//ISO/JTC 1/SC 29 **N2203TF**

Date: 1998-05-15

ISO/IEC CD 14496-3 Subpart 4

ISO/JTC 1/SC 29/WG11

Secretariat:

Information Technology - Coding of Audiovisual Objects

Part 3: Audio

Subpart 4: Time/Frequency Coding

Document type: International standard
Document:sub-type if applicable
Document:stage (20) Préparation
Document:language E

C:\ISOST18\BASICEN.DOT ISOSTD Basic Version 1.8 1996-10-30

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

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

0 Introduction

The MPEG-4 Audio T/F based coding is mainly intended to be used for generic audio coding at all but the lowest bitrates. Typically, T/F based encoding is used for complex music material from 6 kbit/s per channel and for stereo signals from 16 kbit/s per stereo signal up to broadcast quality audio at 64 kbit/s per channel and more. MPEG-2 Advanced Audio Coding (AAC) syntax (including support for multi-channel audio) is fully supported by MPEG-4 Audio T/F based coding. The tools derived from MPEG-2 AAC are available together with other MPEG-4 T/F based coding tools to code audio objects using MPEG-4 functionality (including scalability). MPEG-4 T/F based coding is not restricted to some fixed bitrates but supports a wide range of bitrates and variable rate coding.

The block diagrams of the T/F based encoder and decoder reflect the structure of MPEG-4 T/F coding. In general, there are the MPEG-2 AAC related tools with MPEG-4 add-ons for some of them and the tools related to the Twin-VQ Quantization and Coding. The Twin-VQ is an alternative module for the AAC-type quantization and it is based on an interleave vector quantization and LPC (Linear Predictive Coding) spectral estimation. It covers from 6 kbit/s to over 40 kbit/s with constant bit rate. This feature provides merits in terms of error robustness, scaleability and random accecss.

While efficient mono, stereo and multi-channel coding is possible using the MPEG-2 AAC tools, the document also provides extensions to this toolset further enhancing compression performance, scaleability based on a core coder, mono/stereo scaleability etc..

In this context, the Bit-sliced Arithmetic Coding (BSAC) scheme provides a possibility for noiseless transcoding of an AAC stream into a fine granule scalable stream between 16 kbit/s to 64 kbit/s per channel.With BSAC, the bit rate control can be done with a stepsize of 1 kbit/s, which enables the decoder to stop anywhere between 16 kbit/s and the encoded bit rate with a 1 kbit/s stepsize.

All the features and possibilities of the MPEG-2 AAC standard also apply to MPEG-4. AAC has been tested to allow for ITU-R 'indistinguishable' quality according to [4] at data rates of 320 kb/s for five full-bandwidth channel audio signals.

0.1 Overview of tools

The basic structure of the MPEG-4 T/F system is shown in Figures 1 and 2. The data flow in this diagram is from left to right, top to bottom. The functions of the decoder are to find the description of the quantized audio spectra in the bitstream, decode the quantized values and other reconstruction information, reconstruct the quantized spectra, process the reconstructed spectra through whatever tools are active in the bitstream in order to arrive at the actual signal spectra as described by the input bitstream, and finally convert the frequency domain spectra to the time domain, with or without an optional gain control tool. Following the initial reconstruction and scaling of the spectrum reconstruction, there are many optional tools that modify one or more of the spectra in order to provide more efficient coding. For each of the optional tools that operate in the spectral domain, the option to "pass through" is retained, and in all cases where a spectral operation is omitted, the spectra at its input are passed directly through the tool without modification.

The input to the <u>bitstream demultiplexer tool</u> is the MPEG-4 T/F bitstream. The demultiplexer separates the bitstream into the parts for each tool, and provides each of the tools with the bitstream information related to that tool.
The outputs from the bitstream demultiplexer tool are:

- The quantized (and optionally noiselessly coded) spectra represented by either
    - the sectioning information and the noiselessly coded spectra (AAC) or
    - the BSAC information or
    - a set of indices of code vectors (TwinVQ)
- The M/S decision information (optional)
- The predictor state information (optional)
- The perceptual noise substitution (PNS) information (optional)
- The intensity stereo control information and coupling channel control information (both optional)
- The temporal noise shaping (TNS) information (optional)
- The filterbank control information
- The gain control information (optional)

The AAC noiseless decoding tool takes information from the bitstream demultiplexer, parses that information, decodes the Huffman coded data, and reconstructs the quantized spectra and the Huffman and DPCM coded scalefactors.
The inputs to the noiseless decoding tool are:

- The sectioning information for the noiselessly coded spectra
- The noiselessly coded spectra

The outputs of the noiseless decoding tool are:

- The decoded integer representation of the scalefactors:
- The quantized values for the spectra

The BSAC tool provides an alternative to the AAC noiseless coding tool, which provides fine granule scalability. This tool takes information from bitstream demultiplexer, parses that information, decodes the Arithmetic coded bit-sliced data, and reconstructs the quantized spectra and the scalefactors.
The inputs to the BSAC decoding tool are:

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

The outputs from the BSAC decoding tool are:

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

The inverse quantizer tool takes the quantized values for the spectra, and converts the integer values to the non-scaled, reconstructed spectra. This quantizer is a non-uniform quantizer.
The input to the Inverse Quantizer tool is:

- The quantized values for the spectra

The output of the inverse quantizer tool is:

- The un-scaled, inversely quantized spectra

The scalefactor tool converts the integer representation of the scalefactors to the actual values, and multiplies the un-scaled inversely quantized spectra by the relevant scalefactors.
The inputs to the scalefactors tool are:

- The decoded integer representation of the scalefactors
- The un-scaled, inversely quantized spectra

The output from the scalefactors tool is:

- The scaled, inversely quantized spectra

The M/S tool converts spectra pairs from Mid/Side to Left/Right under control of the M/S decision information, improving stereo imaging quality and sometimes providing coding efficiency.
The inputs to the M/S tool are:

- The M/S decision information
- The scaled, inversely quantized spectra related to pairs of channels

The output from the M/S tool is:

- The scaled, inversely quantized spectra related to pairs of channels, after M/S decoding

Note: The scaled, inversely quantized spectra of individually coded channels are not processed by the M/S block, rather they are passed directly through the block without modification. If the M/S block is not active, all spectra are passed through this block unmodified.

The prediction tool reverses the prediction process carried out at the encoder. This prediction process re-inserts the redundancy that was extracted by the prediction tool at the encoder, under the control of the predictor state information. This tool is implemented as a second order backward adaptive predictor. The inputs to the prediction tool are:

- The predictor state information
- The scaled, inversely quantized spectra

The output from the prediction tool is:

- The scaled, inversely quantized spectra, after prediction is applied.

Note: If the prediction is disabled, the scaled, inversely quantized spectra are passed directly through the block without modification.

Alternatively, there is a low complexity prediction mode and a long term predictor provided.

The perceptual noise substitution (PNS) tool implements noise substitution decoding on channel spectra by providing an efficient representation for noise-like signal components.
The inputs to the perceptual noise substitution tool are:

- The inversely quantized spectra
- The perceptual noise substitution control information

The output from the perceptual noise substitution tool is:

- The inversely quantized spectra

Note: If either part of this block is disabled, the scaled, inversely quantized spectra are passed directly through this part without modification. If the perceptual noise substitution block is not active, all spectra are passed through this block unmodified.

The intensity stereo / coupling tool implements intensity stereo decoding on pairs of spectra. In addition, it adds the relevant data from a dependently switched coupling channel to the spectra at this point, as directed by the coupling control information.
The inputs to the intensity stereo / coupling tool are:

- The inversely quantized spectra
- The intensity stereo control information and coupling control information

The output from the intensity stereo / coupling tool is:

- The inversely quantized spectra after intensity and coupling channel decoding.

Note: If either part of this block is disabled, the scaled, inversely quantized spectra are passed directly through this part without modification. The intensity stereo tool and M/S tools are arranged so that the operation of M/S and Intensity stereo are mutually exclusive on any given scalefactor band and group of one pair of spectra.

The temporal noise shaping (TNS) tool implements a control of the fine time structure of the coding noise. In the encoder, the TNS process has flattened the temporal envelope of the signal to which it has been applied. In the decoder, the inverse process is used to restore the actual temporal envelope(s), under control of the TNS information. This is done by applying a filtering process to parts of the spectral data.
The inputs to the TNS tool are:

- The inversely quantized spectra
- The TNS information

The output from the TNS block is:

- The inversely quantized spectra

Note: If this block is disabled, the inversely quantized spectra are passed through without modification.

The filterbank tool applies the inverse of the frequency mapping that was carried out in the encoder, as indicated by the filterbank control information and the presence or absence of gain control information. An inverse modified discrete cosine transform (IMDCT) is used for the filterbank tool. If the gain control tool is not used, the IMDCT in the standard AAC mode input consists of either 1024 or 128 spectral coefficients, depending of the value of window_sequence (see 1.3, Table 6.11). If the gain control tool is used, the filterbank tool is configured to use four sets of either 256 or 32 coefficients, depending of the value of window_sequence.
The inputs to the filterbank tool are:

- The inversely quantized spectra

- The filterbank control information

The output(s) from the filterbank tool is (are):

- The time domain reconstructed audio signal(s).

Currently five alternative, but very similar versions of this tool are part of the VM.

a) 1024 or 128 shift length type with the option to select two window shapes (AAC)
b) 4 x switchable 256 or 32 shift length type with the option to select two window shapes (AAC)
c) 2048 or 512 or 128 shift length type with a sine window as defined for TwinVQ
d) 960 or 120 shift length type with the option to select two window shapes (AAC-derived)

When present, the gain control tool applies a separate time domain gain control to each of 4 frequency bands that have been created by the gain control PQF filterbank in the encoder. Then, it assembles the 4 frequency bands and reconstructs the time waveform through the gain control tool's filterbank.
The inputs to the gain control tool are:

- The time domain reconstructed audio signal(s)
- The gain control information

The output(s) from the gain control tool is (are):

- The time domain reconstructed audio signal(s)

If the gain control tool is not active, the time domain reconstructed audio signal(s) are passed directly from the filterbank tool to the output of the decoder. This tool is used for the scaleable sampling rate (SSR) profile only.

The spectral normalisation tool converts the reconstructed flat spectra to the actual values at the decoder. The spectral envelope is specified by LPC coefficients, a Bark scale envelope, periodic pulse components, and gain.
The input to the spectral normalization tool is

- The reconstructed flat spectra

The output from the spectral normalization tool is

- The reconstructed actual spectra

The TwinVQ tool converts the vector index to a flattened spectra at the decoder
by means of table look-up of the codebook and inverse interleaving. Quantization noise is minimized by a weighted distortion measure at the encoder instead of an adaptive bit allocation. This is an alternative to the AAC quantization tool.
The input to the TwinVQ tool is:

- A set of indices of the code vector.

The output from the TwinVQ tool is:

- The reconstructed actual spectra

Besides the above mentioned tools, there are a number of building blocks provided to facilitate scaleable coder configurations, like the scalable controller, frequency selective switch and upsampling filter.

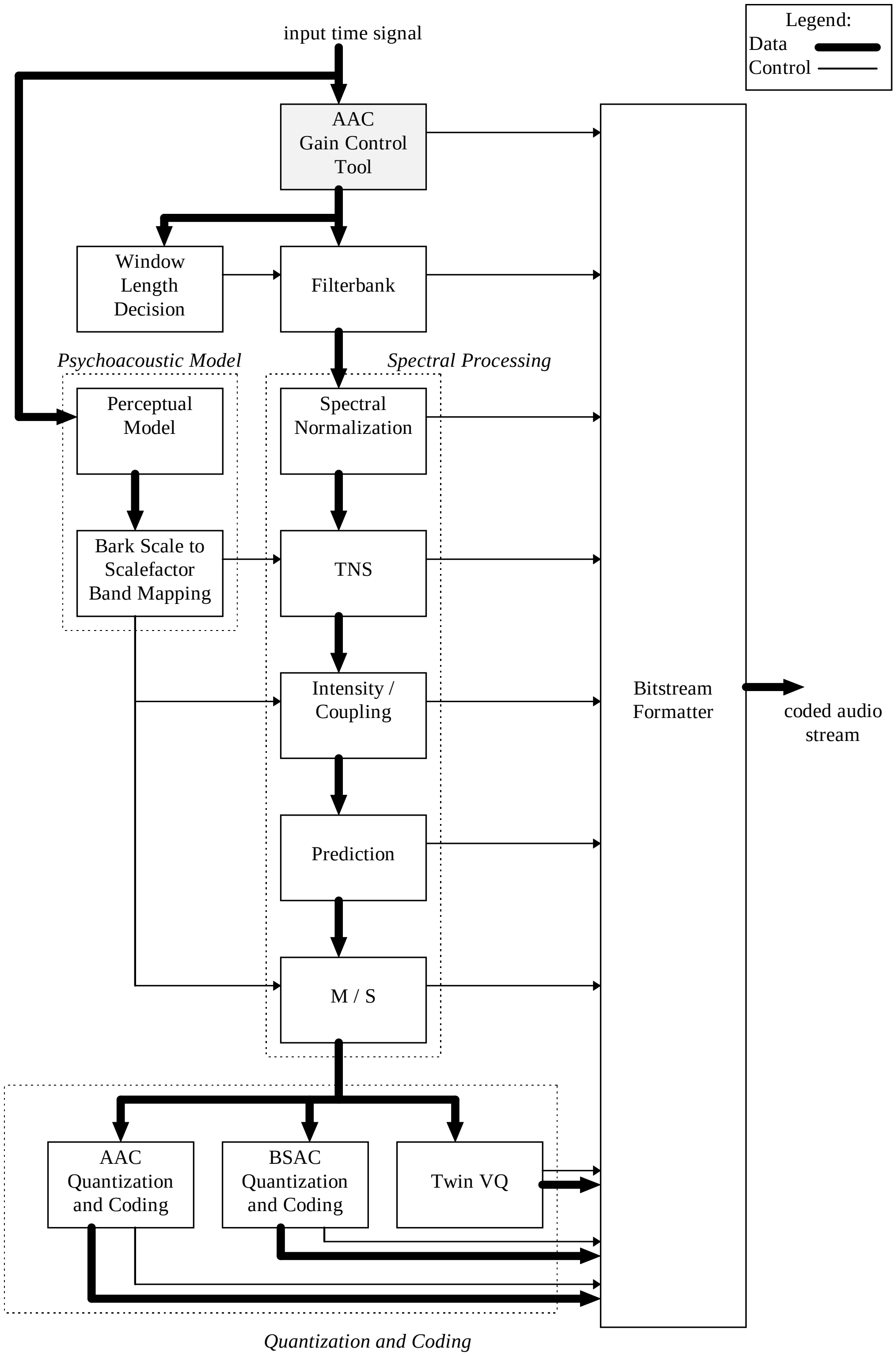


Fig. 1: Blockdiagram TF-based encoder

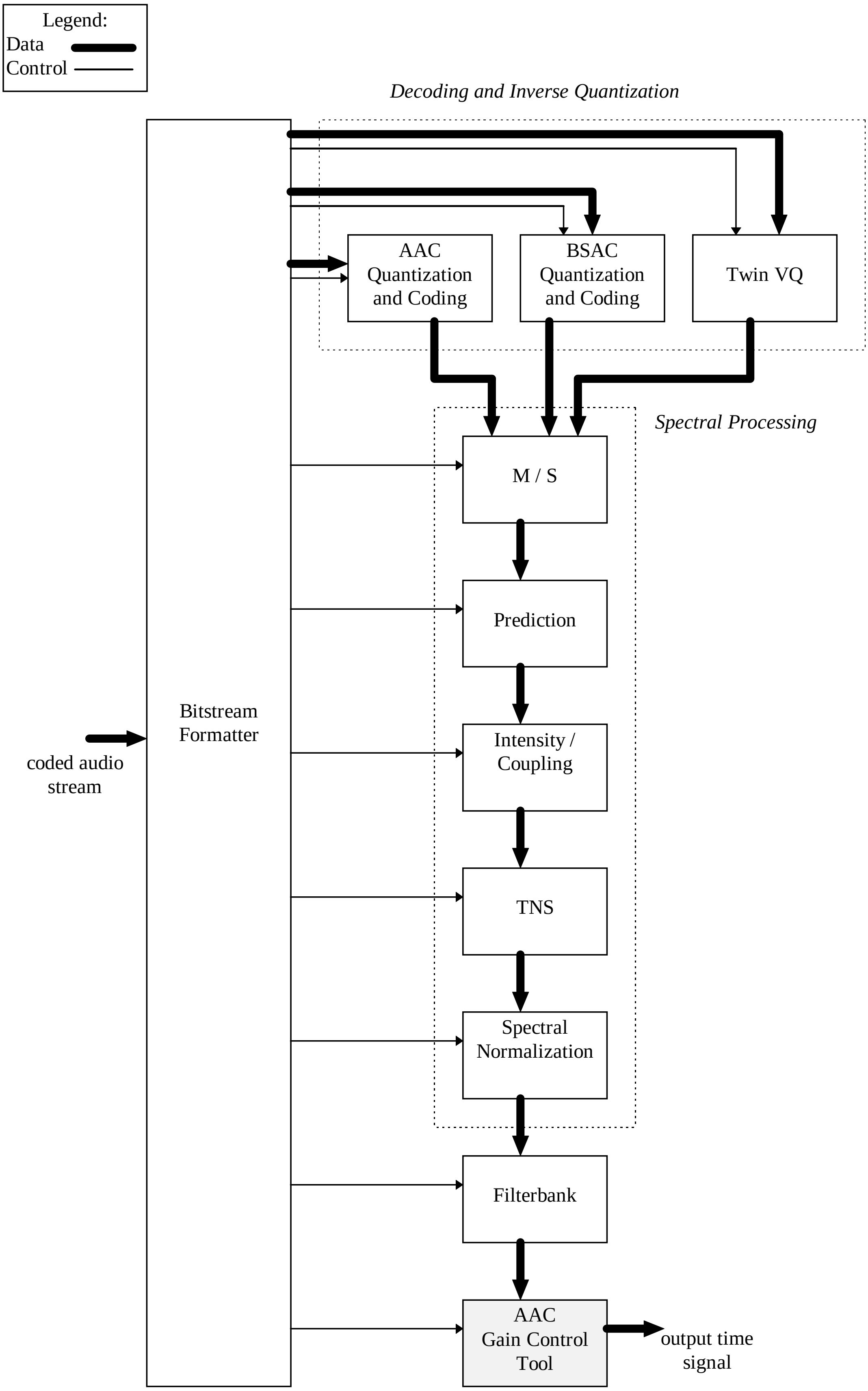


Fig. 2: Blockdiagram of the TF-based decoder

0.2 T/F-specific glossary

0.3 Normative References

[1] ISO/IEC 11172-3:1993, *Information technology - Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s, Part 3: Audio*.

[2] ISO/IEC 13818-3:1997, *Information technology - Generic coding of moving pictures and associated audio, Part 3: Audio.*

[3] M. Bosi, K. Brandenburg, S. Quackenbush, L. Fielder, K. Akagiri, H. Fuchs, M. Dietz, J. Herre, G. Davidson, Y. Oikawa, "ISO/IEC MPEG-2 Advanced Audio Coding", Presented at the 101st AES Convention, Los Angeles, November 1996.

[4] ITU-R Document TG10-2/3- E only, *Basic Audio Quality Requirements for Digital Audio Bit-Rate Reduction Systems for Broadcast Emission and Primary Distribution*, 28 October 1991.

[5] F. J. Harris, *On the Use of Windows For Harmonic Analysis of the Discrete Fourier Transform*, Proc. of the IEEE, Vol. 66, pp. 51- 83, January 1975.

[6] ISO/IEC 13818-1:1996, *Information technology - Generic coding of moving pictures and associated audio: Systems*

1 Syntax

Three types of streams are part of the MPEG-4 T/F coder syntax. These are

1. Interchange format streams
2. Header streams
3. Raw data streams

Raw data streams are intended to be transported via the MPEG-4 Systems layer. These streams contain all information variing on a frame to frame basis and therefore carry the actual audio information.

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

The header and the raw data streams are abstract elements, which define all information for the decoding and parsing of the bitstream. However, for real applications these streams need a transport layer, which cares for the delivery of these streams. Normally this transport mechanism will be handled by MPEG-4 Systems. However, the interface format streams defined in the following section define a simple way of multiplexing the header and the raw data streams.

1.1 Interchange format streams

1.1.1 Audio_Data_Interchange_Format, ADIF

Tabelle 1-1

Syntax	No. of bits	Mnemonic
adif_sequence()		
{		
adif_header()		
raw_data_stream()		

}

Tabelle 1-2 Syntax of adif_header()

Syntax	No. of bits	Mnemonic
adif_header()		
{		
adif_id	**32**	**bslbf**
copyright_id_present	**1**	**bslbf**
if(copyright_id_present)		
copyright_id	**72**	**bslbf**
original_copy	**1**	**bslbf**
home	**1**	**bslbf**
bitstream_type	**1**	**bslbf**
bitrate	**23**	**uimsbf**
num_program_config_elements	**4**	**bslbf**
for (i = 0; i < num_program_config_elements + 1; i++) {		
if(bitstream_type == ‘0’)		
adif_buffer_fullness	**20**	**uimsbf**
program_config_element()		
}		
}		

Tabelle 1-3:Syntax of raw_data_stream()

Syntax	No. of bits	Mnemonic
raw_data_stream()		
{		
while (data_available()) {		
raw_data_block()		
byte_alignment()		
}		
}		

1.1.2 Audio_Data_Transport_Stream frame, ADTS

Tabelle 1-4: Syntax of adts_sequence()

Syntax	No. of bits	Mnemonic
adts_sequence()		
{		
while (nextbits()==syncword) {		
adts_frame()		
}		
}		

Tabelle 1-5: Syntax of adts_frame()

Syntax	No. of bits	Mnemonic
adts_frame()		
{		
byte_alignment()		
adts_fixed_header()		
adts_variable_header()		
adts_error_check()		

```
    for( i=0; i<number_of_raw_data_blocks_in_frame+1; i++) {
        raw_data_block()
    }
}
```

1.1.2.1 Fixed Header of ADTS

Tabelle 1-6: Syntax of adts_fixed_header()

Syntax	No. of bits	Mnemonic
adts_fixed_header()		
{		
syncword	**12**	**bslbf**
ID	**1**	**bslbf**
layer	**2**	**uimsbf**
protection_absent	**1**	**bslbf**
profile	**2**	**uimsbf**
sampling_frequency_index	**4**	**uimsbf**
private_bit	**1**	**bslbf**
channel_configuration	**3**	**uimsbf**
original/copy	**1**	**bslbf**
home	**1**	**bslbf**
emphasis	**2**	**bslbf**
}		

1.1.2.2 Variable Header of ADTS

Tabelle 1-7: Syntax of adts_variable_header()

Syntax	No. of bits	Mnemonic
adts_variable_header()		
{		
copyright_identification_bit	**1**	**bslbf**
copyright_identification_start	**1**	**bslbf**
frame_length	**13**	**bslbf**
adts_buffer_fullness	**11**	**bslbf**
number_of_raw_data_blocks_in_frame	**2**	**uimsfb**
}		

1.1.2.3 Error detection

Tabelle 1-8: Syntax of adts_error_check()

Syntax	No. of bits	Mnemonic
adts_error_check()		
{		
if (protection_absent == ‘0’)		
crc_check	**16**	**rpchof**
}		

1.1.3 Twin-VQ audio sequence

Table 1-9 Syntax of vq_audio_sequence()

Syntax	No. of bits	Mnemonic
vq_audio_sequence()		
{		
vq_header()		
while (nextbit() != NULL){		
vq_single_element()		
}		
}		

Table 1-10 Syntax of vq_scaleable_sequence()

Syntax	No. of bits	Mnemonic
vq_scaleable_sequence()		
{		
vq_header()		
while (nextbit() != NULL){		
vq_scaleable_element()		
}		
}		

1.1.4 AAC-scalable core stream

Tabelle 1-11 Scalable coder stream

Syntax	No. of bits	Mnemonic
core_aac_scalable_stream()		
{		
while (1) {		
syncword	**7**	**bslbf**
if(**syncword** != 0x37) {		
break;		
}		
scal_header()		
bitrate_index	**4**	**bslbf**
padding_bit	**1**	**bslbf**
protection_bit	**1**	**bslbf**
main_data_begin	**10**	**bslbf**
for(ch=0; ch<no_core_ch; ch++) {		
core_coder_stream()		
}		
aac_ scalable_ main_stream()		
for(lay=0; lay<intermediate_layers; lay++) {		
aac_ scalable_ extension_stream()		
}		
for(ch=0; ch<no_core_ch; ch++) {		
core_coder_stream()		
}		
aac_ scalable_ main_stream()		
for(lay=0; lay<intermediate_layers; lay++) {		
aac_ scalable_ extension_stream()		
}		
if(third_core_stream()) {		
for(ch=0; ch<no_core_ch; ch++) {		
core_coder_stream()		
}		
}		
aac_ scalable_ main_stream()		
for(lay=0; lay<intermediate_layers; lay++) {		

```
            aac_ scalable_ extension_stream()
        }
    }
}
```

1.1.5 core BSAC stream

Tabelle 1-12 BSAC Scalable coder stream

Syntax	No. of bits	Mnemonic
core_bsac_stream()		
{		
while (1) {		
syncword	**7**	**bslbf**
if(**syncword** != 0x37) {		
break;		
}		
scal_header()		
bitrate_index	**4**	**bslbf**
padding_bit	**1**	**bslbf**
protection_bit	**1**	**bslbf**
main_data_begin	**10**	**bslbf**
for(ch=0; ch<no_core_ch; ch++) {		
core_coder_stream()		
}		
bsac_raw_data_block()		
for(ch=0; ch<no_core_ch; ch++) {		
core_coder_stream()		
}		
bsac_raw_data_block()		
if(third_core_stream()) {		
for(ch=0; ch<no_core_ch; ch++) {		
core_coder_stream()		
}		
}		
bsac_raw_data_block()		
}		
}		

1.1.6 BSAC stream

Tabelle 1-13:Syntax of bsac_data_stream()

Syntax	No. of bits	Mnemonic
bsac_data_stream()		
{		
nch	**3**	**uimbf**
sampling_frequency_index	**4**	**uimbf**
frame_length_flag	**1**	**uimbf**

```
    while  (data_available()) {
          bsac_raw_data_block()
    }
}
```

1.1.7 Scalable Header

Tabelle 1-14: Syntax of program_config_element()

Syntax	No. of bits	Mnemonic
scal_header()		
{		
op_mode	**4**	**bslbf**
if(low_rate_channel_present)		
ccbrflsre	**4**	**bslbf**
if(7<= op_mode<=13)		
intermediate_layers	**2**	**bslbf**
sampling_frequency_index	**4**	**bslbf**
}		

1.2 T/F Audio Specific Configuration

```
class TFSpecificConfig( uint(4) samplingFrequencyIndex, uint(4) channelConfiguration ) {
        uint(2) TFCodingType;
        uint(1) frameLength;
        uint(1) dependsOnCoreCoder;
        if (dependsOnCoreCoder == 1){
                uint(14)coreCoderDelay
        }
        if (TFCodingType==BSAC) {
                uint(11) lslayer_length
        }
        uint (1) extensionFlag;
        if (channelConfiguration == 0 ){
                program_config_element();
        }
        if (extensionFlag==1){
                <to be defined in mpeg4 phase 2>
        }
}
```

1.2.1 Program config element

Tabelle 1-15: Syntax of program_config_element()

Syntax	No. of bits	Mnemonic
program_config_element()		
{		
element_instance_tag	**4**	**uimsbf**
profile	**2**	**uimsbf**
sampling_frequency_index	**4**	**uimsbf**
num_front_channel_elements	**4**	**uimsbf**
num_side_channel_elements	**4**	**uimsbf**
num_back_channel_elements	**4**	**uimsbf**

num_lfe_channel_elements	**2**	**uimsbf**
num_assoc_data_elements	**3**	**uimsbf**
num_valid_cc_elements	**4**	**uimsbf**
mono_mixdown_present	**1**	**uimsbf**
if (mono_mixdown_present == 1)		
mono_mixdown_element_number	**4**	**uimsbf**
stereo_mixdown_present	**1**	**uimsbf**
if (stereo_mixdown_present == 1)		
stereo_mixdown_element_number	**4**	**uimsbf**
matrix_mixdown_idx_present	**1**	**uimsbf**
if (matrix_mixdown_idx_present == 1) {		
matrix_mixdown_idx	**2**	**uimsbf**
pseudo_surround_enable	**1**	**uimsbf**
}		
for (i = 0; i < num_front_channel_elements; i++) {		
front_element_is_cpe[i];	**1**	**bslbf**
front_element_tag_select[i];	**4**	**uimsbf**
}		
for (i = 0; i < num_side_channel_elements; i++) {		
side_element_is_cpe[i];	**1**	**bslbf**
side_element_tag_select[i];	**4**	**uimsbf**
}		
for (i = 0; i < num_back_channel_elements; i++) {		
back_element_is_cpe[i];	**1**	**bslbf**
back_element_tag_select[i];	**4**	**uimsbf**
}		
for (i = 0; i < num_lfe_channel_elements; i++)		
lfe_element_tag_select[i];	**4**	**uimsbf**
for (i = 0; i < num_assoc_data_elements; i++)		
assoc_data_element_tag_select[i];	**4**	**uimsbf**
for (i = 0; i < num_valid_cc_elements; i++) {		
cc_element_is_ind_sw[i];	**1**	**uimsbf**
valid_cc_element_tag_select[i];	**4**	**uimsbf**
}		
byte_alignment()		
comment_field_bytes	**8**	**uimsbf**
for (i = 0; i < comment_field_bytes; i++)		
comment_field_data[i];	**8**	**uimsbf**
}		

1.3 T/F Bitstream Payload

1.3.1 Top Level Payloads of the AAC-only Profiles

Tabelle 1-16: Syntax of raw_data_block()

Syntax	No. of bits	Mnemonic
raw_data_block()		
{		
while((id = **id_syn_ele**) != ID_END){	**3**	**uimsbf**
switch (id) {		
case ID_SCE: single_channel_element()		
break;		
case ID_CPE: channel_pair_element()		
break;		

```
                case ID_CCE:    coupling_channel_element()
                    break;
                case ID_LFE:    lfe_channel_element()
                    break;
                case ID_DSE:    data_stream_element()
                    break;
                case ID_PCE:    program_config_element()
                    break;
                case ID_FIL:    fill_element()
                    break;
            }
        }
}
```

Tabelle 1-17: Syntax of single_channel_element()

Syntax	No. of bits	Mnemonic
single_channel_element()		
{		
element_instance_tag	**4**	**uimsbf**
individual_channel_stream(0,0)		
}		

Tabelle 1-18: Syntax of channel_pair_element()

Syntax	No. of bits	Mnemonic
channel_pair_element()		
{		
element_instance_tag	**4**	**uimsbf**
common_window	**1**	**uimsbf**
if(common_window) {		
ics_info()		
ms_mask_present	**2**	**uimsbf**
if(ms_mask_present == 1) {		
for(g=0; g < num_window_groups; g++) {		
for(sfb=0; sfb < max_sfb; sfb++) {		
ms_used[g][sfb**]**	**1**	**uimsbf**
}		
}		
}		
}		
individual_channel_stream(common_window,0)		
individual_channel_stream(common_window,0)		
}		

1.3.2 Top Level Payloads of the scalable Profiles

Tabelle 1-19 Syntax of tf_scalable_main_element()

Syntax	**No. of bits**	**Mnemonic**

```
tf_scalable_main_element()
{
    if (TFCodingType != BSAC){
        tf_scalable_main_header(stereo_flag, mono_lay)
    }
    else {
        /* tf_scalable_main_header is included
           in bsac_lstep_stream(0) in case of BSAC */
    }
    if (TFCodingType == AAC_scaleable){
        for (ch=0; ch<(!mono_lay ? 2:1); ch++){
            individual_channel_stream(1, 1)
        }
    } else if (TFCodingType == BSAC){
        bsac_lstep_stream(0)
    } else if (TFCodingType == TwinVQ){
        vq_single_element(0)
    }
}
```

Tabelle 1-20: Syntax of tf_scalable_extension_element()

Syntax	No. of bits	Mnemonic
tf_scalable_extension_element()		
{		
tf_scalable_extension_header(stereo_flag, mono_lay)		
if (TFCodingType == AAC_scaleable){		
for (ch=0; ch<(!mono_lay ? 2:1); ch++){		
individual_channel_stream(1, 1)		
}		
} else if (TFCodingType == BSAC){		
bsac_lstep_stream(lay)		
} else if (TFCodingType == TwinVQ){		
vq_single_element(lay)		
}		
}		

Tabelle 1-21: Syntax of tf_scalable_main_header()

Syntax	No. of bits	Mnemonic
tf_scalable_main_header(stereo_flag, mono_lay)		
{		
if (TFCodingType != TwinVQ) {		
ics_info()		
}else {		
window_sequence	**2**	**bslbf**
window_shape	**1**	**bslbf**
}		
if(stereo_flag && !mono_lay) {		
ms_mask_present	**2**	**bslbf**
if(ms_mask_present == 1 && TFCodingType != BSAC) {		
if (TFCodingType == TwinVQ){		
if (window_sequence == ONLY_SHORT_WINDOW){		
max_sfb	**4**	**bslbf**
}else{		
max_sfb	**6**	**bslbf**
}		

}		
for(g=0; g<num_window_groups; g++) {		
for(sfb=0; sfb<max_sfb; sfb++) {		
ms_used[g][sfb];		
}		
}		
}		
}		
for(ch=0 ch< (stereo_flag ? 2:1); ch++) {		
ltp_data_present	**1**	**bslbf**
if (ltp_data_present)**{**		
ltp_data (last_max_sfb, max_sfb,)		
}		
tns_data_present	**1**	**bslbf**
if(tns_data_present)		
tns_data()		
if(core_flag && ((ch==0) | !mono_lay)		
diff_control_data()		
}		
}		

Tabelle 1-22: Syntax of tf_scalable_extension_header()

Syntax	No. of bits	Mnemonic
tf_scalable_extension_header()		
{		
if (TFCodingType == AAC_scaleable){		
aac_scalable_extension_header()		
}else if (TFCodingType == TwinVQ){		
tvq_scalable_extension_header()		
}		
}		

Tabelle 1-23: Syntax of aac_scalable_extension_header()

Syntax	No. of bits	Mnemonic
aac_scalable_extension_header()		
{		
if(window_sequence == ONLY_SHORT_WINDOW) {		
max_sfb[lay]	**4**	**bslbf**
} else {		
max_sfb[lay]	**6**	**bslbf**
}		
if(!mono_lay) {		
if(stereo_flag)		
ms_mask_present	**2**	**bslbf**
if(ms_mask_present == 1) {		
for(g=0; g<num_window_groups; g++) {		
for(sfb=last_max_sfb_ms; sfb<max_sfb; sfb++) {		
ms_used[g][sfb];	**1**	**bslbf**
}		

```
                }
            }
        }
        for( ch=0; ch<(!mono_flag ? 2:1); ch++ ) {
            ltp_data_present                                    1     bslbf
            if (ltp_data_present)
                ltp_data(last_max_sfb, max_sfb)
            if( mono_stereo_flag ) {
                tns_channel_mono_layer                          1     bslbf
                tns_data_present                                1
                if( tns_data_present
                    tns_data()
                if( block_type != SHORT_WINDOW )
                    for( sfb=0; sfb<max_sfb; sfb++ )
                        if( !ms_used[0][sfb])
                            diff_control_lr[0][sfb]             1     bslbf
                else
                    for( win=0; win<8; win++ )
                            diff_control_lr[win][0]             1     bslbf
            }
        }
}
```

Tabelle 1-24: Syntax of tvq_scalable_extension_header()

```
Syntax                                                 No. of bits  Mnemonic
tvq_scalable_extension_header()
{
        ltp_data_present                                    1     bslbf
        if (ltp_data_present)
            ltp_data(last_max_sfb, max_sfb)
        if( !mono_lay ) {



            if( mono_stereo_flag )
                ms_mask_present                                 2     bslbf
            if( ms_mask_present == 1 ) {
                if (window_sequence == ONLY_SHORT_WINDOW){
                    max_sfb[lay]                                4     bslbf
                } else {
                    max_sfb[lay]                                6     bslbf
                }
                for( g=0; g<num_window_groups; g++ ) {
                    for( sfb=last_max_sfb_ms; sfb<max_sfb; sfb++ ) {
                        ms_used[g][sfb];                        1     bslbf
                    }
                }
            }
        }
}
```

1.3.3 Subsidiary Payloads

Tabelle 1-25: Syntax of ics_info()

```
ics_info()
{
```

ics_reserved_bit	**1**	**bslbf**
window_sequence	**2**	**uimsbf**
window_shape	**1**	**uimsbf**
if(window_sequence == EIGHT_SHORT_SEQUENCE) {		
max_sfb	**4**	**uimsbf**
scale_factor_grouping	**7**	**uimsbf**
}		
else {		
max_sfb	**6**	**uimsbf**
ltp_data_present	**1**	**uimsbf**
if (ltp_data_present)		
ltp_data**()**		
if (common_window) {		
ltp_data_present	**1**	**uimsbf**
if (ltp_data_present)		
ltp_data**()**		
}		
}		
}		
}		
}		

Tabelle 1-26: Syntax of individual_channel_stream()

Syntax	No. of bits	Mnemonic
individual_channel_stream(common_window,scale_flag)		
{		
global_gain	**8**	**uimsbf**
if(!common_window && !scale_flag)		
ics_info()		
section_data()		
scale_factor_data()		
pulse_data_present	**1**	**uismbf**
if(pulse_data_present) {		
pulse_data()		
}		
if(!scale_flag) {		
tns_data_present	**1**	**uimsbf**
if(tns_data_present)		
tns_data()		
gain_control_data_present	**1**	**uimsbf**
if(gain_control_data_present)		
gain_control_data()		
}		
spectral_data()		

}

Tabelle 1-27: Syntax of section_data()

Syntax	No. of bits	Mnemonic
section_data()		
{		
if(window_sequence == EIGHT_SHORT_SEQUENCE)		
sect_esc_val = (1<<3) - 1		
else		
sect_esc_val = (1<<5) - 1		
for(g=0; g < num_window_groups; g++) {		
k=0		
i=0		
while (k<max_sfb) {		
sect_cb[g][i]	**4**	**uimsbf**
sect_len=0		
while (**sect_len_incr** == sect_esc_val)	**3/5**	**uimsbf**
sect_len += sect_esc_val		
sect_len += sect_len_incr		
sect_start[g][i] = k		
sect_end[g][i] = k+sect_len		
for (sfb=k; sfb<k+sect_len; k++)		
sfb_cb[g][sfb] = sect_cb[g][i];		
k += sect_len		
i++		
}		
num_sec[g] = i		
}		
}		

Tabelle 1-28: Syntax of scale_factor_data()

Syntax	No. of bits	Mnemonic
scale_factor_data()		
{		
noise_pcm_flag = 1		
for (g=0; g<num_window_groups; g++) {		
for (sfb=0; sfb<max_sfb; sfb++) {		
if (sect_cb[g][sfb] != ZERO_HCB) {		
if (is_intensity(g,sfb))		
hcod_sf[dpcm_is_position[g][sfb]]	**1..19**	**bslbf**
else if (is_noise(g,sfb)) {		
if (noise_pcm_flag) {		
noise_pcm_flag = 0		
dpcm_noise_energy[g][sfb]	**9**	**uimsbf**
} else		
hcod_sf[dpcm_noise_energy[g][sfb]]	**1..19**	**bslbf**
else		
hcod_sf[dpcm_sf[g][sfb]]	**1..19**	**bslbf**
}		
}		
}		
}		

Tabelle 1-29: Syntax of tns_data()

Syntax	No. of bits	Mnemonic
tns_data()		
{		
for (w=0; w<num_windows; w++) {		
n_filt[w]	**1..2**	**uimsbf**
if (n_filt[w])		
coef_res[w]	**1**	**uimsbf**
for (filt=0; filt<n_filt[w]; filt++) {		
length[w][filt]	**4/6**	**uimsbf**
order[w][filt]	**3/5**	**uimsbf**
if (order[w][filt]) {		
direction[w][filt]	**1**	**uimsbf**
coef_compress[w][filt]	**1**	**uimsbf**
for (i=0; i<order[w][filt]; i++)		
coef[w][filt][i]	**2..4**	**uimsbf**
}		
}		
}		
}		

Tabelle 1-30: Syntax of ltp_data()

Syntax	No. of bits	Mnemonic
ltp_data(start_sfb, stop_sfb)		
{		
ltp_lag	**11**	**uimsbf**
ltp_coef	**3**	**uimsbf**
if (TFcodingType != TwinVQ && TFCodingType != AAC_non_scaleable {		
if(window_sequence==EIGHT_SHORT_SEQUENCE) {		
for (w=0; w<num_windows; w++) {		
ltp_short_used[w]	**1**	**uimsbf**
if (ltp_short_used [w]) {		
ltp_short_lag_present[w]	**1**	
}		
if (ltp_short_lag_present[w]) {		
ltp_short_lag[w]	**4**	**uimsbf**
}		
}		
}		
else {		
for (sfb=start_sfb; sfb< stop_sfb); sfb++) {		
ltp_long_used[sfb]	**1**	**uimsbf**
}		
}		
}		
else {		
for (sfb=start_sfb0; sfb<min(max_sfb, stop_sfb); sfb++) {		
ltp_long_used[sfb]	**1**	**uimsbf**
}		
}		

}		

Tabelle 1-31: Syntax of spectral_data()

Syntax	No. of bits	Mnemonic
spectral_data()		
{		
for(g=0; g<num_window_groups; g++) {		
for (i=0; i<num_sec[g]; i++) {		
if (sect_cb[g][i] != ZERO_HCB &&		
sect_cb[g][i] != NOISE_HCB &&		
sect_cb[g][i] != INTENSITY_HCB &&		
sect_cb[g][i] != INTENSITY_HCB2) {		
for (k=sect_sfb_offset[g][sect_start[g][i]];		
k< sect_sfb_offset[g][sect_end[g][i]];) {		
if (sect_cb[g][i]<FIRST_PAIR_HCB) {		
hcod[sect_cb[g][i]][w][x][y][z]	**1..31**	**bslbf**
if(unsigned_cb[sect_cb[g][i]])		
quad_sign_bits	**0..4**	**bslbf**
k += QUAD_LEN		
}		
else {		
hcod[sect_cb[g][i]][y][z]	**1..31**	**bslbf**
if(unsigned_cb[sect_cb[g][i]])		
pair_sign_bits	**0..2**	**bslbf**
k += PAIR_LEN		
if (sect_cb[g][i]==ESC_HCB) {		
if (y==ESC_FLAG)		
hcod_esc_y	**1..31**	**bslbf**
if (z==ESC_FLAG)		
hcod_esc_z	**1..31**	**bslbf**
}		
}		
}		
}		
}		
}		
}		

Tabelle 1-32: Syntax of pulse_data()

Syntax	No. of bits	Mnemonic
pulse_data() {		
number_pulse	**2**	**uimsbf**
pulse_start_sfb	**6**	**uimsbf**
for (i=0; i<number_pulse+1; i++) {		
pulse_offset[i]	**5**	**uimsbf**
pulse_amp[i]	**4**	**uimsbf**
}		
}		

Tabelle 1-33: Syntax of coupling_channel_element()

Syntax	No. of bits	Mnemonic
coupling_channel_element()		
{		

element_instance_tag	**4**	**uimsbf**
ind_sw_cce_flag	**1**	**uimsbf**
num_coupled_elements	**3**	**uimsbf**
num_gain_element_lists = 0		
for (c=0; c<num_coupled_elements+1; c++) {		
num_gain_element_lists++		
cc_target_is_cpe[c]	**1**	**uimsbf**
cc_target_tag_select[c]	**4**	**uimsbf**
if (cc_target_is_cpe[c]) {		
cc_l[c]	**1**	**uimsbf**
cc_r[c]	**1**	**uimsbf**
if (cc_l[c] && cc_r[c])		
num_gain_element_lists++		
}		
}		
cc_domain	**1**	**uimsbf**
gain_element_sign	**1**	**uimsbf**
gain_element_scale	**2**	**uimsbf**
individual_channel_stream(0,0)		
for (c=1; c<num_gain_element_lists; c++) {		
if (ind_sw_cce_flag) {		
cge = 1		
} else {		
common_gain_element_present[c]	**1**	**uimsbf**
cge = common_gain_element_present[c]		
}		
if (cge)		
hcod_sf[common_gain_element[c]]	**1..19**	**bslbf**
else {		
for (g=0; g<num_window_groups) {		
for (sfb=0; sfb<max_sfb; sfb++) {		
if (sfb_cb[g][sfb] != ZERO_HCB)		
hcod_sf[dpcm_gain_element[c][g][sfb]]	**1..19**	**bslbf**
}		
}		
}		
}		
}		

Tabelle 1-34: Syntax of lfe_channel_element()

Syntax	No. of bits	Mnemonic
lfe_channel_element()		
{		
element_instance_tag	**4**	**uimsbf**
individual_channel_stream(0,0)		
}		

Tabelle 1-35: Syntax of data_stream_element()

Syntax	No. of bits	Mnemonic
data_stream_element()		
{		
element_instance_tag	**4**	**uimsbf**
data_byte_align_flag	**1**	**uimsbf**
cnt = **count**	**8**	**uimsbf**

Syntax	No. of bits	Mnemonic
if (cnt == 255)		
cnt += **esc_count**;	**8**	**uimsbf**
if (data_byte_align_flag)		
byte_alignment()		
for (i=0; i<cnt; i++)		
data_stream_byte[element_instance_tag][i];	**8**	**uimsbf**
}		

Tabelle 1-36: Syntax of fill_element()

Syntax	No. of bits	Mnemonic
fill_element()		
{		
cnt = **count**	**4**	**uimsbf**
if (cnt == 15) {		
cnt += **esc_count** - 1;	**8**	**uimsbf**
}		
for (i=0; i<cnt; i++)		
fill_byte[i];	**8**	**uimsbf**
}		

Tabelle 1-37: Syntax of gain_control_data()

Syntax	No. of bits	Mnemonic
gain_control_data()		
{		
max_band	**2**	**uimsbf**
if (window_sequence == ONLY_LONG_SEQUENCE) {		
for (bd=1; bd<=max_band; bd++) {		
for (wd=0; wd<1; wd++) {		
adjust_num[bd][wd]	**3**	**uimsbf**
for (ad=0; ad<adjust_num[bd][wd]; ad++) {		
alevcode[bd**][**wd**][**ad**]**	**4**	**uimsbf**
aloccode[bd**][**wd**][**ad**]**	**5**	**uimsbf**
}		
}		
}		
}		
else if (window_sequence == LONG_START_SEQUENCE) {		
for (bd=1; bd<=max_band; bd++) {		
for (wd=0; wd<2; wd++) {		
adjust_num[bd**][**wd**]**	**3**	**uimsbf**
for (ad=0; ad<adjust_num[bd][wd]; ad++) {		
alevcode[bd**][**wd**][**ad**]**	**4**	**uimsbf**
if (wd == 0)		
aloccode[bd**][**wd**][**ad**]**	**4**	**uimsbf**
else		
aloccode[bd**][**wd**][**ad**]**	**2**	**uimsbf**
}		
}		
}		
}		
else if (window_sequence == EIGHT_SHORT_SEQUENCE) {		
for (bd=1; bd<=max_band; bd++) {		
for(wd=0; wd<8; wd++) {		
adjust_num[bd**][**wd**]**	**3**	**uimsbf**
for (ad=0; ad<adjust_num[bd][wd]; ad++) {		

```
                        alevcode[bd][wd][ad]                    4        uimsbf
                        aloccode[bd][wd][ad]                    2        uimsbf
                    }
                }
            }
        }
        else if (window_sequence == LONG_STOP_SEQUENCE) {
            for (bd=1; bd<=max_band; bd++) {
                for(wd=0; wd<2; wd++) {
                    adjust_num[bd][wd]                          3        uimsbf
                    for (ad=0; ad<adjust_num[bd][wd]; ad++) {
                        alevcode[bd][wd][ad]                    4        uimsbf
                        if (wd == 0)
                            aloccode[bd][wd][ad]                4        uimsbf
                        else
                            aloccode[bd][wd][ad]                5        uimsbf
                    }
                }
            }
        }
}
```

Tabelle 1-38: Syntax of diff_control_data()

Syntax	No. of bits	Mnemonic
diff_control_data()		
{		
if(block_type == SHORT_WINDOW)		
for(w=0; w<8; w++)		
diff_control[w][0]	**1**	**bslbf**
else		
for(dc_group=0; dc_group<no_of_dc_groups; dc_group++)		
diff_control[0][dc_group]	**2..5**	**bslbf**
}		

Tabelle 1-39: Syntax of Twin-VQ scaleable element

Syntax	No. of bits	Mnemonic
vq_scaleable_element()		
{		
for (lyr=0; lyr<=num_enhancement_lyr; lyr++){		
vq_single_element(lyr)		
}		
}		

Tabelle 1-40 : Syntax of Twin-VQ single element

Syntax	No. of bits	Mnemonic
vq_single_element(lyr)		
{		
window_sequence	**4**	**uimsbf**
switch (window_sequence){		
case ONLY_LONG_WINDOW:		
case LONG_MEDIUM_WINDOW:		
case MEDIUM_LONG_WINDOW:		
case LONG_SHORT_WINDOW:		

```
    case SHORT_LONG_WINDOW:
        vq_data(LONG, lyr)
    case ONLY_MEDIUM_WINDOW:
    case MEDIUM_SHORT_WINDOW:
    case SHORT_MEDIUM_WINDOW:
        vq_data(MEDIUM, lyr)
    case ONLY_SHORT_WINDOW:
        vq_data(SHORT, lyr)
    }
}
```

Tabelle 1-41 : Syntax of Twin-VQ data element

Syntax	No. of bits	Mnemonic
vq_data(b_type, lyr)		
{		
if (b_type == LONG && LTP_ACTIVE){		
ltp_data_present	**1**	**uimsbf**
if (ltp_data_present)		
ltp_data()		
}		
if (b_type != SHORT && lyr == 0)		
optional_info	**2**	**uimsbf**
if (lyr > 0)		
for (i_ch=0; i_ch<n_ch; i_ch++){		
fb_shift[i_ch]	**2**	**uimsbf**
}		
if (lyr == 0){		
for (i_ch=0; i_ch<n_ch; i_ch++){		
index_blim_h[i_ch]	**0,2,3**	**uimsbf**
index_blim_l[i_ch]	**0/1**	**uimsbf**
}		
}		
for (idiv=0; idiv<N_DIV; idiv++){		
index_shape0[idiv]	**6/7**	**uimsbf**
index_shape1[idiv]	**6/7**	**uimsbf**
}		
for (i_ch=0; i_ch<n_ch; i_ch++){		
for (isb=0; isb<N_SF; isb++){		
for (ifdiv=0; ifdiv<FW_N_DIV; ifdiv++){		
index_env[i_ch][isb][ifdiv]	**0,3,5,6**	**uimsbf**
}		
}		
}		
for (i_ch=0; i_ch<n_ch; i_ch++){		
for (isbm=0; isbm<N_SF; isbm++){		
index_fw_alf[i_ch][isbm]	**1**	**uimsbf**
}		
}		
for (i_ch=0; i_ch<n_ch; i_ch++){		
index_gain[i_ch]	**7..10**	**uimsbf**
if (N_SF[b_type]>1){		
for (isbm=0; isbm<N_SF[b_type]; isbm++){		
index_gain_sb[i_ch][isbm]	**4..6**	**uimsbf**
}		
}		
}		
for (i_ch=0; i_ch<n_ch; i_ch++){		

index_lsp0[i_ch]	**1**	**uimsbf**
index_lsp1[i_ch]	**5/6**	**uimsbf**
for (isplt=0; isplt<LSP_SPLIT; isplit++){		
index_lsp2[i_ch][isplt]	**3/4**	**uimsbf**
}		
}		
if (ppc_present){		
for (idiv=0; idiv<N_DIV_P; idiv++){		
index_shape0_p[idiv]	**7**	**uimsbf**
index_shape1_p[idiv]	**7**	**uimsbf**
}		
for (i_ch=0; i_ch<n_ch; i_ch++){		
index_pit[i_ch]	**8/9**	**uimsbf**
index_pgain[i_ch]	**6/7**	**uimsbf**
}		
}		
}		

Tabelle 1-42 : Syntax of bsac_lstep_stream()

Syntax	No. of bits	Mnemonic
bsac_lstep_stream(lslayer)		
{		
for(i=Lstep_offset[lslayer];i<Lstep_offset[lslayer+1];i++)		
BSAC_stream_buf[i]	**8**	**uimsbf**
/* Large step stream is saved in BSAC_stream_buf[].		
BSAC_stream_buf[] is mapped to small step stream,		
bsac_raw_data_block(), f or the actual decoding.		
see the decoding process of BSAC large step scalability		
for more detailed description.		
*/		
}		

Tabelle 1-43 : Syntax of bsac_raw_data_block()

Syntax	No. of bits	Mnemonic
bsac_raw_data_block()		
{		
bsac_main_stream()		
layer=1;		
while(data_available() && layer<=encoded_layer) {		
bsac_layer_stream(nch, layer)		
layer++;		
}		
byte_alignment()		
}		

Tabelle 1-44 : Syntax of bsac_main_stream()

Syntax	No. of bits	Mnemonic
bsac_main_stream()		
{		
switch(nch) {		
case 1 : tf_scalable_main_header(0, 0)		
break		
case 2 : tf_scalable_main_header(1, 0)		

```
            break
    }
    bsac_general_info(nch)
    bsac_layer_stream(nch, 0)
}
```

Tabelle 1-45 : Syntax of bsac_layer_stream()

Syntax	No. of bits	Mnemonic
bsac_layer_stream(nch, layer)		
{		
bsac_side_info(nch, layer)		
bsac_spectral_data(nch, layer)		
}		

Tabelle 1-46 : Syntax of bsac_general_info()

Syntax	No. of bits	Mnemonic
bsac_general_info(nch)		
{		
frame_length	**10/11**	**uimbf**
encoded_layer	**6**	**uimbf**
for(ch=0;ch<nch;ch++) {		
max_scalefactor[ch]	**8**	**uimbf**
scalefactor_model[ch]	**2**	**uimbf**
min_ArModel[ch]	**5**	**uimbf**
ArModel_model[ch]	**2**	**uimbf**
scf_coding[ch]	**1**	**uimbf**
}		
pns_data_present	**1**	**uimbf**
if (pns_data_present)		
pns_start_sfb	**6**	**uimbf**
}		

Tabelle 1-47 : Syntax of bsac_side_info ()

Syntax	No. of bits	Mnemonic
bsac_side_info (nch, layer)		
{		
if (nch==1) {		
if(pns_data_present) {		
for(g = 0; g < num_window_groups; g++) {		
for(sfb=layer_sfb[layer];sfb<layer_sfb[layer+1];sfb++){		
if(sfb>=pns_start_sfb)		
acode_noise_flag	**1**	**bslbf**
}		
}		
}		
}		
else if (ms_mask_present !=2) {		
for(g = 0; g < num_window_groups; g++) {		
for(sfb=layer_sfb[layer];sfb<layer_sfb[layer+1];sfb++){		
if (ms_mask_present==1) {		
acode_ms_used	**1**	**bslbf**
pns_data_present = 0		
}		
else if (ms_mask_present==3) {		

```
                acode_stereo_info                           0..4        bslbf
            }
            if(pns_data_present && sfb>=pns_start_sfb) {
                acode_noise_flag_l                          1           bslbf
                acode_noise_flag_r                          1           bslbf
                if(ms_mask_present==3 && stereo_info==3)  {
                    if(noise_flag_l[g][sfb] && noise_flag_r[g][sfb])
                        acode_noise_mode                    2           bslbf
                }
            }
        }
    }
}

noise_pcm_flag = 1;
for(ch=0;ch<nch;ch++) {
    for(g = 0; g < num_window_group; g++) {
        for(sfb=layer_sfb[layer]; sfb<layer_sfb[layer+1]; sfb++) {
            if (scf_coding[ch]) {
                if (noise_flag[ch][g][sfb]) {
                    if (!noise_pcm_flag)
                        acode_dpcm_noise_energy             0..13       bslbf
                }
                else if (stereo_info[ch][g][sfb]>=2)
                    acode_is_position                       0..13       bslbf
                else
                    acode_scf                               0..13       bslbf
            }
            else {
                if (noise_flag[ch][g][sfb]) {
                    if (!noise_pcm_flag) {
                        acode_dpcm_noise_energy_index       0..14       bslbf
                        if(dpcm_noise_energy_index==ESC_ INDEX)
                            acode_esc_dpcm_noise_energy_index   3..8    bslbf
                    }
                }
                else if (stereo_info[ch][g][sfb]>=2) {
                    acode_is_position_index                 0..14       bslbf
                    if (is_position_index==ESC_ INDEX)
                        acode_esc_is_position_index         3..8        bslbf
                }
                else {
                    acode_scf_index                         1..14       bslbf
                    if (scf_index==ESC_SCF_INDEX)
                        acode_esc_scf_index                 3..8        bslbf
                }
            }
            if (noise_flag[ch][g][sfb] && noise_pcm_flag) {
                acode_pcm_noise_energy                      9           bslbf
                noise_pcm_flag = 0
            }
        }
    }
}

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

```
                for(i=swb_offset[band]; i<swb_offset[band+1]; i+=4) {
                    cband = index2cb(ch, i);
                    if(!decode_cband[ch][cband]) {
                        acode_ArModel                                   1..14       bslbf
                        decode_cband[ch][cband] = 1
                    }
                }
            }
        }
    }
}
```

Tabelle 1-48 :Syntax of bsac_spectral_data ()

Syntax	No. of bits	Mnemonic
bsac_spectral_data(nch, layer)		
{		
for (snf=maxsnf; snf>0; snf--) {		
for (i =0; i <last_index; i +=4) {		
for(ch=0;ch<nch;ch++) {		
if(i >= layer_index[ch]) continue;		
if (cur_snf[ch][i]<snf) continue;		
dim0 = dim1 = 0		
for(k = 0; k < 4; k++)		
if(prestate[ch][i +k]) dim1++		
else dim0++		
if(dim0)		
acode_vec0	**0..14**	**bslbf**
if(dim1)		
acode_vec1	**0..14**	**bslbf**
for(k = 0; k < 4; k++) {		
if(sample[ch][i +k] &&!prestate[ch][i +k]) {		
acode_sign	**1**	**bslbf**
prestate[ch][i +k] = 1		
}		
}		
cur_snf[ch][i]--		
if (total_estimated_bits >= available_bits[layer]) return		
}		
}		
if (total_estimated_bits >= available_bits[layer]) return		
}		
}		

2 General information

2.1 Decoding of interface formats

2.1.1 Audio_Data_Interchange_Format (ADIF), Audio_Data_Transport_Stream (ADTS) and raw_data_block

2.1.1.1 Definitions

Bit stream elements:

adif_sequence()	a sequence according to the Audio_Data_Interchange_Format (Table 6.1)
adif_header()	header of the Audio_Data_Interchange_Format located at the beginning of an adif_sequence (Table 6.2)
raw_data_block()	see subclause **Fehler! Verweisquelle konnte nicht gefunden werden.** and Table 6.8
adif_id	ID that indicates the Audio_Data_Interchange_Format. Its value is 0x41444946 (most significant bit first), the ASCII representation of the string „ADIF“ (Table 6.2)
copyright_id_present	indicates whether **copyright_id** is present or not (Table 6.2)
copyright_id	The field consists of an 8-bit copyright_identifier, followed by a 64-bit copyright_number (Table 6.2). The copyright identifier is given by a Registration Authority as designated by SC 29. The copyright_number is a value which identifies uniquely the copyrighted material. See ISO/IEC 13818-3, subclause 2.5.2.13.
original_copy	see ISO/IEC 11172-3, subclause 2.4.2.3 (Table 6.2)
home	see ISO/IEC 11172-3, subclause 2.4.2.3 (Table 6.2)
bitstream_type	a flag indicating the type of a bitstream (Table 6.2): ‘0’ constant rate bitstream. This bitstream may be transmitted via a channel with constant rate ‘1’ variable rate bitstream. This bitstream is not designed for transmission via constant rate channels
bitrate	a 23 bit unsigned integer indicating either the bitrate of the bitstream in bits/sec in case of constant rate bitstream or the maximum peak bitrate (measured per frame) in case of variable rate bitstreams. A value of 0 indicates that the bitrate is not known (Table 6.2)
num_program_config_element	number of program config elements specified for this adif_sequence() (Table 6.2)
adif_buffer_fullness	number of bits remaining in the encoder buffer after encoding the first raw_data_block in the adif_sequence (Table 6.2)
program_config_element()	contains information about the configuration for one program (Table 6.2). See subclause 2.2.1.
adts_sequence()	a sequence according to Audio_Data_Transport_Stream ADTS (Table 6.3)
adts_frame()	a ADTS frame, consisting of a fixed header, a variable header, an optional error check and a specified number of raw_data_blocks() (Table 6.4)
adts_fixed_header()	fixed header of ADTS. The information in this header does not change from frame to frame. It is repeated every frame to allow random access into a bitstream bitstream (Table 6.5)
adts_variable_header()	variable header of ADTS. This header is transmitted every frame as well as the fixed header, but contains data that changes from frame to frame (Table 6.6)
adts_error_check()	CRC error detection data generated as described in ISO/IEC 11172-3, subclause 2.4.3.1 (Table 6.7) The following bits are protected and fed into the CRC algorithm in order of their appearance: all bits of the headers first 192 bits of any single_channel_element (SCE) channel_pair_element (CPE) coupling_channel_element (CCE) low frequency enhancement channel (LFE) In addition, the first 128 bits of the second individual_channel_stream in the channel_pair_element must be protected. All information in any program configuration element or data element must be protected. For any element where the specified protection length of 128 or 192 bits exceeds its actual length, the element is zero padded to the specified protection length for CRC calculation.
byte_alignment()	if called from within a raw_data_block then align with respect to the first bit of the raw_data_block, else align with respect to the first bit of the header.

syncword The bit string '1111 1111 1111'. See ISO/IEC 11172-3, subclause 2.4.2.3 (Table 6.5)
ID MPEG identifier, set to '1'. See ISO/IEC 11172-3, subclause 2.4.2.3 (Table 6.5)
layer Indicates which layer is used. Set to '00'. See ISO/IEC 11172-3, subclause 2.4.2.3 (Table 6.5)
protection_absent Indicates whether error_check() data is present or not. Same as syntax element 'protection_bit' in ISO/IEC 11172-3, subclause 2.4.1 and 2.4.2 (Table 6.5)
profile profile used. See clause 2. (Table 6.5)
sampling_frequency_index indicates the sampling frequency used according to the following table: (Table 6.5)

sampling_frequency_index	sampling frequeny
0x0	96000
0x1	88200
0x2	64000
0x3	48000
0x4	44100
0x5	32000
0x6	24000
0x7	22050
0x8	16000
0x9	12000
0xa	11025
0xb	8000
0xc	reserved
0xd	reserved
0xe	reserved
0xf	reserved

private_bit see ISO/IEC 11172-3, subclause 2.4.2.3 (Table 6.5)
channel_configuration indicates the channel configuration used. If channel_configuration is greater than 0, the channel configuration is given by the 'Default bitstream index number' in **Fehler! Verweisquelle konnte nicht gefunden werden.**, see subclause 2.2.1. If channel_configuration equals 0, the channel configuration is not specified in the header and must be given by a program_config_element following as first bitstream element in the first raw_data_block after the header, or by the implicit configuration (see subclause 8.5) or must be known in the application. (Table 6.5)
emphasis see ISO/IEC 11172-3, subclause 2.4.2.3 (Table 6.5)
copyright_identification_bit One the bits of the 72-bit copyright identification field (see copyright_id above). The bits of this field are transmitted frame by frame; the first bit is indicated by the copyright_identification_start bit set to '1'. The field consists of an 8-bit copyright_identifier, followed by a 64-bit copyright_number. The copyright identifier is given by a Registration Authority as designated by SC29. The copyright_number is a value which identifies uniquely the copyrighted material. See ISO/IEC 13818-3, subclause 2.5.2.13 (Table 6.6)
copyright_identification_start One bit to indicate that the copyright_identification_bit in this audio frame is the first bit of the 72-bit copyright identification. If no copyright identification is transmitted, this bit should be kept '0'.
'0' no start of copyright identification in this audio frame
'1' start of copyright identification in this audio frame
See ISO/IEC 13818-3, subclause 2.5.2.13 (Table 6.6)
frame_length length of the frame including headers and error_check (Table 6.5) in bytes
adts_buffer_fullness number of 32 bit words remaining in the encoder buffer after encoding the first raw_data_block in the ADTS frame (Table 6.6). A value of hexadecimal 7FF signals that the bitstream is a variable rate bitstream. In this case, buffer fullness is not applicable.
number_of_raw_data_blocks_in_frame number of raw_data_blocks in the ADTS frame (Table 6.6)

Help elements:

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

2.1.1.2 Overview

The raw_data_block() contains all data which belongs to the audio (including ancillary data). Beyond that, additional information like sampling_frequency is needed to fully describe an audio sequence. The Audio_Data_Interchange_Format (ADIF) contains all elements that are necessary to describe a bitstream according to this standard.
For specific applications some or all of the syntax elements like those specified in the header of the ADIF, e.g. sampling_rate, may be known to the decoder by other means and hence do not appear in the bitstream. Furthermore, additional information that varies from block to block (e.g. to enhance the parsability or error resilience) may be required. Therefore transport streams may be designed for a specific application and are not specified in this standard. However, one possible transport stream called Audio_Data_Transport_Stream (ADTS) is given, which may be used for applications.

2.1.1.3 Audio_Data_Interchange_Format ADIF

The Audio_Data_Interchange_Format (ADIF) contains one header at the start of the sequence followed by a raw_data_stream (). The raw_data_stream() may not contain any further channel_configuration_elements.
As such, the ADIF is useful only for systems with a defined start and no need to start decoding from within the audio data stream, such as decoding from disk file. It can be used as an interchange format in that it contains all information necessary to decode and play the audio data.

2.1.1.4 Audio_Data_Transport_Stream ADTS

The Audio_Data_Transport_Stream (ADTS) is similar to syntax used in ISO/IEC 11172-3 and 13818-3. This will be recognized by ISO/IEC 11172-3 decoders as a "Layer 4" bit-stream.
The fixed header of the ADTS contains the syncword plus all parts of the header which are necessary for decoding and which do not change from frame to frame. The variable header of the ADTS contains header data which changes from frame to frame.
The ADTS only supports a raw_data_stream() with only one program. The program may have up to 7 channels plus an independently switched coupling channel.

2.1.2 AAC scalable core stream

The format of the transport stream, which is used in the VM.

2.1.2.1 Definitions

syncword	The sequence 0110111.
protection_bit	See ISO/IEC 11172-3.
padding_bit	See ISO/IEC 11172-3.
main_data_begin	See ISO/IEC 11172-3.
bitrate_index	See ISO/IEC 11172-3. However, the **bitrate_index** table has been redifined in the VM (see table below).
granule	See ISO/IEC 11172-3.
third_core_stream()	A helper function which returns 1, if a third core stream is present in a transport frame, which comprises three granules, or 0 if not.

2.1.2.2 Decoding process

Padding_bit, bitrate_index , and main_data_begin are used in the VM transport multiplexer to implement a stream similar to Layer III of ISO/IEC 11172-3. If used, a core coder is inserted at the start of a granule. This combination allows to send out the core coder stream and to decode the core coder with the optimum delay, without having to wait for the delayed AAC stream.

<u>third_core_stream()</u>

The function third_core_stream() is defined as follows:

Return 1, if two times the core coder frame length is less than three times of half of the AAC frame length. Return 0 else.

2.2 Decoding of the T/F Audio Specific Configuration

2.2.1 General configuration

TFCodingType specifies the TF coding type

	TFCodingType
0x0	AAC scaleable
0x1	BSAC
0x2	TwinVQ
0x3	AAC non scaleable (i.e. multichannel)

`frameLength` specifies the window length of the IMDCT: if set to 0 a 1024 lines IMDCT is used if set to 1 a 960 line IMDCT is used.

`dependsOnCoreCoder` shall be set to 1 if a non MPEG-4 TF coder or a MPEG-4 TF coder at an different sampling rate is used as a core coder in a scaleable bitstream.

`coreCoderDelay` is the delay that has to be applied to the core decoder output befor the MDCT if the core coder and the 1st scaleable TF layer should be decoded.

`lslayer_length` is the length of the BSAC large step scalability layer which is represented in the unit of byte. A BSAC large step scalability layer is conveyed over an elementary stream.

`extensionFlag` is used for use in MPEG-4 phase 2.

`program_config_element()` shall be used only if audio data are MPEG2 AAC data.

2.2.2 Program Config Element (PCE)

profile The two-bit profile index from Table 7.1 (Table 6.21)
sampling_frequency_index Indicates the sampling rate of the program (and all other programs in this bitstream). See definition in 8.1.1 (Table 6.21)
num_front_channel_elements The number of audio syntactic elements in the front channels, front center to back center, symmetrically by left and right, or alternating by left and right in the case of single channel elements (Table 6.21)
num_side_channel_elements Number of elements to the side as above (Table 6.21)
num_back_channel_elements As number of side and front channel elements, for back channels (Table 6.21)
num_lfe_channel_elements number of LFE channel elements associated with this program (Table 6.21)
num_assoc_data_elements The number of associated data elements for this program (Table 6.21)
num_valid_cce_elements The number of cce's that can add to the audio data for this program (Table 6.21)
mono_mixdown_present One bit, indicating the presence of the mono mixdown element (Table 6.21)
mono_mixdown_element_number The number of a specified SCE that is the mono mixdown (Table 6.21)
stereo_mixdown_present One bit, indicating that there is a stereo mixdown present (Table 6.21)
stereo_mixdown_element_number The number of a specified CPE that is the stereo mixdown element (Table 6.21)
matrix_mixdown_idx_present One bit, indicating the presence of matrix mixdown information (Table 6.21)
matrix_mixdown_idx Two bit field, specifying the index of the surround mixdown coefficient (Table 6.21)
pseudo_surround_enable One bit, indicating the possibility of mixdown for pseudo surround reproduction (Table 6.21)
front_element_is_cpe indicates whether a SCE or a CPE is addressed as a front element (Table 6.21)
'0' selects an SCE
'1' selects an CPE
The instance of the SCE or CPE addressed is given by **front_element_tag_select**
front_element_tag_select the instance_tag of the SCE/CPE addressed as a front element (Table 6.21)

side_element_is_cpe see **front_element_is_cpe**, but for side elements (Table 6.21)
side_element_tag_select see **front_element_tag_select**, but for side elements (Table 6.21)
back_element_is_cpe see **front_element_is_cpe**, but for back elements (Table 6.21)
back_element_tag_select see **front_element_tag_select**, but for back elements (Table 6.21)
lfe_element_tag_select instance_tag of the LFE addressed (Table 6.21)
assoc_data_element_tag_select instance_tag of the DSE addressed (Table 6.21)
valid_cce_element_tag_select instance_tag of the CCE addressed (Table 6.21)
cc_element_is_ind_sw One bit, indicating that the corresponding CCE is an independently switched coupling channel (Table 6.21)
comment_field_bytes The length, in bytes, of the following comment field (Table 6.21)
comment_field_data The data in the comment field (Table 6.21)

SCE or CPE elements within the PCE are addressed with two syntax elements. First, an is_cpe syntax element selects whether a SCE or CPE is addressed. Second, a tag_select syntax element selects the instance_tag of a SCE/CPE. LFE, CCE and DSE elements are directly addressed with their instance_tag.

2.2.2.1 Implicit and defined channel configurations

The MPEG-2 AAC audio syntax provides two ways to convey the mapping of channels within a set of syntactic elements to physical locations of speakers. The first way is a default mapping based on the specific set of syntactic elements received and the order in which they are received. The most common mappings are further defined in**Fehler! Verweisquelle konnte nicht gefunden werden.**. If a mapping shown in **Fehler! Verweisquelle konnte nicht gefunden werden.** is not used, the following methods describe the default determination of channel mapping:

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

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

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

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

In case of this default (or implicit) mapping the number and order of SCE's and CPE's and the resulting configuration may not change within the bitstream without sending a program_configuration_element, i.e. an implicit reconfiguration is not allowed.

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

If reliable mapping of channel set to speaker geometry is a concern, then it is recommended that an implicit mapping from **Fehler! Verweisquelle konnte nicht gefunden werden.** or a program configuration element be used.

For more complicated configurations a **Program Configuration Element** (PCE) is defined. There are 16 available PCE's, and each one can specify a distinct program that is present in the raw data stream. All available PCE's within a raw_data_block must come before all other syntactic elements. Programs may or may not share audio syntactic elements, for example, programs could share a channel_pair_element and use distinct coupling channels for voice over in different languages. A given program configuration element contains information pertaining to only one program out of many that may be included in the raw data stream. Included in the PCE are "list of front channels", again using the rule center outwards, left before right. In this list, a center channel SCE, if any, must come first, and any other SCE's must appear in pairs, constituting an LR pair. If only two SCE's are specified, this signifies one LR stereophonic pair.

After the list of front channels, there is a list of “side channels” consisting of CPE’s, or of pairs of SCE’s. These are listed in the order of front to back. Again, in the case of a pair of SCE’s, the first is a left channel, the second a right channel.

After the list of side channels, a list of back channels is available, listed from outside in. Any SCE’s except the last SCE must be paired, and the presence of exactly two SCE’s (alone or preceeded by a CPE) indicates that the two SCE’s are Left and Right Rear center, respectively.

The configuration indicated by the PCE takes effect at the raw_data_block containing the PCE. The number of front, side and back channels as specified in the PCE must be present in that block and all subsequent raw_data_blocks until a raw_data_block containing a new PCE is transmitted.

Other elements are also specified. A list of one or more LFE’s is specified for application to this program. A list of one or more CCE’s (profile-dependent) is also provided, in order to allow for dialog management as well as different intensity coupling streams for different channels using the same main channels. A list of data streams associated with the program can also associate one or more data streams with a program. The program configuration element also allows for the specification of one monophonic and one stereophonic simulcast mixdown channel for a program.

Note that the MPEG-2 Systems standard [6] supports alternate methods of simulcast.

The PCE element is not intended to allow for rapid program changes. At any time when a given PCE, as selected by its element_instance_tag, defines a new (as opposed to repeated) program, the decoder is not obliged to provide audio signal continuity.

2.2.3 AAC/BSAC scalable core header

Configuration data for the scalable decoder. This is part of the MPEG-4 audio element identifier definition and will be covered there, when the final definition of this element is available. The following tables and definitions reflect the status of the current VM.

2.2.3.1 Definitions

op_mode	**definition**
0	core layer only (mono)
1	core layer only (stereo)
2	high layer only (mono)
3	high layer only (stereo)
4	core and high (mono, mono)
5	core and high (mono, stereo)
6	core and high (stereo, stereo)
7	core and intermediate and high (mono, mono, mono)
8	core and intermediate and high (mono, mono, stereo)
9	core and intermediate and high (mono, stereo, stereo)
10	core and intermediate and high (stereo, stereo, stereo)
11	intermediate and high (mono, mono)
12	intermediate and high (mono, stereo)
13	intermediate and high (stereo, stereo)
14	reserved
15	reserved

ccbrflsre	core coder bitrate (kbit/s)	core frame length (ms)	number of bytes per frame for the core coder	window length (samples)	example core coder
0	8	10	10	1920	G7.29
1	6.3	30	24	1920	G7.23

2	5.3	30	20	1920	G7.23
3	3.8	30	14	1920	MPEG-4 CELP core
4	4.9	20	12	1920	MPEG-4 CELP core
5	6	20	15	1920	MPEG-4 CELP core
6	7.7	20	19	1920	MPEG-4 CELP core
7	9.9	20	25	1920	MPEG-4 CELP core
8	11	10	14	1920	MPEG-4 CELP core
9	12.2	10	15	1920	MPEG-4 CELP core
10	4.8	30	18	1920	FS1016
11	6				MPEG-4 parametric IL core
12					
13					
14					
15					

intermediate_layers The number of enhancement layers -1. Only transmitted if more than one AAC layer is used

2.2.4 Twin-VQ header

2.2.4.1 Definitions

function indicates flags of supported functionalities.
samplingFrequencyIndex descriptor elment which indicates sampling rate.
bitrate indicates bitrate.

2.2.4.2 Decoder configuration

Configuration mode parameter MODE_VQ, bitrate parameter BITRATE, and sampling frequency parameter SAMPLING_FREQUENCY are set by syntax elements **bitrate, frameLength** and **samplingFrequencyIndex**.

```
if (lyr == 0){
   switch (samplingFrequencyIndex){
      case 24000:
         SAMPLING_FREQUENCY = 24000
         if (bitrate >= 6) {
            if (frameLength == 0) {
               MODE_VQ = 24_06;
            }
            else MODE_VQ = 24_06_960;
         break;
         }
      case 16000:
         SAMPLING_FREQUENCY = 16000.;
         if (bitrate >= 16) MODE_VQ = 16_16;
         break;
      case 8000:
         SAMPING_FREQUENCY = 8000.;
         if (bitrate >= 6) MODE_VQ = 08_06;
         }
         break;
      default:
         break;
   }
}
else{
   if (lyr < 2){
      if (frameLength == 0) {
         MODE_VQ = SCL_1;
      }
```

```
      else{
         MODE_VQ = SCL_1_960;
      }
   }
   else{
      if (frameLength == 0) {
         MODE_VQ = SCL_2;
      }
      else{
         MODE_VQ = SCL_2_960;
      }
   }
}
```

Lower limit of syntax elements **bitrate** is 6 for 48 kHz sampling, 16 kbit/s for 16 kbit/s, and 6 for 8 kHz sampling in the non-scaleable case. In the scaleable case, the lower limit of the **bitrate** is 8.

2.3 Decoding of the T/F Bitstream payload

2.3.1 Definitions

raw_data_stream() sequence of raw_data_block()'s

raw_data_block() block of raw data that contains audio data for a time period of 1024 samples, related information and other data. There are 8 bitstream elements, identified as bitstream element id_syn_ele. The audio elements in one raw data stream and one raw data block must have one and only one sampling rate. In the raw data block, several instances of the same id_syn_ele may occur, but each such instance of an id_syn_ele except for a data_stream_element must have a different 4-bit element_instance_tag. Therefore, in one raw data block, there can be from 0 to at most 16 of any id_syn_ele. The exceptions to this are the data_stream_element, the fill_element and the terminator element. If multiple data stream elements occur which have unique element_instance_tags then they are part of distinct data streams. If multiple data stream elements occur which have the same element_instance_tag then they are part of the same data stream. The fill_element has no element_instance_tag (since the content does not require subsequent reference) and can occur any number of times. The terminator element has no element_instance_tag and must occur exactly once , as it marks the end of the raw_data_block (Table 6.8).

id_syn_ele a bitstream element that identifies one of the following syntactic elements: (Table 6.8)

Syntactic Element	ID name	encoding	Abbreviation
single_channel_element	ID_SCE	0x0	SCE
channel_pair_element	ID_CPE	0x1	CPE
coupling_channel_element	ID_CCE	0x2	CCE
lfe_channel_element	ID_LFE	0x3	LFE
data_stream_element	ID_DSE	0x4	DSE
program_config_element	ID_PCE	0x5	PCE
fill_element	ID_FIL	0x6	FIL
terminator	ID_END	0x7	TERM

single_channel_element() abbreviaton SCE. Syntactic element of the bitstream containing coded data for a single audio channel. A single_channel_element() basically consists of an individual_channel_stream(). There may be up to 16 such elements per raw data block, each one must have a unique element_instance_tag (Table 6.9)

channel_pair_element() abbreviation CPE. Syntactic element of the bitstream containing data for a pair of channels. A channel_pair_element consists of two individual_channel_streams and additional joint channel coding information. The two channels may share common side information. The channel_pair_element has the same restrictions as

	the single channel element as far as element_instance_tag, and number of occurrances (Table 6.10).
coupling_channel_element()	Abbreviation CCE. Syntactic element that contains audio data for a coupling channel. A coupling channel represents the information for multi-channel intensity for one block, or alternately for dialogue for multilingual programming. The rules for number of coupling_channel_elements and instance tags are as for single_channel_elements (Table 6.18). See clause 12.3
lfe_channel_element()	Abbreviation LFE. Syntactic element that contains a low sampling frequency enhancement channel. The rules for the number of lfe_channel_elements and instance tags are as for single_channel_elements (Table 6.19). See clause 8.4
program_config_element()	Abbreviation PCE. Syntactic element that contains program configuration data. The rules for the number of program_config_elements and element instance tags are the same as for single_channel_elements (Table 6.21). PCE's must come before all other syntactic elements in a raw_data_block(). See clause 8.5
fill_element()	Abbreviation FIL. Syntactic element that contains fill data. There may be any number of fill elements, that can come in any order in the raw data block (Table 6.22). See clause 8.7
data_stream_element()	Abbreviation DSE. Syntactic element that contains data. Again, there are 16 element_instance_tags. There is, however, no restriction on the number of data_stream_elements with any one instance tag, as a single data stream may continue across multiple data_stream_elements with the same instance tag (Table 6.20). See clause 8.6
terminator (ID_END)	The terminator id_syn_ele ID_END indicates the end of a raw data block. There must be one and only one terminator per raw data block. (Table 6.8)
element_instance_tag	unique instance tag for syntactic elements other than terminator element and fill element. All syntactic elements containing instance tags may occur more than once, but, except for data_stream_elements, must have an unique element_instance_tag in each raw_data_block. This tag is also used to reference audio syntactic elements in a coupling_channel_element, and single_channel_elements, channel_pair_elements, lfe_channel_elements, data_channel_elements, and coupling_channel_elements inside a program_config_element, and provides the possibility of up to 16 independent program_config_elements (tables 6.9, 6.10, 6.18, 6.19, 6.20, 6.21, 6.22)
audio_channel_element	generic term for single_channel_element, channel_pair_element, coupling_channel_element and lfe_channel_element.

2.3.2 Buffer requirements

<u>Minimum decoder input buffer:</u>

The following rules are used to calculate the maximum number of bits in the input buffer both for the bitstream as a whole, for any given program, or for any given SCE/CPE/CCE:

The input buffer size is 6144 bits per SCE or independently switched CCE, plus 12288 bits per CPE. Both the total buffer and the individual buffer sizes are limited, so that the buffering limit can be calculated for either the entire bitstream, any entire program, or the individual audio elements permitting the decoder to break a multichannel bitstream into separate mono and stereo bitstreams which are decoded by separate mono and stereo decoders, respectively. Although the 6144 bit/CCE must be obeyed for dependent CCE's as well, any bits for dependent CCE's must be supplied from the total buffer requirements based on the independent CCE's, SCE's, and CPE's.

<u>Bit reservoir:</u>

The bit reservoir is controlled at the encoder. The maximum bit reservoir in the encoder depends on the number of channels and the mean bitrate. The maximum bit reservoir size for constant rate channels can be calculated by subtracting the mean number of bits per block from the minimum decoder input buffer size. For example, at 96 kbit/s for a stereo signal at 48 kHz sampling frequency the maximum bit reservoir size is 12288 bit- (96000 kbit/s / 48000 1/s * 1024) = 10240 bit. For variable bitrate channels the encoder must operate in a way that the input buffer requirements do not exceed the minimum decoder input buffer.

The state of the bit reservoir is transmitted in the buffer_fullness field as the number of available bits in the bit reservoir divided by the number of audio channels divided by 32 and truncated to an integer value.

<u>Maximum bit rate:</u>

The maximum bit rate depends on the audio sampling rate. The maximum bit rate per channel can be calculated based on the minimum input buffer according to the formula:

$$\frac{6144\,\frac{bit}{block}}{1024\,\frac{samples}{block}}\cdot sampling\_frequency$$

So, for example, this leads to the following maximum bit rates:

sampling_frequency	maximum bit rate / channel
48 kHz	288 kbit/sec
44.1 kHz	264.6 kbit/sec
32 kHz	192 kbit/sec

2.3.3 Decoding process

Assuming that the start of a raw_data_block is known, it can be decoded without any additional “transport-level” information and produces 1024 audio samples per output channel. The sampling rate of the audio signal may be specified in a program_config_element or it may be implied in the specific application domain. Assuming that the start of the first raw_data_block in a raw_data_stream is known, the sequence can be decoded without any additional “transport-level” information and produces 1024 audio samples per raw_data_block per output channel.

The raw_data_stream supports encoding for both constant rate and variable rate channels. In each case the structure of the bitstream and the operation of the decoder are identical except for some minor qualifications. For constant rate channels, the encoder may have to insert a FIL element to adjust the rate upwards to exactly the desired rate. A decoder reading from a constant rate channel must accumulate a minimum number of bits in its input buffer prior to the start of decoding so that output buffer underrun does not occur. In the case of variable rate, demand read channels, each raw_data_block can have the minimum length (rate) such that the desired audio quality is achieved, and in the decoder there is no minimum input data requirement prior to the start of decoding.

Examples of the simplest possible bitstreams are

bitstream segment	**output signal**
<SCE><TERM><SCE><TERM>…	mono signal
<CPE><TERM><CPE><TERM>…	stereo signal
<SCE><CPE><CPE><LFE><TERM><SCE><CPE><CPE><LFE><TERM>…	5.1 channel signal

where angle brackets (< >) are used to delimit syntactic elements. For the mono signal each SCE must have the same value in its **element_instance_tag**, and similarly, for the stereo signal each CPE must have the same value in its **element_instance_tag**. For the 5.1 channel signal each SCE must have the same value in its **element_instance_tag**, each CPE associated with the front channel pair must have the same value in its **element_instance_tag**, and each CPE associated with the back channel pair must have the same value in its **element_instance_tag**.

If these bitstreams are to be transmitted over a constant rate channel then they might include a fill_element to adjust the instantaneous bit rate. In this case an example of a coded stereo signal is

<CPE><FIL><TERM><CPE><FIL><TERM>…

If the bitstreams are to carry ancillary data and run over a constant rate channel then an example of a coded stereo signal is

<CPE><DSE><FIL><TERM><CPE><DSE><FIL><TERM>…

All data_stream_elements have the same element_instance_tag if they are part of the same data stream.

2.3.4 Decoding of a single_channel_element (SCE), channel_pair_element (CPE) and individual_channel_stream (ICS)

2.3.4.1 Definitions

Bit stream elements:

individual_channel_stream()	contains data necessary to decode one channel (Table 6.12)
ics_info()	contains side information necessary to decode an individual_channel_stream. The individual_channel_streams of a channel_pair_element may share one common ics_info (Table 6.11)
common_window	a flag indicating whether the two individual_channel_streams share a common ics_info or not. In case of sharing, the ics_info is part of the channel_pair_element and must be used for both channels. Otherwise, the ics_info is part of each individual_channel_stream (Table 6.10)
ics_reserved_bit	bit reserved for future use
window_sequence	indicates the sequence of windows as defined in Table 2.2 (Table 6.11)
window_shape	A 1 bit field that determines what window is used for the trailing part of this analysis window (Table 6.11)
max_sfb	number of scalefactor bands transmitted per group (Table 6.11)
scale_factor_grouping	A bit field that contains information about grouping of short spectral data (Table 6.11)

Help elements:

scalefactor window band	term for scalefactor bands within a window, given in table Table 2.3 to Table 2.5
scalefactor band	term for scalefactor band within a group. In case of EIGHT_SHORT_SEQUENCE and grouping a scalefactor band may contain several scalefactor window bands of corresponding frequency. For all other window_sequences scalefactor bands and scalefactor window bands are identical.
g	group index
win	window index within group
sfb	scalefactor band index within group
swb	scalefactor window band index within window
bin	coefficient index
num_window_groups	number of groups of windows which share one set of scalefactors
window_group_length[g]	number of windows in each group.
bit_set(bit_field,bit_num)	function that returns the value of bit number bit_num of a bit_field (most right bit is bit 0)
num_windows	number of windows of the actual window sequence
num_swb_long_window	number of scalefactor bands for long windows. This number has to be selected depending on the sampling frequency. See clause 2.3.8.
num_swb_short_window	number of scalefactor window bands for short windows. This number has to be selected depending on the sampling frequency. See clause 2.3.8.
num_swb	number of scalefactor window bands for shortwindows in case of EIGHT_SHORT_SEQUENCE, number of scalefactor window bands for long windows otherwise
swb_offset_long_window[swb]	table containing the index of the lowest spectral coefficient of scalefactor band sfb for long windows. This table has to be selected depending on the sampling frequency. See clause 2.3.8.
swb_offset_short_window[swb]	table containing the index of the lowest spectral coefficient of scalefactor band sfb for short windows. This table has to be selected depending on the sampling frequency. See clause 2.3.8.
swb_offset[swb]	table containing the index of the lowest spectral coefficient of scalefactor band sfb for short windows in case of EIGHT_SHORT_SEQUENCE, otherwise for long windows
sect_sfb_offset[g][section]	table that gives the number of the start coefficient for the section_data() within a group. This offset depends on the window_sequence and scale_factor_grouping.
sampling_frequency_index	see clause 2.1.1.1

2.3.4.2 Decoding process

Single_channel_element and channel_pair_element
A single_channel_element is composed of an element_instance_tag and an individual_channel_stream. In this case ics_info is always located in the individual_channel_stream.
A channel_pair_element begins with an element_instance_tag and common_window flag. If the common_window equals '1', then ics_info is shared amongst the two individual_channel_stream elements and the MS information is transmitted. If common_window equals '0', then there is an ics_info within each individual_channel_stream and there is no MS information.

Decoding an individual_channel_stream (ICS)
In the individual_channel_stream, the order of decoding is:

Get global_gain
Get ics_info (parse bitstream if common information is not present)
Get Section Data
Get Scalefactor Data, if present
Get pulse data if present
Get TNS data, if present
Get gain control data, if present
Get Spectral Data, if present.

The process of recovering pulse_data is described in section 9, tns_data in section 14, and gain_control data in section 16. An overview of how to decode ics_info (clause 8.3), section data (clause 9), scalefactor data (clause 9 and 11), and spectral data (clause 9) will be given here.

Recovering ics_info
For single_channel_elements ics_info is always located immediately after the global_gain in the inidividual_channel_stream. For a channel pair element there are two possible locations for the ics_info. If each individual channel in the pair window switch together then the ics_info is located immediately after common_window in the channel_pair_element() and common_window is set to 1. Otherwise there is an ics_info immediately after global_gain in each of the two individual_channel_stream() in the channel_pair_element and common_window is set to 0.

ics_info carries window information associated with an ICS and thus permits channels in a channel_pair to switch separately if desired. In addition it carries the max_sfb which places an upper limit on the number of ms_used[] and predictor_used[] bits that must be transmitted. If the window_sequence is EIGHT_SHORT_SEQUENCE then scale_factor_grouping is transmitted. If a set of short windows form a group then they share scalefactors as well as intensity stereo positions and have their spectral coefficients interleaved. The first short window is always a new group so no grouping bit is transmitted. Subsequent short windows are in the same group if the associated grouping bit is 1. A new group is started if the associated grouping bit is 0. It is assumed that grouped short windows have similar signal statistics. Hence their spectra are interleaved so as to place correlated coefficients next to each other. The manner of interleaving is indicated in figure 8.3. ics_info also carries the prediction data for the individual channel or channel pair (see clause 13).

Recovering Sectioning Data
In the ICS, the information about one long window, or eight short windows, is recovered. The sectioning data is the first field to be decoded, and describes the Huffman codes that apply to the scalefactor bands in the ICS (see clause 9 and 11). The form of the section data is:

sect_cb The codebook for the section

and

sect_len The length of the section. This length is recovered by reading the bitstream sequentially for a section length, adding the escape value to the total length of the section until a non-escape value is found, which is added to establish the total length of the section. This process is clearly explained in the C-like syntax description. Note that within each group the sections must delineate the scalefactor bands from zero to **max_sfb** so that the first section within each group starts at bands zero and the last section within each group ends at **max_sfb**.
The sectioning data describes the codebook, and then the length of the section using that codebook, starting from the first scalefactor band and continuing until the total number of scalefactor bands is reached.
After this description is provided, all scalefactors and spectral data corresponding to codebook zero are zeroed, and no values corresponding to these scalefactors or spectral data will be transmitted. When scanning for scale-factor data it is important to note that scalefactors for any scalefactor bands whose Huffman codebook is zero

will be omitted. Similarly, all spectral data associated with Huffman codebook zero are omitted (see clause 9 and 11)
In addition spectral data associated with the scalefactor bands that have an intensity codebook will not be transmitted, but intensity steering coefficients will be transmitted in place of the scalefactors, as described in section 12.2

Scalefactor Data Parsing and Decoding
For each scalefactor band that is not in a section coded with the zero codebook (ZERO_HCB), a scalefactor is transmitted. These will be denoted as 'active' scalefactor bands and the associated scalefactors as active scalefactors. Global gain, the first bitstream element in an ICS, is typically the value of the first active scalefactor. All scalefactors (and steering coefficients) are transmitted using Huffman coded DPCM relative to the previous active scalefactor (see clause 9 and 11). The first active scalefactor is differentially coded relative to the global gain. Note that it is not illegal, merely inefficient, to provide a global_gain that is different from the first active scalefactor and then a non-zero DPCM value for the first scalefactor DPCM value. If any intensity steering coefficients are received interspersed with the DPCM scalefactor elements, they are sent to the intensity stereo module, and are not involved in the DPCM coding of scalefactor values (see clause 12.2). The value of the first active scalefactor is usually transmitted as the global_gain with the first DPCM scalefactor having a zero value. Once the scalefactors are decoded to their integer values, the actual values are found via a power function (see clause 11).

Spectral Data Parsing and Decoding
The spectral data is recovered as the last part of the parsing of an ICS. It consists of all the non-zeroed coefficients remaining in the spectrum or spectra, ordered as described in the ICS_info. For each non-zero, non-intensity codebook, the data are recovered via Huffman decoding in quads or pairs, as indicated in the noiseless coding tool (see clause 9). If the spectral data is associated with an unsigned Huffman codebook, the necessary sign bits follow the Huffman codeword (see section 9.3). In the case of the ESCAPE codebook, if any escape value is received, a corresponding escape sequence will appear after that Huffman code. There may be zero, one or two escape sequences for each codeword in the ESCAPE codebook, as indicated by the presence of escape values in that decoded codeword. For each section the Huffman decoding continues until all the spectral values in that section have been decoded. Once all sections have been decoded, the data is multiplied by the decoded scalefactors and deinterleaved if necessary.

2.3.4.3 Windows and window sequences

Quantization and coding is done in the frequency domain. For this purpose, the time signal is mapped into the frequency domain in the encoder. The decoder performs the inverse mapping as described in clause 15. Depending on the signal, the coder may change the time/frequency resolution by using two different windows: LONG_WINDOW and SHORT_WINDOW. To switch between windows, the transition windows LONG_START_WINDOW and LONG_STOP_WINDOW are used. Table 2.1 lists the windows, specifies the corresponding transform length and shows the shape of the windows schematically. Two transform lengths are used: 1024 (referred to as long transform) and 128 coefficients (refered to as short transform).
Window sequences are composed of windows in a way that a raw_data_block always contains data representing 1024 output samples. The bitstream element **window_sequence** indicates the window sequence that is actually used. Table 2.2 lists how the window sequences are composed of individual windows. Refer to clause 15 for more detailed information about the transform and the windows.

2.3.4.4 Scalefactor bands and grouping

Many tools of the decoder perform operations on groups of consecutive spectral values called scalefactor bands (abbreviation 'sfb'). The width of the scalefactor bands is built in imitation of the critical bands of the human auditory system. For that reason the number of scalefactor bands in a spectrum and their width depend on the transform length and the sampling frequency. Table 2.3 to Table 2.5 list the offset to the beginning of each scalefactor band for the transform lengths 1024 and 128 and the different sampling frequencies, respectively.
To reduce the amount of side information in case of sequences which contain SHORT_WINDOWS, consecutive SHORT_WINDOWs may be grouped (see figure 8.1). The information about the grouping is contained in the **scale_factor_grouping** bitstream element. Grouping means that only one set of scalefactors is transmitted for all grouped windows as if there was only one window. The scalefactors are then applied to the corresponding spectral data in all grouped windows. To increase the efficiency of the noiseless coding (see clause 9), the spectral data of a group is transmitted in an interleaved order given in clause 8.3.5. The interleaving is done on a scalefactor band by scalefactor band basis, so that the spectral data can be grouped to form a virtual scalefactor band to which the common scalefactor can be applied. Within this document the expression 'scalefactor band'

(abbreviation ‘sfb’) denotes these virtual scalefactor bands. If the scalefactor bands of the single windows are referred to, the expression ‘scalefactor window band’ (abbreviation ‘swb’) is used. Due to its influence on the scalefactor bands, grouping affects the meaning of section_data (see clause 9), the order of spectral data (see clause 8.3.5), and the total number of scalefactor bands. For a LONG_WINDOW scalefactor bands and scalefactor window bands are identical since there is only one group with only one window.
To reduce the amount of information needed for the transmission of side information specific to each scalefactor band, the bitstream element **max_sfb** is transmitted. Its value is one greater than the highest active scalefactor band in all groups. **max_sfb** has influence on the interpretation of section data (see clause 9), the transmission of scalefactors (see clause 9 and 11), the transmission of predictor data (see clause 13) and the transmission of the ms_mask (see clause 12.1).

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

- how to determine the number of windows in a window_sequence *num_windows*
- how to determine the number of window_groups *num_window_groups*
- how to determine the number of windows in each group *window_group_length[g]*
- how to determine the total number of scalefactor window bands *num_swb* for the actual window type
- how to determine *swb_offset[swb]*, the offset of the first coefficient in scalefactor window band *swb* of the window actually used
- how to determine *sect_sfb_offset[g][section]*, the offset of the first coefficient in section *section*. This offset depends on **window_sequence** and **scale_factor_grouping** and is needed to decode the spectral_data().

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

```
fs_index = sampling_frequency_index;
switch( window_sequence ) {
   case ONLY_LONG_SEQUENCE:
   case LONG_START_SEQUENCE:
   case LONG_STOP_SEQUENCE:
      num_windows = 1;
      num_window_groups = 1;
      window_group_length[num_window_groups-1] = 1;
      num_swb = num_swb_long_window[fs_index];
      /* preparation of sect_sfb_offset for long blocks */
      /* also copy the last value! */
      for( i=0; i< max_sfb + 1; i++ ) {
         sect_sfb_offset[0][i] = swb_offset_long_window[fs_index][i];
         swb_offset[i] = swb_offset_long_window[fs_index][i];
      }
      break;
   case EIGHT_SHORT_SEQUENCE:
      num_windows = 8;
      num_window_groups = 1;
      window_group_length[num_window_groups-1] = 1;
      num_swb = num_swb_short_window[fs_index];
      for( i=0; i< num_swb_short_window[fs_index] + 1; i++ )
         swb_offset[i] = swb_offset_short_window[fs_index][i];
      for( i=0; i< num_windows-1; i++) {
         if( bit_set(scale_factor_grouping,6-i)) == 0 ) {
            num_window_groups += 1;
            window_group_length[num_window_groups-1] = 1;
         }
         else {
            window_group_length[num_window_groups-1] += 1;
         }
      }
      /* preparation of sect_sfb_offset for short blocks */
      for( g=0; g<num_window_groups; g++ ) {
         sect_sfb = 0;
         offset = 0;
         for( i=0; i< max_sfb; i++ ) {
            width = swb_offset_short_window[fs_index][i+1] -
                    swb_offset_short_window[fs_index][i];
            width *= window_group_length[g];
```

```
            sect_sfb_offset[g][sect_sfb++] = offset;
            offset += width;
         }
         sect_sfb_offset[g][sect_sfb] = offset;
      }
      break;
   default:
      break;
}
```

2.3.4.5 Order of spectral coefficients in spectral_data

For ONLY_LONG_SEQUENCE windows (num_window_groups = 1, window_group_length[0] = 1) the spectral data is in ascending spectral order, as shown in figure 8.2.

For the EIGHT_SHORT_SEQUENCE window, the spectral order depends on the grouping in the following manner:

- Groups are ordered sequentially
- Within a group, a scalefactor band consists of the spectral data of all grouped SHORT_WINDOWs for the associated scalefactor window band. To clarify via example, the length of a group is in the range of one to eight SHORT_WINDOWs.
    - If there are eight groups each with length one (num_window_groups = 8, window_group_length[0] = 1), the result is a sequence of eight spectrums, each in ascending spectral order.
    - If there is only one group with length eight (num_window_groups = 1, window_group_length[0] = 1), the results is that spectral data of all eight SHORT_WINDOWs is interleaved by scalefactor window bands.
    - Figure 8.3 shows the spectral ordering for an EIGHT_SHORT_SEQUENCE with grouping of SHORT_WINDOWs according to figure 8.1 (num_window_groups = 4).
- Within a scalefactor window band, the coefficients are in ascending spectral order.

2.3.4.6 Output word length

The global gain for each audio channel is scaled such that the integer part of the output of the IMDCT can be used directly as a 16-bit PCM audio output to a digital-to-analog (D/A) converter. This is the default mode of operation and will result in correct audio levels. If the decoder has a D/A converter that has greater than 16-bit resolution then the output of the IMDCT can be scaled up such that the appropriate number of fractional bits are included to form the desired D/A word size. In this case the level of the converter output would be matched to that of a 16-bit D/A, but would have the advantage of greater signal dynamic range and lower converter noise floor. Similarly, shorter D/A word lengths can be accommodated.

2.3.4.7 Use of emphasis

This standard does not support pre-emphasis and de-emphasis and no signalling bits are provided to transport such information in the bitstream.

2.3.4.8 Matrix-mixdown Method

2.3.4.8.1 Description

The matrix-mixdown method applies only for mixing a 3-front/2-back speaker configuration, 5-channel program, down to a stereo or a mono program. It is not applicable to any program with other than the 3/2 configuration and only decoders capable of decoding a 3/2 configuration must be able to decode this mode.

2.3.4.8.2 Definitions

matrix_mixdown_idx_present One bit indicating that a stereo matrix coefficient index is present (see Table 6.21). For all configurations other than the 3/2 format this bit must be zero.

matrix_mixdown_idx A two bit field that indicates that the coefficient to be used in the 5-channel to 2-channel matrix-mixdown. Possible matrix coefficients are listed in section 8.3.8.5.

pseudo_surround_enable One bit indicating that pseudo surround decoding is possible.

2.3.4.8.3 Matrix-mixdown process

A derived stereo signal can be generated within a matrix-mixdown decoder by use of one of the two following sets of equations.

Set 1:

$$L' = \frac{1}{1+1/\sqrt{2}+A} \cdot [L + C/\sqrt{2} + A \cdot L_S]$$

$$R' = \frac{1}{1+1/\sqrt{2}+A} \cdot [R + C/\sqrt{2} + A \cdot R_S]$$

Set 2:

$$L' = \frac{1}{1+1/\sqrt{2}+2 \cdot A} \cdot [L + C/\sqrt{2} - A \cdot (L_S + R_S)]$$

$$R' = \frac{1}{1+1/\sqrt{2}+2 \cdot A} \cdot [R + C/\sqrt{2} + A \cdot (L_S + R_S)]$$

Where L, C, R, LS and RS are the source signals, L' and R' are the derived stereo signals and A is the matrix coefficient indicated by matrix_mixdown_idx. LFE channels are omitted from the mixdown.

If pseudo_surround_enable is not set, then only set 1 should be used. If pseudo_surround_enable is set, then either set 1 or set 2 equations can be used, depending on whether the receiver has facilities to invoke some form of surround synthesis.

As further information it should be noted that one can derive a mono signal using the following equation:

$$M = \frac{1}{3+2 \cdot A} \cdot \left[L + C + R + A \cdot (L_S + R_S)\right]$$

2.3.4.8.4 Advisory

The matrix-mixdown provision enables a mode of operation which may be beneficial to some operators in some circumstances. However, it is advised that this method should not be used. The psychoacoustic principles on which the audio coding are based are violated by this form of post-processing, and a perceptually faithful reconstruction of the signal cannot be guaranteed. The preferred method is to use the stereo or mono mixdown channels in the AAC syntax to provide stereo or mono programming which is specifically created by conventional studio mixing prior to bitrate reduction.

The stereo and mono mixdown channels additionally enable the content provider to separately optimize the stereo and multichannel program mixes - this is not possible by using the matrix-mixdown method.

It is additionally relevant to note that, due to the algorithms used for the multichannel and stereo mixdown coding, a better combination of quality and bitrate is usually provided by use of the stereo mixdown channels than can be provided by the matrix-mixdown process.

2.3.4.8.5 Tables

Matrix-mixdown coefficients

matrix_mixdown_idx	A
0	$1/\sqrt{2}$
1	$1/2$
2	$1/(2\sqrt{2})$
3	0

2.3.5 Low Frequency Enhancement Channel (LFE)

2.3.5.1 General

In order to maintain a regular structure of the decoder, the lfe_channel_element is defined as a standard individual_channel_stream(0) element, i.e. equal to a single_channel_element. Thus, decoding can be done using the standard procedure for decoding a single_channel_element.

In order to accomodate a more bitrate and hardware efficient implementation of the LFE decoder, however, several restrictions apply to the options used for the encoding of this element:

- The window_shape field is always set to 0, i.e. sine window (see 6.3, Table 6.11)
- The window_sequence field is always set to 0 (ONLY_LONG_SEQUENCE) (see 6.3, Table 6.11)
- The index of the highest non-zero spectral coefficient present in the element is 12
- No Temporal Noise Shaping is used, i.e. tns_data_present is set to 0 (see 6.3, Table 6.12)
- No prediction is used, i.e. predictor_data_present is set to 0 (see 6.3, Table 6.11)

The presence of LFE channels depends on the profile used. Refer to clause 7 for detailed information.

2.3.6 Data stream element (DSE)

Bitstream elements:

data_byte_align_flag	One bit indicating that a byte aligment is performed within the data stream element (Table 6.20)
count	Initial value for length of data stream (Table 6.20)
esc_count	Incremental value of length of data or padding element (Table 6.20)
data_stream_byte	A data stream byte extracted from bitstream (Table 6.20)

A data element contains any additional data, e.g. auxiliary information, that is not part of the audio information itself. Any number of data elements with the same element_instance_tag or up to 16 data elements with different element_instance_tags are possible. The decoding process of the data element is described in this clause.

Decoding process:

The first syntactic element to be read is the the 1 bit **data_byte_align_flag**. Next is the 8 bit value **count**. It contains the initial byte-length of the data stream. If **count** equals 255, its value is incremented by a second 8 bit value, **esc_count**, this final value represents the number of bytes in the data stream element. If **data_byte_align_flag** is set, a byte alignment is performed. The bytes of the data stream follow.

2.3.7 Fill element (FIL)

Bitstream elements:

count	Initial value for length of fill data (Table 6.22)
esc_count	Incremental value of length of fill data (Table 6.22)
fill_byte	byte to be discarded by the decoder (Table 6.22)

Fill elements have to be added to the bitstream if the bitsum of all audio data together with all additional data is lower than the minimum allowed number of bits in this frame necessary to reach the target bitrate. Under normal conditions fill bits are avoided and free bits are used to fill up the bit reservoir. Only if the bitreservoir is full, fill bits are written. Any number of fill elements are allowed.

Decoding process:

The syntactic element **count** gives the initial value of the length of the fill data. In the same way as for the data element this value is incremented with the value of **esc_count** if **count** equals 15. The resulting number gives the number of **fill_byte**s to be read.

2.3.8 Scalable core + AAC/BSAC elements

The scalable core plus AAC elements provides one way of achieving bit rate scalability. It is based on the calculation of a difference signal between the outüut signal of a core coder and the original input signal. At least one enhancement layer based on the AAC/BSAC based VM modules is used. Joint stereo coding and mixed mono/stereo configurations are possible. All AAC joint stereo modes are available in the combined coder.

2.3.8.1 Definitions

diff_control_lr	Element used in a mono T/F / stereo T/F configuration, to control the interaction of the M-channel with the L and R channel.
stereo_flag	Set, if there is a stereo enhancement stage.
mono_layer	Signals a mono T/F layer.
mono_stereo_flag	Set, if it is the first stereo layer.
last_max_sfb	max_sfb of the previous scalability layer.
last_max_sfb_ms	max_sfb of the previous stereo scalability layer.
core_flag	Set, if a core coder is present

2.3.8.2 Decoding / Specifications

Core coder integration

There is no principal limitation on to which core coder can be used. However, in general the core coder should encode the waveform of the input signal, to allow a useful difference signal to be calculated. For all CELP-type speech coders the post-filter has to be switched off for the use of it's output signal in the scalable coder. If a continuous bitstream is desired (in opposition to a packetized transmission of core stream and enhancement layer streams), common bitstream frames are desirable. In order to allow for these common frames, the T/F modules provide -in addition to the standard 2048 length - also a 1920 samples per frame option. This allows for AAC frame lengths of a multiple of 5 and 10 ms. The following table gives an overview:

Sampling rate (Hz):	96	64	48	32	24	16	8
Frame length (1920, ms)	10	15	20	30	40	60	120

These frame lengths allow for an easy integration of the MPEG-4 VM CELP core, and forto the construction of bitstream frames which integrate speech coders standardized elsewhere, which usually have a freame length of a multiple of 10 ms (information on the integration of some standard CELP coders are given in the informative annex). Some combinations of core coders and T/F coder sampling rates require different numbers of core coder frames and T/F frames to build common integrated frames. An example is given in the transport layer used in the reference software, which is described in section 1.1.

Furthermore also common bitstream frames at sampling rates of 44.1 and 22.05 can be achieved, by adjusting the sampling rate of the core coder to match either AAC frames with either 2048 or 1920 samples window length.

Sampling rates of 12 and 11.025 kHz are possible, if a core coder operating at these sampling rates is used. The available combinations are/will be defined in the element identifier elements. The current status, which is used in the reference software, uses the preliminary table given in the header section 1.2.

The ratio of the sampling rate of the core coder and the sampling rate of the enhancement coder must be an integer, to allow for the upsampling tool defined in section 3 to be used.

The example of a transport multiplex given in section 1.1 allows for a delay optimized multiplex of core and enhancement layer streams. This stream allows the original delay of the core coder to be utilized, if only the core decoder is decoded. The bitstream of the core layer then has to be delayed by the equivivalent amount of the maximum bit buffer allowed.

Enhancement layers

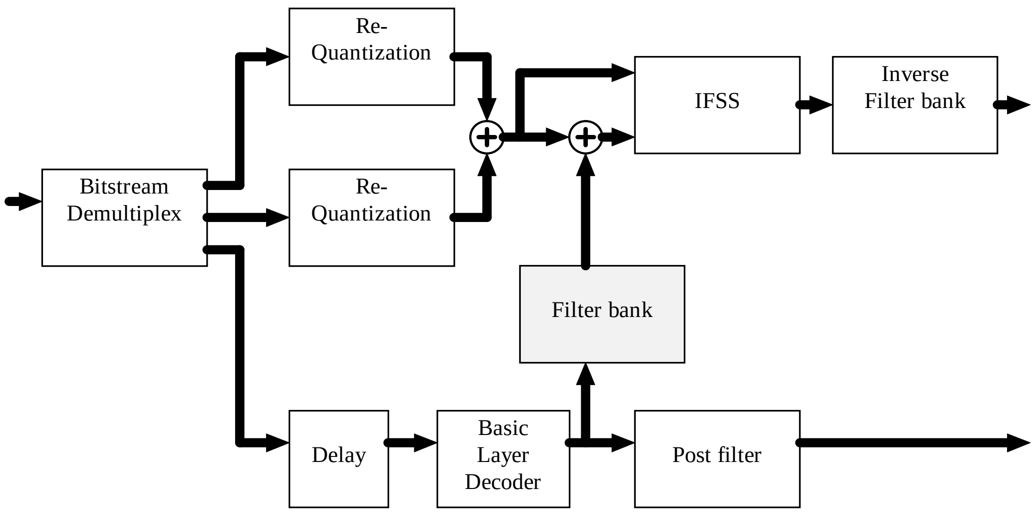
Both, AAC-like enhancement layers, and BSAC scalable coding can be used to enhance the quality of the core stream. In the case of AAC multiple enhancement Quantization&Coding (Q&C) stages are possible by difference encoding in the spectrum.

Up to five enhancement layers are defined in the scalable header syntax (To be changed depending on profiles/levels). The bitrate of the additional layers can be any bit rate possible for the AAC/BSAC enhancement stages.

Different basic configurations are possible:

1.	mono core	mono scal. T/F Q&C	
2.	stereo core	stereo scal. T/F Q&C	
3.	mono core	stereo scal. T/F Q&C	
4.	mono core	mono scal. T/F Q&C	stereo scal. T/F Q&C
5.		mono scal. T/F Q&C	stereo scal. T/F Q&C
6.		mono scal. T/F Q&C	

Figure 1 below shows the basic configuration of the mono / mono or stereo/stereo configuration. In the case of stereo the appropriate inverse joint stereo operations have to be performed before the inverse filter bank is applied.



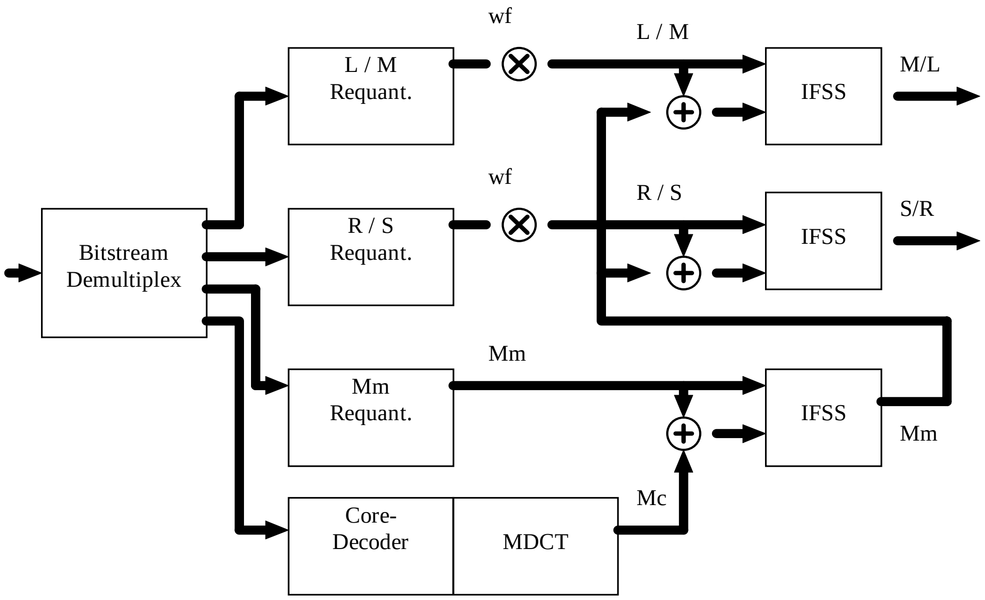
Decoding of the mono/mono and stereo/stereo modes 1-2:

Reconstruct all T/F coded spectra of in all scale factor bands as specified in the „Noiseless Coding“ and „Scalefactors“ or BSAC sections, respectively. Add all these contributions to form the combined T/F spectrum ‘S_{TF}’.

If a core channel is present, decode the core and up-sample and transform the decoded core signal into the frequency domain using the upsampling (filterbank) tool to get the core spectrum ‘S_{C}’. Apply the Inverse Frequency Selective Switch (IFSS) tool to get the complete output spectrum ‘S_{O}’. If TNS is used the encoder TNS filter is applied to the core spectrum S_C. The normal decoder TNS-filter is applied to S_O before the calculation of the inverse Filterbank.

In the stereo/stereo case (mode 2) instead of S_{TF} two signals $S_{TF-L/M}$ and $S_{TF-R /S}$ are available The usual inverse joint stereo procedures are applied to calculate S_{TF-L} and S_{TF-R} . These signals are then combined in IFFS(left channel) and IFSS(right channel) with the two core spectra S_{C-L} and S_{C-R}.to get the output spectra S_{O-L} and S_{O-R}.

Figure 2 below shows the block diagram for the combined mono/stereo modes 3-5.



<u>Decoding of the mixed mono / stereo modes 3-5:</u>

Reconstruct the spectra M'/S, or L'/R', in all scale factor bands as specified in the „Noiseless Coding“ and „Scalefactors“ sections or BSAC sections, respectively. If a separate T/F coded M-channel is present in mode 4 and 5 also decode this channel as described above to get the signal Mm.

If (in mode 3 and 4) a core channel is present, decode the core, up-sample and transform into the frequency domain as described above to get the signal Mc.

In mode 4 further apply the Inverse Frequency Selective Switch (IFSS) as described for the „Core + T/F scalability“ tool to get the signal Mm'. In this case diff_control_m(win,sb) is used to control the switching. In mode 4 Mm' is identical to Mc. In mode 5 Mm' is identical to Mm.

For all bands where M/S coding is selected only the IFSS information for the M-switch is transmitted in diff_control[0][win][sfb]. Diff_control[1][win][sfb] is not transmitted, but set to 1, in order to avoid processing of the S signal.

Generate the spectra M/S or L/R for each scalefactor band using diff_control(ch,win,sb) to control the switching. If ms_used[sfb] shows L/R encoding, *wf* = 2.0 (for reversing the MS-Matrix factor 0.5), and if it shows M/S coding *wf* = 1.0 is used to modify the spectra before feeding them to the IFFS units.

Finally, on the L/R, or M/S output signals the inverse M/S matrix is applied as described in the „Joint coding“ tool to get the final L and R spectra.

2.3.9 VQ single element and VQ scaleable element

2.3.9.1 Definitions

vq_single_element() data element to decode TwinVQ data.
vq_scaleable_element() data element to decode scaleable TwinVQ data.
window_sequence indicates type of window sequence.

num	**window_sequence**
0	ONLY_LONG_SEQUENCE

1	LONG_SHORT_SEQUENCE
2	ONLY_SHORT_SEQUENCE
3	SHORT_LONG_SEQUENCE
4	SHORT_MEDIUM_SEQUENCE
5	MEDIUM_LONG_SEQUENCE
6	LONG_MEDIUM_SEQUENCE
7	MEDIUM_SHORT_SEQUENCE
8	ONLY_MEDIUM_SEQUENCE

After receiving the window sequence information, a parameter BLEN_TYPE is set as listed as follows:

window_sequence	BLEN_TYPE
ONLY_LONG_SEQUENCE LONG_SHORT_SEQUENCE SHORT_LONG_SEQUENCE LONG_MEDIUM_SEQUENCE MEDIUM_LONG_SEQUENCE	LONG
ONLY_MEDIUM_SEQUENCE MEDIUM_SHORT_SEQUENCE SHORT_MEDIUM_SEQUENCE	MEDIUM
ONLY_SHORT_SEQUENCE	SHORT

vq_data() A data block containing one frame of syntax elements for VQ base layer element and VQ enhancement layer element.

optional_info indicates postfilter switch.

fb_shift indicates location of active frequency band in enhancement layer

index_blim_h indicates the upper boundary for the bandwidth control process of the spectrum normalization tool.

index_blim_l indicates the lower boundary for the bandwidth control process of the spectrum normalization tool

index_shape0 indicates the codevector number of the shape codebook 1 and polarity of the codevector for the interleaved vector quantization tool.

index_shape1 indicates the codevector number of the shape codebook 1 of the interleaved vector quantization tool.

index_env indicates the codevector number of the Bark-scale envelope codebook of the spectrum normalization tool.

index_fw_alf indicates the prediction switch of the Bark-scale envelope coding of the spectrum normalization tool.

index_gain indicates the gain factor of the spectrum normalization tool.

index_gain_sb indicates the sub-block gain factor of the spectrum normalization tool.

index_lsp0 indicates LSP MA prediction switch of the spectrum normalization tool.

index_lsp1 indicates codevector number of the first-stage LSP VQ in the spectrum normlization tool.

index_lsp2 indicates codevector number of the second-stage LSP VQ in the spectrum normalization tool.

index_shape0_p indicates the codevector number of codebook 0 of the periodic peak component coding in the spectrum normalization tool.

index_shape1_p indicates the codevector number of codebook 1 of the periodic peak component coding in the spectrum normalization tool.

index_pit indicates the base frequency of the periodic peak component in the spectrum normalization tool.

index_pgain indicates the gain factor of the periodic peak component in the spectrum normalization tool.

2.3.9.2 Parameter setting

Following parameters are used for decoding of VQ base layer element and the VQ enhancement layer element:

N_CH	number of channels defined by the system layer
N_DIV	number of sub-vector division for interleaved vector quantization
N_SF	number of filterbank subblocks in a frame
FW_N_DIV	number of codebook division for the Bark-scale envelope quantization
LSP_SPLIT	number of subvectors for LSP VQ.
ppc_present	switch for the periodic peak component coding
N_DIV_P	number of sub-vector division for periodic peak component coding

These parameters are also referenced from the interleave vector quantization tool and the spectrum normalization tool.

2.3.9.2.1 N_DIV

Number of sub-vector division for interleaved vector quantization, N_DIV is calculated according to the decoder status. This calculation is defined in section 2.3.9.3.2.

2.3.9.2.2 N_SF

Number of filterbank subblocks, N_SF, is set according to the parameters MODE_VQ and BLEN_TYPE as listed below:

MODE_VQ	BLEN_TYPE	N_SF
	LONG	1
24_06 24_06_960 SCL_1 SCL_1_960 SCL_2 SCL_2_960	SHORT	8
16_16 08_06	MEDIUM	2
	SHORT	8

2.3.9.2.3 FW_N_DIV

Number of sub-vector division for the Bark-scale envelope coding, FW_N_DIV, is set according to the parameters MODE_VQ and BLEN_TYPE as listed below:

MODE_VQ	BLEN_TYPE	FW_N_DIV
24_06/ 24_06_960	LONG SHORT	7 1
16_16	LONG MEDIUM SHORT	3 2 1
08_06	LONG MEDIUM SHORT	3 2 1
SCL_1/ SCL_1_960	LONG SHORT	8 1
SCL_2/ SCL_2_960	LONG SHORT	8 1

2.3.9.2.4 LSP_SPLIT

This parameter defines the number of sub-vectors for 2nd-stage LSP vector quantization in spectrum normalization tool. Values are asigned according to the parameter MODE_VQ:

MODE_VQ	LSP_SPLIT
24_06	3
24_06_960	3
16_16	3
08_06	3
SCL_1	3
SCL_1_960	3
SCL_2	3
SCL_2_960	3

2.3.9.2.5 ppc_present

This parameter activates the periodic peak component coding process of the spectrum normalization tool. The value is set as follows:

```
if (BLEN_TYPE == LONG && (MODE_VQ == 16_16 || MODE_VQ == 08_06))
   ppc_present = TRUE;
else
   ppc_present = FALSE;
```

2.3.9.2.6 N_DIV_P

This parameter indicates of number of sub-vector division of the periodic peak component coding in spectrum normalization tool. The value is always set to 2.

2.3.9.3 Bit allocation

2.3.9.3.1 Spectrum normalization tool

For syntax elements listed below, the number of bits is set according to parameters MODE_VQ and BLEN_TYPE:

index_blim_h
index_blim_l
index_env
index_gain
index_gain_sb
index_lsp1
index_lsp2
index_pit
index_pgain

Number of bits are set as follows:

MODE_VQ	BLEN_TYPE	**env**	**blim_h**	**blim_l**	**gain**	**gain_sb**	**lsp1**	**lsp2**
24_06	LONG	6	0	0	9	4	6	4
24_06_960	SHORT	0						
16_16	LONG	6	2	1	8	5	6	4
	MEDIUM	5						
	SHORT	6						

08_06	LONG	6	0	0	8	4	5	3
	MEDIUM	6						
	SHORT	3						
SCL_1	LONG	6	0	0	8	4	6	4
SCL_1_960	SHORT	0						
SCL_2	LONG	6	0	0	7	4	6	4
SCL_2_960	SHORT	0						

The number of bits for index_pit is 9 for MODE_VQ == 16_16, 8 for MODE_VQ == 08_06.
The number of bits for index_pgain is 7 for MODE_VQ == 16_16, 6 for MODE_VQ == 08_06.

2.3.9.3.2 Interleaved vector quantization tool

Parameter N_DIV, Number of bits of shape code index 0, bits0, and number of bits of shape code index 1, bits1, are calculated as following procedure:

First, number of bits for side information, bits_for_side_information is calculated as follows:

```
bits_for_side_information =
   4 + OPT_TBIT + LSP_TBIT + GAIN_TBIT + FW_TBIT + PIT_TBIT+ used_bits
```

where used_bit is number of bits used by tools other than spectrum normalization tool. OPT_TBIT, LSP_TBIT, GAIN_TBIT, FW_TBIT, and PIT_TBIT is number of bits for optional information, lsp coding, gain coding, Bark-scale envelope coding, and periodic peak components coding respectively. They are set as follows:

```
LSP_TBIT = (LSP_BIT0+LSP_BIT1+(LSP_BIT2*LSP_SPLIT)) * N_CH;
if (FW_N_BIT>0){
   FW_TBIT = ((FW_N_BIT * FW_N_DIV + 1) * N_SF) * N_CH;
}
else{
   FW_TBIT = 0;
}

switch(BLEN_TYPE){
   case SHORT:
      OPT_TBIT = 0;
      GAIN_TBIT = (GAIN_BIT + SUB_GAIN_BIT * N_SF) * N_CH;
      PIT_TBIT = 0;
   break;
   case MEDIUM:
      OPT_TBIT = 2;
      GAIN_TBIT = (GAIN_BIT + SUB_GAIN_BIT * N_SF) * N_CH;
      PIT_TBIT = 0;
      break;
   default:
      OPT_TBIT = 2;
      GAIN_TBIT = GAIN_BIT * N_CH;
      if (ppc_present == TRUE}
         PIT_TBIT = PIT_N_BIT + (BASF_BIT + PGAIN_BIT) * N_CH;
      else
         PIT_TBIT = 0;
   break;
}
```

Parameters LSP_BIT0, LSP_BIT1, LSP_BIT2, LSP_SPLIT, FW_N_BIT, FW_N_DIV, GAIN_BIT, SUB_GAIN_BIT, SUB_GAIN_BIT, PIT_N_BIT, BASF_BIT, and PGAIN_BIT is set according to the parameters BLEN_TYPE and MODE_VQ as listed in tables from 3.9.1 to 3.9.7.

Number of available bits, bits_available_vq is calculated as follows:

```
available_vq =
   (int)(FRAME_SIZE * BITRATE/SAMPLING_FREQUENCY) -
   bits_for_side_information
```

Finally, number of shape sub-vector, `N_DIV` and number of bits for shape code indexes, bits0 and bits1, are calculated as follows:

```
N_DIV = ((int)((bits_available_vq + MAXBIT*2-1)/(MAXBIT*2)));
bits = (bits_available_vq + N_DIV - 1 - idiv) / N_DIV;
bits0 = (int)(bits+1) / 2;
bits1 = (int)bits/2;
```

where MAXBIT is maximum number of shape code bit. The MAXBIT is always set to 7.

2.3.10 Decoding Process for BSAC large step scalability

2.3.10.1 Definitions

Bit stream elements:

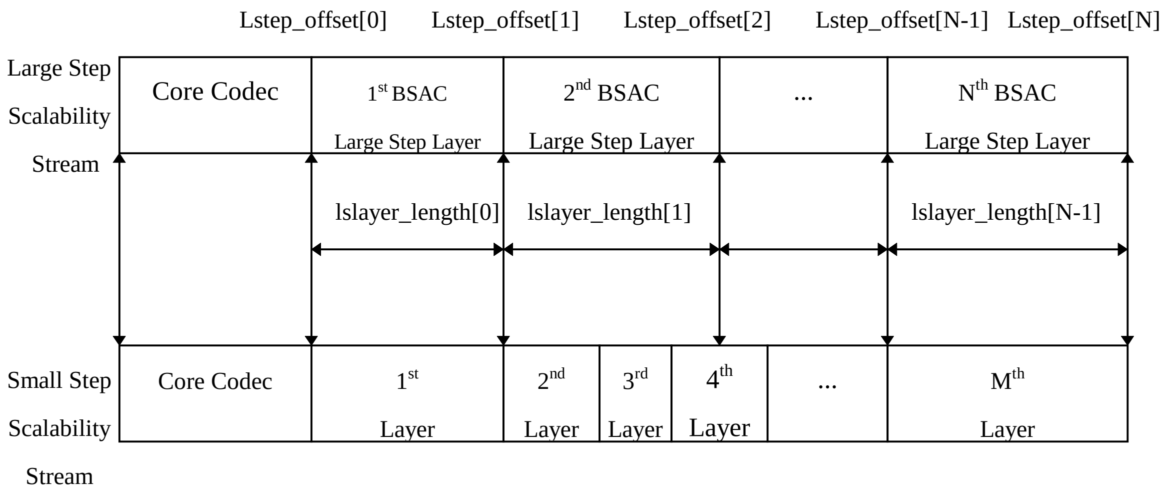
bsac_lstep_data_block()	block of raw data that contains audio data for a time period of 1024(960) samples, related information and other data. A bsac_lstep_data_block() basically consists of several bsac_lstep_stream().
bsac_lstep_stream()	abbreviaton BLSS. Syntactic element of the large step layer bitstream containing coded audio data for a time period of 1024(960) samples, related information and other data.
BSAC_stream_buf[]	a bitstream buffer for large step scalability bitstream. This buffer is mapped to small step scalability bitstream for the actual decoding

Help elements:

data_available()	function that returns ‘1’ as long as data is available, otherwise ‘0
Lstep_offset[]	array containing the offset of the bits to be used in the large step scalability layer. See clause 2.3.10.2
lslayer	large step scalability layer index

2.3.10.2 Decoding process

- In BSAC decoder, large step scalability bitstream can be obtained from each FLEXmuxPDU payload. Large step scalability bitstream should be mapped to small step scalability bitstream, since we decode small step scalability bitstream to make the reconstructed signal.
- In BSAC encoder, small step BSAC bitstream is generated. BSAC would create large overhead if one would try to transmit small step scalability over multiple elementary streams. So, some small step enhancement layers can grouped into a large step enhancement layer in order to behave like large step scalability. The relationship between small step BSAC bitstream and large step bitstream is shown in the following figure.



Mapping of **BSAC_stream_buf[]** from **Large step scalability stream** to **small step scalability stream**

In Encoder, the length of the large step enhancement layer depends upon the number of the large step enhancement layer which the user is going to make. The length of the large step layer is conveyed in the TFSpecificConig : _lslayer_length_ for each large layer.
Since the large step layer bitstream is byte-aligned, its length is represented in the unit of byte.
Lstep_offset[n] is the offset value of nth layer when each large step scalable bitstream is concatenated together. If the large step layer is not the first BSAC layer, Lstep_offset[n] is calculated using the length of the large step layer and the length of the previous layer, Lstep_offset[n-1] as follows :

Lstep_offset[n] (byte) = Lstep_offset[n-1] (byte) + lslayer_length[n]. (byte)

If the large step layer is the first BSAC layer, Lstep_offset[n] depends on the core codec. If the core codec is present, Lstep_offset[0] is initialized to the size of the core codec. Otherwise Lstep_offset[0] is set to 0.

2.3.11 Decoding Process for BSAC small step scalability

2.3.11.1 Definitions

Bit stream elements:

bsac_data_stream()	nch, sampling_frequency_index, frame_length_flag and sequence of syncword and bsac_raw_data_block()'s
nch	a bitstream element that identifies the number of the channel.
sampling_frequency_index	indicates the sampling frequency. See the sampling frequency table in clause 2.1.1.1
frame_length_flag	1 bit flag which specifies the window length of the IMDCT: if set to 0 a 1024 lines IMDCT is used if set to 1 a 960 line IMDCT is used.
bsac_raw_data_block()	block of raw data that contains audio data for a time period of 1024(960) samples, related information and other data. A bsac_raw_data_block() basically consists of bsac_main_stream() and several bsac_layer_stream().
bsac_main_stream()	abbreviaton BMS. Syntactic element of the base layer bitstream containing coded audio data for a time period of 1024(960) samples, related information and other data. There are 2 bitstream elements, identified as bitstream element nch.
bsac_layer_stream()	abbreviaton BLS. Syntactic element of the enhancement layer bitstream containing coded audio data for a time period of 1024(960) samples, related information and other data.
tf_scalable_main_header()	contains header data used for all t/f scalable coding
ms_mask_present	this two bit field (see) indicates 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.

ics_info()	contains side information necessary to decode an bsac_channel_stream. The bsac_channel_stream of a bsac_pair_main_stream may share one common ics_info.
ics_reserved_bit	bit reserved for future use
window_sequence	indicates the sequence of windows as defined in Table 2.2
window_shape	A 1 bit field that determines what window is used for the trailing part of this analysis window
max_sfb	number of scalefactor bands transmitted per group
scale_factor_grouping	A bit field that contains information about grouping of short spectral data

Help elements:

data_available()	function that returns '1' as long as each layer's bitstream is available, otherwise '0'
nch	a bitstream element that identifies the number of the channel.
encoded_layer	top scalability layer index to be encoded. If top layer index, encoded_layer is smaller than 63, the number of the layer is smaller than 64 and is (encoded+1). Otherwise, since the number of the layer is larger than or equal to 64, encoded_layer value is reset to 1000.
scalefactor window band	term for scalefactor bands within a window, given in Table 2.3 to 2.15.
scalefactor band	term for scalefactor band within a group. In case of EIGHT_SHORT_SEQUENCE and grouping a scalefactor band may contain several scalefactor window bands of corresponding frequency. For all other window_sequences scalefactor bands and scalefactor window bands are identical.
g	group index
win	window index within group
sfb	scalefactor band index within group
swb	scalefactor window band index within window
num_window_groups	number of groups of windows which share one set of scalefactors. See clause 2.3.11.4
window_group_length[g]	number of windows in each group. See clause 2.3.11.4
bit_set(bit_field,bit_num)	function that returns the value of bit number bit_num of a bit_field (most right bit is bit 0)
num_windows	number of windows of the actual window sequence. See clause 2.3.11.4
num_swb_long_window	number of scalefactor bands for long windows. This number has to be selected depending on the sampling frequency. See clause 2.4
num_swb_short_window	number of scalefactor window bands for short windows. This number has to be selected depending on the sampling frequency. See clause 2.4
num_swb	number of scalefactor window bands for shortwindows in case of EIGHT_SHORT_SEQUENCE, number of scalefactor window bands for long windows otherwise. See clause 2.3.11.4
swb_offset_long_window[swb]	table containing the index of the lowest spectral coefficient of scalefactor band sfb for long windows. This table has to be selected depending on the sampling frequency. See clause 2.4
swb_offset_short_window[swb]	table containing the index of the lowest spectral coefficient of scalefactor band sfb for short windows. This table has to be selected depending on the sampling frequency. See clause 2.4
swb_offset[swb]	table containing the index of the lowest spectral coefficient of scalefactor band sfb for short windows in case of EIGHT_SHORT_SEQUENCE, otherwise for long windows. See clause 2.3.11.4

2.3.11.2 Decoding process

bsac_raw_data_block

A total BSAC stream, bsac_raw_data_block has the layered structure. First, bsac_main_stream is parsed and decoded which is the bitstream for 1st BSAC scalability layer. Then, bsac_layer_stream for the next enhancement layer is parsed and decoded. bsac_layer_stream decoding routine is repeated until the decoded bitstream data is available and layer is smaller than or equal to the top layer, **encoded_layer.**

bsac_main_stream

A bsac_main_stream is made up of tf_scalable_main_header, bsac_general_info and bsac_layer_stream().If nch(the number of the channel) is 1, tf_scalable_main_header is parsed whose input parameters, stereo_flag and mono_lay, are 0 and 0, respectively. If nch is 2, tf_scalable_main_header is parsed whose input parameters, stereo_flag and mono_lay, are 1 and 0, respectively. bsac_general_info and bsac_layer_stream are parsed sequentially. And, the decoded samples is reconstructed with the decoded bit-sliced data.

An overview of how to decode bsac_general_info and bsac_layer_stream and reconstruct the decoded samples will be given here.

bsac_layer_stream

A bsac_layer_stream is an enhancement layer bitstream and composed of a bsac_side_info() and bsac_spectral_data(). Decoding process of bsac_layer_stream is as follows :

Decode bsac_side_info
Decode bsac_spectral_data
Reconstruct the decoded samples from the decoded bit-sliced data.

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

Decoding of stereo_info, ms_used or noise_flag.
Decoding of scalefactors
Decoding of arithmetic model index

An overview of how to decode stereo_info, scalefactor and arithmetic model index will be given in clause 3.13.

Decoding a tf_scalable_main_header

In the tf_scalable_main_header, the order of decoding is :

Get ics_info()
Get ms_mask_present, if present
Get ltp_data_present
Get ltp data, if present
Get tns_data_present
Get TNS data, if present
Get gain_control_data_present
Get gain control data, if present

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

Get ltp data, if present
Get tns_data_present
Get TNS data, if present
Get gain_control_data_present
Get gain control data, if present

The process of recovering gain_control_data and tns_data is described in clause 3.12 and 3.8, respectively. An overview of how to decode ics_info will be given in clause 2.3.4.2.

Recovering bsac_general_info

BSAC provides a 1-kits/sec/ch fine granule scalability whose bitstream has the layered structure, one BSAC base layer and several enhancement layers. BSAC base layer contains the common side information for all small step layers, the specific side information for only the base layer and the audio data. The common side information is transmitted in the syntax of bsac_general_info().

bsac_general_info consists of frame_length, encoded_layer, max_scalefactor, scalefactor_model and scf_coding necessary for decoding the scalefactor, min_ArModel and Armodel_model necessary for decoding the arithmetic model and pns_data_present and pns_start_sfb for Perceptual Noise Substitution(pns). All the bitstream elements are included in the form of the unsigned integer.

First, frame length is parsed from syntax. It represents the length of the frame including headers in bytes If the number of the channel is 1, frame length has 10 bits. Otherwise it has 11 bits.

Next, encoded_layer is parsed which preresents the top scalability layer index to be encoded. If top layer index, encoded_layer is smaller than 63, the number of the layer is smaller than 64 and is (encoded+1). Otherwise, since the number of the layer is larger than or equal to 64, encoded_layer value is reset to 1000.
max_scalefactor, scalefactor_model, min_ArModel, ArModel_model and scf_coding are parsed sequentially whose length are 8, 6, 2, 5, 2 and 1bits, respectively. If the number of the channel is not 1, all elements are parsed one more.
And, pns_data_present is parsed from syntax. If the value of the parsed pns_data_present is '1', pns_start_sfb is parsed.

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

If pns data is not present, decoding process is as follows :
  If ms_mask_present is 0, arithmetic decoding of stereo_info or ms_used is not needed.
  If ms_mask_present is 2, all ms_used values are ones in this case. So, M/S stereo processing of AAC is done at all scalefactor band.
  If ms_mask_present is 1, 1 bit mask of max_sfb bands of ms_used is conveyed in this case. So, ms_used is arithmetic decoded. M/S stereo processing of AAC is done according to the decoded ms_used.
  If ms_mask_present is 3, stereo_info is arithmetic decoded. stereo_info is two-bit flag per scalefactor band indicating the M/S coding or Intensity coding mode. If stereo_info is not 0, M/S stereo or intensity stereo of AAC is done with these decoded data.

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

If pns data is present and the number of channel is 2, decoding process is as follows :
  If ms_mask_present is 0, noise flag for pns is arithmetic decoded. Perceptual noise substitution of independent mode is done according to the decoded noise flag.
  If ms_mask_present is 2, all ms_used values are ones in this case. So, M/S stereo processing of AAC is done at all scalefactor band. However, there is no pns processing regardless of pns_data_present flag
  If ms_mask_present is 1, 1 bit mask of max_sfb bands of ms_used is conveyed in this case. So, ms_used is arithmetic decoded. M/S stereo processing of AAC is done according to the decoded ms_used. However, there is no pns processing regardless of pns_data_present flag
  If ms_mask_present is 3, stereo_info is arithmetic decoded. If stereo_info is 1 or 2, M/S stereo or intensity stereo processing of AAC is done with these decoded data and there is no pns processing. If stereo_info is 3 and scalefactor band is smaller than pns_start_sfb, out_of_phase intensity stereo processing is done. If stereo_info is 3 and scalefactor band is larger than or equal to pns_start_sfb, noise flag for pns is arithmetic decoded. And then if the both noise flags of two channel are 1, noise substitution mode is arithmetic decoded. The perceptual noise is substituted or out_of_phase intensity stereo processing is done according to the substitution mode. Otherwise, the perceptual noise is substituted only if noise flag is 1.

Decoding of scalefactors
The spectral coefficients are divided into scalefactor bands that contain a multiple of 4 quantized spectral coefficients. Each scalefactor band has a scalefactor. The noiseless coding has two ways to represent the scalefactors.

One way is to use coding scheme similar to AAC. For all scalefactors the difference to the preceding value is mapped into new value using Table B.1. If the newly mapped value 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**.

Another way is BSAC scalefactor coding method. For all scalefactors the difference to the offset value is arithmetic-coded using the arithmetic model. The arithmetic model used for coding differential scalefactors is given as a 2-bit unsigned integer in the bitstream element, **scalefactor_model**.

Decoding of arithmetic model index
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.
arithmetic model index is the model information used for encoding/decoding the bit-sliced data of each coding band.
For all arithmetic model indexes the difference to the offset value is arithmetic-decoded using the arithmetic model.

Bit-Sliced Spectral Data Parsing and Decoding
A quantized sequence is mapped into a bit-sliced sequence. Four-dimension vectors are formed from the bit-sliced sequence of the quantized spectrum and are divided into two subvectors depending upon the previous states. Noiseless coding of the subvectors relies on the arithmetic model of the coding band, the dimension , the significance of the sub-vector and the previous states.

One- to four-dimensional subvector of bit-sliced sequence are arithmetic coded and transmitted from MSB to LSB, starting from the lowest-frequency coefficient and progressing to the highest-frequency coefficient. 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 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. The spectral information for all scalefactor bands equal to or greater than **max_sfb** is set to zero.

Reconstruction of the decoded sample from bit-sliced data
The result of arithmetic decoding each bit-sliced sequence is the codeword index. This index is translated to the bit values as specified in the following pseudo C code:

pre_state[] = State that indicates whether the current decoded value is 0 or not.
snf = the significance of the vector to be decoded.
idx0 = codeword index whose previous states are 0
idx1 = codeword index whose previous states are 1
sample[] = data to be decoded
start_i = start frequence line of the decoded vectors.

```
for (i=start_i; i < (start_i+4); i++) {
    if (pre_state[i]) {
        if (idx1 & 0x01)
            sample[i] |= (1<<(snf-1))
        idx1 >>= 1;
    }
    else {
        if (idx0 & 0x01)
            sample[i] |= (1<<(snf-1))
        idx0 >>= 1;
    }
}
```

And if the sign bit of the decoded sample is 1, the decoded sample y has the negative value as follows :

```
if (y != 0)
    if (sign_bit == 1)
        y = -y
```

2.3.11.3 Windows and window sequences for BSAC

Quantization and coding is done in the frequency domain. For this purpose, the time signal is mapped into the frequency domain in the encoder. Depending on the signal, the coder may change the time/frequency resolution by using two different windows: LONG_WINDOW and SHORT_WINDOW. To switch between windows, the transition windows LONG_START_WINDOW and LONG_STOP_WINDOW are used. Refer to clause 2.3.4.3 for more detailed information about the transform and the windows as BSAC has the same transform and windows with AAC.

2.3.11.4 Scalefactor bands, grouping and coding bands for BSAC

Many tools of the AAC/BSAC decoder perform operations on groups of consecutive spectral values called scalefactor bands (abbreviation _sfb_). The width of the scalefactor bands is built in imitation of the critical bands of the human auditory system. For that reason the number of scalefactor bands in a spectrum and their width depend on the transform length and the sampling frequency. Refer to clause 2.3.4.4 for more detailed information about the scalefactor bands and grouping as BSAC has the same process with AAC.
BSAC decoding tool performs operations on groups of consecutive spectral values called coding bands (abbreviation _cband_). To increase the efficiency of the noiseless coding, the width of the coding bands is fixed as 32 irrespective of the transform length and the sampling frequency. In case of sequences which contain LONG_WINDOW, 32 spectral data are simply grouped into a coding band. Since the spectral data are transmitted in an interleaved order in case of sequences which contain SHORT_WINDOWs, the interleaved spectral data are grouped into a coding band. Each spectral index is mapped into a coding band with a mapping function, index2cb(ch, i), which returns the coding band using the mapping table index2cband[][] in case of EIGHT_SHORT_SEQUENCE, otherwise i/32. The mapping table depends on **window_sequence** and **scalefactor_grouping.**

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

- how to determine the number of windows in a window_sequence *num_windows*
- how to determine the number of window_groups *num_window_groups*
- how to determine the number of windows in each group *window_group_length[g]*
- how to determine the total number of scalefactor window bands *num_swb* for the actual window type
- how to determine *swb_offset[swb]*, the offset of the first coefficient in scalefactor window band *swb* of the window actually used
- how to determine *index2cband[i],* the mapping table from the spectral index to the coding band . This mapping table depends on **window_sequence** and **scale_factor_grouping** and is needed to decode the bsac_spectral_data().

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

```
fs_index = sampling_frequency_index;
switch( window_sequence ) {
   case ONLY_LONG_SEQUENCE:
   case LONG_START_SEQUENCE:
   case LONG_STOP_SEQUENCE:
      num_windows = 1;
      num_window_groups = 1;
      window_group_length[num_window_groups-1] = 1;
      num_swb = num_swb_long_window[fs_index];
      for( sfb=0; sfb< max_sfb+1; sfb++ ) {
            swb_offset[sfb] = swb_offset_long_window[fs_index][sfb];
      }

      /* preparation of index2cband for long blocks */
      for( sfb=0; sfb< max_sfb; sfb++ ) {
         for (i= swb_offset[sfb]; i< swb_offset[sfb+1]; i+=4){
            index2cband[i] = i / 32;
      }

      break;
   case EIGHT_SHORT_SEQUENCE:
```

```
      num_windows = 8;
      num_window_groups = 1;
      window_group_length[num_window_groups-1] = 1;
      num_swb = num_swb_short_window[fs_index];
      for( i=0; i< num_windows-1; i++) {
         if( bit_set(scale_factor_grouping,6-i)) == 0 ) {
            num_window_groups += 1;
            window_group_length[num_window_groups-1] = 1;
         }
         else {
            window_group_length[num_window_groups-1] += 1;
         }
      }

      startRegion[0] = 0;
      endRegion[num_window_groups-1] = 8;
      for( i=0; i< num_window_groups-1; i++) {
         endRegion[i] = startRegion[i] + window_group_length[i];
         startRegion[i+1] = endRegion[i];
      }

      swb_offset[0] = 0;
      b = 1
      for(i = 0; i < max_sfb; i++) {
         for(w = 0; w < num_window_groups; w++, b++) {
            width = swb_offset_short_window[fs_index][i+1]
            width -= swb_offset_short_window[fs_index][i]
            width *= window_group_length[w];
            swb_offset[b] = swb_offset[b-1] + width;
         }
      }

      /* preparation of index2cband for short blocks */
      for(qband=0; qband<max_sfb; qband++) {
         for (i=swb_offset[qband]; i<swb_offset[qband+1]; i+=4){
            for(w=0; w<num_window_groups; w++)  {
               cband = i*(endRegion[w]-startRegion[w])/32;
               for (b=startRegion[w]; b<endRegion[w]; b++) {
                  for (k=0; k<4; k++) {
                     tempband0[128*b+i+k] = 24*w+cband;
                  }
               }
            }
            j = 0;
            for(i = 0; i < swb_offset_short[maxSfb]; i+=4) {
               for(b = 0; b < 8; b++) {
                  for(k = 0; k < 4; k++, j++)
                     index2cband[j] = tempband0[128*b+i+k];
               }
            }
         }
      }
      break;

   default:
      break;
}
```

2.3.11.5 BSAC small step scalability layer

BSAC provides a 1-kits/sec/ch fine granule scalability whose bitstream has the layered structure, one BSAC base layer and various enhancement layers. BSAC base layer is made up of the common side information for all small step layers, the specific side information for only the base layer and the audio data. BSAC enhancement layers contain the layer side information and the audio data.
In order to provide the small step scalability, BSAC has the fixed band-limit according to the small step layer. Table 2.18 and Table 2.19 list the scalefactor band offset to the band-limit of each layer for the transform lengths 1024(960) and 128(120) and the different sampling frequencies, respectively. Table 2.16 and Table 2.17 list the spectral component offset to the band-limit of each layer for the transform lengths 1024(960) and 128(120) and the different sampling frequencies, respectively.

Some help variables are needed to describe the BSAC decoding process. These help variables depend on sampling_frequency, **nch** and **frame_length** and must be built up for each bsac_raw_data_block. The pseudo code shown below describes

- how to determine *layer_length*, the length of each small step enhancement layer :
  `layer_length = BLOCK_SIZE_SAMPLES_IN_FRAME *1000 * nch / SAMPLING_FREQUENCY`
- how to determine *layer_index[]*, the spectral component offset to the band-limit of each layer
- how to determine *available_bits[0]*, the maximum available bits to be used in the BSAC base layer

```
available_bits[0] = base_layer_bitrate * layer_length / 1000
if (available_bits[0] > frame_length*8)
    available_bits[0] = frame_length*8
```

where, base_layer_bitrate is 16000 bits/s.

- initialization of *layer_index[ch][0]*, the spectral component offset to the band-limit of the base layer
  `layer_index[ch][0] = 0`
- initialization of *layer_sfb[ch][0]*, the scalefactor band offset to the band-limit of the base layer
  `layer_sfb[ch][0] = 0`

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

- how to determine *available_bits[i]*, the available maximum size of the bitstream from the BSAC base layer to the *i*-th layer.
- how to determine *layer_sfb[][]*, the scalefactor band offset to the band-limit of each layer
- how to determine *layer_index[]*, the spectral component offset to the band-limit of each layer
- how to determine *last_index*, the highest spectral index of **nch** channel band-limits

```
layer = BSAC_small_step_layer_index
available_bits[layer+1] = available_bits[layer] + layer_length
if (available_bits[layer+1] > frame_length*8)
    available_bits[layer+1] = frame_length*8

fs_index = sampling_frequency_index
last_index = 0

for(ch = 0; ch < nch; ch++) {
    switch( window_sequence ) {
        case ONLY_LONG_SEQUENCE:
        case LONG_START_SEQUENCE:
        case LONG_STOP_SEQUENCE:
            if (layer_sfb_offset_long [fs_index][layer] < max_sfb[ch] && layer<encoded_layer)
                layer_sfb[ch][layer+1] = layer_sfb_offset_long[fs_index][layer];
            else
                layer_sfb[ch][layer+1] = max_sfb[ch];

            sfb = layer_sfb[ch][layer+1];
            layer_index[ch][layer+1] = layer_index_offset_long[fs_index][layer];
            if (swb_offset[ch][sfb] < layer_index[ch][layer+1])
                    layer_index[ch][layer+1] = swb_offset[ch][sfb];
        break;


        case EIGHT_SHORT_SEQUENCE:
            if (layer_sfb_offset_short[fs_index][layer] < max_sfb[ch] && layer<encoded_layer)
                layer_sfb[ch][layer+1] = layer_sfb_offset_short[fs_index][layer];
            else
                layer_sfb[ch][layer+1] = max_sfb[ch];

            sfb = layer_sfb[ch][layer+1] * num_window_groups[ch];
            layer_index[ch][layer+1] = layer_index_offset_short[fs_index][layer];
            if (swb_offset[ch][sfb] < layer_index[ch][layer+1])
```

```
            layer_index[ch][layer+1] = swb_offset[ch][sfb];
        break;

      default:
        break;
    }

    /*   find last index  */
    qband = layer_sfb[ch][layer+1] * num_window_groups[ch]
    if(last_index < swb_offset[ch][qband])
      last_index = swb_offset[ch][qband]
}
```

BSAC scalable coding scheme has the fixed band-limit according to the small step layer. The spectral band is extended more and more as the number of the enhancement layer is increased. So, the new spectral componets are added to be decoded in each layer. some help variables and arrays are needed to describe the bit-sliced decoding process of the spectral values in each BSAC small step layer. cur_snf[ch][*i*] is initialized as the allocated bit to the coding band *cband*, Abit[ch][cband] as shown below, where we can get Abit[][] from **ArModel[ch][cband]** and map *i* into *cband* using the function *index2cb(ch, i)*. And, we need the offset significance to start the decoding of the bit-sliced data in each layer. The maximum significance, *maxsnf*, is used as the offset.
These help variables and arrays must be built up for each bsac_spectral_data(). The pseudo code shown below describes

- how to initialize *cur_snf[][]* , the current significance of the 4-dimensional vectors to be added newly. due to the spectral band extension in each enhancement scalability layer.
- how to determine *maxsnf*, the maximum significance of all vectors to be decoded.

```
maxsnf = 0;
for(ch = 0; ch < nch; ch++) {
    /* set current snf */
    for(sfb=layer_sfb[ch][layer]; sfb<layer_sfb[ch][layer+1]; sfb++) {
        for(w = 0; w < num_window_groups[ch]; w++) {
            qband = (sfb * num_window_groups[ch]) + w
            for (i=swb_offset [ch][qband]; i<swb_offset[ch][qband+1]; i+=4) {
                cband = index2cb(ch, i);
                cur_snf[ch][i] = Abit[ch][cband]
            }
        }
    }

    /*   find maximum snf  */
    qband = layer_sfb[ch][layer+1] * num_window_groups[ch]
    for(i = 0; i< swb_offset[ch][qband]; i+=4)
        if (maxsnf < cur_snf[ch][i]) maxsnf = cur_snf[ch][i]
}
```

2.3.11.6 Order of spectral coefficients in spectral_data

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

spectral coefficients →

sfb 0	sfb 1	sfb 2	. . .	sfb (num_sfb-1)

Order of scalefactor bands for ONLY_LONG_SEQUENCE

For the EIGHT_SHORT_SEQUENCE window, each 4 spectral data in each block is interleaving in ascending spectral order, as shown in the following :

spectral coefficients →

B0 0..3	B1 0..3	B2 0..3	. . .	B7 0..3	B0 4..7	B1 4..7	B2 4..7	. . .	B7 4..7	. . .	B0 80...83	B1 80...83	B2 80...83	. .	B7 80...83	. .

Order of spectral data for EIGHT_SHORT_SEQUENCE

2.4 Tables

Table 2.1 – Transform windows (for 48 kHz)

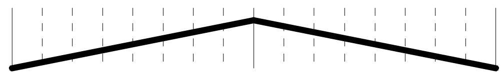
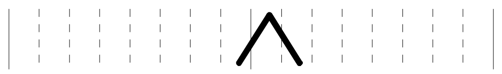
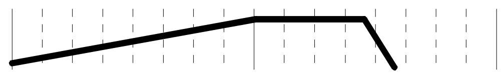
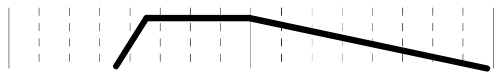
window	num_swb	#coeffs	looks like
LONG_WINDOW	49	1024	
SHORT_WINDOW	14	128	
LONG_START_WINDOW	49	1024	
LONG_STOP_WINDOW	49	1024	

Table 2.2 – Window Sequences

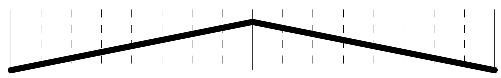
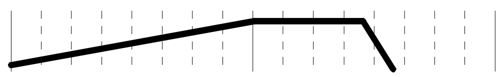
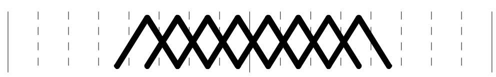
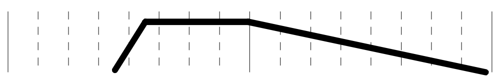
value	window_sequence	num_ windows	looks like
0	ONLY_LONG_SEQUENCE = LONG_WINDOW	1	
1	LONG_START_SEQUENCE = LONG_START_WINDOW	1	
2	EIGHT_SHORT_SEQUENCE = 8 * SHORT_WINDOW	8	
3	LONG_STOP_SEQUENCE = LONG_STOP_WINDOW	1	

Table 2.3 – scalefactor bands for a window length of 2048 and 1920 (values for 1920 in brackets) for LONG_WINDOW, LONG_START_WINDOW, LONG_STOP_WINDOW at 44.1 and 48 kHz

fs [kHz]	44.1,48
num_swb_long_ window	49

swb	swb_offset_long _window
0	0
1	4
2	8
3	12
4	16
5	20

swb	swb_offset_long_ window
25	216
26	240
27	264
28	292
29	320
30	352

6	24	31	384
7	28	32	416
8	32	33	448
9	36	34	480
10	40	35	512
11	48	36	544
12	56	37	576
13	64	38	608
14	72	39	640
15	80	40	672
16	88	41	704
17	96	42	736
18	108	43	768
19	120	44	800
20	132	45	832
21	144	46	864
22	160	47	896
23	176	48	928
24	196		1024 (960)

Table 2.4 – scalefactor bands for a window length of 256 and 240 (values for 240 in brackets) for SHORT_WINDOW at 32, 44.1 and 48 kHz

fs [kHz]	32,44.1,48		
num_swb_short_window	14		
swb	swb_offset_short_window	swb	swb_offset_short_window
0	0	8	44
1	4	9	56
2	8	10	68
3	12	11	80
4	16	12	96
5	20	13	112
6	28		128 (120)
7	36		

Table 2.5 – scalefactor bands for a window length of 2048 and 1920 (values for 1920 in brackets) for LONG_WINDOW, LONG_START_WINDOW, LONG_STOP_WINDOW at 32 kHz

fs [kHz]	32		
num_swb_long_window	51		
swb	swb_offset_long_window	swb	swb_offset_long_window
0	0	26	240
1	4	27	264
2	8	28	292
3	12	29	320
4	16	30	352
5	20	31	384
6	24	32	416
7	28	33	448
8	32	34	480
9	36	35	512
10	40	36	544
11	48	37	576
12	56	38	608
13	64	39	640
14	72	40	672
15	80	41	704
16	88	42	736
17	96	43	768
18	108	44	800
19	120	45	832
20	132	46	864

21	144	47	896
22	160	48	928
23	176	49	960
24	196	50	992 (960)
25	216		1024 (960)

Table 8.6 – scalefactor bands for a window length of 2048 and 1920 (values for 1920 in brackets) for LONG_WINDOW, LONG_START_WINDOW, LONG_STOP_WINDOW at 8 kHz

fs [kHz]	8		
num_swb_long_window	40		
swb	swb_offset_long_window	swb	swb_offset_long_window
0	0	21	288
1	12	22	308
2	24	23	328
3	36	24	348
4	48	25	372
5	60	26	396
6	72	27	420
7	84	28	448
8	96	29	476
9	108	30	508
10	120	31	544
11	132	32	580
12	144	33	620
13	156	34	664
14	172	35	712
15	188	36	764
16	204	37	820
17	220	38	880
18	236	39	944
19	252		1024 (960)
20	268		

Table 8.7 – scalefactor bands for a window length of 256 and 240 (values for 240 in brackets) for SHORT_WINDOW at 8 kHz

fs [kHz]	8		
num_swb_short_window	15		
swb	swb_offset_short_window	swb	swb_offset_short_window
0	0	8	36
1	4	9	44
2	8	10	52
3	12	11	60
4	16	12	72
5	20	13	88
6	24	14	108
7	28		128 (120)

Table 2.8 – scalefactor bands for a window length of 2048 and 1920 (values for 1920 in brackets) for LONG_WINDOW, LONG_START_WINDOW, LONG_STOP_WINDOW at 11.025, 12 and 16 kHz

fs [kHz]	11.025, 12, 16		
num_swb_long_window	43		
swb	swb_offset_long	swb	swb_offset_long_

	_window		window
0	0	22	228
1	8	23	244
2	16	24	260
3	24	25	280
4	32	26	300
5	40	27	320
6	48	28	344
7	56	29	368
8	64	30	396
9	72	31	424
10	80	32	456
11	88	33	492
12	100	34	532
13	112	35	572
14	124	36	616
15	136	37	664
16	148	38	716
17	160	39	772
18	172	40	832
19	184	41	896
20	196	42	960
21	212		1024 (960)

Table 2.9 – scalefactor bands for a window length of 256 and 240 (values for 240 in brackets) for SHORT_WINDOW at 11.025, 12 and 16 kHz

fs [kHz]	11.025, 12, 16		
num_swb_short_ window	15		
swb	swb_offset_short _window	swb	swb_offset_short_ window
0	0	8	32
1	4	9	40
2	8	10	48
3	12	11	60
4	16	12	72
5	20	13	88
6	24	14	108
7	28		128 (120)

Table 2.10 – scalefactor bands for a window length of 2048 and 1920 (values for 1920 in brackets) for LONG_WINDOW, LONG_START_WINDOW, LONG_STOP_WINDOW at 22.05 and 24 kHz

fs [kHz]	22.05 and 24		
num_swb_long_ window	47		
swb	swb_offset_long _window	swb	swb_offset_long_ window
0	0	24	160
1	4	25	172
2	8	26	188
3	12	27	204
4	16	28	220
5	20	29	240
6	24	30	260
7	28	31	284
8	32	32	308

9	36	33	336
10	40	34	364
11	44	35	396
12	52	36	432
13	60	37	468
14	68	38	508
15	76	39	552
16	84	40	600
17	92	41	652
18	100	42	704
19	108	43	768
20	116	44	832
21	124	45	896
22	136	46	960
23	148		1024 (960)

Table 2.11 – scalefactor bands for a window length of 256 and 240 (values for 240 in brackets) for SHORT_WINDOW at 22.05 and 24 kHz

fs [kHz]	22.05 and 24
num_swb_short_window	15

swb	swb_offset_short_window	swb	swb_offset_short_window
0	0	8	36
1	4	9	44
2	8	10	52
3	12	11	64
4	16	12	76
5	20	13	92
6	24	14	108
7	28		128 (120)

Table 2.12 – scalefactor bands for a window length of 2048 and 1920 (values for 1920 in brackets) for LONG_WINDOW, LONG_START_WINDOW, LONG_STOP_WINDOW at 64 kHz

fs [kHz]	64
num_swb_long_window	47

swb	swb_offset_long_window	swb	swb_offset_long_window
0	0	24	172
1	4	25	192
2	8	26	216
3	12	27	240
4	16	28	268
5	20	29	304
6	24	30	344
7	28	31	384
8	32	32	424
9	36	33	464
10	40	34	504
11	44	35	544
12	48	36	584
13	52	37	624
14	56	38	664
15	64	39	704
16	72	40	744

17	80
18	88
19	100
20	112
21	124
22	140
23	156
41	784
42	824
43	864
44	904
45	944
46	984 (960)
	1024 (960)

Table 2.13 – scalefactor bands for a window length of 256 and 240 (values for 240 in brackets) for SHORT_WINDOW at 64 kHz

fs [kHz]	64
num_swb_short_window	12
swb	swb_offset_short_window
0	0
1	4
2	8
3	12
4	16
5	20
6	24
7	32
8	40
9	48
10	64
11	92
	128 (120)

Table 2.14 – scalefactor bands for a window length of 2048 and 1920 (values for 1920 in brackets) for LONG_WINDOW, LONG_START_WINDOW, LONG_STOP_WINDOW at 88.2 and 96 kHz

fs [kHz]	88.2 and 96
num_swb_long_window	41
swb	swb_offset_long_window
0	0
1	4
2	8
3	12
4	16
5	20
6	24
7	28
8	32
9	36
10	40
11	44
12	48
13	52
14	56
15	64
16	72
17	80
18	88
19	96
20	108
21	120
22	132
23	144
24	156
25	172
26	188
27	212
28	240
29	276
30	320
31	384
32	448
33	512
34	576
35	640
36	704
37	768
38	832
39	896
40	960
	1024 (960)

Table 2.15 – scalefactor bands for a window length of 256 and 240 (values for 240 in brackets) for SHORT_WINDOW at 88.2 and 96 kHz

fs [kHz]	88.2 and 96
num_swb_short_	12

window	
swb	swb_offset_short_window
0	0
1	4
2	8
3	12
4	16
5	20
6	24
7	32
8	40
9	48
10	64
11	92
	128 (120)

Table 2.16 BSAC layer index for a window length of 2048 and 1920 for LONG_WINDOW, LONG_START_WINDOW, LONG_STOP_WINDOW at 48, 44.1, 32, 24, 22.05, 16, 12, 11.025, 8 kHz

layer	48 kHz	44.1 kHz	32 kHz	24 kHz	22.05 kHz	16 kHz	12 kHz	11.025 kHz	8 kHz
0	160	176	240	336	364	492	664	716	1024(960)
1	168	184	264	348	396	532	716	772	1024(960)
2	180	192	292	364	432	572	772	832	1024(960)
3	192	208	320	380	468	616	832	896	1024(960)
4	200	216	336	396	488	664	896	960	1024(960)
5	212	232	352	432	508	716	960	960	1024(960)
6	224	240	368	468	540	772	960	1024(960)	1024(960)
7	232	256	384	508	572	832	1024(960)	1024(960)	1024(960)
8	244	264	416	544	600	896	1024(960)	1024(960)	1024(960)
9	256	280	440	576	632	960	1024(960)	1024(960)	1024(960)
10	264	288	464	608	664	960	1024(960)	1024(960)	1024(960)
11	276	304	488	652	696	960	1024(960)	1024(960)	1024(960)
12	288	312	512	704	728	1024(960)	1024(960)	1024(960)	1024(960)
13	296	320	536	736	768	1024(960)	1024(960)	1024(960)	1024(960)
14	308	336	560	768	808	1024(960)	1024(960)	1024(960)	1024(960)
15	320	352	584	800	848	1024(960)	1024(960)	1024(960)	1024(960)
16	328	360	608	832	896	1024(960)	1024(960)	1024(960)	1024(960)
17	340	368	624	848	912	1024(960)	1024(960)	1024(960)	1024(960)
18	352	384	640	864	928	1024(960)	1024(960)	1024(960)	1024(960)
19	360	392	656	880	944	1024(960)	1024(960)	1024(960)	1024(960)
20	372	408	672	896	960	1024(960)	1024(960)	1024(960)	1024(960)
21	384	416	696	912	960	1024(960)	1024(960)	1024(960)	1024(960)
22	392	424	720	928	992 (960)	1024(960)	1024(960)	1024(960)	1024(960)
23	404	440	744	944	992 (960)	1024(960)	1024(960)	1024(960)	1024(960)
24	416	456	768	960	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)
25	424	464	776	960	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)
26	436	472	788	992 (960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)
27	448	488	800	992 (960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)
28	456	496	808	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)
29	468	512	820	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)
30	480	520	832	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)
31	488	528	848	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)
32	500	544	864	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)
33	512	560	896	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)
34	520	568	928	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)
35	532	576	960	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)
36	544	592	992 (960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)
37	552	600	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)
38	564	616	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)
39	576	624	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)
40	584	632	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)
41	596	648	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)
42	608	664	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)
43	616	672	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)
44	628	680	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)

45	640	696	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)
46	648	704	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)
47	660	720	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)
48	672	728	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)
> 48	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)	1024(960)

Table 2.17 BSAC layer index for a window length of 256 and 240 for SHORT_WINDOW at 48, 44.1, 32, 24, 22.05, 16, 12, 11.025, 8 kHz

layer	48 kHz	44.1 kHz	32 kHz	24 kHz	22.05 kHz	16 kHz	12 kHz	11.025 kHz	8 kHz
0	20	20	28	36	44	60	80	88	128(120)
1	20	20	32	40	48	64	88	96	128(120)
2	20	24	36	44	52	68	96	104	128(120)
3	24	24	40	44	56	76	104	112	128(120)
4	24	24	40	48	60	80	112	120	128(120)
5	24	28	44	52	60	88	120	120	128(120)
6	28	28	44	56	64	96	120	128(120)	128(120)
7	28	32	48	60	68	104	128(120)	128(120)	128(120)
8	28	32	52	64	68	112	128(120)	128(120)	128(120)
9	32	36	52	72	72	120	128(120)	128(120)	128(120)
10	32	36	56	72	76	120	128(120)	128(120)	128(120)
11	32	36	60	80	80	120	128(120)	128(120)	128(120)
12	36	40	64	88	88	128(120)	128(120)	128(120)	128(120)
13	36	40	64	92	88	128(120)	128(120)	128(120)	128(120)
14	36	40	68	96	92	128(120)	128(120)	128(120)	128(120)
15	40	44	72	100	96	128(120)	128(120)	128(120)	128(120)
16	40	44	76	100	96	128(120)	128(120)	128(120)	128(120)
17	40	44	76	104	100	128(120)	128(120)	128(120)	128(120)
18	44	48	80	108	104	128(120)	128(120)	128(120)	128(120)
19	44	48	80	108	108	128(120)	128(120)	128(120)	128(120)
20	44	52	84	112	112	128(120)	128(120)	128(120)	128(120)
21	48	52	84	112	120	128(120)	128(120)	128(120)	128(120)
22	48	52	88	116	120	128(120)	128(120)	128(120)	128(120)
23	48	56	92	116	120	128(120)	128(120)	128(120)	128(120)
24	52	56	96	120	120	128(120)	128(120)	128(120)	128(120)
25	52	56	96	120	128(120)	128(120)	128(120)	128(120)	128(120)
26	52	60	96	120	128(120)	128(120)	128(120)	128(120)	128(120)
27	56	60	100	120	128(120)	128(120)	128(120)	128(120)	128(120)
28	56	64	100	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)
29	56	64	100	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)
30	60	64	104	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)
31	60	68	104	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)
32	60	68	108	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)
33	64	68	112	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)
34	64	72	116	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)
35	64	72	120	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)
36	68	76	124	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)
37	68	76	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)
38	68	76	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)
39	72	80	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)
40	72	80	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)
41	72	80	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)
42	76	84	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)
43	76	84	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)
44	76	84	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)
45	80	88	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)
46	80	88	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)
47	80	88	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)
48	84	92	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)
> 48	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)	128(120)

Table 2.18 BSAC layer scalefactor band for a window length of 2048 and 1920 for LONG_WINDOW, LONG_START_WINDOW, LONG_STOP_WINDOW at 48, 44.1, 32, 24, 22.05, 16, 12, 11.025, 8 kHz

layer	48 kHz	44.1 kHz	32 kHz	24 kHz	22.05 kHz	16 kHz	12 kHz	11.025 kHz	8 kHz
0	22	23	26	33	34	33	37	38	40
1	23	24	27	34	35	34	38	39	40
2	24	24	28	34	36	35	39	40	40
3	24	25	29	35	37	36	40	41	40
4	25	25	30	35	38	37	41	42	40
5	25	26	30	36	38	38	42	42	40
6	26	26	31	37	39	39	42	43	40
7	26	27	31	38	40	40	43	43	40
8	27	27	32	39	40	41	43	43	40
9	27	28	33	40	41	42	43	43	40
10	27	28	34	41	42	42	43	43	40
11	28	29	35	41	42	42	43	43	40
12	28	29	35	42	43	43	43	43	40
13	29	29	36	43	43	43	43	43	40
14	29	30	37	43	44	43	43	43	40
15	29	30	38	44	45	43	43	43	40
16	30	31	38	44	45	43	43	43	40
17	30	31	39	45	46	43	43	43	40
18	30	31	39	45	46	43	43	43	40
19	31	32	40	45	46	43	43	43	40
20	31	32	40	45	46	43	43	43	40
21	31	32	41	46	46	43	43	43	40
22	32	33	42	46	47	43	43	43	40
23	32	33	43	46	47	43	43	43	40
24	32	34	43	46	47	43	43	43	40
25	33	34	44	46	47	43	43	43	40
26	33	34	44	47	47	43	43	43	40
27	33	35	44	47	47	43	43	43	40
28	34	35	45	47	47	43	43	43	40
29	34	35	45	47	47	43	43	43	40
30	34	36	45	47	47	43	43	43	40
31	35	36	46	47	47	43	43	43	40
32	35	36	46	47	47	43	43	43	40
33	35	37	47	47	47	43	43	43	40
34	36	37	48	47	47	43	43	43	40
35	36	37	49	47	47	43	43	43	40
36	36	38	50	47	47	43	43	43	40
37	37	38	51	47	47	43	43	43	40
38	37	39	51	47	47	43	43	43	40
39	37	39	51	47	47	43	43	43	40
40	38	39	51	47	47	43	43	43	40
41	38	40	51	47	47	43	43	43	40
42	38	40	51	47	47	43	43	43	40
43	39	40	51	47	47	43	43	43	40
44	39	41	51	47	47	43	43	43	40
45	39	41	51	47	47	43	43	43	40
46	40	41	51	47	47	43	43	43	40
47	40	42	51	47	47	43	43	43	40
48	40	42	51	47	47	43	43	43	40
> 48	max_sfb	max_sfb	max_sfb	max_sfb	max_sfb	max_sfb	max_sfb	max_sfb	max_sfb

Table 2.19 BSAC layer scalefactor band for a window length of 256 and 240 for SHORT_WINDOW at 48, 44.1, 32, 24, 22.05, 16, 12, 11.025, 8 kHz

layer	48 kHz	44.1 kHz	32 kHz	24 kHz	22.05 kHz	16 kHz	12 kHz	11.025 kHz	8 kHz
0	5	5	6	8	9	11	13	13	15
1	5	5	7	9	10	12	13	14	15

2	5	6	7	9	10	12	14	14	15
3	6	6	8	9	11	13	14	15	15
4	6	6	8	10	11	13	15	15	15
5	6	6	8	10	11	13	15	15	15
6	6	6	8	11	11	14	15	15	15
7	6	7	9	11	12	14	15	15	15
8	6	7	9	11	12	15	15	15	15
9	7	7	9	12	12	15	15	15	15
10	7	7	9	12	12	15	15	15	15
11	7	7	10	13	13	15	15	15	15
12	7	8	10	13	13	15	15	15	15
13	7	8	10	13	13	15	15	15	15
14	7	8	10	14	13	15	15	15	15
15	8	8	11	14	14	15	15	15	15
16	8	8	11	14	14	15	15	15	15
17	8	8	11	14	14	15	15	15	15
18	8	9	11	14	14	15	15	15	15
19	8	9	11	14	14	15	15	15	15
20	8	9	12	15	15	15	15	15	15
21	9	9	12	15	15	15	15	15	15
22	9	9	12	15	15	15	15	15	15
23	9	9	12	15	15	15	15	15	15
24	9	9	12	15	15	15	15	15	15
25	9	9	12	15	15	15	15	15	15
26	9	10	12	15	15	15	15	15	15
27	9	10	13	15	15	15	15	15	15
28	9	10	13	15	15	15	15	15	15
29	9	10	13	15	15	15	15	15	15
30	10	10	13	15	15	15	15	15	15
31	10	10	13	15	15	15	15	15	15
32	10	10	13	15	15	15	15	15	15
33	10	10	13	15	15	15	15	15	15
34	10	11	14	15	15	15	15	15	15
35	10	11	14	15	15	15	15	15	15
36	10	11	14	15	15	15	15	15	15
37	10	11	14	15	15	15	15	15	15
38	10	11	14	15	15	15	15	15	15
39	11	11	14	15	15	15	15	15	15
40	11	11	14	15	15	15	15	15	15
41	11	11	14	15	15	15	15	15	15
42	11	12	14	15	15	15	15	15	15
43	11	12	14	15	15	15	15	15	15
44	11	12	14	15	15	15	15	15	15
45	11	12	14	15	15	15	15	15	15
46	12	12	14	15	15	15	15	15	15
47	12	12	14	15	15	15	15	15	15
48	12	12	14	15	15	15	15	15	15
> 48	max_sfb	max_sfb	max_sfb	max_sfb	max_sfb	max_sfb	max_sfb	max_sfb	max_sfb

2.5 Figures

window_sequence =EIGHT_SHORT_SEQUENCE

num_windows = 8
grouping_bits = '1100101'
num_window_groups = 4
window_group_length[] = { 3, 1, 2, 2 }

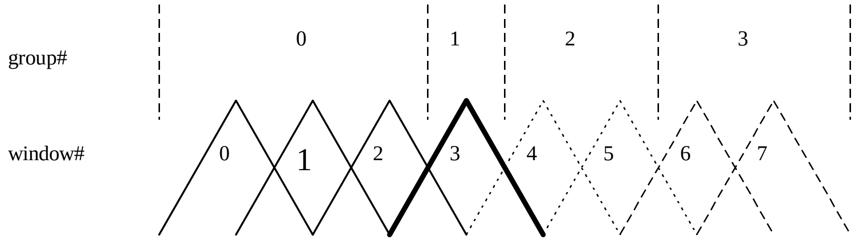
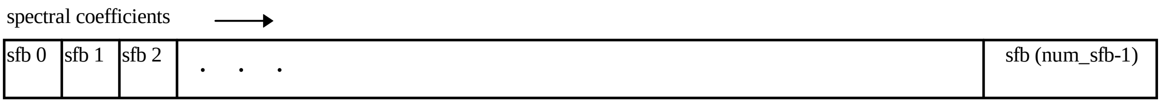
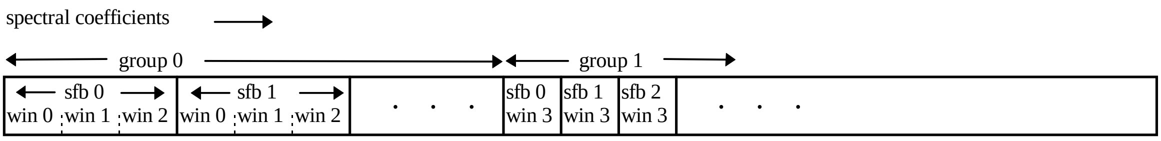


Figure 2.1 – Example for short window grouping



Order of scalefactor bands for ONLY_LONG_SEQUENCE

Figure 2.2 – Spectral order of scalefactor bands in case of ONLY_LONG_SEQUENCE



Order of scale factor bands for EIGHT_SHORT_SEQUENCE
window_group_length[] = { 3, 1, ... }

Figure 2.3 – Spectral order of scalefactor bands in case of EIGHT_SHORT_SEQUE