the arithmetic model index of the coding band, the significance, the dimension of the bit-sliced vector and the previous state.

The significance of the bit-sliced data is the bit-position of the vector to be coded.
The previous states are updated with coding the vectors from MSB to LSB. They are initialized to 0. And they are set to 1 when bit-value is non-zero.

The arithmetic model index for encoding the bit-sliced data within each coding band is transmitted starting from the lowest frequency coding band and progressing to the highest frequency coding band. For the detailed description of the arithmetic model index, see clause 3.13.4. Table 3.13.4 lists 32 arithmetic models which are used for encoding/decoding the bit-sliced data. The BSAC arithmetic model consists of several sub-models. Sub-models are classified and chosen according to the significance, the previous state and the dimension of the vector to be decoded as shown Table B.18 to Table B.48. Two subvectors are arithmetic encoded using the sub-model chosen from a set of several possible sub-models of BSAC arithmetic model.

3.13.1.2 Definitions

Bit stream elements:

acod_vec0[ArModel[i]][snf][subvector0] Arithmetic codeword from arithmetic coding with the model **BSAC_arith_model[ArModel[i]][snf][dim0]** that encodes the next subvector0, where subvector0 is composed of bit-values whose previous state is 0 among 4-tuple (w,x,y,z) which is formed from the bit-sliced sequence of the quantized spectral coefficients. Thus, w = x_quant[sfb][bin][Abit[i]] & (1<<(snf-1)), x = x_quant[sfb][bin+1][Abit[i]] & (1<<(snf-1)), y = x_quant[sfb][bin+2][Abit[i]] & (1<<(snf-1)) and z = x_quant[sfb][bin+3][Abit[i]] & (1<<(snf-1)) , where snf is the significance of the bit. N-tuples progress from low to high frequency within the current coding band and from MSB to LSB.

acod_vec1[ArModel[i]][snf][subvector1] Arithmetic codeword from arithmetic coding with the model **BSAC_arith_model[ArModel[i]][snf][dim1]** that encodes the next subvector1, where subvector1 is composed of bit-values whose previous state is 1 among 4-tuple (w,x,y,z) which is formed from the bit-sliced sequence of the quantized spectral coefficients. Thus, w = x_quant[sfb][bin][Abit[i]] & (1<<(snf-1)), x = x_quant[sfb][bin+1][Abit[i]] & (1<<(snf-1)), y = x_quant[sfb][bin+2][Abit[i]] & (1<<(snf-1)) and z = x_quant[sfb][bin+3][Abit[i]] & (1<<(snf-1)) , where snf is the significance of the bit. N-tuples progress from low to high frequency within the current coding band and from MSB to LSB.

acod_sign Arithmetic codeword from arithmetic coding sign_bit with the **sign_arith_model** given in Table B.15. sign_bit indicates sign bit for non-zero coefficient. A ‘1’ indicates a negative coefficient, a ‘0’ a positive one. When the bit value of the quantized signal is assigned 1 for the first time, sign bit is arithmetic coded and sent.

Help elements:

cur_snf [i] current significance of the *i*-th vector. cur_snf[] is initialized to Abit[cband]. See clause 2.3.11.5.

maxsnf maximum of current significance of the vectors to be decoded. See clause 2.3.11.5.

snf significance index

last_index maximum of layer_index values of each channel. See clause 2.3.11.5.

layer_index[ch][layer] array containing the index of the highest spectral coefficient of band-limit band in each layer for short windows in case of EIGHT_SHORT_SEQUENCE, otherwise for long windows
layer_index_offset_long[layer] table containing the index of the highest spectral coefficient of band-limit band in each layer for long windows. See Table 2.16
layer_index_offset_short[layer] table containing the index of the highest spectral coefficient of band-limit band in each layer for short windows. See Table 2.17
dim0 dimension of subvector 0
dim1 dimension of subvector 1
sample[ch][i] quantized spectral coefficients reconstructed from the decoded bit-sliced data of spectral line i in channel ch. See clause 2.3.11.5
prestate[ch][i] previous state that indicates whether the previously decoded value of frequency line i is 0 or not in channel ch. See clause 3.13.1.3
total_estimated_bits Total bits estimated to be used for decoding the bitstream. See clause 3.13.1.3
available_bits[layer] array containing the available bits to be used in the scalability layer. See clause 2.3.11.5.
sign_bit sign bit for non-zero coefficient. A ‘1’ indicates a negative coefficient, a ‘0’ a positive one. When the bit value of the quantized signal is assigned 1 for the first time, sign bit is arithmetic coded and sent.
layer scalability layer index
snf significance of vector to be decoded.
ch channel index
nch the number of channel

3.13.1.3 Decoding Process

Decoding of bit-sliced data
In BSAC encoder, a quantized sample is bit-sliced. In order to further reduce the redundancy of bit-sliced data, the vector is formed which consists of successive non-overlapping 4-tuples of the MSB data starting from the lowest-frequency coefficient and progressing to the highest-frequency coefficient..
The vectors are divided into two subvectors depending upon the previous states. One- to four-dimensional subvector of bit-sliced sequence are arithmetic coded and transmitted. For the case of multiple windows per block, the concatenated and possibly grouped and interleaved set of spectral coefficients is treated as a single set of coefficients that progress from low to high. This set of spectral coefficients may need to be de-interleaved after they are decoded. The spectral information for all scalefactor bands equal to or greater than **max_sfb** is set to zero.
After all MSB data are encoded from the lowest frequency line to the highest, the same encoding process is repeated until LSB data is encoded.

Four-dimensional vector is the basic unit in encoding/decoding the bit-sliced data. The 4-dimensional vector is made up of two sub-vectors upon the previous states. *prestate[]* indicates the state of the sample whose bit-sliced data has been decoded previously. Before the decoding of the bit-sliced data is started, all previous states are set to 0. The previous states are updated with coding the bit-sliced data of the sample from MSB to LSB. And they are set to 1 when bit-value is non-zero. The dimension of two subvectors depend on the previous state as follows :

```
offset = the frequency line offset of the vector
dim0 = 0 /* the dimension of the subvector 0 */
dim1 = 0 /* the dimension of the subvector 1 */
for (k=0; k<4; k++) {
    if (prestate[offset+k])
       dim1++
    else
       dim0++
}
```

After the dimension of the subvector is determined, the first subvector of the bit-sliced data is decoded with the arithmetic model which depends on the arithmetic model index, the dimension and the previous state value. And, the second vector is decoded.

Detailed arithmetic decoding procedure will be described in this clause. But, the model used for arithmetic decoding should be defined in order to decode the subvector. Arithmetic model of the bit-sliced data relies on the

arithmetic model index of the coding band, the dimension and the significance of the sub-vector and the previous states, as listed in Table B.18 to Table B.48.
The dimension of the subvector and the previous state of the decoded sample can calculated in the previous procedure.
There are 32 arithmetic models which can be used for encoding/decoding the bit-sliced data. In order to transmit the arithmetic model information used in encoding process, the index of arithmetic model is coded and included in the syntax of bsac_side_info(). After the model index is decoded, the decoding of the bit-sliced data shall be started.
The current significance of the sub-vector represent the bit-position of the vector to be decoded. Table 3.13.4 shows the allocated bit of the decoded sample according to the decoded model index. Current significance, *cur_snf[]* of all vector within the coding band are initialized to the allocated bit. For the detailed initialization process, see clause 2.3.11.5.

The sign bits associated with non-zero coefficients follow the arithmetic codeword when the bit-value of the quantized spectral coefficient is 1 for the first time, with a _1_ indicating a negative coefficient and a _0_ indicating a positive one. For example, if an arithmetic codeword has been decoded and the decoded bit-value of the quantized spectral coefficient is 1 for the first time, then immediately following this in the bitstream is

acod_sign

sign_bit of a sample can be arithmetic decoded from the bitstream using sign_arithmetic model given in Table B.15.

Two decoded subvectors of the bit-sliced data need to be de-interleaved and reconstructed to the sample. For the detailed reconstruction of the blt-sliced data, see **Reconstruction of the decoded sample from bit-sliced data** part in clause 2.3.11.2

Arithmetic decoding procedure

The pseudo code fragment shown below describes

- how to decode the arithmetic codeword
- the summary of the arithmetic decoding procedure
- how to use the arithemtic model in arithmetic decoding procedure
- how to estimate the bits necessary for decoding the arithmetic codeword.

```
while (1) {
       if (high < Half) {
          /* nothing */
       }
       else if (low >= Half ) {
          low -= Half;
          high -= Half;
          value -= Half;
       }
       else if (low >= First_qtr &&
          high <Third_qtr) {
          low -= First_qtr;
          high -= First_qtr;
          value -= First_qtr;
       }
       else
          break;

       low = 2*low;        /* Scale up code range. */
       high = 2*high+1;
       value = 2*value;

       value += next_bit(); /* Move in next input bit. */
    }

    range = (long)(high-low) + 1;
    cum = (((long)(value-low)+1)*16384-1)/range; /* Find cum freq */

    for (symbol=0; armodel[symbol]>cum; symbol++);

    /* Narrow the code region to that allotted to this symbol. */
    if (symbol>0) {
       high  = low + (range * armodel[symbol-1])/16384 - 1;
```

```
    }
    low  = low + (range * armodel[symbol])/16384;
```

Symbol is the value decoded from arithmetic codeword. next_bit() is the function that extract 1 bit value from the bit-stream. Half, First_qtr and Third_qtr are defined as 8192, 4096 and 12288 respectively. The frequency range for the *i*th symbol with the exception of the 0^{th} symbol is from armodel[i-1] to armodel[i]. In case of the 0^{th} symbol, the frequency range is from 16384 to armodel[0]. As *i* decreases, armodel[i] increases.

The estimated bits are accumulated in order to calculate the total bits as follows.

```
    if (symbol>0)
       prob = (double)(armodel[symbol-1]-armodel[symbol])/16384;
    else
    prob = (double)(16384-armodel[symbol])/16384;

    estimated_bits += (double)((int)(-log(prob)/log((double)2.)*4096))/4096.;
```

3.13.2 Arithmetic Decoding of stereo_info, ms_used or noise_flag

3.13.2.1 Tool description

The BSAC scalable coding scheme includes the noiseless coding which is different from MPEG-4 AAC coding and further reduce the redundancy of the stereo-related data.
Decoding of the stereo-related data and Perceptual Noise Substitution(pns) data is depended on pns_data_present and ms_mask_present which indicates the stereo mask. Since the decoded data is the same value with MPEG-4 AAC, the MPEG-4 AAC stereo-related and pns processing follows the decoding of the stereo-related data and pns data.

3.13.2.2 Definition

Bit stream elements:

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

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

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

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

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

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

11 reserved

Help elements:

ch	channel index
g	group index
sfb	scalefacotor band index within group
layer_sfb[layer]	array containing the index of the lowest scalefactor band to be added newly in the scalability layer for short windows in case of EIGHT_SHORT_SEQUENCE, otherwise for long windows. See clause 2.3.11.5
layer_sfb_offset_long[layer]	table containing the index of the lowest scalefactor band to be added newly in the scalability layer for long windows. See Table 2.18.
layer_sfb_offset_short[layer]	table containing the index of the lowest scalefactor band to be added newly in the scalability layer for short windows. See Table 2.19.
num_window_groups	number of groups of windows which share one set of scalefactors. See clause 2.3.11.4.
layer	scalability layer index
nch	the number of channel
ms_mask_present	this two bit field indeicates that the stereo mask is 00 Independent 01 1 bit mask of max_sfb bands of ms_used is located in the layer side information part. 10 All ms_used are ones 11 2 bit mask of max_sfb bands of stereo_info is located in the layer side information part.

3.13.2.3 Decoding Process

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

00 Independent
01 1 bit mask of max_sfb bands of ms_used is located in the layer side information part.
10 All ms_used are ones
11 2 bit mask of max_sfb bands of stereo_info is located in the layer side information part.

Decoding process is classified as follows :

- 1 channel, no pns data
  If the number of channel is 1 and pns data is not present, there is no bit-stream elements related to stereo or pns.
- 1 channel, pns data
  If the number of channel is 1 and pns data is present, noise flag of the scalefactor bands between **pns_start_sfb** to **max_sfb** is arithmetic decoded using model shown in Table B.16. Perceptual noise substitution is done according to the decoded noise flag.
- 2 channel, ms_mask_present=0 (Independent), No pns data
  If ms_mask_present is 0 and pns data is not present, arithmetic decoding of stereo_info or ms_used is not needed.
- 2 channel, ms_mask_present=0 (Independent), pns data
  If ms_mask_present is 0 and pns data is present, noise flag for pns is arithmetic decoded using model shown in Table B.16. Perceptual noise substitution of independent mode is done according to the decoded noise flag.
- 2 channel, ms_mask_present=2 (all ms_used), pns data or no pns data
  All ms_used values are ones in this case. So, M/S stereo processing of AAC is done at all scalefactor band. And naturally there can be no pns processing regardless of pns_data_present flag.
- 2 channel, ms_mask_present=1 (optional ms_used), pns data or no pns data
  1 bit mask of max_sfb bands of ms_used is conveyed in this case. So, ms_used is arithmetic decoded using the ms_used model given in Table B.13. M/S stereo processing of AAC is done or not according to the decoded ms_used. And there is no pns processing regardless of pns_data_present flag

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

  At first, stereo_info is arithmetic decoded using the stereo_info model given in Table B.14.
  stereo_info is is two-bit flag per scalefactor band indicating that M/S coding or Intensity coding is being used in window group g and scalefactor band sfb as follows :

  00 Independent
  01 ms_used
  10 Intensity_in_phase
  11 Intensity_out_of_phase

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

- 2 channel, ms_mask_present=3 (optional ms_used/intensity/pns), pns data

  stereo_info is arithmetic decoded using the stereo_info model given in Table B.14.
  If stereo_info is 1 or 2, M/S stereo or intensity stereo processing of AAC is done with these decoded data and there is no pns processing.
  If stereo_info is 3 and scalefactor band is larger than or equal to pns_start_sfb, noise flag for pns is arithmetic decoded using model given in Table B.16. And then if the both noise flags of two channel are 1, noise substitution mode is arithmetic decoded using model given in Table B.17. The perceptual noise is substituted or out_of_phase intensity stereo processing is done according to the substitution mode. Otherwise, the perceptual noise is substituted only if noise flag is 1.
  If stereo_info is 3 and scalefactor band is smaller than pns_start_sfb, out_of_phase intensity stereo processing is done.

3.13.3 Arithmetic Decoding of scalefactors

3.13.3.1 Tool description

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

The noiseless coding has two ways to represent the scalefactors. One way is to use coding scheme similar to AAC. The max_scalefactor is coded as an 8 bit unsigned integer. The first scalefactor associated with the quantized spectrum is differentially coded relative to the max_scalefactor value and the arithmetic coded using the differential scalefactor arithmetic model. The remaining scalefactors are differentially coded relative to the previous scalefactor and then Arithmetic coded using the differential scalefactor model.
Another way is the BSAC scalefactor coding method. The max_scalefactor is coded as an 8 bit unsigned integer. The scalefactors are differentially coded relative to the offset value, max_scalefactor, and then arithmetic coded using the scalefactor arithmetic model.

3.13.3.2 Definitions

Bit stream element:

acode_scf[ch][g][sfb] Arithmetic codeword from the coding of the differential scalefactors.
acode_scf_index[ch][g][sfb] Arithmetic codeword from the coding of the index which is converted from the differential scalefactor.
acode_esc_scf_index[ch][g][sfb] Arithmetic codeword from the coding of the escape code for the index which is converted from the differential scalefactor.
acode_dpcm_noise_energy[ch][g][sfb] Arithmetic codeword from the coding of the differential noise energy for PNS.
acode_dpcm_noise_energy_index[ch][g][sfb] Arithmetic codeword from the coding of the index which is converted from the the differential noise energy.
acode_esc_dpcm_noise_energy_index[ch][g][sfb] Arithmetic codeword from the coding of the escape code for the index which is converted from the differential noise energy.
acode_is_position[ch][g][sfb] Arithmetic codeword from the coding of the differential intensiy stereo position.
acode_is_position_index[ch][g][sfb] Arithmetic codeword from the coding of the index which is converted from the differential intensiy stereo position.
acode_esc_is_position_index[ch][g][sfb] Arithmetic codeword from the coding of the escape code for the index which is converted from the differential intensity stereo position.

acode_pcm_noise_energy[ch][g][sfb] Arithmetic codeword from the coding of the noise energy for the first PNS scalefactor band. Arithmetic model for coding pcm_noise_energy is given as follows :

```
arithmetic_model[0] = 16384 - 32;
for (index=1; index<512;  index++)
        arithmetic_model[index] = arithmetic_model[index-1] - 32;
```

Help elements:

ch	channel index
g	group index
sfb	scalefacotor band index within group
layer_sfb[layer]	array containing the index of the lowest scalefactor band to be added newly in the scalability layer for short windows in case of EIGHT_SHORT_SEQUENCE, otherwise for long windows. See clause 2.3.11.5
layer_sfb_offset_long[layer]	table containing the index of the lowest scalefactor band to be added newly in the scalability layer for long windows. See Table 2.18
layer_sfb_offset_short[layer]	table containing the index of the lowest scalefactor band to be added newly in the scalability layer for short windows. See Table 2.19.
num_window_groups	number of groups of windows which share one set of scalefactors. See clause 2.3.11.4
layer	scalability layer index
nch	the number of channel
scf_coding[ch]	indicates the decoding method of the scalefactors.

The noiseless coding of the scalefactor requires two constants, ESC_SCF_INDEX and ESC_INDEX whose values are defined as 54. (See bsac_side_info())

3.13.3.3 Decoding Process

The spectral coefficients are divided into scalefactor bands that contain a multiple of 4 quantized spectral coefficients. Each scalefactor band has a scalefactor. The noiseless coding has two ways to represent the scalefactors. **scf_coding[ch]** indicates which method the scalefactor is coded with.

One way is to use coding scheme similar to AAC. For all scalefactors the difference to the preceding value is mapped into new index using Table B.1. If the newly mapped index is smaller than 54, it is arithmetic-coded using the arithmetic model given in Table B.3. Otherwise, the escape value 54 is arithmetic coded using the scalefactor arithmetic model given in Table B.3 and the difference to escape value 54 is arithmetic coded using the arithmetic model given in Table B.4. The initial preceding value is given explicitly as a 8 bit PCM in the bitstream element **max_scalefactor**.
The max_scalefactor is coded as an 8 bit unsigned integer. The first scalefactor associated with the quantized spectrum is differentially coded relative to the max_scalefactor value and arithmetic coded using the differential scalefactor arithmetic model. The remaining scalefactors are differentially coded relative to the previous scalefactor and then arithmetic coded using the differential scalefactor model, as shown in Table 3.13.2.

A second way is BSAC scalefactor coding method. For all scalefactors the difference to the offset value is arithmetic-coded using the arithmetic model, as shown in Table 3.13.3. The arithmetic model used for coding differential scalefactors is given as a 2-bit unsigned integer in the bitstream element, **scalefactor_model**. The offset value is given explicitly as a 8 bit PCM in the bitstream element **max_scalefactor**.

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

```
for (ch=0; ch<nch; ch++) {
  if (scf_coding[ch]==1) {
     for (g=0; g<num_window_group; g++) {
        for( sfb=layer_sfb[layer]; sfb<layer_sfb[layer+1]; sfb++ ) {
```

```
          diff_scf = arithmetic_decoding();
          sf[ch][g][sfb] = max_scalefactor – diff_scf;
        }
      }
    }
    else {
      for (g=0; g<num_window_group; g++) {
        for( sfb=layer_sfb[layer]; sfb<layer_sfb[layer+1]; sfb++ ) {
        diff_scf_index = arithmetic_decoding();
          if (diff_scf_index==ESC_SCF_INDEX) {
            esc_scf_index = arithmetic_decoding();
            diff_scf_index += esc_scf_index;
          }
          if (sfb==0)
             scf_index = max_scalefactor – diff_scf_index;
          else
             scf_index = sf[ch][g][sfb-1] – diff_scf_index;
          sf[ch][g][sfb] = index2sf[scf_index];
        }
      }
    }
  }
```

where, layer_sfb[layer] is the start scalefactor band and layer_sfb[layer+1] is the end scalefactor band for decoding the scalefactors in each layer.

3.13.4 Arithmetic Decoding of arithmetic model index

3.13.4.1 Tool description

In BSAC scalable coding scheme, the spectral coefficients are divided into coding bands which contain 32 quantized spectral coefficients for the noiseless coding. Coding bands are the basic units used for the noiseless coding. The set of bit-sliced sequence is divided into coding bands. The arithmetic model index for encoding the bit-sliced data within each coding band is transmitted starting from the lowest frequency coding band and progressing to the highest frequency coding band. For all arithmetic model indexes the difference to the offset value is arithmetic-coded using the arithmetic model **ArModel_model,** as shown in Table 3.13.3.

3.13.4.2 Definition

Bit stream element:

acode_ArModel[ch][cband] Arithmetic codeword from the arithmetic coding of arithmeticmodel index for coding-band cband.

Help elements:

g	group index
sfb	scalefacotor band index within group
layer_sfb[layer]	array containing the index of the lowest scalefactor band to be added newly in the scalability layer for short windows in case of EIGHT_SHORT_SEQUENCE, otherwise for long windows
num_window_group	number of groups of windows which share one set of scalefactors
swb_offset[sfb]	array containing the index of the lowest spectral coefficient of scalefactor band sfb for short windows in case of EIGHT_SHORT_SEQUENCE, otherwise for long windows
cband	coding band index within group
index2cb(ch, i)	function returning coding band which the index of the spectral coefficient i is mapped into by the mapping table, index2cband[][].
layer	scalability layer index
ch	channel index
nch	the number of channel

3.13.4.3 Decoding Process

For all arithmetic model indexes the difference to the offset value is arithmetic-coded using the arithmetic model **ArModel_model,** as shown in Table 3.13.3. The arithmetic model used for coding differential arithmetic model index is given as a 2-bit unsigned integer in the bitstream element, **ArModel_model**. The offset value is given explicitly as a 5 bit PCM in the bitstream element **min_ArModel**.

The following pseudo code describes how to decode the arithmetic model index *ArModel[cband]* in base layer and each enhancement layer:

```
for (ch=0; ch<nch; ch++)
  for( sfb=layer_sfb[layer]; sfb<layer_sfb[layer+1]; sfb++ )
     for (g=0; g<num_window_groups; g++) {
        band = (sfb * num_window_groups) + g
        for (i=0; swb_offset[band]; i<swb_offset[band+1]; i+=4) {
           cband = index2cb(ch, i);
           if (!decode_cband[ch][cband]) {
              ArModel[ch][cband] = min_ArModel + arithmetic_decoding();
              decode_cband[ch][cband] = 1;
           }
        }
     }
```

where, layer_sfb[layer] is the start scalefactor band and layer_sfb[layer+1] is the end scalefactor band for decoding the arithmetic model index in each layer, and decode_cband[ch][cband] is flag indicating whether the arithmetic model has been decoded (1) or not (0).

3.13.5 Tables

Table 3.13.1 Arithmetic Model 0 of mapped Scalefactor

Model Number	Dimension of Codebook	Range of values	Model listed in Table
0	1	0 to 54	B.3
1	1	0 to 66	B.4

Table 3.13.2 Arithmetic Model 1 of Differential Scalefactor

Model Number	Largest Differential Scalefactor	Model listed in Table
0	7	B.5
1	15	B.6
2	31	B.7
3	63	B.8

Table 3.13.3 Arithmetic Model of Differential ArModel

Model Number	Largest Differential ArModel	Model listed in Table
0	3	B.9
1	7	B.10
2	15	B.11
3	31	B.12

Table 3.1.34 BSAC Arithmetic Model Parameters

Arithmetic Model index	allocated bit / sample within coding band	Model listed in Table	Arithmetic model index	allocated bit / sample within coding band	Model listed in Table

0	0	B.18	16	8	B.33
1	-	not used	17	8	B.34
2	1	B.19	18	9	B.35
3	1	B.20	19	9	B.36
4	2	B.21	20	10	B.37
5	2	B.22	21	10	B.38
6	3	B.23	22	11	B.39
7	3	B.24	23	11	B.40
8	4	B.25	24	12	B.41
9	4	B.26	25	12	B.42
10	5	B.27	26	13	B.43
11	5	B.28	27	13	B.44
12	6	B.29	28	14	B.45
13	6	B.30	29	14	B.46
14	7	B.31	30	15	B.47
15	7	B.32	31	15	B.48

3.14 Perceptual Noise Substitution (PNS)

3.14.1 Tool description

This tool is used to implement perceptual noise substitution coding within an ICS. Thus, certain sets of spectral coefficients are derived from random vectors rather than from Huffman coded symbols and an inverse quantization process. This is done selectively on a scalefactor band and group basis when perceptual noise substitution is flagged as active.

3.14.2 Definitions

hcod_sf[]	Huffman codeword from the Huffman code table used for coding of scalefactors (see ISO 13818-7 clause 4.2)
dpcm_noise_nrg[][]	Differentially encoded noise energy
noise_nrg[group][sfb]	Noise energy for each group and scalefactor band
spec[]	Array containing the channel spectrum of the respective channel

3.14.3 Decoding Process

The use of the percpetual noise substitution tool is signaled by the use of the pseudo codebook NOISE_HCB (13).

Furthermore, if the same scalefactor band and group is coded by perceptual noise substitution in both channels of a channel pair, the correlation of the noise signal can be controlled by means of the ms_used field: While the default noise generation process works independently for each channel (separate generation of random vectors), the same random vector is used for both channels if ms_used[] is set for a particular scalefactor band and group. In this case, no M/S stereo coding is carried out (because M/S stereo coding and noise substitution coding are mutually exclusive).

The energy information for percpetual noise substitution decoding is represented by a "noise energy" value indicating the overall power of the substituted spectral coefficients in steps of 1.5 dB. If noise substitution coding is active for a particular group and scalefactor band, a noise energy value is transmitted instead of the scalefactor of the respective channel.

Noise energies are coded just like scalefactors, i.e. by Huffman coding of differential values:

- the start value for the DPCM decoding is given by global_gain.
- Differential decoding is done separately between scalefactors, intensity stereo positions and noise energies. In other words, the noise energy decoder ignores interposed scalefactors and intensity stereo position values and vice versa (see ISO 13818-7 clause 6.3.2)

The same codebook is used for coding of noise energies as for scalefactors.

One pseudo function is defined for use in perceptual noise substitution decoding:

```
function is_noise(group,sfb) {
   1    for window groups / scalefactor bands with
           codebook sfb_cb[group][sfb] == NOISE_HCB
   0    otherwise
   }
```

The noise substitution decoding process for one channel is defined by the following pseudo code:

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

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

   /* Do perceptual noise substitution decoding */
   for (b=0; b<window_group_length[g]; b++)  {
      for (sfb=0; sfb<max_sfb; sfb++)  {
         if (is_noise(g,sfb))  {

            offs = swb_offset[sfb];
            size = swb_offset[sfb+1] - offs;

            /* Generate random vector */
            gen_rand_vector( &spec[g][b][sfb][0], size );
            scale = 1/(size * sqrt(MEAN_NRG));
            scale *= 2.0^(0.25*noise_nrg [g][sfb]);
            /* Scale random vector to desired target energy */
            for (i=0; i<len; i++)
               spec[g][b][sfb][i] *= scale;

         }

      }
   }

}
```

The constant NOISE_OFFSET is used to adapt the range of average noise energy values to the usual range of scalefactors and has a value of 90.

The function gen_rand_vector(addr, size) generates a vector of length <size> with signed random values of average energy MEAN_NRG per random value. A suitable random number generator can be realized using one multiplication/accumulation per random value.

3.14.4 Diagrams

3.14.5 Tables

3.14.6 Integration with Intra Channel Prediction Tools

For scalefactor bands coded using PNS the corresponding predictors are switched to “off“, thus effectively overriding the status specified by the prediction_used mask. In addition, for scalefactor bands coded by perceptual noise substitution the predictors belonging to the corresponding spectral coefficients are reset (see ISO 13818-7 clause 8.3.3). The update of these predictors is done by feeding a value of zero as the "last quantized value" $x_{rec}(n-1)$.

In Long Term Prediction, the scalefactor bands coded using PNS are not predicted.

3.14.7 Integration with other AAC Tools

The following interactions between the perceptual noise substitution tool and other AAC tools take place:

Definition of a new pseudo Huffman codebook number NOISE_HCB = 13

- During Huffman decoding of the quantized spectral coefficients, the Huffman codebook table NOISE_HCB is treated exactly like the zero codebook ZERO_HCB, i.e. no Huffman codewords are read for the corresponding scalefactor band and group.
- If the same scalefactor band and group is coded by perceptual noise substitution in both channels of a channel pair, no M/S stereo decoding is carried out for this scalefactor band and group .
- The pseudo noise components generated by the perceptual noise substitution tool are injected into the output spectrum prior to the temporal noise shaping (TNS) processing step.

3.14.8 Integration into a Scalable AAC-based Coder

The following rules apply for usage of the perceptual noise substitution tool in a scalable AAC-based coder:

If a particular scalefactor band and group is coded by perceptual noise substitution, its contribution to the spectral components of the reconstructed output signal for the update of the intra channel predictor is omitted.

- If a particular scalefactor band and group is coded by perceptual noise substitution, its contribution to the spectral components of the output signal is omitted if spectral coefficients are transmitted for this scalefactor band and group in any of the higher (enhancement) layers by means of a non-zero codebook number (i.e. a Huffman codebook != ZERO_HCB).
- If a particular scalefactor band and group is coded by perceptual noise substitution in both channels of a channel pair, the higher (enhancement) layers may still use the M/S stereo flag ms_used[][] to signal the use of M/S stereo decoding.

3.15 Frequency Selective Switch Module

3.15.1 Definitions

dc_group	Four consecutive scalefactor bands if the window type is not SHORT_WINDOW. One band of diff_short_lines, if the window type is SHORT_WINDOW.
no_of_dc_groups	If the window type is not SHORT_WINDOW, the number of groups depending on the sampling frequency is given in table 2 below. If the window type is SHORT_WINDOW, no_of_dc_groups is '1'.
diff_short_lines	Only used, if the window type is SHORT_WINDOW. The number of spectral lines in the single dc_group per window, depending on the sampling rate, is given in table 2 below.
diff_control[w][dc_group]	for each window and dc_group the switch control information for one dc_group. If the window type is not SHORT_WINDOW **diff_control[w][dc_group]** is huffman encoded using table 1 in the bitstream.
diff_control_sfb[w][sfb]	Only applies, if the window type is not SHORT_WINDOW. The decoded **diff_control[w][dc_group]** values; 1 bit or each scale factor band.

The Inverse Frequency Selective Switching Unit (IFFS) connects one of two input signals to the output, depending on diff_control[w][dc_group].

In the bitstream diff_control[w][dc_group] is huffman encoded using the following table:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Code	0	20	21	22	23	24	25	8	9	26	27	28	29	30	31	1
Length	2	5	5	5	5	5	5	4	4	5	5	5	5	5	5	2

Table 1: diff_control_hc_tab

Sampling Rate	96	88.2	64	48	44.1	32	24	22.05	16	12	11.025	8
Core Sampling Rate	8	7.35	8	8	7.35	8	8	7.35	8	12	11.025	8
no_of_dc_groups	4	4	5	5	5	6	8	8	8	10	10	9
diff_short_lines	8	8	13	18	18	26	36	36	54	104	104	104

Table 2: number of dc_groups for each sampling rate

3.15.2 Decoding Process

Decoding if the window type is not SHORT_WINDOW:

After huffman decoding diff_control[w][dc_group] from the bitstream, the array diff_control_sfb[w][sfb] is generated according to:

```
if ( ! SHORT_WINDOW ) {
   dc_group = 0;
   while( dc_group < no_of_ dc_groups ) {
       for( i=0; i<4; i++ ) {
           diff_control_sfb[0][dc_group*4+i] = diff_control[0][dc_group]  & 0x8;
           diff_control[0][dc_group] <<= 1;
       }
   }
}
```

For all scale factor bands which did not get a value assigned in diff_control_sfb[w][sfb] in the above procedure, diff_control_sfb[w][sfb] is set to ‘1’;

Finally the switching for all scale factor bands is done according to:

```
if ( diff_control_sfb[w][sfb] == 0 ) {
   spec_out[w][sfb] = spec_requantized[w][sfb] + spec_core[w][sfb];
} else {
   spec_out[w][sfb] = spec_requantized[w][sfb];
}
```

Decoding if the window type is SHORT_WINDOW:

If the window type is SHORT_WINDOW, there is only one band of diff_short_lines per window, where the diff control mechanism is applied.

For spectral lines #0 to #diff_short_lines:

```
if ( diff_control_sfb[w][0] == 0 ) {
   spec_out[w] = spec_requantized[w] - spec_core[w];
} else {
   spec_out[w] = spec_requantized[w];
}
```

For the remaining lines the output of the switch is identical to the input:

```
   spec_out[w] = spec_requantized[w];
```

3.16 Upsampling Filter Tool

3.16.1 Tool Description

The Upsampling filter tool is used to adapt the sampling rate of the core coder to the sampling rate of the time/frequency coder. The upsampling filter is implemented based on the MDCT filterbank of the AAC-derived encoder, which is used without any changes. This filterbank is very similar to the IMDCT filterbank already used in the decoder.

The filterbank takes a block of time samples of the core coder output and inserts an appropriate number of zeroes between these samples to generate a signal at the desired higher sampling rate. The upsampled samples are then modulated by an appropriate window function, and the MDCT is performed. Each block of input samples is overlapped by 50% with the immediately preceding block and the following block. The transform input block length N is set to either 2048 (1920) or 256 (240) samples. The filterbank is switched synchronously to the IMDCT using both window_sequence and window_shape.

The output of the filterbank is connected to the FSS module, which only uses the output values in the FSS-bands. Since the upper FSS band doesn't exceed half of the lower sampling rate, there are no aliasing effects.

3.16.2 Definitions

The syntax elements for the filterbank are identical to those used for the IMDCT filterbank. They consist of the control information **window_sequence** and **window_shape**.

window_sequence	2 bit indicating which window sequence (i.e. block size) is used (see 1.3, Table 6.11).
window_shape	1 bit indicating which window function is selected (see 1.3, Table 6.11).
up-sampling-factor	ratio of T/F coder sampling rate and core coder sampling rate.
$x_{in\text{-}mdct\text{-}core}$	temporal data field, which is used to hold the up-sampled input to the MDCT filterbank
$x_{out\text{-}core}[i]$	Output samples of the core decoder

3.16.3 Decoding Process

The analysis window length N for the transform is a function of the syntax element **window_sequence** and the algorthmic context. It is derived in an identical way to the procedure described for the Filterbank and Blockswitching tool.

3.16.3.1 Upsampling by insertion of zeroes

The input to the filterbank is generated by :

$x_{in\text{-}mdct\text{-}core}[k] = 0$ for k = [0 : N/2-1]

$x_{in\text{-}mdct\text{-}core}[\text{up-sampling-factor}*i] = x_{out\text{-}core}[i]$ for i = [0 : N/2/up-sampling-factor-1]

3.16.3.2 Windowing and block switching

The adaptation of the time-frequency resolution of the filterbank is done by shifting between transforms whose input lengths are either 2048 (1920) or 256 (240) samples, synchronously to the decoder IMDCT filterbank. The selection between the 2048/256 or the 1920/240 pairs is done depending on the frame length of the core coder. The 1920/240 pair is used for all core coders having a frame length of a multiple of 10 ms.

The windowed time domain values can be calculated in using exactly the same windows w(n) as defined for the IMDCT filterbank.

The windowed coefficients are calculated by

$$z_{i,n} = w(n) \cdot x'_{\text{in\_mdct\_core}}(n);$$

Finally the windowed coefficients are transformed with the MDCT as defined in the following paragraph.

3.16.3.3 MDCT

The spectral coefficient, $X_{i,k}$, are defined as follows:

$$X_{i,k} = 2 \cdot \sum_{n=0}^{N-1} z_{i,n} \cos\left(\frac{2\pi}{N} (n + n_0) \left(k + \frac{1}{2} \right) \right) \text{ for } 0 \le k < N/2 .$$

where:

z_{in} = windowed input sequence

n = sample index

k = spectral coefficient index

i = block index

N = window length of the one transform window based on the window_ sequence value

n_0 = (N / 2 + 1) / 2

Only the output values from 0 to N/ 2/ up-sampling-factor-1can be used without aliasing distortions. This is ensured by the subsequently following FSS module.

4

Annex A

Table A.1 – Scalefactor Huffman Codebook

index	length	codeword (hexadecimal)	index	length	codeword (hexadecimal)
0	18	3ffe8	61	4	a
1	18	3ffe6	62	4	c
2	18	3ffe7	63	5	1b
3	18	3ffe5	64	6	39
4	19	7fff5	65	6	3b
5	19	7fff1	66	7	78
6	19	7ffed	67	7	7a
7	19	7fff6	68	8	f7
8	19	7ffee	69	8	f9
9	19	7ffef	70	9	1f6
10	19	7fff0	71	9	1f9
11	19	7fffc	72	10	3f4
12	19	7fffd	73	10	3f6
13	19	7ffff	74	10	3f8
14	19	7fffe	75	11	7f5
15	19	7fff7	76	11	7f4
16	19	7fff8	77	11	7f6
17	19	7fffb	78	11	7f7
18	19	7fff9	79	12	ff5
19	18	3ffe4	80	12	ff8
20	19	7fffa	81	13	1ff4
21	18	3ffe3	82	13	1ff6
22	17	1ffef	83	13	1ff8
23	17	1fff0	84	14	3ff8
24	16	fff5	85	14	3ff4
25	17	1ffee	86	16	fff0
26	16	fff2	87	15	7ff4
27	16	fff3	88	16	fff6
28	16	fff4	89	15	7ff5
29	16	fff1	90	18	3ffe2
30	15	7ff6	91	19	7ffd9
31	15	7ff7	92	19	7ffda
32	14	3ff9	93	19	7ffdb
33	14	3ff5	94	19	7ffdc
34	14	3ff7	95	19	7ffdd
35	14	3ff3	96	19	7ffde
36	14	3ff6	97	19	7ffd8
37	14	3ff2	98	19	7ffd2
38	13	1ff7	99	19	7ffd3
39	13	1ff5	100	19	7ffd4
40	12	ff9	101	19	7ffd5
41	12	ff7	102	19	7ffd6
42	12	ff6	103	19	7fff2
43	11	7f9	104	19	7ffdf
44	12	ff4	105	19	7ffe7
45	11	7f8	106	19	7ffe8
46	10	3f9	107	19	7ffe9
47	10	3f7	108	19	7ffea
48	10	3f5	109	19	7ffeb
49	9	1f8	110	19	7ffe6
50	9	1f7	111	19	7ffe0

51	8	fa	112	19	7ffe1
52	8	f8	113	19	7ffe2
53	8	f6	114	19	7ffe3
54	7	79	115	19	7ffe4
55	6	3a	116	19	7ffe5
56	6	38	117	19	7ffd7
57	5	1a	118	19	7ffec
58	4	b	119	19	7fff4
59	3	4	120	19	7fff3
60	1	0			

Table A.2 – Spectrum Huffman Codebook 1

index	**length**	**codeword** (hexadecimal)	**index**	**length**	**codeword** (hexadecimal)
0	11	7f8	41	5	14
1	9	1f1	42	7	65
2	11	7fd	43	5	16
3	10	3f5	44	7	6d
4	7	68	45	9	1e9
5	10	3f0	46	7	63
6	11	7f7	47	9	1e4
7	9	1ec	48	7	6b
8	11	7f5	49	5	13
9	10	3f1	50	7	71
10	7	72	51	9	1e3
11	10	3f4	52	7	70
12	7	74	53	9	1f3
13	5	11	54	11	7fe
14	7	76	55	9	1e7
15	9	1eb	56	11	7f3
16	7	6c	57	9	1ef
17	10	3f6	58	7	60
18	11	7fc	59	9	1ee
19	9	1e1	60	11	7f0
20	11	7f1	61	9	1e2
21	9	1f0	62	11	7fa
22	7	61	63	10	3f3
23	9	1f6	64	7	6a
24	11	7f2	65	9	1e8
25	9	1ea	66	7	75
26	11	7fb	67	5	10
27	9	1f2	68	7	73
28	7	69	69	9	1f4
29	9	1ed	70	7	6e
30	7	77	71	10	3f7
31	5	17	72	11	7f6
32	7	6f	73	9	1e0
33	9	1e6	74	11	7f9
34	7	64	75	10	3f2
35	9	1e5	76	7	66
36	7	67	77	9	1f5
37	5	15	78	11	7ff
38	7	62	79	9	1f7
39	5	12	80	11	7f4
40	1	0			

Table A.3 – Spectrum Huffman Codebook 2

index	length	codeword (hexadecimal)	index	length	codeword (hexadecimal)
0	9	1f3	41	5	7
1	7	6f	42	6	1d
2	9	1fd	43	5	b
3	8	eb	44	6	30
4	6	23	45	8	ef
5	8	ea	46	6	1c
6	9	1f7	47	7	64
7	8	e8	48	6	1e
8	9	1fa	49	5	c
9	8	f2	50	6	29
10	6	2d	51	8	f3
11	7	70	52	6	2f
12	6	20	53	8	f0
13	5	6	54	9	1fc
14	6	2b	55	7	71
15	7	6e	56	9	1f2
16	6	28	57	8	f4
17	8	e9	58	6	21
18	9	1f9	59	8	e6
19	7	66	60	8	f7
20	8	f8	61	7	68
21	8	e7	62	9	1f8
22	6	1b	63	8	ee
23	8	f1	64	6	22
24	9	1f4	65	7	65
25	7	6b	66	6	31
26	9	1f5	67	4	2
27	8	ec	68	6	26
28	6	2a	69	8	ed
29	7	6c	70	6	25
30	6	2c	71	7	6a
31	5	a	72	9	1fb
32	6	27	73	7	72
33	7	67	74	9	1fe
34	6	1a	75	7	69
35	8	f5	76	6	2e
36	6	24	77	8	f6
37	5	8	78	9	1ff
38	6	1f	79	7	6d
39	5	9	80	9	1f6
40	3	0			

Table A.4 – Spectrum Huffman Codebook 3

index	length	codeword (hexadecimal)	index	length	codeword (hexadecimal)
0	1	0	41	10	3ef
1	4	9	42	9	1f3
2	8	ef	43	9	1f4
3	4	b	44	11	7f6
4	5	19	45	9	1e8
5	8	f0	46	10	3ea
6	9	1eb	47	13	1ffc

7	9	1e6	48	8	f2
8	10	3f2	49	9	1f1
9	4	a	50	12	ffb
10	6	35	51	10	3f5
11	9	1ef	52	11	7f3
12	6	34	53	12	ffc
13	6	37	54	8	ee
14	9	1e9	55	10	3f7
15	9	1ed	56	15	7ffe
16	9	1e7	57	9	1f0
17	10	3f3	58	11	7f5
18	9	1ee	59	15	7ffd
19	10	3ed	60	13	1ffb
20	13	1ffa	61	14	3ffa
21	9	1ec	62	16	ffff
22	9	1f2	63	8	f1
23	11	7f9	64	10	3f0
24	11	7f8	65	14	3ffc
25	10	3f8	66	9	1ea
26	12	ff8	67	10	3ee
27	4	8	68	14	3ffb
28	6	38	69	12	ff6
29	10	3f6	70	12	ffa
30	6	36	71	15	7ffc
31	7	75	72	11	7f2
32	10	3f1	73	12	ff5
33	10	3eb	74	16	fffe
34	10	3ec	75	10	3f4
35	12	ff4	76	11	7f7
36	5	18	77	15	7ffb
37	7	76	78	12	ff7
38	11	7f4	79	12	ff9
39	6	39	80	15	7ffa
40	7	74			

Table A.5 – Spectrum Huffman Codebook 4

index	**length**	**codeword** (hexadecimal)	**index**	**length**	**codeword (hexadecimal)**
0	4	7	41	7	6b
1	5	16	42	8	e3
2	8	f6	43	7	69
3	5	18	44	9	1f3
4	4	8	45	8	eb
5	8	ef	46	8	e6
6	9	1ef	47	10	3f6
7	8	f3	48	7	6e
8	11	7f8	49	7	6a
9	5	19	50	9	1f4
10	5	17	51	10	3ec
11	8	ed	52	9	1f0
12	5	15	53	10	3f9
13	4	1	54	8	f5
14	8	e2	55	8	ec
15	8	f0	56	11	7fb
16	7	70	57	8	ea
17	10	3f0	58	7	6f

18	9	1ee	59	10	3f7
19	8	f1	60	11	7f9
20	11	7fa	61	10	3f3
21	8	ee	62	12	fff
22	8	e4	63	8	e9
23	10	3f2	64	7	6d
24	11	7f6	65	10	3f8
25	10	3ef	66	7	6c
26	11	7fd	67	7	68
27	4	5	68	9	1f5
28	5	14	69	10	3ee
29	8	f2	70	9	1f2
30	4	9	71	11	7f4
31	4	4	72	11	7f7
32	8	e5	73	10	3f1
33	8	f4	74	12	ffe
34	8	e8	75	10	3ed
35	10	3f4	76	9	1f1
36	4	6	77	11	7f5
37	4	2	78	11	7fe
38	8	e7	79	10	3f5
39	4	3	80	11	7fc
40	4	0			

Table A.6 – Spectrum Huffman Codebook 5

index	**length**	**codeword** (hexadecimal)	**index**	**length**	**codeword** (hexadecimal)
0	13	1fff	41	4	a
1	12	ff7	42	7	71
2	11	7f4	43	8	f3
3	11	7e8	44	11	7e9
4	10	3f1	45	11	7ef
5	11	7ee	46	9	1ee
6	11	7f9	47	8	ef
7	12	ff8	48	5	18
8	13	1ffd	49	4	9
9	12	ffd	50	5	1b
10	11	7f1	51	8	eb
11	10	3e8	52	9	1e9
12	9	1e8	53	11	7ec
13	8	f0	54	11	7f6
14	9	1ec	55	10	3eb
15	10	3ee	56	9	1f3
16	11	7f2	57	8	ed
17	12	ffa	58	7	72
18	12	ff4	59	8	e9
19	10	3ef	60	9	1f1
20	9	1f2	61	10	3ed
21	8	e8	62	11	7f7
22	7	70	63	12	ff6
23	8	ec	64	11	7f0
24	9	1f0	65	10	3e9
25	10	3ea	66	9	1ed
26	11	7f3	67	8	f1
27	11	7eb	68	9	1ea
28	9	1eb	69	10	3ec

29	8	ea	70	11	7f8
30	5	1a	71	12	ff9
31	4	8	72	13	1ffc
32	5	19	73	12	ffc
33	8	ee	74	12	ff5
34	9	1ef	75	11	7ea
35	11	7ed	76	10	3f3
36	10	3f0	77	10	3f2
37	8	f2	78	11	7f5
38	7	73	79	12	ffb
39	4	b	80	13	1ffe
40	1	0			

Table A.7 – Spectrum Huffman Codebook 6

index	**length**	**codeword (hexadecimal)**	**index**	**length**	**codeword (hexadecimal)**
0	11	7fe	41	4	3
1	10	3fd	42	6	2f
2	9	1f1	43	7	73
3	9	1eb	44	9	1fa
4	9	1f4	45	9	1e7
5	9	1ea	46	7	6e
6	9	1f0	47	6	2b
7	10	3fc	48	4	7
8	11	7fd	49	4	1
9	10	3f6	50	4	5
10	9	1e5	51	6	2c
11	8	ea	52	7	6d
12	7	6c	53	9	1ec
13	7	71	54	9	1f9
14	7	68	55	8	ee
15	8	f0	56	6	30
16	9	1e6	57	6	24
17	10	3f7	58	6	2a
18	9	1f3	59	6	25
19	8	ef	60	6	33
20	6	32	61	8	ec
21	6	27	62	9	1f2
22	6	28	63	10	3f8
23	6	26	64	9	1e4
24	6	31	65	8	ed
25	8	eb	66	7	6a
26	9	1f7	67	7	70
27	9	1e8	68	7	69
28	7	6f	69	7	74
29	6	2e	70	8	f1
30	4	8	71	10	3fa
31	4	4	72	11	7ff
32	4	6	73	10	3f9
33	6	29	74	9	1f6
34	7	6b	75	9	1ed
35	9	1ee	76	9	1f8
36	9	1ef	77	9	1e9
37	7	72	78	9	1f5
38	6	2d	79	10	3fb
39	4	2	80	11	7fc

40	4	0			

Table A.8 – Spectrum Huffman Codebook 7

index	**length**	**codeword (hexadecimal)**	**index**	**length**	**codeword (hexadecimal)**
0	1	0	32	8	f3
1	3	5	33	8	ed
2	6	37	34	9	1e8
3	7	74	35	9	1ef
4	8	f2	36	10	3ef
5	9	1eb	37	10	3f1
6	10	3ed	38	10	3f9
7	11	7f7	39	11	7fb
8	3	4	40	9	1ed
9	4	c	41	8	ef
10	6	35	42	9	1ea
11	7	71	43	9	1f2
12	8	ec	44	10	3f3
13	8	ee	45	10	3f8
14	9	1ee	46	11	7f9
15	9	1f5	47	11	7fc
16	6	36	48	10	3ee
17	6	34	49	9	1ec
18	7	72	50	9	1f4
19	8	ea	51	10	3f4
20	8	f1	52	10	3f7
21	9	1e9	53	11	7f8
22	9	1f3	54	12	ffd
23	10	3f5	55	12	ffe
24	7	73	56	11	7f6
25	7	70	57	10	3f0
26	8	eb	58	10	3f2
27	8	f0	59	10	3f6
28	9	1f1	60	11	7fa
29	9	1f0	61	11	7fd
30	10	3ec	62	12	ffc
31	10	3fa	63	12	fff

Table A.9 – Spectrum Huffman Codebook 8

index	**length**	**codeword (hexadecimal)**	**index**	**length**	**codeword (hexadecimal)**
0	5	e	32	7	71
1	4	5	33	6	2b
2	5	10	34	6	2d
3	6	30	35	6	31
4	7	6f	36	7	6d
5	8	f1	37	7	70
6	9	1fa	38	8	f2
7	10	3fe	39	9	1f9
8	4	3	40	8	ef
9	3	0	41	7	68
10	4	4	42	6	33
11	5	12	43	7	6b
12	6	2c	44	7	6e
13	7	6a	45	8	ee

14	7	75	46	8	f9
15	8	f8	47	10	3fc
16	5	f	48	9	1f8
17	4	2	49	7	74
18	4	6	50	7	73
19	5	14	51	8	ed
20	6	2e	52	8	f0
21	7	69	53	8	f6
22	7	72	54	9	1f6
23	8	f5	55	9	1fd
24	6	2f	56	10	3fd
25	5	11	57	8	f3
26	5	13	58	8	f4
27	6	2a	59	8	f7
28	6	32	60	9	1f7
29	7	6c	61	9	1fb
30	8	ec	62	9	1fc
31	8	fa	63	10	3ff

Table A.10 – Spectrum Huffman Codebook 9

index	**length**	**codeword (hexadecimal)**	**index**	**length**	**codeword (hexadecimal)**
0	1	0	85	12	fda
1	3	5	86	12	fe3
2	6	37	87	12	fe9
3	8	e7	88	13	1fe6
4	9	1de	89	13	1ff3
5	10	3ce	90	13	1ff7
6	10	3d9	91	11	7d3
7	11	7c8	92	10	3d8
8	11	7cd	93	10	3e1
9	12	fc8	94	11	7d4
10	12	fdd	95	11	7d9
11	13	1fe4	96	12	fd3
12	13	1fec	97	12	fde
13	3	4	98	13	1fdd
14	4	c	99	13	1fd9
15	6	35	100	13	1fe2
16	7	72	101	13	1fea
17	8	ea	102	13	1ff1
18	8	ed	103	13	1ff6
19	9	1e2	104	11	7d2
20	10	3d1	105	10	3d4
21	10	3d3	106	10	3da
22	10	3e0	107	11	7c7
23	11	7d8	108	11	7d7
24	12	fcf	109	11	7e2
25	12	fd5	110	12	fce
26	6	36	111	12	fdb
27	6	34	112	13	1fd8
28	7	71	113	13	1fee
29	8	e8	114	14	3ff0
30	8	ec	115	13	1ff4
31	9	1e1	116	14	3ff2
32	10	3cf	117	11	7e1
33	10	3dd	118	10	3df

34	10	3db	119	11	7c9
35	11	7d0	120	11	7d6
36	12	fc7	121	12	fca
37	12	fd4	122	12	fd0
38	12	fe4	123	12	fe5
39	8	e6	124	12	fe6
40	7	70	125	13	1feb
41	8	e9	126	13	1fef
42	9	1dd	127	14	3ff3
43	9	1e3	128	14	3ff4
44	10	3d2	129	14	3ff5
45	10	3dc	130	12	fe0
46	11	7cc	131	11	7ce
47	11	7ca	132	11	7d5
48	11	7de	133	12	fc6
49	12	fd8	134	12	fd1
50	12	fea	135	12	fe1
51	13	1fdb	136	13	1fe0
52	9	1df	137	13	1fe8
53	8	eb	138	13	1ff0
54	9	1dc	139	14	3ff1
55	9	1e6	140	14	3ff8
56	10	3d5	141	14	3ff6
57	10	3de	142	15	7ffc
58	11	7cb	143	12	fe8
59	11	7dd	144	11	7df
60	11	7dc	145	12	fc9
61	12	fcd	146	12	fd7
62	12	fe2	147	12	fdc
63	12	fe7	148	13	1fdc
64	13	1fe1	149	13	1fdf
65	10	3d0	150	13	1fed
66	9	1e0	151	13	1ff5
67	9	1e4	152	14	3ff9
68	10	3d6	153	14	3ffb
69	11	7c5	154	15	7ffd
70	11	7d1	155	15	7ffe
71	11	7db	156	13	1fe7
72	12	fd2	157	12	fcc
73	11	7e0	158	12	fd6
74	12	fd9	159	12	fdf
75	12	feb	160	13	1fde
76	13	1fe3	161	13	1fda
77	13	1fe9	162	13	1fe5
78	11	7c4	163	13	1ff2
79	9	1e5	164	14	3ffa
80	10	3d7	165	14	3ff7
81	11	7c6	166	14	3ffc
82	11	7cf	167	14	3ffd
83	11	7da	168	15	7fff
84	12	fcb			

Table A.11 – Spectrum Huffman Codebook 10

index	**length**	**codeword** (hexadecimal**)**	**index**	**length**	**codeword** **(**hexadecimal**)**
0	6	22	85	9	1c7

1	5	8	86	9	1ca
2	6	1d	87	9	1e0
3	6	26	88	10	3db
4	7	5f	89	10	3e8
5	8	d3	90	11	7ec
6	9	1cf	91	9	1e3
7	10	3d0	92	8	d2
8	10	3d7	93	8	cb
9	10	3ed	94	8	d0
10	11	7f0	95	8	d7
11	11	7f6	96	8	db
12	12	ffd	97	9	1c6
13	5	7	98	9	1d5
14	4	0	99	9	1d8
15	4	1	100	10	3ca
16	5	9	101	10	3da
17	6	20	102	11	7ea
18	7	54	103	11	7f1
19	7	60	104	9	1e1
20	8	d5	105	8	d4
21	8	dc	106	8	cf
22	9	1d4	107	8	d6
23	10	3cd	108	8	de
24	10	3de	109	8	e1
25	11	7e7	110	9	1d0
26	6	1c	111	9	1d6
27	4	2	112	10	3d1
28	5	6	113	10	3d5
29	5	c	114	10	3f2
30	6	1e	115	11	7ee
31	6	28	116	11	7fb
32	7	5b	117	10	3e9
33	8	cd	118	9	1cd
34	8	d9	119	9	1c8
35	9	1ce	120	9	1cb
36	9	1dc	121	9	1d1
37	10	3d9	122	9	1d7
38	10	3f1	123	9	1df
39	6	25	124	10	3cf
40	5	b	125	10	3e0
41	5	a	126	10	3ef
42	5	d	127	11	7e6
43	6	24	128	11	7f8
44	7	57	129	12	ffa
45	7	61	130	10	3eb
46	8	cc	131	9	1dd
47	8	dd	132	9	1d3
48	9	1cc	133	9	1d9
49	9	1de	134	9	1db
50	10	3d3	135	10	3d2
51	10	3e7	136	10	3cc
52	7	5d	137	10	3dc
53	6	21	138	10	3ea
54	6	1f	139	11	7ed
55	6	23	140	11	7f3
56	6	27	141	11	7f9
57	7	59	142	12	ff9

58	7	64	143	11	7f2
59	8	d8	144	10	3ce
60	8	df	145	9	1e4
61	9	1d2	146	10	3cb
62	9	1e2	147	10	3d8
63	10	3dd	148	10	3d6
64	10	3ee	149	10	3e2
65	8	d1	150	10	3e5
66	7	55	151	11	7e8
67	6	29	152	11	7f4
68	7	56	153	11	7f5
69	7	58	154	11	7f7
70	7	62	155	12	ffb
71	8	ce	156	11	7fa
72	8	e0	157	10	3ec
73	8	e2	158	10	3df
74	9	1da	159	10	3e1
75	10	3d4	160	10	3e4
76	10	3e3	161	10	3e6
77	11	7eb	162	10	3f0
78	9	1c9	163	11	7e9
79	7	5e	164	11	7ef
80	7	5a	165	12	ff8
81	7	5c	166	12	ffe
82	7	63	167	12	ffc
83	8	ca	168	12	fff
84	8	da			

Table A.12 – Spectrum Huffman Codebook 11

index	**length**	**codeword (hexadecimal)**	**index**	**length**	**codeword (hexadecimal)**
0	4	0	145	10	38d
1	5	6	146	10	398
2	6	19	147	10	3b7
3	7	3d	148	10	3d3
4	8	9c	149	10	3d1
5	8	c6	150	10	3db
6	9	1a7	151	11	7dd
7	10	390	152	8	b4
8	10	3c2	153	10	3de
9	10	3df	154	9	1a9
10	11	7e6	155	9	19b
11	11	7f3	156	9	19c
12	12	ffb	157	9	1a1
13	11	7ec	158	9	1aa
14	12	ffa	159	9	1ad
15	12	ffe	160	9	1b3
16	10	38e	161	10	38b
17	5	5	162	10	3b2
18	4	1	163	10	3b8
19	5	8	164	10	3ce
20	6	14	165	10	3e1
21	7	37	166	10	3e0
22	7	42	167	11	7d2
23	8	92	168	11	7e5
24	8	af	169	8	b7

25	9	191	170	11	7e3
26	9	1a5	171	9	1bb
27	9	1b5	172	9	1a8
28	10	39e	173	9	1a6
29	10	3c0	174	9	1b0
30	10	3a2	175	9	1b2
31	10	3cd	176	9	1b7
32	11	7d6	177	10	39b
33	8	ae	178	10	39a
34	6	17	179	10	3ba
35	5	7	180	10	3b5
36	5	9	181	10	3d6
37	6	18	182	11	7d7
38	7	39	183	10	3e4
39	7	40	184	11	7d8
40	8	8e	185	11	7ea
41	8	a3	186	8	ba
42	8	b8	187	11	7e8
43	9	199	188	10	3a0
44	9	1ac	189	9	1bd
45	9	1c1	190	9	1b4
46	10	3b1	191	10	38a
47	10	396	192	9	1c4
48	10	3be	193	10	392
49	10	3ca	194	10	3aa
50	8	9d	195	10	3b0
51	7	3c	196	10	3bc
52	6	15	197	10	3d7
53	6	16	198	11	7d4
54	6	1a	199	11	7dc
55	7	3b	200	11	7db
56	7	44	201	11	7d5
57	8	91	202	11	7f0
58	8	a5	203	8	c1
59	8	be	204	11	7fb
60	9	196	205	10	3c8
61	9	1ae	206	10	3a3
62	9	1b9	207	10	395
63	10	3a1	208	10	39d
64	10	391	209	10	3ac
65	10	3a5	210	10	3ae
66	10	3d5	211	10	3c5
67	8	94	212	10	3d8
68	8	9a	213	10	3e2
69	7	36	214	10	3e6
70	7	38	215	11	7e4
71	7	3a	216	11	7e7
72	7	41	217	11	7e0
73	8	8c	218	11	7e9
74	8	9b	219	11	7f7
75	8	b0	220	9	190
76	8	c3	221	11	7f2
77	9	19e	222	10	393
78	9	1ab	223	9	1be
79	9	1bc	224	9	1c0
80	10	39f	225	10	394
81	10	38f	226	10	397

82	10	3a9	227	10	3ad
83	10	3cf	228	10	3c3
84	8	93	229	10	3c1
85	8	bf	230	10	3d2
86	7	3e	231	11	7da
87	7	3f	232	11	7d9
88	7	43	233	11	7df
89	7	45	234	11	7eb
90	8	9e	235	11	7f4
91	8	a7	236	11	7fa
92	8	b9	237	9	195
93	9	194	238	11	7f8
94	9	1a2	239	10	3bd
95	9	1ba	240	10	39c
96	9	1c3	241	10	3ab
97	10	3a6	242	10	3a8
98	10	3a7	243	10	3b3
99	10	3bb	244	10	3b9
100	10	3d4	245	10	3d0
101	8	9f	246	10	3e3
102	9	1a0	247	10	3e5
103	8	8f	248	11	7e2
104	8	8d	249	11	7de
105	8	90	250	11	7ed
106	8	98	251	11	7f1
107	8	a6	252	11	7f9
108	8	b6	253	11	7fc
109	8	c4	254	9	193
110	9	19f	255	12	ffd
111	9	1af	256	10	3dc
112	9	1bf	257	10	3b6
113	10	399	258	10	3c7
114	10	3bf	259	10	3cc
115	10	3b4	260	10	3cb
116	10	3c9	261	10	3d9
117	10	3e7	262	10	3da
118	8	a8	263	11	7d3
119	9	1b6	264	11	7e1
120	8	ab	265	11	7ee
121	8	a4	266	11	7ef
122	8	aa	267	11	7f5
123	8	b2	268	11	7f6
124	8	c2	269	12	ffc
125	8	c5	270	12	fff
126	9	198	271	9	19d
127	9	1a4	272	9	1c2
128	9	1b8	273	8	b5
129	10	38c	274	8	a1
130	10	3a4	275	8	96
131	10	3c4	276	8	97
132	10	3c6	277	8	95
133	10	3dd	278	8	99
134	10	3e8	279	8	a0
135	8	ad	280	8	a2
136	10	3af	281	8	ac
137	9	192	282	8	a9
138	8	bd	283	8	b1

139	8	bc	284	8	b3
140	9	18e	285	8	bb
141	9	197	286	8	c0
142	9	19a	287	9	18f
143	9	1a3	288	5	4
144	9	1b1			

Table A.13 – *Kaiser-Bessel window for SSR profile EIGHT_SHORT_SEQUENCE*

i	*w(i)*	*i*	*w(i)*
0	0.0000875914060105	16	0.7446454751465113
1	0.0009321760265333	17	0.8121892962974020
2	0.0032114611466596	18	0.8683559394406505
3	0.0081009893216786	19	0.9125649996381605
4	0.0171240286619181	20	0.9453396205809574
5	0.0320720743527833	21	0.9680864942677585
6	0.0548307856028528	22	0.9827581789763112
7	0.0871361822564870	23	0.9914756203467121
8	0.1302923415174603	24	0.9961964092194694
9	0.1848955425508276	25	0.9984956609571091
10	0.2506163195331889	26	0.9994855586984285
11	0.3260874142923209	27	0.9998533730714648
12	0.4089316830907141	28	0.9999671864476404
13	0.4959414909423747	29	0.9999948432453556
14	0.5833939894958904	30	0.9999995655238333
15	0.6674601983218376	31	0.9999999961638728

Table A.14 – *Kaiser-Bessel window for SSR profile for other window sequences.*

i	*w(i)*	*i*	*w(i)*
0	0.0005851230124487	128	0.7110428359000029
1	0.0009642149851497	129	0.7188474364707993
2	0.0013558207534965	130	0.7265597347077880
3	0.0017771849644394	131	0.7341770687621900
4	0.0022352533849672	132	0.7416968783634273
5	0.0027342299070304	133	0.7491167073477523
6	0.0032773001022195	134	0.7564342060337386
7	0.0038671998069216	135	0.7636471334404891
8	0.0045064443384152	136	0.7707533593446514
9	0.0051974336885144	137	0.7777508661725849
10	0.0059425050016407	138	0.7846377507242818
11	0.0067439602523141	139	0.7914122257259034
12	0.0076040812644888	140	0.7980726212080798
13	0.0085251378135895	141	0.8046173857073919
14	0.0095093917383048	142	0.8110450872887550
15	0.0105590986429280	143	0.8173544143867162
16	0.0116765080854300	144	0.8235441764639875
17	0.0128638627792770	145	0.8296133044858474
18	0.0141233971318631	146	0.8355608512093652
19	0.0154573353235409	147	0.8413859912867303
20	0.0168678890600951	148	0.8470880211822968
21	0.0183572550877256	149	0.8526663589032990
22	0.0199276125319803	150	0.8581205435445334
23	0.0215811201042484	151	0.8634502346476508
24	0.0233199132076965	152	0.8686552113760616
25	0.0251461009666641	153	0.8737353715068081
26	0.0270617631981826	154	0.8786907302411250
27	0.0290689473405856	155	0.8835214188357692

28	0.0311696653515848	156	0.8882276830575707
29	0.0333658905863535	157	0.8928098814640207
30	0.0356595546648444	158	0.8972684835130879
31	0.0380525443366107	159	0.9016040675058185
32	0.0405466983507029	160	0.9058173183656508
33	0.0431438043376910	161	0.9099090252587376
34	0.0458455957104702	162	0.9138800790599416
35	0.0486537485902075	163	0.9177314696695282
36	0.0515698787635492	164	0.9214642831859411
37	0.0545955386770205	165	0.9250796989403991
38	0.0577322144743916	166	0.9285789863994010
39	0.0609813230826460	167	0.9319635019415643
40	0.0643442093520723	168	0.9352346855155568
41	0.0678221432558827	169	0.9383940571861993
42	0.0714163171546603	170	0.9414432135761304
43	0.0751278431308314	171	0.9443838242107182
44	0.0789577503982528	172	0.9472176277741918
45	0.0829069827918993	173	0.9499464282852282
46	0.0869763963425241	174	0.9525720912004834
47	0.0911667569410503	175	0.9550965394547873
48	0.0954787380973307	176	0.9575217494469370
49	0.0999129187977865	177	0.9598497469802043
50	0.1044697814663005	178	0.9620826031668507
51	0.1091497100326053	179	0.9642224303060783
52	0.1139529881122542	180	0.9662713777449607
53	0.1188797973021148	181	0.9682316277319895
54	0.1239302155951605	182	0.9701053912729269
55	0.1291042159181728	183	0.9718949039986892
56	0.1344016647957880	184	0.9736024220549734
57	0.1398223211441467	185	0.9752302180233160
58	0.1453658351972151	186	0.9767805768831932
59	0.1510317475686540	187	0.9782557920246753
60	0.1568194884519144	188	0.9796581613210076
61	0.1627283769610327	189	0.9809899832703159
62	0.1687576206143887	190	0.9822535532154261
63	0.1749063149634756	191	0.9834511596505429
64	0.1811734433685097	192	0.9845850806232530
65	0.1875578769224857	193	0.9856575802399989
66	0.1940583745250518	194	0.9866709052828243
67	0.2006735831073503	195	0.9876272819448033
68	0.2074020380087318	196	0.9885289126911557
69	0.2142421635060113	197	0.9893779732525968
70	0.2211922734956977	198	0.9901766097569984
71	0.2282505723293797	199	0.9909269360049311
72	0.2354151558022098	200	0.9916310308941294
73	0.2426840122941792	201	0.9922909359973702
74	0.2500550240636293	202	0.9929086532976777
75	0.2575259686921987	203	0.9934861430841844
76	0.2650945206801527	204	0.9940253220113651
77	0.2727582531907993	205	0.9945280613237534
78	0.2805146399424422	206	0.9949961852476154
79	0.2883610572460804	207	0.9954314695504363
80	0.2962947861868143	208	0.9958356402684387
81	0.3043130149466800	209	0.9962103726017252
82	0.3124128412663888	210	0.9965572899760172
83	0.3205912750432127	211	0.9968779632693499
84	0.3288452410620226	212	0.9971739102014799
85	0.3371715818562547	213	0.9974465948831872
86	0.3455670606953511	214	0.9976974275220812

87	0.3540283646950029	215	0.9979277642809907
88	0.3625521080463003	216	0.9981389072844972
89	0.3711348353596863	217	0.9983321047686901
90	0.3797730251194006	218	0.9985085513687731
91	0.3884630932439016	219	0.9986693885387259
92	0.3972013967475546	220	0.9988157050968516
93	0.4059842374986933	221	0.9989485378906924
94	0.4148078660689724	222	0.9990688725744943
95	0.4236684856687616	223	0.9991776444921379
96	0.4325622561631607	224	0.9992757396582338
97	0.4414852981630577	225	0.9993639958299003
98	0.4504336971855032	226	0.9994432036616085
99	0.4594035078775303	227	0.9995141079353859
100	0.4683907582974173	228	0.9995774088586188
101	0.4773914542472655	229	0.9996337634216871
102	0.4864015836506502	230	0.9996837868076957
103	0.4954171209689973	231	0.9997280538466377
104	0.5044340316502417	232	0.9997671005064359
105	0.5134482766032377	233	0.9998014254134544
106	0.5224558166913167	234	0.9998314913952471
107	0.5314526172383208	235	0.9998577270385304
108	0.5404346525403849	236	0.9998805282555989
109	0.5493979103766972	237	0.9999002598526793
110	0.5583383965124314	238	0.9999172570940037
111	0.5672521391870222	239	0.9999318272557038
112	0.5761351935809411	240	0.9999442511639580
113	0.5849836462541291	241	0.9999547847121726
114	0.5937936195492526	242	0.9999636603523446
115	0.6025612759529649	243	0.9999710885561258
116	0.6112828224083939	244	0.9999772592414866
117	0.6199545145721097	245	0.9999823431612708
118	0.6285726610088878	246	0.9999864932503106
119	0.6371336273176413	247	0.9999898459281599
120	0.6456338401819751	248	0.9999925223548691
121	0.6540697913388968	249	0.9999946296375997
122	0.6624380414593221	250	0.9999962619864214
123	0.6707352239341151	251	0.9999975018180320
124	0.6789580485595255	252	0.9999984208055542
125	0.6871033051160131	253	0.9999990808746198
126	0.6951678668345944	254	0.9999995351446231
127	0.7031486937449871	255	0.9999998288155155

5 Annex B

Table B.1 transition table 0 (differential scalefactor to index)

DIFF	INDEX	DIFF	INDEX	DIFF	INDEX	DIFF	INDEX	DIFF	INDEX	DIFF	INDEX	DIFF	INDEX	DIFF	INDEX
0	68	16	87	32	46	48	25	64	9	80	40	96	96	112	112
1	69	17	88	33	47	49	19	65	10	81	43	97	97	113	113
2	70	18	89	34	48	50	20	66	12	82	44	98	98	114	114
3	71	19	72	35	49	51	14	67	13	83	45	99	99	115	115
4	75	20	90	36	50	52	15	68	17	84	52	100	100	116	116
5	76	21	73	37	51	53	16	69	18	85	53	101	101	117	117
6	77	22	65	38	41	54	11	70	21	86	63	102	102	118	118
7	78	23	66	39	42	55	7	71	22	87	56	103	103	119	119
8	79	24	58	40	35	56	8	72	26	88	64	104	104	120	120

9	80	25	67	41	36	57	5	73	27	89	57	105	105	121	121
10	81	26	59	42	37	58	2	74	28	90	74	106	106	122	122
11	82	27	60	43	29	59	1	75	31	91	91	107	107	123	123
12	83	28	61	44	38	60	0	76	32	92	92	108	108	124	124
13	84	29	62	45	30	61	3	77	33	93	93	109	109	125	125
14	85	30	54	46	23	62	4	78	34	94	94	110	110	126	126
15	86	31	55	47	24	63	6	79	39	95	95	111	111	127	127

Table B.2 transition table 1 (index to differential scalefactor)

INDEX	DIFF	INDEX	DIFF	INDEX	DIFF	INDEX	DIFF	INDEX	DIFF	INDEX	DIFF	INDEX	DIFF	INDEX	DIFF
0	60	16	53	32	76	48	34	64	88	80	9	96	96	112	112
1	59	17	68	33	77	49	35	65	22	81	10	97	97	113	113
2	58	18	69	34	78	50	36	66	23	82	11	98	98	114	114
3	61	19	49	35	40	51	37	67	25	83	12	99	99	115	115
4	62	20	50	36	41	52	84	68	0	84	13	100	100	116	116
5	57	21	70	37	42	53	85	69	1	85	14	101	101	117	117
6	63	22	71	38	44	54	30	70	2	86	15	102	102	118	118
7	55	23	46	39	79	55	31	71	3	87	16	103	103	119	119
8	56	24	47	40	80	56	87	72	19	88	17	104	104	120	120
9	64	25	48	41	38	57	89	73	21	89	18	105	105	121	121
10	65	26	72	42	39	58	24	74	90	90	20	106	106	122	122
11	54	27	73	43	81	59	26	75	4	91	91	107	107	123	123
12	66	28	74	44	82	60	27	76	5	92	92	108	108	124	124
13	67	29	43	45	83	61	28	77	6	93	93	109	109	125	125
14	51	30	45	46	32	62	29	78	7	94	94	110	110	126	126
15	52	31	75	47	33	63	86	79	8	95	95	111	111	127	127

Table B.3 differential scalefactor arithmetic model 0

size	cumulative frequencies
55	8192, 6144, 5120, 4096, 3072, 2560, 2048, 1792, 1536, 1280, 1024, 896, 768, 640, 576, 512, 448, 384, 320, 288, 256, 224, 192, 176, 160, 144, 128, 112, 96, 88, 80, 72, 64, 56, 48, 44, 40, 36, 32, 28, 24, 22, 20, 18, 16, 14, 13, 12, 11, 10, 9, 8, 7, 6, 0,

Table B.4 differential scalefactor arithmetic model 1

size	cumulative frequencies
67	15018, 13653, 12288, 10922, 10240, 9557, 8874, 8192, 7509, 6826, 6144, 5802, 5461, 5120, 4949, 4778, 4608, 4437, 4266, 4096, 3925, 3840, 3754, 3669, 3584, 3498, 3413, 3328, 3242, 3157, 3072, 2986, 2901, 2816, 2730, 2645, 2560, 2474, 2389, 2304, 2218, 2133, 2048, 1962, 1877, 1792, 1706, 1621, 1536, 1450, 1365, 1280, 1194, 1109, 1024, 938, 853, 768, 682, 597, 512, 426, 341, 256, 170, 85, 0,

Table B.5 differential scalefactor arithmetic model 2

size	cumulative frequencies
8	1342, 790, 510, 344, 214, 127, 57, 0,

Table B.6 differential scalefactor arithmetic model 3

size	cumulative frequencies
16	2441, 2094, 1798, 1563, 1347, 1154, 956, 818, 634, 464, 342, 241, 157, 97, 55, 0,

Table B.7 differential scalefactor arithmetic model 4

size	cumulative frequencies
32	3963, 3525, 3188, 2949, 2705, 2502, 2286, 2085, 1868, 1668, 1515, 1354, 1207, 1055, 930, 821, 651, 510, 373, 269, 192, 134, 90, 58, 37, 29, 24, 15, 10, 8, 5, 0,

Table B.8 differential scalefactor arithmetic model 5

size	cumulative frequencies
64	13587, 13282, 12961, 12656, 12165, 11721, 11250, 10582, 10042, 9587, 8742, 8010, 7256, 6619, 6042, 5480, 4898, 4331, 3817, 3374, 3058, 2759, 2545, 2363, 2192, 1989, 1812, 1582, 1390, 1165, 1037, 935, 668, 518, 438, 358, 245, 197, 181, 149, 144, 128, 122, 117, 112, 106, 101, 85, 80, 74, 69, 64, 58, 53, 48, 42, 37, 32, 26, 21, 16, 10, 5, 0,

Table B.9 differential ArModel arithmetic model 0

size	cumulative frequencies
4	9868, 3351, 1676, 0,

Table B.10 differential ArModel arithmetic model 1

size	cumulative frequencies
8	12492, 8600, 5941, 3282, 2155, 1028, 514, 0,

Table B.11 differential ArModel arithmetic model 2

size	cumulative frequencies
16	14316, 12248, 9882, 7516, 6399, 5282, 4183, 3083, 2247, 1411, 860, 309, 185, 61, 31, 0,

Table B.12 differential ArModel arithmetic model 3

size	cumulative frequencies
40	12170, 7956, 6429, 4901, 4094, 3287, 2982, 2677, 2454, 2230, 2062, 1894, 1621, 1348, 1199, 1050, 854, 658, 468, 278, 169, 59, 38, 18, 17, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0,

Table B.13 MS_used model

size	cumulative frequencies
2	8192, 0

Table B.14 stereo_info model

size	cumulative frequencies
4	13926, 4096, 1638, 0

Table B.15 sign arithmetic model

size	cumulative frequencies
2	8192, 0

Table B.16 noise_flag arithmetic model

size	cumulative frequencies
2	8192, 0

Table B.17 noise_mode arithmetic model

size	cumulative frequencies
4	12288, 8192,4096, 0

Table B.18 BSAC arithmetic model 0
Allocated bit = 0

BSAC arithmetic model 1
not used

Table B.19 BSAC arithmetic model 2
Allocated bit = 1

snf	pre_state	dimension	cumulative frequencies
1	0	4	14858, 13706, 12545, 11546, 10434, 9479, 8475, 7619, 6457, 5456, 4497, 3601, 2600, 1720, 862, 0,

Table B.20 BSAC arithmetic model 3
Allocated bit = 1

snf	pre_state	dimension	cumulative frequencies
1	0	4	5476, 4279, 3542, 3269, 2545, 2435, 2199, 2111, 850, 739, 592, 550, 165, 132, 21, 0,

Table B.21 BSAC arithmetic model 4
Allocated bit = 2

snf	pre_state	dimension	cumulative frequencies
2	0	4	4292, 3445, 2583, 2473, 1569, 1479, 1371, 1332, 450, 347, 248, 219, 81, 50, 15, 0,
1	0	4	15290, 14389, 13434, 12485, 11559, 10627, 9683, 8626, 7691, 6727, 5767, 4655, 3646, 2533, 1415, 0,
		3	15139, 13484, 11909, 9716, 8068, 5919, 3590, 0,
		2	14008, 10384, 6834, 0,
		1	11228, 0
	1	4	10355, 9160, 7553, 7004, 5671, 4902, 4133, 3433, 1908, 1661, 1345, 1222, 796, 714, 233, 0,
		3	8328, 6615, 4466, 3586, 1759, 1062, 321, 0,
		2	4631, 2696, 793, 0,
		1	968, 0,

Table B.22 BSAC arithmetic model 5

Allocated bit = 2

snf	pre_state	dimension	cumulative frequencies
2	0	4	3119, 2396, 1878, 1619, 1076, 1051, 870, 826, 233, 231, 198, 197, 27, 26, 1, 0,
1	0	4	3691, 2897, 2406, 2142, 1752, 1668, 1497, 1404, 502, 453, 389, 368, 131, 102, 18, 0,
		3	11106, 8393, 6517, 4967, 2739, 2200, 608, 0,
		2	10771, 6410, 2619, 0,
		1	6112, 0
	1	4	11484, 10106, 7809, 7043, 5053, 3521, 2756, 2603, 2296, 2143, 1990, 1531, 765, 459, 153, 0,
		3	10628, 8930, 6618, 4585, 2858, 2129, 796, 0,
		2	7596, 4499, 1512, 0,
		1	4155, 0,

Table B.23 BSAC arithmetic model 6

Allocated bit = 3

snf	pre_state	dimension	cumulative cumulative frequencies
3	0	4	2845, 2371, 1684, 1524, 918, 882, 760, 729, 200, 198, 180, 178, 27, 25, 1, 0,
2	0	4	1621, 1183, 933, 775, 645, 628, 516, 484, 210, 207, 188, 186, 39, 35, 1, 0,
		3	8800, 6734, 4886, 3603, 1326, 1204, 104, 0,
		2	8869, 5163, 1078, 0,
		1	3575, 0,
	1	4	12603, 12130, 10082, 9767, 8979, 8034, 7404, 6144, 4253, 3780, 3150, 2363, 1575, 945, 630, 0,
		3	10410, 8922, 5694, 4270, 2656, 1601, 533, 0,
		2	8459, 5107, 1670, 0
		1	4003, 0,
1	0	4	5185, 4084, 3423, 3010, 2406, 2289, 2169, 2107, 650, 539, 445, 419, 97, 61, 15, 0,
		3	13514, 11030, 8596, 6466, 4345, 3250, 1294, 0,
		2	13231, 8754, 4635, 0,
		1	9876, 0,
	1	4	14091, 12522, 11247, 10299, 8928, 7954, 6696, 6024, 4766, 4033, 3119, 2508, 1594, 1008, 353, 0,
		3	12596, 10427, 7608, 6003, 3782, 2580, 928, 0,
		2	10008, 6213, 2350, 0,
		1	5614, 0,

Table B.24 BSAC arithmetic model 7

Allocated bit = 3

snf	pre_state	dimension	cumulative frequencies
3	0	4	3833, 3187, 2542, 2390, 1676, 1605, 1385, 1337, 468, 434, 377, 349, 117, 93, 30, 0,
2	0	4	6621, 5620, 4784, 4334, 3563, 3307, 2923, 2682, 1700, 1458, 1213, 1040, 608, 431, 191, 0,
		3	11369, 9466, 7519, 6138, 3544, 2441, 1136, 0,
		2	11083, 7446, 3439, 0,
		1	8823, 0,
	1	4	12027, 11572, 9947, 9687, 9232, 8126, 7216, 6176, 4161, 3705, 3055, 2210, 1235, 780, 455, 0,
		3	9566, 7943, 4894, 3847, 2263, 1596, 562, 0,
		2	7212, 4217, 1240, 0,

		1	3296, 0,
1	0	4	14363, 13143, 12054, 11153, 10220, 9388, 8609, 7680, 6344, 5408, 4578, 3623, 2762, 1932, 1099, 0,
		3	14785, 13256, 11596, 9277, 7581, 5695, 3348, 0,
		2	14050, 10293, 6547, 0,
		1	10948, 0,
	1	4	13856, 12350, 11151, 10158, 8816, 7913, 6899, 6214, 4836, 4062, 3119, 2505, 1624, 1020, 378, 0,
		3	12083, 9880, 7293, 5875, 3501, 2372, 828, 0,
		2	8773, 5285, 1799, 0,
		1	4452, 0,

Table B.25 BSAC arithmetic model 8

Allocated bit = 4

snf	pre_state	dimension	cumulative frequencies
4	0	4	2770, 2075, 1635, 1511, 1059, 1055, 928, 923, 204, 202, 190, 188, 9, 8, 1, 0,
3	0	4	1810, 1254, 1151, 1020, 788, 785, 767, 758, 139, 138, 133, 132, 14, 13, 1, 0,
		3	7113, 4895, 3698, 3193, 1096, 967, 97, 0,
		2	6858, 4547, 631, 0,
		1	4028, 0,
	1	4	13263, 10922, 10142, 9752, 8582, 7801, 5851, 5071, 3510, 3120, 2730, 2340, 1560, 780, 390, 0,
		3	12675, 11275, 7946, 6356, 4086, 2875, 1097, 0,
		2	9473, 5781, 1840, 0,
		1	3597, 0,
2	0	4	2600, 1762, 1459, 1292, 989, 983, 921, 916, 238, 233, 205, 202, 32, 30, 3, 0,
		3	10797, 8840, 6149, 5050, 2371, 1697, 483, 0,
		2	10571, 6942, 2445, 0,
		1	7864, 0,
	1	4	14866, 12983, 11297, 10398, 9386, 8683, 7559, 6969, 5451, 4721, 3484, 3007, 1882, 1208, 590, 0,
		3	12611, 10374, 8025, 6167, 4012, 2608, 967, 0,
		2	10043, 6306, 2373, 0,
		1	5766, 0,
1	0	4	6155, 5057, 4328, 3845, 3164, 2977, 2728, 2590, 1341, 1095, 885, 764, 303, 188, 74, 0,
		3	12802, 10407, 8142, 6263, 3928, 3013, 1225, 0,
		2	13131, 9420, 4928, 0,
		1	10395, 0,
	1	4	14536, 13348, 11819, 11016, 9340, 8399, 7135, 6521, 5114, 4559, 3521, 2968, 1768, 1177, 433, 0,
		3	12735, 10606, 7861, 6011, 3896, 2637, 917, 0,
		2	9831, 5972, 2251, 0,
		1	4944, 0,

Table B.26 BSAC arithmetic model 9

Allocated bit = 4

snf	pre_state	dimension	cumulative frequencies
4	0	4	3383, 2550, 1967, 1794, 1301, 1249, 1156, 1118, 340, 298, 247, 213, 81, 54, 15, 0,
3	0	4	7348, 6275, 5299, 4935, 3771, 3605, 2962, 2814, 1295, 1143, 980, 860, 310, 230, 75, 0,
		3	9531, 7809, 5972, 4892, 2774, 1782, 823, 0,

		2	11455, 7068, 3383, 0,
		1	9437, 0,
	1	4	12503, 9701, 8838, 8407, 6898, 6036, 4527, 3664, 2802, 2586, 2371, 2155, 1293, 431, 215, 0,
		3	11268, 9422, 6508, 5277, 3076, 2460, 1457, 0,
		2	7631, 4565, 1506, 0,
		1	2639, 0,
2	0	4	11210, 9646, 8429, 7389, 6252, 5746, 5140, 4692, 3350, 2880, 2416, 2014, 1240, 851, 404, 0,
		3	12143, 10250, 7784, 6445, 3954, 2528, 1228, 0,
		2	10891, 7210, 3874, 0,
		1	9537, 0,
	1	4	14988, 13408, 11860, 10854, 9631, 8992, 7834, 7196, 5616, 4793, 3571, 2975, 1926, 1212, 627, 0,
		3	12485, 10041, 7461, 5732, 3669, 2361, 940, 0,
		2	9342, 5547, 1963, 0,
		1	5140, 0,
1	0	4	14152, 13258, 12486, 11635, 11040, 10290, 9740, 8573, 7546, 6643, 5903, 4928, 4005, 2972, 1751, 0,
		3	14895, 13534, 12007, 9787, 8063, 5761, 3570, 0,
		2	14088, 10108, 6749, 0,
		1	11041, 0,
	1	4	14817, 13545, 12244, 11281, 10012, 8952, 7959, 7136, 5791, 4920, 3997, 3126, 2105, 1282, 623, 0,
		3	12873, 10678, 8257, 6573, 4186, 2775, 1053, 0,
		2	9969, 6059, 2363, 0,
		1	5694, 0,

Table B.27 BSAC arithmetic model 10

Allocated bit (Abit) = 5

snf	pre_state	dimension	cumulative frequencies
Abit	0	4	2335, 1613, 1371, 1277, 901, 892, 841, 833, 141, 140, 130, 129, 24, 23, 1, 0,
Abit-1	0	4	1746, 1251, 1038, 998, 615, 611, 583, 582, 106, 104, 101, 99, 3, 2, 1, 0,
		3	7110, 5230, 4228, 3552, 686, 622, 46, 0,
		2	6101, 2575, 265, 0,
		1	1489, 0,
	1	4	13010, 12047, 11565, 11083, 9637, 8673, 6264, 5782, 4336, 3855, 3373, 2891, 2409, 1927, 963, 0,
		3	10838, 10132, 8318, 7158, 5595, 3428, 2318, 0,
		2	8209, 5197, 1287, 0,
		1	4954, 0,
Abit-2	0	4	2137, 1660, 1471, 1312, 1007, 1000, 957, 951, 303, 278, 249, 247, 48, 47, 1, 0,
		3	9327, 7413, 5073, 4391, 2037, 1695, 205, 0,
		2	8658, 5404, 1628, 0,
		1	5660, 0,
	1	4	13360, 12288, 10727, 9752, 8484, 7899, 7119, 6631, 5363, 3900, 3023, 2535, 1852, 1267, 585, 0,
		3	13742, 11685, 8977, 7230, 5015, 3426, 1132, 0,
		2	10402, 6691, 2828, 0,
		1	5298, 0,
Abit-3	0	4	4124, 3181, 2702, 2519, 1959, 1922, 1733, 1712, 524, 475, 425, 407, 78, 52, 15, 0,
		3	10829, 8581, 6285, 4865, 2539, 1920, 594, 0,

		2	11074, 7282, 3092, 0,
		1	8045, 0,
	1	4	14541, 13343, 11637, 10862, 9328, 8783, 7213, 6517, 5485, 5033, 4115, 3506, 2143, 1555, 509, 0,
		3	13010, 11143, 8682, 7202, 4537, 3297, 1221, 0,
		2	9941, 5861, 2191, 0,
		1	5340, 0,
other snf	0	4	9845, 8235, 7126, 6401, 5551, 5131, 4664, 4320, 2908, 2399, 1879, 1506, 935, 603, 277, 0,
		3	13070, 11424, 9094, 7203, 4771, 3479, 1486, 0,
		2	13169, 9298, 5406, 0,
		1	10371, 0,
	1	4	14766, 13685, 12358, 11442, 10035, 9078, 7967, 7048, 5824, 5006, 4058, 3400, 2350, 1612, 659, 0,
		3	13391, 11189, 8904, 7172, 4966, 3183, 1383, 0,
		2	10280, 6372, 2633, 0,
		1	5419, 0,

Table B.28 BSAC arithmetic model 11

Allocated bit (Abit) = 5

snf	pre_state	dimension	cumulative frequencies
Abit	0	4	2872, 2294, 1740, 1593, 1241, 1155, 1035, 960, 339, 300, 261, 247, 105, 72, 34, 0,
Abit-1	0	4	3854, 3090, 2469, 2276, 1801, 1685, 1568, 1505, 627, 539, 445, 400, 193, 141, 51, 0,
		3	10654, 8555, 6875, 4976, 3286, 2229, 826, 0,
		2	10569, 6180, 2695, 0,
		1	6971, 0,
	1	4	11419, 11170, 10922, 10426, 7943, 6950, 3723, 3475, 1737, 1489, 1241, 992, 744, 496, 248, 0,
		3	11013, 9245, 6730, 4962, 3263, 1699, 883, 0,
		2	6969, 4370, 1366, 0,
		1	3166, 0,
Abit-2	0	4	9505, 8070, 6943, 6474, 5305, 5009, 4290, 4029, 2323, 1911, 1591, 1363, 653, 443, 217, 0,
		3	11639, 9520, 7523, 6260, 4012, 2653, 1021, 0,
		2	12453, 8284, 4722, 0,
		1	9182, 0,
	1	4	13472, 12295, 10499, 9167, 7990, 7464, 6565, 6008, 4614, 3747, 2818, 2477, 1641, 1084, 557, 0,
		3	13099, 10826, 8476, 6915, 4488, 2966, 1223, 0,
		2	9212, 5772, 2053, 0,
		1	4244, 0,
Abit-3	0	4	14182, 12785, 11663, 10680, 9601, 8758, 8135, 7353, 6014, 5227, 4433, 3727, 2703, 1818, 866, 0,
		3	13654, 11814, 9714, 7856, 5717, 3916, 2112, 0,
		2	12497, 8501, 4969, 0,
		1	10296, 0,
	1	4	15068, 13770, 12294, 11213, 10230, 9266, 8439, 7438, 6295, 5368, 4361, 3620, 2594, 1797, 895, 0,
		3	13120, 10879, 8445, 6665, 4356, 2794, 1047, 0,
		2	9311, 5578, 1793, 0,
		1	4695, 0
other snf	0	4	15173, 14794, 14359, 13659, 13224, 12600, 11994, 11067, 10197, 9573, 9081, 7624, 6697, 4691, 3216, 0,
		3	15328, 13985, 12748, 10084, 8587, 6459, 4111, 0,

		2	14661, 11179, 7924, 0,
		1	11399, 0,
	1	4	14873, 13768, 12458, 11491, 10229, 9164, 7999, 7186, 5992, 5012, 4119, 3369, 2228, 1427, 684, 0,
		3	13063, 10913, 8477, 6752, 4529, 3047, 1241, 0,
		2	10101, 6369, 2615, 0,
		1	5359, 0,

Table B.29 BSAC arithmetic model 12
same as BSAC arithmetic model 10, but Allocated bit (Abit) = 6

Table B.30 BSAC arithmetic model 13
same as BSAC arithmetic model 11, but Allocated bit (Abit) = 6

Table B.31 BSAC arithmetic model 14
same as BSAC arithmetic model 10, but Allocated bit (Abit) = 7

Table B.32 BSAC arithmetic model 15
same as BSAC arithmetic model 11, but Allocated bit (Abit) = 7

Table B.33 BSAC arithmetic model 16
same as BSAC arithmetic model 10, but Allocated bit (Abit) = 8

Table B.34 BSAC arithmetic model 17
same as BSAC arithmetic model 11, but Allocated bit (Abit) = 8

Table B.35 BSAC arithmetic model 18
same as BSAC arithmetic model 10, but Allocated bit (Abit) = 9

Table B.36 BSAC arithmetic model 19
same as BSAC arithmetic model 11, but Allocated bit (Abit) = 9

Table B.37 BSAC arithmetic model 20
same as BSAC arithmetic model 10, but Allocated bit (Abit) = 10

Table B.38 BSAC arithmetic model 21
same as BSAC arithmetic model 11, but Allocated bit (Abit) = 10

Table B.39 BSAC arithmetic model 22
same as BSAC arithmetic model 10, but Allocated bit (Abit) = 11

Table B.40 BSAC arithmetic model 23
same as BSAC arithmetic model 11, but Allocated bit (Abit) = 11

Table B.41 BSAC arithmetic model 24
same as BSAC arithmetic model 10, but Allocated bit (Abit) = 12

Table B.42 BSAC arithmetic model 25
same as BSAC arithmetic model 11, but Allocated bit (Abit) = 12

Table B.43 BSAC arithmetic model 26
same as BSAC arithmetic model 10, but Allocated bit (Abit) = 13

Table B.44 BSAC arithmetic model 27
same as BSAC arithmetic model 11, but Allocated bit (Abit) = 13

Table B.45 BSAC arithmetic model 28
same as BSAC arithmetic model 10, but Allocated bit (Abit) = 14

Table B.46 BSAC arithmetic model 29
same as BSAC arithmetic model 11, but Allocated bit (Abit) = 14

Table B.47 BSAC arithmetic model 30
same as BSAC arithmetic model 10, but Allocated bit (Abit) = 15

Table B.48 BSAC arithmetic model 31
same as BSAC arithmetic model 11, but Allocated bit (Abit) = 15

6 Annex C

Contents are available in w1903tvq.doc.