ETSI TS 103 190 V1.1.1 (2014-04)

Technical Specification

Digital Audio Compression (AC-4) Standard

EBU

OPERATING EUROVISION

Reference

DTS/JTC-025

Keywords

audio, broadcasting, codec, content, digital, distribution

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° 7803/88

Contents

Intellectual Property Rights

Foreword

This Technical Specification (TS) has been produced by Joint Technical Committee (JTC) Broadcast of the European Broadcasting Union (EBU), Comité Européen de Normalisation ELECtrotechnique (CENELEC) and the European Telecommunications Standards Institute (ETSI).

NOTE: The EBU/ETSI JTC Broadcast was established in 1990 to co-ordinate the drafting of standards in the specific field of broadcasting and related fields. Since 1995 the JTC Broadcast became a tripartite body by including in the Memorandum of Understanding also CENELEC, which is responsible for the standardization of radio and television receivers. The EBU is a professional association of broadcasting organizations whose work includes the co-ordination of its members' activities in the technical, legal, programme-making and programme-exchange domains. The EBU has active members in about 60 countries in the European broadcasting area; its headquarters is in Geneva.

European Broadcasting Union
CH-1218 GRAND SACONNEX (Geneva)
Switzerland
Tel: +41 22 717 21 11
Fax: +41 22 717 24 81

The symbolic source code for tables referenced throughout the present document is contained in archive ts_103190v010101p0.zip which accompanies the present document.

Introduction

Motivation

Digital entertainment has been progressing at a rapid pace during the past 20 years. During this period, digital audio and video compression technologies have mostly been designed and rolled-out first in the context of analog to digital migration context for broadcast services. Then, technologies evolved in order to expand this digital world with greater quality through high definition TV and surround sound. Later on, further to the rapid development pace of internet entertainment services accessible to an ever increasing number of consumers, with increasing connection speeds, in some cases through hybrid broadcast-broadband technologies bridging the gap between one-to-many and one-to-one applications, non-linear services and more recently interactive services significantly transformed the entertainment experience for the end-consumer. Place shifting and time shifting became possible. Multiple screen services and more recently companion screens and screen-to-screen applications are being made available to end consumers for truly personalized experiences. When this digital world became a reality, a post-digital era started and is still on-going which will be more than ever focusing on delivering experiences, combined with increasing pressure to implement digitally efficient delivery systems, and multiple device environments. These two aspects of this post-digital experience era continue to drive a need for innovation in delivery formats. At the same time, compatibility and standardization of formats is necessary for effective audience reach.

In this context, AC-4 does not only provide the compression efficiency required for the broad variety of tomorrow's broadcast and broadband delivery environments but also system integration features in order to address particular challenges of modern media distribution, all with the flexibility to support future audio experiences:

- Coded audio frame alignment with video framing (configurable)
  Aligning audio and video frames greatly simplifies audio and video timebase correction (A/V sync management) in the compressed domain for contribution and distribution applications. In addition, audio elementary stream chunking/fragmentation processes for multi-screen delivery can be precisely aligned with video elementary streams to eliminate the complexity of managing A/V sync end-to-end.

- Built in Dialog Enhancement
  Dialog enhancement algorithms allow users to modify the dialog level guided by information from the encoder or content creator, both with and without a clean (separate) dialog track presented to the encoder.

- Designed for adaptive streaming and advertisement insertions
  Bitrate and channel configuration can be switched without audible glitches.

- Next step in non-destructive loudness and dynamic range management
  Intelligent and independent dynamic range and loudness controls act across a wide range of devices and applications (Home Theatre to Mobile) and can be configured to align with numerous worldwide standards and/or recommendations. Level management is completely automated and leverages a new signaling framework to ensure compliant programing from the content creator is passed without cascaded processing.

- Dual-ended post-processing
  Metadata driven post-processing leverages media intelligence to optimize the experience across device types and ensures that only a single instance of each post processing algorithm is enabled throughout the entire chain.

- Support for lossy/low bitrate up to high quality/lossless audio.

Encoding

The AC-4 encoder accepts PCM audio and produces an encoded bitstream consistent with the present document. The specifics of the audio encoding process are not normative requirements of the present document. Nevertheless, the encoder is expected to produce a bitstream matching the syntax described in clause 4, which, when decoded according to clauses 5 and 6, produces audio of sufficient quality for the intended application.

Decoding

The decoding process is basically the inverse of the encoding process. The decoder will de-format the various types of data such as the encoded spectral components and apply the relevant decoding tools.

Structure of the present document

The present document is structured as follows. Clause 4.2 specifies the details of the AC-4 bitstream syntax; clause 4.3 interprets bits into values used elsewhere in the present document. Clause 5 describes the various tools used in the AC-4 decoder. Finally clause 6 mentions all the tools in context and connects them into a working decoder.

1 Scope

The present document specifies a coded presentation of audio information, and specifies the decoding process. The coded presentation specified herein is suitable for use in digital audio transmission and storage applications. The coded presentation may convey full bandwidth audio signals, along with a low frequency enhancement signal, for multichannel playback. Additional presentations can be included, targeting e.g. listeners with visual or hearing disabilities. A wide range of encoded bit-rates is supported by decoders implemented according to the present document, ranging from state-of-the-art compression to perceptually lossless rates. The coded presentation is designed with system features such as robust operation, video frame synchronicity, and seamless switching of presentations.

2 References

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the reference document (including any amendments) applies.

Referenced documents which are not found to be publicly available in the expected location might be found at http://docbox.etsi.org/Reference.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

2.1 Normative references

The following referenced documents are necessary for the application of the present document.

[1] ISO/IEC 23009-1:2012: "Information technology -- Dynamic Adaptive Streaming over HTTP (DASH) -- Part 1: Media presentation description and segment formats".

[2] ISO/IEC 14496-12:2012/Amd 1:2013: "Information technology -- Coding of audio-visual objects -- Part 12: ISO base media file format"/"Various enhancements including support for large metadata".

[3] ETSI TS 102 822-3-1 (V1.4.1) - (2007-11): "Broadcast and On-line Services: Search, select, and rightful use of content on personal storage systems ("TV-Anytime"); Part 3: Metadata; Sub-part 1: Phase 1 - Metadata schemas".

[4] IETF BCP 47: "Tags for Identifying Languages".

[5] ISO 639: "Codes for the representation of names of languages".

2.2 Informative references

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.1] ETSI TS 102 366: "Digital Audio Compression (AC-3, Enhanced AC-3) Standard".

[i.2] ISO/IEC 13818-1 (2000): "Information technology - Generic coding of moving pictures and associated audio information: Systems".

[i.3] Recommendation ITU-R BS.1770: "Algorithms to measure audio programme loudness and true-peak audio level".

[i.4] Recommendation ITU-R BS.1771: "Requirements for loudness and true-peak indicating meters".

[i.5] ATSC Standard A/85: "Techniques for Establishing and Maintaining Audio Loudness for Digital Television".

[i.6] Recommendation EBU R128: "Loudness normalisation and permitted maximum level of audio signals".

[i.7] Technical Report ARIB TR-B32: "Operational Guidelines for Loudness of Digital Television Programs".

[i.8] FREE TV AUSTRALIA OPERATIONAL PRACTICE OP-59: "Measurement and Management of Loudness in Soundtracks for Television Broadcasting".

[i.9] Dolby Laboratories Speech Gating Reference Code and Information.

NOTE: Available at http://www.dolby.com/us/en/professional/technology/broadcast/dialogue-intelligence.aspx.

[i.10] Recommendation EBU Tech 3342: "Loudness Range: A Measure to supplement loudness normalization in accordance with EBU R128".

[i.11] Recommendation EBU Tech 3341: "Loudness Metering: 'EBU Mode' metering to supplement loudness normalisation in accordance with EBU R 128".

3 Definitions, symbols, abbreviations and conventions

3.1 Definitions

For the purposes of the present document, the following terms and definitions apply:

A-SPX time slot: time range represented by one or two QMF time slots

NOTE: The actual size of an A-SPX time slot depends on the ASF block length.

(A-SPX) time slot group: group of A-SPX time slots

A-SPX delay line: time range represented by 3 A-SPX time slots

NOTE: The A-SPX delay line is used to align A-SPX and QMF domain processing.

A-SPX interval: temporal extent of A-SPX time slots processed by the A-SPX tool

(audio) channel: data representing an audio signal for a dedicated speaker position

(audio) track: data representing an audio signal

average bitrate stream: Stream complying to a buffer model

bin: position of the frequency coefficient in the MDCT domain, as in frequency bin number *n*

bitstream element: variable, array or matrix described by a series of one or more bits in an AC-4 bitstream

block: part of a frame

block length: temporal extent of a block

channel element: bitstream element containing information on one or several jointly decoded channels

codec: encoder-decoder chain

coefficient: time domain sample transformed into frequency domain

constant bitrate stream: Stream with equal-sized frames

dB_{FS}: dB relative to full scale

dBTP: signal amplitude relative to 100 % full scale, true-peak measurement expressed in dB (see [i.3])

envelope: vector of scale factor values

NOTE: An envelope can be of two kinds: signal or noise.

frame: cohesive section of bits in the bitstream

frame length: temporal extent of a frame when decoded to PCM

frame rate: number of frames decoded per second in realtime operation

frame size: extent of a frame in the bitstream domain

framing: method to determine the QMF time slot group borders of signal or noise envelopes in A-SPX

(full-bandwidth) channel: audio channel capable of full audio bandwidth

NOTE: All channels (left, centre, right, left surround, right surround, left back, right back) except the lfe channel are full-bandwidth channels.

helper element: variable, array or matrix derived from a bitstream element

I-frame: independently decodeable frame

low frequency effects (lfe) channel: optional single channel of limited (< 120 Hz) bandwidth

NOTE: The optional lfe channel is intended to be reproduced at a level +10 dB with respect to the full-bandwidth channels. It allows high sound pressure levels to be provided for low frequency sounds.

LUFS: loudness units relative to nominal full scale (see [i.11])

noise scale factor: average noise floor within the region in a QMF time/frequency matrix defined by a noise envelope and a scale factor subband group

P-Frame: predictively coded frame

presentation: set of one or more AC-4 substreams to be presented simultaneously

presentation configuration: set of metadata to describe how a presentation has to be decoded

QMF matrix: representation of the QMF time/frequency domain as a matrix

NOTE: The columns of a QMF matrix represent elements at the same time and rows represent elements at the same frequency.

QMF time slot: time range represented by one column in a QMF matrix

(QMF) subband: frequency range represented by one row in a QMF matrix, carrying a subsampled signal

(QMF) subband group: grouping of adjacent QMF subbands

(QMF) subsample: single element of a QMF matrix

signal scale factor: average energy of the signal within the region in a QMF matrix defined by an A-SPX envelope and an A-SPX scale factor subband group

spectral data: frequency domain data

spectral frontend: tool used to decode the encoded spectral data before feeding it into the windowing and IMDCT blocks

NOTE: There are two different frontends in AC-4: The Speech Spectral Frontend (SSF), and the Audio Spectral Frontend (ASF).

spectral line: frequency coefficient

substream: part of an AC-4 bitstream, contains audio data and corresponding metadata

tiling: method to determine the QMF subband group structure for each of the envelopes in A-SPX

transform length: block length

variable bitrate stream: stream with unconstrained frame sizes, or where the buffer model is unknown

window: weighting function associated with the IMDCT transform of a block

3.2 Symbols

For the purposes of the present document, the following symbols apply:

X^*	complex conjugate of value X, if X is a scalar; and conjugate transpose if X is a vector
$\lceil x \rceil$	round x towards plus infinity
$\lfloor x \rfloor$	round x towards minus infinity

3.3 Abbreviations

For the purposes of the present document, the following abbreviations apply:

ABR	Average Bitrate
A-CPL	Advanced CouPLing
ASF	Audio Spectral Frontend
A-SPX	Advanced SPectral eXtension tool
ARIB	Association of Radio Industries and Business
ATSC	Advanced Television Systems Committee
AVR	Audio/Video Receiver
BCP	Best Current Practice
CB	Codebook
CBR	Constant Bitrate
CDF	Cumulative Distribution Function
CM	Complete Main
DASH	Dynamic Adaptive Streaming over HTTP
dBTP	dB True Peak
DC	Direct Current
DE	Dialog Enhancement
DRC	Dynamic Range Control
DSI	Decoder Specific Information
EBU	European Broadcasting Union
HF	High Frequency
HI	Hearing Impaired
HSF	High Sampling Frequency
HTTP	Hypertext Transfer Protocol
IFFT	Inverse Fast Fourier Transform
IIR	Infinite Impulse Response
IMDCT	Inverse Modified Discrete Cosine Transform
ISO	International Organization for Standardization
IWC	Interleaved Waveform Coding
KBD	Kaiser-Bessel derived
LFE	Low Frequency Effects
LKFS	Loudness, K weighted, relative to nominal full scale
LU	Loudness Unit (see [i.11])
LUFS	Loudness Units relative to Full Scale
LUT	Look-Up Table
M/S	Mid/Side
MDCT	Modified Discrete Cosine Transform
ME	Music and Effects
MPD	Media Presentation Description
MPEG	Motion Picture Experts Group
MSB	Most Significant Bit
PCM	Pulse Code Modulation
QMF	Quadrature Mirror Filter

RMS	Root Mean Square
SAP	Stereo Audio Processing
SEC	Spectral Extension Components
SNF	Spectral Noise Fill
SSF	Speech Spectral Frontend
TOC	Table of Contents
UMD	Universal MetaData
VI	Visually Impaired
VO	Voice Over
VBR	Variable Bitrate
WCC	Waveform Coded Components
XML	eXtensible Markup Language

Please also refer to table D.1 for a listing of speaker location abbreviations.

3.4 Conventions

Unless otherwise stated, the following convention regarding the notation is used:

- Constants are indicated by upper-case italic, e.g. *NOISE_FLOOR_OFFSET*.
- Tables are indicated as *TABLE[idx]*.
- Functions are indicated as *func(x).*
- Variables are indicated by italic, e.g. *variable*.
- Vectors and vector components are indicated by bold lower-case names, e.g. **vector** or $\mathbf{vector}_{\mathbf{idx}}$.
- Matrices (and vectors of vectors) are indicated by bold upper-case single letter names, e.g. **M** or $\mathbf{M}_{\mathbf{row,column}}$.
- Indices to tables, vectors and matrices are zero based. The top left element of matrix **M** is $\mathbf{M}_{\mathbf{0,0}}$.
- Bitstream syntactic elements are indicated by the use of a different font, e.g. `dynrng`. All bitstream elements are described in clause 4.3.
- Normal pseudo-code interpretation of flowcharts is assumed, with no rounding or truncation unless explicitly stated.
- Units of [dB$_2$] refer to the approximation $1dB \equiv factor\ of\ \sqrt[6]{2{,}0}$, i.e. $6dB \equiv factor\ of\ 2{,}0$.
- Fractional frame rates are written in "shorthand notation" as defined in table 1.

Table 1: Shorthand notation for frame rates

Fractional framerate	Shorthand
$24 \times {}^{1000}/_{1001}$	23,976
$30 \times {}^{1000}/_{1001}$	29,97
$48 \times {}^{1000}/_{1001}$	47,952
$60 \times {}^{1000}/_{1001}$	59,94
$120 \times {}^{1000}/_{1001}$	119,88

4 Bitstream syntax

4.1 Semantics of syntax specification

The following pseudocode within syntax boxes describes the order of arrival of information within the bitstream. This pseudocode is roughly based on C language syntax, but simplified for ease of reading. For bitstream elements which are larger than 1 bit, the order of the bits in the serial bitstream is either most-significant-bit-first (for numerical values), or left-bit-first (for bit-field values). Fields or elements contained in the bitstream are indicated with **bold** type. Syntactic elements are typographically distinguished by the use of a different font (e.g. dynrng).

Some AC-4 bitstream elements naturally form arrays. This syntax specification treats all bitstream elements individually, whether or not they would naturally be included in arrays. Arrays are thus described as multiple elements (as in **companding_active[ch]** as opposed to simply **companding_active** or **companding_active[ ]**), and control structures such as *for* loops are employed to increment the index ([ch] for channel in this example).

When the number of bits is variable and not bound by constants, it is indicated as VAR.

4.2 Syntax specification

4.2.1 raw_ac4_frame - Raw AC-4 frame

Table 2: Syntax of raw_ac4_frame()

Syntax	No. of bits
raw_ac4_frame()	
{	
ac4_toc();	
fill_area;	VAR
byte_align;	0…7
for (i = 0; i < n_substreams; i++) {	
ac4_substream_data(); /* sub stream data stays byte aligned */	
}	
fill_area;	VAR
byte_align;	0…7
}	

4.2.2 variable_bits - Variable bits

Table 3: Syntax of variable_bits()

Syntax	No. of bits
variable_bits(n_bits)	
{	
value = 0;	
do {	
value += **read**;	**n_bits**
if (**b_read_more**) {	**1**
value <<= n_bits;	
value += (1<<n_bits);	
}	
} while(b_read_more);	
return value;	
}	

4.2.3 AC-4 frame info

4.2.3.1 ac4_toc - AC-4 table of contents

Table 4: Syntax of ac4_toc()

Syntax	No. of bits
ac4_toc()	
{	
bitstream_version;	2
if (bitstream_version == 3) {	
bitstream_version += **variable_bits(2);**	
}	
sequence_counter;	10
if (**b_wait_frames**) {	1
wait_frames;	3
if (wait_frames > 0)	
reserved;	2
}	
}	
fs_index;	1
frame_rate_index;	4
b_iframe_global;	1
b_single_presentation;	1
if (b_single_presentation == 1) {	
n_presentations = 1;	
}	
else {	
if (**b_more_presentations** == 1) {	1
n_presentations = **variable_bits(2)** + 2;	
} else {	
n_presentations = 0;	
}	
}	
payload_base = 0;	
if (**b_payload_base**) {	1
payload_base = **payload_base_minus1** + 1;	5
if (payload_base == 0x20) {	
payload_base += **variable_bits(3);**	
}	
}	
for (i = 0; i < n_presentations; i++) {	
ac4_presentation_info();	
}	
substream_index_table();	
byte_align;	0...7
}	

4.2.3.2 ac4_presentation_info - AC-4 presentation information

Table 5: Syntax of ac4_presentation_info()

Syntax	No. of bits
ac4_presentation_info()	
{	
if (**b_single_substream** != 1) {	1
presentation_config	3
if (presentation_config == 7) {	
presentation_config += **variable_bits(2)**;	
}	
}	
presentation_version();	
if (b_single_substream != 1 && presentation_config == 6) {	
b_add_umd_substreams = 1;	
} else {	
b_mdcompat;	1
if (**b_belongs_to_presentation_group**) {	1
presentation_group = **variable_bits(2)**;	
}	
frame_rate_multiply_info();	
umd_info();	
if (b_single_substream == 1) {	
ac4_substream_info();	
} else {	
b_hsf_ext;	1
switch(presentation_config) {	
case 0: /* Dry main + Dialog */	
ac4_substream_info(); /* Main */	
if (b_hsf_ext == 1) {	
ac4_hsf_ext_substream_info(); /* Main HSF */	
}	
ac4_substream_info(); /* Dialog */	
break;	
case 1: /* Main + DE */	
ac4_substream_info(); /* Main */	
if (b_hsf_ext == 1) {	
ac4_hsf_ext_substream_info(); /* Main HSF */	
}	
ac4_substream_info(); /* DE */	
break;	
case 2: /* Main + Associate */	
ac4_substream_info(); /* Main */	
if (b_hsf_ext == 1) {	
ac4_hsf_ext_substream_info(); /* Main HSF */	
}	
ac4_substream_info(); /* Associate */	
break;	
case 3: /* Dry Main + Dialog + Associate */	
ac4_substream_info(); /* Main */	
if (b_hsf_ext == 1) {	
ac4_hsf_ext_substream_info(); /* Main HSF */	
}	
ac4_substream_info(); /* Dialog */	
ac4_substream_info(); /* Associate */	
break;	
case 4: /* Main + DE Associate */	
ac4_substream_info(); /* Main */	
if (b_hsf_ext == 1) {	
ac4_hsf_ext_substream_info(); /* Main HSF */	
}	
ac4_substream_info(); /* DE */	
ac4_substream_info(); /* Associate */	
break;	
case 5: /* Main + HSF ext */	
ac4_substream_info(); /* Main */	
if (b_hsf_ext == 1) {	
ac4_hsf_ext_substream_info(); /* Main HSF */	
}	
break;	
default:	
presentation_config_ext_info();	
}	
}	
b_pre_virtualized;	1
b_add_umd_substreams;	1

}	
if (b_add_umd_substreams) {	
if (**n_add_umd_substreams** == 0) {	2
n_add_umd_substreams = **variable_bits(2)** + 4;	
}	
for (i = 0; i < n_add_umd_substreams; i++) {	
umd_info();	
}	
}	
}	

4.2.3.3 presentation_version - Presentation version information

Table 6: Syntax of presentation_version()

Syntax	No. of bits
presentation_version()	
{	
val = 0;	
while (**b_tmp** == 1) {	1
val++;	
}	
return val;	
}	

4.2.3.4 frame_rate_multiply_info - Frame rate multiplier information

Table 7: Syntax of frame_rate_multiply_info()

Syntax	No. of bits
frame_rate_multiply_info(frame_rate_index)	
{	
switch (frame_rate_index) {	
case 2:	
case 3:	
case 4:	
if (**b_multiplier**) {	1
multiplier_bit;	1
}	
break;	
case 0:	
case 1:	
case 7:	
case 8:	
case 9:	
b_multiplier;	1
break;	
default:	
break;	
}	
}	
NOTE: frame_rate_index is specified in ac4_toc().	

4.2.3.5 umd_info - Universal metadata information

Table 8: Syntax of umd_info()

Syntax	No. of bits
umd_info()	
{	
umd_version;	2
if (umd_version == 3) {	
umd_version += **variable_bits(2);**	
}	
key_id;	3
if (key_id == 7) {	
key_id += **variable_bits(3);**	
}	
if (**b_umd_payloads_substream_info**) {	1
umd_payloads_substream_info();	
}	
umd_protection();	
}	

4.2.3.6 ac4_substream_info - AC-4 substream information

Table 9: Syntax of ac4_substream_info()

Syntax	No. of bits
ac4_substream_info()	
{	
channel_mode;	1/2/4/7
if (channel_mode == 0b1111111) {	
channel_mode += **variable_bits(2);**	
}	
if (fs_index == 1) { /* 48 kHz */	
if (**b_sf_multiplier**) {	1
sf_multiplier;	1
}	
}	
if (**b_bitrate_info**) {	1
bitrate_indicator;	3/5
}	
if (channel_mode == 0b1111010 || channel_mode == 0b1111011 ||	
channel_mode == 0b1111100 || channel_mode == 0b1111101) {	
add_ch_base;	1
if (**b_content_type**) {	1
content_type();	
}	
for (i = 0; i < frame_rate_factor; i++) {	
b_iframe;	1
}	
if (**substream_index** == 3) {	2
substream_index += **variable_bits(2);**	
}	
}	
NOTE: fs_index is specified in ac4_toc() and frame_rate_factor in clause 4.3.3.5.3.	

4.2.3.7 content_type - Content type

Table 10: Syntax of content_type()

Syntax	No. of bits
`content_type()`	
`{`	
`    content_classifier;`	3
`    if (b_language_indicator) {`	1
`        if (b_serialized_language_tag) {`	1
`            b_start_tag;`	1
`            language_tag_chunk;`	16
`        } else {`	
`            n_language_tag_bytes;`	6
`            for (i = 0; i < n_language_tag_bytes; i++) {`	
`                language_tag_bytes;`	8
`            }`	
`        }`	
`    }`	
`}`	

4.2.3.8 presentation_config_ext_info - Presentation configuration extended information

Table 11: Syntax of presentation_config_ext_info()

Syntax	No. of bits
`presentation_config_ext_info()`	
`{`	
`    n_skip_bytes;`	5
`    if (b_more_skip_bytes) {`	1
`        n_skip_bytes += variable_bits(2) << 5;`	
`    }`	
`    for (i = 0; i < n_skip_bytes; i++) {`	
`        reserved;`	8
`    }`	
`}`	

4.2.3.9 ac4_hsf_ext_substream_info - AC-4 HSF extension substream information

Table 12: Syntax of ac4_hsf_ext_substream_info()

Syntax	No. of bits
`ac4_hsf_ext_substream_info()`	
`{`	
`    if (substream_index == 3) {`	2
`        substream_index += variable_bits(2);`	
`    }`	
`}`	

4.2.3.10 umd_payloads_substream_info - Universal metadata payloads substream information

Table 13: Syntax of umd_payloads_substream_info()

Syntax	No. of bits
`umd_payloads_substream_info()`	
`{`	
`    if (substream_index == 3) {`	2
`        substream_index += variable_bits(2);`	
`    }`	
`}`	

4.2.3.11 substream_index_table - Substream index table

Table 14: Syntax of substream_index_table()

```
Syntax                                                          No. of bits
substream_index_table()
{
    if (n_substreams == 0) {                                        2
        n_substreams = variable_bits(2) + 4;
    }
    if (n_substreams == 1) {
        b_size_present;                                             1
    } else {
        b_size_present = 1;
    }
    if (b_size_present) {
        for (s = 0; s < n_substreams; s++) {
            b_more_bits;                                            1
            substream_size[s];                                      10
            if (b_more_bits) {
                substream_size[s] += (variable_bits(2) << 10);
            }
        }
    }
}
```

4.2.4 AC-4 substreams

The `ac4_substream_data()` element for a specific substream index depends on the type of info element which refers to this specific substream. The mapping for the info elements defined in the present document is given in table 15.

Table 15: ac4_substream_data mapping

info element type referencing the substream	ac4_substream_data element
ac4_substream_info()	ac4_substream()
ac4_hsf_ext_substream_info()	ac4_hsf_ext_substream()
umd_payloads_substream_info()	umd_payloads_substream()

4.2.4.1 ac4_substream - AC-4 substream

Table 16: Syntax of ac4_substream()

```
Syntax                                                          No. of bits
ac4_substream()
{
    audio_size = audio_size_value;                                  15
    if (b_more_bits) {                                              1
        audio_size += variable_bits(7) << 15;
    }
    byte_align;                                                     0...7
    audio_data(channel_mode, b_iframe);
    fill_bits;                                                      VAR
    byte_align;                                                     0...7
    metadata(b_iframe);
    byte_align;                                                     0...7
}
```

NOTE: channel_mode and b_iframe are specified in the ac4_substream_info() which refers to the ac4_substream() via its substream_index.

4.2.4.2 ac4_hsf_ext_substream - AC-4 high sampling frequency extension substream

In case a substream is coded in 96/192 kHz, this additional substream holds the scale factor and spectral data beyond 24 kHz. A 48 kHz decoder may skip this substream.

Table 17: Syntax of ac4_hsf_ext_substream()

```
Syntax                                                                        No. of bits
ac4_hsf_ext_substream()
{
    for (g = 0; g < num_window_groups; g++) {
        for (i = num_sec_lsf[g]; i < num_sec[g]; i++) {
            if (sect_cb[g][i] != 0 && sect_cb[g][i] <= 11) {
                sect_start_line = sect_sfb_offset[g][sect_start[g][i]];
                sect_end_line = sect_sfb_offset[g][sect_end[g][i]];
                for (k = sect_start_line; k < sect_end_line;) {
                    if (CB_DIM[sect_cb[g][i]] == 4) {
                        quad_qspec_lines = huff_decode( sect_cb[g][i],
                                                        asf_qspec_hcw);       1…16
                        quant_spec[k]   = get_qline(quad_qspec_lines, 1);
                        quant_spec[k+1] = get_qline(quad_qspec_lines, 2);
                        quant_spec[k+2] = get_qline(quad_qspec_lines, 3);
                        quant_spec[k+3] = get_qline(quad_qspec_lines, 4);
                        if (UNSIGNED_CB[sect_cb[g][i]])
                            quad_sign_bits;                                   0…4
                        k += 4;
                    }
                    else {  /* (CB_DIM[sect_cb[g][i]] == 2) */
                        pair_qspec_lines = huff_decode( sect_cb[g][i],
                                                        asf_qspec_hcw);       1…15
                        quant_spec[k]   = get_qline(pair_spec_lines, 1);
                        quant_spec[k+1] = get_qline(pair_spec_lines, 2);
                        if (UNSIGNED_CB[sect_cb[g][i]])
                            pair_sign_bits;                                   0…2
                        if (sect_cb[g][i] == 11) {
                            if (quant_spec[k] == 16)
                                quant_spec[k] = ext_decode(ext_code);         5…21
                            if (quant_spec[k+1] == 16)
                                quant_spec[k+1] = ext_decode(ext_code);       5…21
                        }
                        k += 2;
                    }
                }
            }
        }
    }
    for (g = 0; g < num_window_groups; g++) {
        start_sfb = num_sfb_48(get_transf_length(g));
        for (sfb = start_sfb; sfb < get_max_sfb(g); sfb++) {
            if (sfb_cb[g][sfb] != 0) {
                if (max_quant_idx[g][sfb] > 0) {
                    if (first_scf_found == 1)
                        dpcm_sf[g][sfb] = huff_decode(  ASF_HCB_SCALEFAC,
                                                        asf_sf_hcw);          1…17
                    else
                        first_scf_found = 1;
                }
            }
        }
    }
    if (b_snf_data_exists) {
        for (g = 0; g < num_window_groups; g++) {
            start_sfb = num_sfb_48(get_transf_length(g));
            for (sfb = start_sfb; sfb < get_max_sfb(g); sfb++) {
                if ((sfb_cb[g][sfb] == 0) ||
                    (max_quant_idx[g][sfb] == 0)) {
                    dpcm_snf[g][sfb] = huff_decode( ASF_HCB_SNF,
                                                    asf_snf_hcw);             3…8
                }
            }
        }
    }
    byte_align;                                                               0…7
}
```

4.2.4.3 umd_payloads_substream - Universal metadata payloads substream

Table 18: Syntax of umd_payloads_substream()

Syntax	No. of bits
umd_payloads_substream()	
{	
while (**umd_payload_id** != 0) {	5
if (umd_payload_id == 31) {	
umd_payload_id += **variable_bits(5);**	
}	
umd_payload_config();	
umd_payload_size = **variable_bits(8);**	
for (i = 0; i < umd_payload_size; i++) {	
umd_payload_byte[i];	8
}	
}	
byte_align;	0…7
}	

4.2.5 audio_data - Audio data

Table 19: Syntax of audio_data()

Syntax	No. of bits
audio_data(channel_mode, b_iframe)	
{	
switch (channel_mode) {	
case mono:	
single_channel_element(b_iframe);	
break;	
case stereo:	
channel_pair_element(b_iframe);	
break;	
case 3.0:	
3_0_channel_element(b_iframe);	
break;	
case 5.0:	
5_X_channel_element(0, b_iframe);	
break;	
case 5.1:	
5_X_channel_element(1, b_iframe);	
break;	
case 7.0:	
case 7.1:	
7_X_channel_element(channel_mode, b_iframe);	
break;	
}	
}	

4.2.6 Channel elements

4.2.6.1 single_channel_element - Single channel element

Table 20: Syntax of single_channel_element()

Syntax	No. of bits
single_channel_element(b_iframe)	
{	
mono_codec_mode;	1
if (b_iframe) {	
if (mono_codec_mode == ASPX) {	
aspx_config();	
}	
}	
if (mono_codec_mode == SIMPLE) {	
mono_data(0);	
}	
else {	
companding_control(1);	
mono_data(0);	
aspx_data_1ch();	
}	
}	

4.2.6.2 mono_data - Mono data

Table 21: Syntax of mono_data()

Syntax	No. of bits
mono_data(b_lfe)	
{	
if (b_lfe) {	
spec_frontend = ASF;	
sf_info_lfe();	
}	
else {	
spec_frontend;	1
sf_info(spec_frontend, 0, 0);	
}	
sf_data(spec_frontend);	
}	

4.2.6.3 channel_pair_element - Channel pair element

Table 22: Syntax of channel_pair_element()

```
Syntax                                                          No. of bits
channel_pair_element(b_iframe)
{
    stereo_codec_mode;                                               2

    if (b_iframe) {
        if (stereo_codec_mode == ASPX) {
            aspx_config();
        }
        if (stereo_codec_mode == ASPX_ACPL_1) {
            aspx_config();
            acpl_config_1ch(PARTIAL);
        }
        if (stereo_codec_mode == ASPX_ACPL_2) {
            aspx_config();
            acpl_config_1ch(FULL);
        }
    }

    switch (stereo_codec_mode) {
    case SIMPLE:
        stereo_data();
        break;
    case ASPX:
        companding_control(2);
        stereo_data();
        aspx_data_2ch();
        break;
    case ASPX_ACPL_1:
        companding_control(1);
        if (b_enable_mdct_stereo_proc) {                             1
            spec_frontend_m = ASF;
            spec_frontend_s = ASF;
            sf_info(ASF, 1, 0);
            chparam_info();
        }
        else {
            spec_frontend_m;                                         1
            sf_info(spec_frontend_m, 0, 0);
            spec_frontend_s;                                         1
            sf_info(spec_frontend_s, 0, 1);
        }
        sf_data(spec_frontend_m);
        sf_data(spec_frontend_s);
        aspx_data_1ch();
        acpl_data_1ch();
        break;
    case ASPX_ACPL_2:
        companding_control(1);
        spec_frontend;                                               1
        sf_info(spec_frontend, 0, 0);
        sf_data(spec_frontend);
        aspx_data_1ch();
        acpl_data_1ch();
        break;
    }
}
```

4.2.6.4 stereo_data - Stereo data

Table 23: Syntax of stereo_data()

```
Syntax                                              No. of bits
stereo_data()
{
    if (b_enable_mdct_stereo_proc) {                     1
        spec_frontend_l = ASF;
        spec_frontend_r = ASF;
        sf_info(ASF, 0, 0);
        chparam_info();
    }
    else {
        spec_frontend_l;                                 1
        sf_info(spec_frontend_l, 0, 0);
        spec_frontend_r;                                 1
        sf_info(spec_frontend_r, 0, 0);
    }
    sf_data(spec_frontend_l);
    sf_data(spec_frontend_r);
}
```

4.2.6.5 3_0_channel_element - 3.0 channel element

Table 24: Syntax of 3_0_channel_element()

```
Syntax                                              No. of bits
3_0_channel_element(b_iframe)
{
    3_0_codec_mode;                                      1
    if (3_0_codec_mode == ASPX) {
        if (b_iframe) {
            aspx_config();
        }
        companding_control(3);
    }
    switch (3_0_coding_config) {                         1
    case 0:
        stereo_data();
        mono_data(0);
        break;
    case 1:
        three_channel_data();
        break;
    }
    if (3_0_codec_mode == ASPX) {
        aspx_data_2ch();
        aspx_data_1ch();
    }
}
```

4.2.6.6 5_X_channel_element - 5.X channel element

Table 25: Syntax of 5_X_channel_element()

Syntax	No. of bits
5_X_channel_element(b_has_lfe, b_iframe)	
{	
5_X_codec_mode;	3
if (b_iframe) {	
if (5_X_codec_mode == ASPX ||	
5_X_codec_mode == ASPX_ACPL_1 ||	
5_X_codec_mode == ASPX_ACPL_2 ||	
5_X_codec_mode == ASPX_ACPL_3) {	
aspx_config();	
}	
if (5_X_codec_mode == ASPX_ACPL_1) {	
acpl_config_1ch(PARTIAL);	
}	
if (5_X_codec_mode == ASPX_ACPL_2) {	
acpl_config_1ch(FULL);	
}	
if (5_X_codec_mode == ASPX_ACPL_3) {	
acpl_config_2ch();	
}	
}	
if (b_has_lfe) {	
mono_data(1);	
}	
switch (5_X_codec_mode) {	
case SIMPLE:	
case ASPX:	
if (5_X_codec_mode == ASPX) {	
companding_control(5);	
}	
switch (**coding_config**) {	2
case 0:	
2ch_mode;	1
two_channel_data();	
two_channel_data();	
mono_data(0);	
break;	
case 1:	
three_channel_data();	
two_channel_data();	
break;	
case 2:	
four_channel_data();	
mono_data(0);	
break;	
case 3:	
five_channel_data();	
break;	
}	
if (5_X_codec_mode == ASPX) {	
aspx_data_2ch();	
aspx_data_2ch();	
aspx_data_1ch();	
}	
break;	
case ASPX_ACPL_1:	
case ASPX_ACPL_2:	
companding_control(3);	
if (**coding_config**) {	1
three_channel_data();	
} else {	
two_channel_data();	
}	
if (5_X_codec_mode == ASPX_ACPL_1) {	
max_sfb_master;	n_side_bits
chparam_info();	
chparam_info();	
sf_data(ASF);	
sf_data(ASF);	
}	
if (coding_config == 0) {	
mono_data(0);	

```
        }
        aspx_data_2ch();
        aspx_data_1ch();
        acpl_data_1ch();
        acpl_data_1ch();
        break;
    case ASPX_ACPL_3:
        companding_control(2);
        stereo_data();
        aspx_data_2ch();
        acpl_data_2ch();
        break;
    }
}
```
NOTE: n_side_bits is derived by taking the largest signalled transform length from three_channel_data() or two_channel_data() above. This largest transform length determines the number of bits as per table 104.	

4.2.6.7 two_channel_data - Two channel data

Table 26: Syntax of two_channel_data()

Syntax	No. of bits

```
two_channel_data()
{
    if (b_enable_mdct_stereo_proc) {
        sf_info(ASF, 0, 0);
        chparam_info();
    }
    else {
        sf_info(ASF, 0, 0);
        sf_info(ASF, 0, 0);
    }
    sf_data(ASF);
    sf_data(ASF);
}
```
	
	1

4.2.6.8 three_channel_data - Three channel data

Table 27: Syntax of three_channel_data()

Syntax	No. of bits
```
three_channel_data()
{
    sf_info(ASF, 0, 0);
    three_channel_info();
    sf_data(ASF);
    sf_data(ASF);
    sf_data(ASF);
}
```

4.2.6.9 four_channel_data - Four channel data

Table 28: Syntax of four_channel_data()

Syntax	No. of bits
```
four_channel_data()
{
    sf_info(ASF, 0, 0);
    four_channel_info();
    sf_data(ASF);
    sf_data(ASF);
    sf_data(ASF);
    sf_data(ASF);
}
```

4.2.6.10 five_channel_data - Five channel data

Table 29: Syntax of five_channel_data()

Syntax	No. of bits
`five_channel_data()`	
`{`	
`    sf_info(ASF, 0, 0);`	
`    five_channel_info();`	
`    sf_data(ASF);`	
`    sf_data(ASF);`	
`    sf_data(ASF);`	
`    sf_data(ASF);`	
`    sf_data(ASF);`	
`}`	

4.2.6.11 three_channel_info - Three channel info

Table 30: Syntax of three_channel_info()

Syntax	No. of bits
`three_channel_info()`	
`{`	
`    chel_matsel;`	4
`    chparam_info();`	
`    chparam_info();`	
`}`	

4.2.6.12 four_channel_info - Four channel info

Table 31: Syntax of four_channel_info()

Syntax	No. of bits
`four_channel_info()`	
`{`	
`    chparam_info();`	
`    chparam_info();`	
`    chparam_info();`	
`    chparam_info();`	
`}`	

4.2.6.13 five_channel_info - Five channel info

Table 32: Syntax of five_channel_info()

Syntax	No. of bits
`five_channel_info()`	
`{`	
`    chel_matsel;`	4
`    chparam_info();`	
`    chparam_info();`	
`    chparam_info();`	
`    chparam_info();`	
`    chparam_info();`	
`}`	

4.2.6.14 7_X_channel_element - 7.X channel element

Table 33: Syntax of 7_X_channel_element()

```
Syntax                                                                          No. of bits
7_X_channel_element(channel_mode, b_iframe)
{
    7_X_codec_mode;                                                                  2
    if (b_iframe) {
        if (7_X_codec_mode != SIMPLE) {
            aspx_config();
        }
        if (7_X_codec_mode == ASPX_ACPL_1) {
            acpl_config_1ch(PARTIAL);
        }
        if (7_X_codec_mode == ASPX_ACPL_2) {
            acpl_config_1ch(FULL);
        }
    }

    if (channel_mode == "7.1") {
        mono_data(1);       /* LFE */
    }

    if (7_X_codec_mode == ASPX_ACPL_1 || 7_X_codec_mode == ASPX_ACPL_2) {
        companding_control(5);
    }

    switch (coding_config) {                                                         2
    case 0:
        2ch_mode;                                                                    1
        two_channel_data();
        two_channel_data();
        break;
    case 1:
        three_channel_data();
        two_channel_data();
        break;
    case 2:
        four_channel_data();
        break;
    case 3:
        five_channel_data();
        break;
    }

    if (7_X_codec_mode == SIMPLE || 7_X_codec_mode == ASPX) {
        if (b_use_sap_add_ch) {                                                      1
            chparam_info();
            chparam_info();
        }
        two_channel_data(); /* additional channels */
    }
    if (7_X_codec_mode == ASPX_ACPL_1) {
        max_sfb_master;                                                         n_side_bits
        chparam_info();
        chparam_info();
        sf_data(ASF);
        sf_data(ASF);
    }
    if (coding_config == 0 || coding_config == 2) {
        mono_data(0);
    }
    if (7_X_codec_mode != SIMPLE) {
        aspx_data_2ch();
        aspx_data_2ch();
        aspx_data_1ch();
    }
    if (7_X_codec_mode == ASPX) {
        aspx_data_2ch();
    }
    if (7_X_codec_mode == ASPX_ACPL_1 || 7_X_codec_mode == ASPX_ACPL_2) {
        acpl_data_1ch();
        acpl_data_1ch();
    }
}
```

NOTE:	n_side_bits is derived by taking the largest signalled transform length from the respective X_channel_data() under the 'switch (coding_config)' statement above. For coding_config == 0, this depends on which channel pair the additional channels are derived from. This largest transform length determines the number of bits as per table 104.

4.2.7 Spectral frontend

4.2.7.1 sf_info - Spectral frontend info

Table 34: Syntax of sf_info()

Syntax	No. of bits

`sf_info(spec_frontend, b_dual_maxsfb, b_side_limited)`
`{`
`    if (spec_frontend == ASF) {`
`        `**`asf_transform_info();`**
`        `**`asf_psy_info(b_dual_maxsfb, b_side_limited);`**
`    }`
`}`	

4.2.7.2 sf_info_lfe - Spectral frontend info for LFE

Table 35: Syntax of sf_info_lfe()

Syntax	No. of bits

`sf_info_lfe()`
`{`
`    b_long_frame = 1;  /* transform length = frame_length */`
`    `**`max_sfb[0];`**
`    num_window_groups = 1;`
`}`
	n_msfbl_bits

NOTE: n_msfbl_bits is defined in clause 4.3.6.2.1.	

4.2.7.3 sf_data - Spectral frontend data

Table 36: Syntax of sf_data()

Syntax	No. of bits

`sf_data(spec_frontend)`
`{`
`    if (spec_frontend == ASF) {`
`        `**`asf_section_data();`**
`        `**`asf_spectral_data();`**
`        `**`asf_scalefac_data();`**
`        `**`asf_snf_data();`**
`    }`
`    else {`
`        `**`ssf_data(b_iframe);`**
`    }`
`}`	

4.2.8 Audio spectral frontend

4.2.8.1 asf_transform_info - ASF transform info

Table 37: Syntax of asf_transform_info()

Syntax	No. of bits
`asf_transform_info()`	
`{`	
`    if (frame_len_base >= 1536) {`	
`        b_long_frame;`	1
`        if (b_long_frame == 0) {`	
`            transf_length[0];`	2
`            transf_length[1];`	2
`        }`	
`    } else {`	
`        transf_length;`	2
`    }`	
`}`	

4.2.8.2 asf_psy_info - ASF scale factor band info

Table 38: Syntax of asf_psy_info()

Syntax	No. of bits
`asf_psy_info(b_dual_maxsfb, b_side_limited)`	
`{`	
`    b_different_framing = 0;`	
`    if (frame_len_base >= 1536 && (b_long_frame == 0) &&`	
`        (transf_length[0] != transf_length[1])) {`	
`        b_different_framing = 1;`	
`    }`	
`    max_sfb[0];`	**n_msfb_bits**
`    if (b_dual_maxsfb) {`	
`        if (b_side_limited) {`	
`            max_sfb_side[0];`	**n_side_bits**
`        } else {`	
`            max_sfb_side[0];`	**n_msfb_bits**
`        }`	
`    }`	
`    if (b_different_framing) {`	
`        max_sfb[1];`	**n_msfb_bits**
`        if (b_dual_maxsfb) {`	
`            if (b_side_limited) {`	
`                max_sfb_side[1];`	**n_side_bits**
`            } else {`	
`                max_sfb_side[1];`	**n_msfb_bits**
`            }`	
`        }`	
`    }`	
`    for (i = 0; i < n_grp_bits; i++) {`	
`        scale_factor_grouping[i] =` **`scale_factor_grouping_bit;`**	1
`    }`	
`}`	
NOTE: n_msfb_bits and n_side_bits are defined in clause 4.3.6.2.1 and n_grp_bits is defined in clause 4.3.6.2.4.	

4.2.8.3 asf_section_data - ASF section data

Table 39: Syntax of asf_section_data()

Syntax	No. of bits
`asf_section_data()`	
`{`	
`    for (g = 0; g < num_window_groups; g++) {`	
`        transf_length_g = get_transf_length(g);`	
`        if (transf_length_g <= 2) {`	
`            sect_esc_val = ( 1 << 3 ) - 1; // = 7`	
`        }`	
`        else {`	
`            sect_esc_val = ( 1 << 5 ) - 1; // = 31`	
`        }`	
`        k = 0;`	
`        i = 0;`	
`        num_sec_lsf[g] = 0;`	
`        max_sfb = get_max_sfb(g);`	
`        while (k < max_sfb) {`	
`            sect_cb[g][i];`	4
`            sect_len = 1;`	
`            while (sect_len_incr == sect_esc_val) {`	3/5
`                sect_len += sect_esc_val;`	
`            }`	
`            sect_len += sect_len_incr;`	
`            sect_start[g][i] = k;`	
`            sect_end[g][i] = k + sect_len;`	
`            if (sect_end[g][i] >= num_sfb_48(transf_length_g)) {`	
`                num_sec_lsf[g] = i + 1;`	
`                if (sect_end[g][i] > num_sfb_48(transf_length_g)) {`	
`                    sect_end[g][i] = num_sfb_48(transf_length_g);`	
`                    i++;`	
`                    sect_start[g][i] = num_sfb_48(transf_length_g);`	
`                    sect_end[g][i] = k + sect_len;`	
`                    sect_cb[g][i] = sect_cb[g][i-1];`	
`                }`	
`            }`	
`            for (sfb = k; sfb < k + sect_len; sfb++)`	
`                sfb_cb[g][sfb] = sect_cb[g][i];`	
`            k += sect_len;`	
`            i++;`	
`        }`	
`        num_sec[g] = i;`	
`        if (num_sec_lsf[g] == 0)`	
`            num_sec_lsf[g] = num_sec[g];`	
`    }`	
`}`	
NOTE: num_sfb_48(transf_length_g) is the number of the scale factor bands for the respective transform length at 48 kHz sampling frequency as defined in table B.1.	

4.2.8.4 asf_spectral_data - ASF spectral data

Table 40: Syntax of asf_spectral_data()

```
Syntax                                                                          No. of bits
asf_spectral_data()
{
    for (g = 0; g < num_window_groups; g++) {
        for (i = 0; i < num_sec_lsf[g]; i++) {
            if (sect_cb[g][i] != 0 && sect_cb[g][i] <= 11) {
                sect_start_line = sect_sfb_offset[g][sect_start[g][i]];
                sect_end_line = sect_sfb_offset[g][sect_end[g][i]];
                for (k = sect_start_line; k < sect_end_line;) {
                    if (CB_DIM[sect_cb[g][i]] == 4) {
                        quad_qspec_lines = huff_decode( sect_cb[g][i],
                                                        asf_qspec_hcw);         1…16
                        quant_spec[k]   = get_qline(quad_qspec_lines, 1);
                        quant_spec[k+1] = get_qline(quad_qspec_lines, 2);
                        quant_spec[k+2] = get_qline(quad_qspec_lines, 3);
                        quant_spec[k+3] = get_qline(quad_qspec_lines, 4);
                        if (UNSIGNED_CB[sect_cb[g][i]])
                            quad_sign_bits;                                     0…4
                        k += 4;
                    }
                    else {  /* (CB_DIM[sect_cb[g][i]] == 2) */
                        pair_qspec_lines = huff_decode( sect_cb[g][i],
                                                        asf_qspec_hcw);         1…15
                        quant_spec[k]   = get_qline(pair_spec_lines, 1);
                        quant_spec[k+1] = get_qline(pair_spec_lines, 2);
                        if (UNSIGNED_CB[sect_cb[g][i]])
                            pair_sign_bits;                                     0…2
                        if (sect_cb[g][i] == 11) {
                            if (quant_spec[k] == 16)
                                quant_spec[k] = ext_decode(ext_code);           5…21
                            if (quant_spec[k+1] == 16)
                                quant_spec[k+1] = ext_decode(ext_code);         5…21
                        }
                        k += 2;
                    }
                }
            }
        }
    }
}
```

4.2.8.5 asf_scalefac_data - ASF scale factor data

Table 41: Syntax of asf_scalefac_data()

```
Syntax                                                                          No. of bits
asf_scalefac_data()
{
    reference_scale_factor;                                                     8
    first_scf_found = 0;
    for (g = 0; g < num_window_groups; g++) {
        max_sfb = min(get_max_sfb(g), num_sfb_48(get_transf_length(g));
        for (sfb = 0; sfb < max_sfb; sfb++) {
            if (sfb_cb[g][sfb] != 0) {
                if (max_quant_idx[g][sfb] > 0) {
                    if (first_scf_found == 1)
                        dpcm_sf[g][sfb] = huff_decode(  ASF_HCB_SCALEFAC,
                                                        asf_sf_hcw);            1…17
                    else
                        first_scf_found = 1;
                }
            }
        }
    }
}
```

NOTE: max_quant_idx[g][sfb] is the maximum of the absolute values of the quantized spectral lines for group g and scale factor band sfb.

4.2.8.6 asf_snf_data - ASF spectral noise fill data

Table 42: Syntax of asf_snf_data()

Syntax	No. of bits
asf_snf_data(b_iframe)	
{	
if (**b_snf_data_exists**) {	1
for (g = 0; g < num_window_groups; g++) {	
transf_length_g = get_transf_length(g);	
max_sfb = min(get_max_sfb(g), num_sfb_48(transf_length_g);	
for (sfb = 0; sfb < max_sfb; sfb++) {	
if ((sfb_cb[g][sfb] == 0) ||	
(max_quant_idx[g][sfb] == 0)) {	
dpcm_snf[g][sfb] = huff_decode(ASF_HCB_SNF,	
asf_snf_hcw);	3...8
}	
}	
}	
}	
}	

4.2.9 Speech spectral frontend

4.2.9.1 ssf_data - Speech spectral frontend data

Table 43: Syntax of ssf_data()

Syntax	No. of bits
ssf_data(b_iframe)	
{	
if (b_iframe) {	
b_ssf_iframe = 1;	
} else {	
b_ssf_iframe;	1
}	
ssf_granule(b_ssf_iframe);	
if (frame_len_base >= 1536) {	
ssf_granule(0);	
}	
}	

4.2.9.2 ssf_granule - Speech spectral frontend granule

Table 44: Syntax of ssf_granule()

```
Syntax                                                                  No. of bits
ssf_granule(b_iframe)
{
    stride_flag;                                                             1

    if (b_iframe == 1) {
        num_bands_minus12;                                                   3
        num_bands = num_bands_minus12 + 12;
    }

    start_block = 0;
    end_block = 0;
    if ((stride_flag == LONG_STRIDE) && (b_iframe == 0)) {
        end_block = 1;
    }
    if (stride_flag == SHORT_STRIDE) {
        end_block = 4;
        if (b_iframe ==1) {
            start_block = 1;
        }
    }

    for (block = start_block; block < end_block; block++) {
        predictor_presence_flag[block];                                      1
        if (predictor_presence_flag[block] == 1) {
            if ((start_block == 1) && (block == 1)) {
                delta_flag[block] = 0;
            } else {
                delta_flag[block];                                           1
            }
        }
    }

    ssf_st_data();
    ssf_ac_data();
}
```

4.2.9.3 ssf_st_data - Speech spectral frontend static data

Table 45: Syntax of ssf_st_data()

Syntax	No. of bits
ssf_st_data()	
{	
env_curr_band0_bits;	**5**
if ((b_iframe == 1) && (stride_flag == SHORT_STRIDE)) {	
env_startup_band0_bits;	**5**
}	
if (stride_flag == SHORT_STRIDE) {	
for (block = 0; block < 4; block++) {	
gain_bits[block];	**4**
}	
}	
num_blocks = (stride_flag == SHORT_STRIDE) ? 4 : 1;	
for (block = 0; block < num_blocks; block++) {	
if (block >= start_block && block < end_block) {	
if (predictor_presence_flag[block] == 1) {	
if (delta_flag[block] == 1) {	
predictor_lag_delta_bits[block];	**4**
}	
else {	
predictor_lag_bits[block];	**9**
}	
}	
}	
variance_preserving_flag[block];	**1**
alloc_offset_bits[block];	**5**
}	
}	

4.2.9.4 ssf_ac_data - Speech spectral frontend arithmetic coded data

Table 46: Syntax of ssf_ac_data()

Syntax	No. of bits
ssf_ac_data()	
{	
env_curr_ac_bits; // num_bands-1 values	**VAR**
if ((b_iframe == 1) && (stride_flag == SHORT_STRIDE)) {	
env_startup_ac_bits; // num_bands-1 values	**VAR**
}	
num_blocks = (stride_flag == SHORT_STRIDE) ? 4 : 1;	
for (block = 0; block < num_blocks; block++) {	
if (((b_iframe == 1) && (block > 0)) || (b_iframe == 0)) {	
if (predictor_presence_flag[block] == 1) {	
predictor_gain_ac_bits[block]; // 1 value	**VAR**
}	
}	
q_mdct_coefficients_ac_bits[block]; // num_bins values	**VAR**
}	
}	

4.2.10 Stereo audio processing

4.2.10.1 chparam_info -Stereo information

Table 47: Syntax of chparam_info()

Syntax	No. of bits
chparam_info()	
{	
sap_mode;	2
if (sap_mode == 1) {	
for (g = 0; g < num_window_groups; g++) {	
for (sfb = 0; sfb < max_sfb[g]; sfb++) {	
ms_used[g][sfb];	1
}	
}	
}	
if (sap_mode == 3) {	
sap_data();	
}	
}	

4.2.10.2 sap_data - Stereo audio processing data

Table 48: Syntax of sap_data()

Syntax	No. of bits
sap_data()	
{	
sap_coeff_all;	1
if (sap_coeff_all == 0) {	
for (g = 0; g < num_window_groups; g++) {	
for (sfb = 0; sfb < max_sfb[g]; sfb += 2) {	
sap_coeff_used[g][sfb];	1
if ((sfb+1) < max_sfb[g]) {	
sap_coeff_used[g][sfb+1] = sap_coeff_used[g][sfb];	
}	
}	
}	
}	
else {	
for (g = 0; g < num_window_groups; g++) {	
for (sfb = 0; sfb < max_sfb[g]; sfb++) {	
sap_coeff_used[g][sfb] = 1;	
}	
}	
}	
if (num_window_groups != 1) {	
delta_code_time;	1
}	
for (g = 0; g < num_window_groups; g++) {	
for (sfb = 0; sfb < max_sfb[g]; sfb += 2) {	
if (sap_coeff_used[g][sfb]) {	
dpcm_alpha_q[g][sfb] = huff_decode(ASF_HCB_SCALEFAC,	
sap_hcw);	1...19
}	
}	
}	
}	

4.2.11 Companding control

Table 49: Syntax of companding_control()

Syntax	No. of bits
companding_control(num_chan)	
{	
sync_flag = 0;	
if (num _chan > 1) {	
sync_flag;	1
}	
b_need_avg = 0;	
nc = sync_flag ? 1 : num_chan;	
for (ch = 0; ch < nc; ch++) {	
b_compand_on[ch];	1
if (!b_compand_on[ch]) {	
b_need_avg = 1;	
}	
}	
if (b_need_avg) {	
b_compand_avg;	1
}	
}	

4.2.12 Advanced spectral extension - A-SPX

4.2.12.1 aspx_config - A-SPX configuration

Table 50: Syntax of aspx_config()

Syntax	No. of bits
aspx_config()	
{	
aspx_quant_mode_env;	1
aspx_start_freq;	3
aspx_stop_freq;	2
aspx_master_freq_scale;	1
aspx_interpolation;	1
aspx_preflat;	1
aspx_limiter;	1
aspx_noise_sbg;	2
aspx_num_env_bits_fixfix;	1
aspx_freq_res_mode;	2
}	

4.2.12.2 aspx_data_1ch - A-SPX 1-channel data

Table 51: Syntax of aspx_data_1ch()

Syntax	No. of bits
`aspx_data_1ch(b_iframe)`	
`{`	
`    if (b_iframe)`	
`        aspx_xover_subband_offset;`	3
`    aspx_framing(0);`	
`    aspx_delta_dir(0);`	
`    aspx_hfgen_iwc_1ch();`	
`    aspx_data_sig[0] = aspx_ec_data(SIGNAL,`	
`                                    aspx_num_env[0],`	
`                                    aspx_freq_res[0],`	
`                                    aspx_quant_mode_env,`	
`                                    0,`	
`                                    aspx_sig_delta_dir[0]);`	
`    aspx_data_noise[0] = aspx_ec_data(  NOISE,`	
`                                        aspx_num_noise[0],`	
`                                        0,`	
`                                        0,`	
`                                        0,`	
`                                        aspx_noise_delta_dir[0]);`	
`}`	

4.2.12.3 aspx_data_2ch - A-SPX 2-channel data

Table 52: Syntax of aspx_data_2ch()

Syntax	No. of bits
`aspx_data_2ch(b_iframe)`	
`{`	
`    if (b_iframe)`	
`        aspx_xover_subband_offset;`	3
`    aspx_framing(0);`	
`    aspx_balance;`	1
`    if (aspx_balance)`	
`        aspx_framing(1);`	
`    aspx_delta_dir(0);`	
`    aspx_delta_dir(1);`	
`    aspx_hfgen_iwc_2ch(aspx_balance);`	
`    aspx_data_sig[0] = aspx_ec_data(SIGNAL,`	
`                                    aspx_num_env[0],`	
`                                    aspx_freq_res[0],`	
`                                    aspx_quant_mode_env,`	
`                                    LEVEL,`	
`                                    aspx_sig_delta_dir[0]);`	
`    aspx_data_sig[1] = aspx_ec_data(SIGNAL,`	
`                                    aspx_num_env[1],`	
`                                    aspx_freq_res[1],`	
`                                    aspx_quant_mode_env,`	
`                                    aspx_balance ? BALANCE : LEVEL,`	
`                                    aspx_sig_delta_dir[1]);`	
`    aspx_data_noise[0] = aspx_ec_data(  NOISE,`	
`                                        aspx_num_noise[0],`	
`                                        0,`	
`                                        0,`	
`                                        LEVEL,`	
`                                        aspx_noise_delta_dir[0]);`	
`    aspx_data_noise[1] = aspx_ec_data(  NOISE,`	
`                                        aspx_num_noise[1],`	
`                                        0,`	
`                                        0,`	
`                                        aspx_balance ? BALANCE : LEVEL,`	
`                                        aspx_noise_delta_dir[1]);`	
`}`	

4.2.12.4 aspx_framing - A-SPX framing

Table 53: Syntax of aspx_framing()

Syntax	No. of bits
aspx_framing(ch, envbits, aspx_freq_res_mode)	
{	
aspx_num_rel_left = 0;	
aspx_num_rel_right = 0;	
aspx_int_class[ch];	1...3
switch (aspx_int_class[ch]) {	
case FIXFIX:	
aspx_num_env[ch] = 2^**tmp_num_env**; // Note 1	envbits+1
if(aspx_freq_res_mode == 0)	
aspx_freq_res[ch][0];	1
break;	
case FIXVAR:	
aspx_var_bord_right[ch];	2
aspx_num_rel_right[ch] = **tmp** + 1; // Note 2	2 (1)
for (rel = 0; rel < aspx_num_rel_right[ch]; rel++)	
aspx_rel_bord_right[ch][rel] = 2***tmp** + 2; // Note 2	2 (1)
break;	
case VARVAR:	
if(b_iframe)	
aspx_var_bord_left[ch];	2
aspx_num_rel_left[ch];	2
for (rel = 0; rel < aspx_num_rel_left[ch]; rel++)	
aspx_rel_bord_left[ch][rel] = 2***tmp** + 2; // Note 2	2 (1)
aspx_var_bord_right[ch];	2
aspx_num_rel_right[ch];	2
for (rel = 0; rel < aspx_num_rel_right[ch]; rel++)	
aspx_rel_bord_right[ch][rel] = 2***tmp** + 2; // Note 2	2 (1)
break;	
case VARFIX:	
if(b_iframe)	
aspx_var_bord_left[ch];	2
aspx_num_rel_left[ch] = **tmp** + 1; // Note 2	2 (1)
for (rel = 0; rel < aspx_num_rel_left[ch]; rel++)	
aspx_rel_bord_left[ch][rel] = 2***tmp** + 2; // Note 2	2 (1)
break;	
}	
if(aspx_int_class[ch] != FIXFIX) {	
aspx_num_env[ch] = aspx_num_rel_left[ch] + aspx_num_rel_right[ch] + 1;	
ptr_bits = ceil(log(aspx_num_env[ch]+2) / log(2)); // Note 3	
aspx_tsg_ptr[ch] = **tmp** - 1;	ptr_bits
if(aspx_freq_res_mode == 0)	
for (env = 0; env < aspx_num_env[ch]; env++)	
aspx_freq_res[ch][env]	1
}	
if (aspx_num_env[ch] > 1)	
aspx_num_noise[ch] = 2;	
else	
aspx_num_noise[ch] = 1;	
}	
NOTE 1: aspx_num_env is restricted according to clause 4.3.10.4.11. NOTE 2: The value within parenthesis applies when num_aspx_timeslots ≤ 12. NOTE 3: The division (/) is a float division without rounding or truncation.	

4.2.12.5 aspx_delta_dir - A-SPX direction of envelope delta coding

Table 54: Syntax of aspx_delta_dir()

Syntax	No. of bits
aspx_delta_dir(ch)	
{	
for (env = 0; env < aspx_num_env[ch]; env++) // Note	
aspx_sig_delta_dir[ch][env];	1
for (env = 0; env < aspx_num_noise[ch]; env++) // Note	
aspx_noise_delta_dir[ch][env];	1
}	
NOTE: aspx_num_env and aspx_num_noise are defined in clause 4.2.12.4.	

4.2.12.6 aspx_hfgen_iwc_1ch - A-SPX 1-channel HF generation and interleaved waveform coding

Table 55: Syntax of aspx_hfgen_iwc_1ch()

Syntax	No. of bits
`aspx_hfgen_iwc_1ch()`	
`{`	
`    for (n = 0; n < num_sbg_noise; n++)                      // Note 1`	
`        **aspx_tna_mode[n];**`	2
`    if (**aspx_ah_present**) {`	1
`        for (n = 0; n < num_sbg_sig_highres; n++)            // Note 1`	
`            **aspx_add_harmonic[n];**`	1
`    }`	
`    /* initialize frequency interleaved coding flags to zero */`	
`    for (n = 0; n < num_sbg_sig_highres; n++)                // Note 1`	
`        aspx_fic_used_in_sfb[n] = 0;`	
`    /* read frequency interleaved coding flags */`	
`    if (**aspx_fic_present**) {`	1
`        for (n = 0; n < num_sbg_sig_highres; n++) {          // Note 1`	
`            **aspx_fic_used_in_sfb[n];**`	1
`        }`	
`    }`	
`    /* initialize time interleaved coding flags to zero */`	
`    for (n = 0; n < num_aspx_timeslots; n++) {               // Note 2`	
`        aspx_tic_used_in_slot[n] = 0;`	
`    }`	
`    /* read time interleaved coding flags */`	
`    if (**aspx_tic_present**) {`	1
`        for (n = 0; n < num_aspx_timeslots; n++) {           // Note 2`	
`            **aspx_tic_used_in_slot[n];**`	1
`        }`	
`    }`	
`}`	
NOTE 1: Variables num_sbg_sig_highres and num_sbg_noise are derived according to clause 5.7.6.3.1. NOTE 2: Variable num_aspx_timeslots is derived in clause 5.7.6.3.3.	

4.2.12.7 aspx_hfgen_iwc_2ch - A-SPX 2-channel HF generation and interleaved waveform coding

Table 56: Syntax of aspx_hfgen_iwc_2ch()

```
Syntax                                                                         No. of bits
aspx_hfgen_iwc_2ch(aspx_balance)
{
    for (n = 0; n < num_sbg_noise; n++)                     // Note 1
        aspx_tna_mode[0][n];                                                   2
    if (aspx_balance == 0) {
        for (n = 0; n < num_sbg_noise; n++)                 // Note 1
            aspx_tna_mode[1][n];                                               2
    } else {
        for (n = 0; n < num_sbg_noise; n++)                 // Note 1
            aspx_tna_mode[1][n] = aspx_tna_mode[0][n]
    }

    if (aspx_ah_left) {                                                        1
        for (n = 0; n < num_sbg_sig_highres; n++)           // Note 1
            aspx_add_harmonic[0][n];                                           1
    }
    if (aspx_ah_right) {                                                       1
        for (n = 0; n < num_sbg_sig_highres; n++)           // Note 1
            aspx_add_harmonic[1][n];                                           1
    }

    /* initialize frequency interleaved coding flags to zero */
    for (n = 0; n < num_sbg_sig_highres; n++)               // Note 1
        aspx_fic_used_in_sfb[0][n] = aspx_fic_used_in_sfb[1][n] = 0;

    /* read frequency interleaved coding flags */
    if (aspx_fic_present) {                                                    1
        if (aspx_fic_left) {                                                   1
            for (n = 0; n < num_sbg_sig_highres; n++)       // Note 1
                aspx_fic_used_in_sfb[0][n];                                    1
        }
        if (aspx_fic_right) {                                                  1
            for (n = 0; n < num_sbg_sig_highres; n++){      // Note 1
                aspx_fic_used_in_sfb[1][n];                                    1
        }
    }

    /* initialize time interleaved coding flags to zero */
    for (n = 0; n < num_aspx_timeslots; n++)
        aspx_tic_used_in_slot[0][n] = aspx_tic_used_in_slot[1][n] = 0;

    aspx_tic_left = aspx_tic_right = 0;

    /* read time interleaved coding flags */
    if (aspx_tic_present) {                                                    1
        aspx_tic_copy;                                                         1
        if (aspx_tic_copy == 0) {
            aspx_tic_left;                                                     1
            aspx_tic_right;                                                    1
        }
        if (aspx_tic_copy || aspx_tic_left) {
            for (n = 0; n < num_aspx_timeslots; n++)        // Note 2
                aspx_tic_used_in_slot[0][n];                                   1
        }
        if (aspx_tic_right) {
            for (n = 0; n < num_aspx_timeslots; n++)        // Note 2
                aspx_tic_used_in_slot[1][n];                                   1
        }
        if (aspx_tic_copy) {
            for (n = 0; n < num_aspx_timeslots; n++)        // Note 2
                aspx_tic_used_in_slot[1][n]=aspx_tic_used_in_slot[0][n];
        }
    }
}
```

NOTE 1: Variables num_sbg_sig_highres and num_sbg_noise are derived according to clause 5.7.6.3.1.
NOTE 2: Variable num_aspx_timeslots is derived in clause 5.7.6.3.3.

4.2.12.8 aspx_ec_data - A-SPX entropy coded data

Table 57: Syntax of aspx_ec_data()

```
Syntax                                                                        No. of bits
aspx_ec_data(data_type, num_env, freq_res, quant_mode, stereo_mode,
direction)
{
    for (env = 0; env < num_env; env++) {
        if (data_type == SIGNAL) {
            if (freq_res[env])
                num_sbg = num_sbg_sig_highres;              // Note
            else
                num_sbg = num_sbg_sig_lowres;               // Note
        } else {
            num_sbg = num_sbg_noise;                        // Note
        }
        sm = stereo_mode;
        dir = direction[env];
        a_huff_data[env] = aspx_huff_data(data_type,num_sbg,qm,sm,dir);
    }
    return a_huff_data;
}
```

NOTE: Variables num_sbg_sig_highres and num_sbg_sig_lowres are derived in clause 5.7.6.3.1.2 and num_sbg_noise is derived according to clause 5.7.6.3.1.3.

4.2.12.9 aspx_huff_data - A-SPX Huffman data

Table 58: Syntax of aspx_huff_data()

```
Syntax                                                                          No. of bits
aspx_huff_data(data_type, num_sbg, quant_mode, stereo_mode, direction)
{
    if (direction == 0) {   // FREQ
        aspx_hcb = get_aspx_hcb(data_type, quant_mode, stereo_mode, F0);
        a_huff_data[0] = huff_decode(aspx_hcb, aspx_hcw);                          1…x
        aspx_hcb = get_aspx_hcb(data_type, quant_mode, stereo_mode, DF);
        for (i = 1; i < num_sbg; i++) {
            a_huff_data[i] = huff_decode_diff(aspx_hcb, aspx_hcw);                 1…x
        }
    }
    else {  // TIME
        aspx_hcb = get_aspx_hcb(data_type, quant_mode, stereo_mode, DT);
        for (i = 0; i < num_sbg; i++) {
            a_huff_data[i] = huff_decode_diff (aspx_hcb, aspx_hcw);                1…x
        }
    }
    return a_huff_data;
}
```

NOTE: The function get_aspx_hcb() is defined in clause 5.7.6.3.4.

4.2.13 Advanced coupling - A-CPL

4.2.13.1 acpl_config_1ch - A-CPL 1-channel configuration

Table 59: Syntax of acpl_config_1ch()

```
Syntax                                                                        No. of bits
acpl_config_1ch(acpl_1ch_mode)
{
    acpl_num_param_bands_id;                                                      2
    acpl_quant_mode;                                                              1
    if (acpl_1ch_mode == PARTIAL) {
        acpl_qmf_band;                                                            3
    }
}
```

NOTE: acpl_1ch_mode is a helper element defined in table 137.

4.2.13.2 acpl_config_2ch - A-CPL 2-channel configuration

Table 60: Syntax of acpl_config_2ch()

Syntax	No. of bits
acpl_config_2ch()	
{	
acpl_num_param_bands_id;	2
acpl_quant_mode_0;	1
acpl_quant_mode_1;	1
}	

4.2.13.3 acpl_data_1ch - A-CPL 1-channel data

Table 61: Syntax of acpl_data_1ch()

Syntax	No. of bits
acpl_data_1ch()	
{	
acpl_framing_data();	
num_bands = acpl_num_param_bands_id;	
acpl_alpha1 = **acpl_ec_data(ALPHA, num_bands, acpl_quant_mode);**	
acpl_beta1 = **acpl_ec_data(BETA, num_bands, acpl_quant_mode);**	
}	

4.2.13.4 acpl_data_2ch - A-CPL 2-channel data

Table 62: Syntax of acpl_data_2ch()

Syntax	No. of bits
acpl_data_2ch()	
{	
acpl_framing_data();	
num_bands = acpl_num_param_bands_id;	
acpl_alpha1 = **acpl_ec_data(ALPHA, num_bands, acpl_quant_mode_0);**	
acpl_alpha2 = **acpl_ec_data(ALPHA, num_bands, acpl_quant_mode_0);**	
acpl_beta1 = **acpl_ec_data(BETA, num_bands, acpl_quant_mode_0);**	
acpl_beta2 = **acpl_ec_data(BETA, num_bands, acpl_quant_mode_0);**	
acpl_beta3 = **acpl_ec_data(BETA3, num_bands, acpl_quant_mode_0);**	
acpl_gamma1 = **acpl_ec_data(GAMMA, num_bands, acpl_quant_mode_1);**	
acpl_gamma2 = **acpl_ec_data(GAMMA, num_bands, acpl_quant_mode_1);**	
acpl_gamma3 = **acpl_ec_data(GAMMA, num_bands, acpl_quant_mode_1);**	
acpl_gamma4 = **acpl_ec_data(GAMMA, num_bands, acpl_quant_mode_1);**	
acpl_gamma5 = **acpl_ec_data(GAMMA, num_bands, acpl_quant_mode_1);**	
acpl_gamma6 = **acpl_ec_data(GAMMA, num_bands, acpl_quant_mode_1);**	
}	

4.2.13.5 acpl_framing_data - A-CPL framing data

Table 63: Syntax of acpl_framing_data()

Syntax	No. of bits
acpl_framing_data()	
{	
acpl_interpolation_type;	1
acpl_num_param_sets_cod;	1
if (acpl_interpolation_type == 1) {	
for (ps = 0; ps < acpl_num_param_sets_cod + 1; ps++) {	
acpl_param_subsample[ps];	5
}	
}	
}	

4.2.13.6 acpl_ec_data - A-CPL entropy coded data

Table 64: Syntax of acpl_ec_data()

Syntax	No. of bits
`acpl_ec_data(data_type, data_bands, quant_mode)`	
`{`	
`for (ps = 0; ps < acpl_num_param_sets_cod + 1; ps++) {`	
`a_param_set[ps] =` **`acpl_huff_data(   data_type, data_bands,`**	
`quant_mode);`	
`}`	
`return a_param_set;`	
`}`	

4.2.13.7 acpl_huff_data - A-CPL Huffman data

Table 65: Syntax of acpl_huff_data()

Syntax	No. of bits
`acpl_huff_data(data_type, data_bands, quant_mode)`	
`{`	
`diff_type;`	1
`if (diff_type == 0) {   // DIFF_FREQ`	
`acpl_hcb = get_acpl_hcb(data_type, quant_mode, F0);`	
`a_huff_data[0] = huff_decode(acpl_hcb,` **`acpl_hcw`**`);`	1…x
`acpl_hcb = get_acpl_hcb(data_type, quant_mode, DF);`	
`for (i = 1; i < data_bands; i++) {`	
`a_huff_data[i] = huff_decode_diff(acpl_hcb,` **`acpl_hcw`**`);`	1…x
`}`	
`}`	
`else {  // DIFF_TIME`	
`acpl_hcb = get_acpl_hcb(data_type, quant_mode, DT);`	
`for (i = 0; i < data_bands; i++) {`	
`a_huff_data[i] = huff_decode_diff(acpl_hcb,` **`acpl_hcw`**`);`	1…x
`}`	
`}`	
`return a_huff_data;`	
`}`	
NOTE: The function get_acpl_hcb() is defined in clause 4.3.11.6.1.	

4.2.14 Metadata

4.2.14.1 metadata() - Metadata

Table 66: Syntax of metadata()

Syntax	No. of bits
`metadata(b_iframe)`	
`{`	
`basic_metadata();`	
`extended_metadata();`	
`tools_metadata_size =` **`tools_metadata_size_value;`**	7
`if (`**`b_more_bits`**`) {`	1
`tools_metadata_size +=` **`variable_bits(3)`** `<< 7;`	
`}`	
`drc_frame(b_iframe);`	
`dialog_enhancement(b_iframe);`	
`if (`**`b_umd_payloads_substream`**`) {`	1
`umd_payloads_substream();`	
`}`	
`}`	

4.2.14.2 basic_metadata - Basic metadata

Table 67: Syntax of basic_metadata()

```
Syntax                                                          No. of bits
basic_metadata(channel_mode)
{
    dialnorm_bits;                                                   7
    if (b_more_basic_metadata) {                                     1
        if (b_further_loudness_info) {                               1
            further_loudness_info();
        }
        if (channel_mode == stereo) {
            if (b_prev_dmx_info) {                                   1
                pre_dmixtyp_2ch;                                     3
                phase90_info_2ch;                                    2
            }
        }
        if (channel_mode > stereo) {
            if (b_dmx_coeff) {                                       1
                loro_center_mixgain;                                 3
                loro_surround_mixgain;                               3
                if (b_loro_dmx_loud_corr) {                          1
                    loro_dmx_loud_corr;                              5
                }
                if (b_ltrt_mixinfo) {                                1
                    ltrt_center_mixgain;                             3
                    ltrt_surround_mixgain;                           3
                }
                if (b_ltrt_dmx_loud_corr) {                          1
                    ltrt_dmx_loud_corr;                              5
                }
                if (channel_mode_contains_Lfe()) {
                    if (b_lfe_mixinfo) {                             1
                        lfe_mixgain;                                 5
                    }
                }
                preferred_dmx_method;                                2
            }
            if (channel_mode == 5_X) {
                if (b_predmxtyp_5ch) {                               1
                    pre_dmixtyp_5ch;                                 3
                }
                if (b_preupmixtyp_5ch) {                             1
                    pre_upmixtyp_5ch;                                4
                }
            }
            if (channel_mode == 7_X) {
                if (b_upmixtyp_7ch) {                                1
                    if (3/4/0) {
                        pre_upmixtyp_3_4;                            2
                    } else if (5/2/2) {
                        pre_upmixtyp_3_2_2;                          1
                    }
                }
            }
            phase90_info_mc;                                         2
            b_surround_attenuation_known;                            1
            b_lfe_attenuation_known;                                 1
        }
        if (b_dc_blocking) {                                         1
            dc_block_on;                                             1
        }
    }
}
```

4.2.14.3 further_loudness_info - Additional loudness information

Table 68: Syntax of further_loudness_info()

Syntax	No. of bits
`further_loudness_info()`	
`{`	
`    loudness_version;`	2
`    if (loudness_version == 3) {`	
`        loudness_version += extended_loudness_version;`	4
`    }`	
`    loud_prac_type;`	4
`    if (loud_prac_type != 0) {`	
`        if (b_loudcorr_dialgate) {`	1
`            dialgate_prac_type;`	3
`        }`	
`        b_loudcorr_type;`	1
`    }`	
`    if (b_loudrelgat) {`	1
`        loudrelgat;`	11
`    }`	
`    if (b_loudspchgat) {`	1
`        loudspchgat;`	11
`        dialgate_prac_type;`	3
`    }`	
`    if (b_loudstrm3s) {`	1
`        loudstrm3s;`	11
`    }`	
`    if (b_max_loudstrm3s) {`	1
`        max_loudstrm3s;`	11
`    }`	
`    if (b_truepk) {`	1
`        truepk;`	11
`    }`	
`    if (b_max_truepk) {`	1
`        max_truepk;`	11
`    }`	
`    if (b_prgmbndy) {`	1
`        prgmbndy = 1;`	
`        prgmbndy_bit = 0;`	
`        while (prgmbndy_bit == 0) {`	
`            prgmbndy <<= 1;`	
`            prgmbndy_bit;`	1
`        }`	
`        b_end_or_start;`	1
`        if (b_prgmbndy_offset) {`	1
`            prgmbndy_offset;`	11
`        }`	
`    }`	
`    if (b_lra) {`	1
`        lra;`	10
`        lra_prac_type;`	3
`    }`	
`    if (b_loudmntry) {`	1
`        loudmntry;`	11
`    }`	
`    if (b_max_loudmntry) {`	1
`        max_loudmntry;`	11
`    }`	
`    if (b_extension) {`	1
`        e_bits_size;`	5
`        if (e_bits_size == 31) {`	
`            e_bits_size += variable_bits(4);`	
`        }`	
`        extension_bits;`	e_bits_size
`    }`	
`}`	

4.2.14.4 extended_metadata - Extended metadata

Table 69: Syntax of extended_metadata()

```
Syntax                                                        No. of bits
extended_metadata(channel_mode, b_associated, b_dialog)
{
    if (b_associated) {
        if (b_scale_main) {                                        1
            scale_main;                                            8
        }
        if (b_scale_main_center) {                                 1
            scale_main_center;                                     8
        }
        if (b_scale_main_front) {                                  1
            scale_main_front;                                      8
        }
        if (channel_mode == mono) {
            pan_associated;                                        8
        }
    }
    if (b_dialog) {
        if (b_dialog_max_gain) {                                   1
            dialog_max_gain;                                       2
        }
        if (b_pan_dialog_present) {                                1
            if (n_channels == 1) {
                pan_dialog;                                        8
            } else {
                pan_dialog[0];                                     8
                pan_dialog[1];                                     8
                reserved;                                          2
            }
        }
    }
    if (b_channels_classifier) {                                   1
        if (channel_mode_contains_c()) {
            if (b_c_active) {                                      1
                b_c_has_dialog;                                    1
            }
        }
        if (channel_mode_contains_lr()) {
            if (b_l_active) {                                      1
                b_l_has_dialog;                                    1
            }
            if (b_r_active) {                                      1
                b_r_has_dialog;                                    1
            }
        }
        if (channel_mode_contains_LsRs()) {
            b_ls_active;                                           1
            b_rs_active;                                           1
        }
        if (channel_mode_contains_LrsRrs()) {
            b_lrs_active;                                          1
            b_rrs_active;                                          1
        }
        if (channel_mode_contains_LwRw()) {
            b_lw_active;                                           1
            b_rw_active;                                           1
        }
        if (channel_mode_contains_VhlVhr()) {
            b_vhl_active;                                          1
            b_vhr_active;                                          1
        }
        if (channel_mode_contains_Lfe()) {
            b_lfe_active;                                          1
        }
    }
    if (b_event_probability) {                                     1
        event_probability;                                         4
    }
}
```

4.2.14.5 drc_frame - DRC frame

Table 70: Syntax of drc_frame()

Syntax	No. of bits
drc_frame(b_iframe)	
{	
if (**b_drc_present**) {	1
if (b_iframe) {	
drc_config();	
}	
drc_data();	
}	
}	

4.2.14.6 drc_config - DRC configuration

Table 71: Syntax of drc_config()

Syntax	No. of bits
drc_config()	
{	
drc_decoder_nr_modes;	3
for (p = 0; p <= drc_decoder_nr_modes; p++) {	
drc_decoder_mode_config(p);	
}	
drc_eac3_profile;	3
}	

4.2.14.7 drc_decoder_mode_config - DRC decoder mode_config

Table 72: Syntax of drc_decoder_mode_config()

Syntax	No. of bits
drc_decoder_mode_config(pcount)	
{	
drc_decoder_mode_id[pcount];	3
if (drc_decoder_mode_id[pcount] > 3) {	
drc_output_level_from;	5
drc_output_level_to;	5
}	
if (**drc_repeat_profile_flag**) {	1
drc_repeat_id;	3
drc_compression_curve_flag[drc_decoder_mode_id[pcount]] = 1;	
} else {	
if (!**drc_default_profile_flag**) {	1
if (**drc_compression_curve_flag[drc_decoder_mode_id[pcount]]**)	1
{	
drc_compression_curve();	
} else {	
drc_gains_config[drc_decoder_mode_id[pcount]];	2
}	
}	
else {	
drc_compression_curve_flag[drc_decoder_mode_id[pcount]] = 1;	
}	
}	
}	

4.2.14.8 drc_compression_curve - Compression curve parameters

Table 73: Syntax of drc_compression_curve()

Syntax	No. of bits
drc_compression_curve()	
{	
drc_lev_nullband_low;	4
drc_lev_nullband_high;	4
drc_gain_max_boost;	4
if (drc_gain_max_boost > 0) {	
drc_lev_max_boost;	5
drc_nr_boost_sections;	1
if (drc_nr_boost_sections > 0) {	
drc_gain_section_boost;	4
drc_lev_section_boost;	5
}	
}	
drc_gain_max_cut;	5
if (drc_gain_max_cut > 0) {	
drc_lev_max_cut;	6
drc_nr_cut_sections;	1
if (drc_nr_cut_sections > 0) {	
drc_gain_section_cut;	5
drc_lev_section_cut;	5
}	
}	
drc_tc_default_flag;	1
if (!drc_tc_default_flag) {	
drc_tc_attack;	8
drc_tc_release;	8
drc_tc_attack_fast;	8
drc_tc_release_fast;	8
drc_adaptive_smoothing_flag;	1
if (drc_adaptive_smoothing_flag) {	
drc_attack_threshold;	5
drc_release_threshold;	5
}	
}	
}	

4.2.14.9 drc_data -DRC frame-based data

Table 74: Syntax of drc_data()

Syntax	No. of bits
drc_data()	
{	
curve_present = 0;	
for (p = 0; p <= drc_decoder_nr_modes; p++) {	
if (drc_compression_curve_flag[drc_decoder_mode_id[p]] == 0) {	
drc_gainset_size = **drc_gainset_size_value**;	**6**
if (**b_more_bits**) {	**1**
drc_gainset_size += **variable_bits(2)** << 6;	
}	
drc_version;	**2**
used_bits = 0;	
if (drc_version <= 1) {	
used_bits = **drc_gains(drc_decoder_mode_id[p])**;	
}	
if (drc_version >= 1) {	
bits_left = drc_gainset_size - 2 - used_bits;	
drc2_bits;	**bits_left**
}	
} else {	
curve_present = 1;	
}	
}	
if (curve_present) {	
drc_reset_flag;	**1**
drc_reserved;	**2**
}	
}	

4.2.14.10 drc_gains - DRC gains

Table 75: Syntax of drc_gains()

Syntax	No. of bits
drc_gains(mode)	
{	
drc_gain[0][0][0] = **drc_gain_val**;	**7**
if (drc_gains_config[mode] > 0) {	
for (ch = 0; ch < nr_drc_channels; ch++) {	
for (band = 0; band < nr_drc_bands; band++)	
for (sf = 0; sf < nr_drc_subframes; sf++) {	
if (sf != 0 || band != 0 || ch != 0) {	
diff = huff_decode_diff(DRC_HCB, **drc_gain_code**);	**1...x**
drc_gain[ch][sf][band] = ref_drc_gain + diff;	
}	
ref_drc_gain = drc_gain[ch][sf][band];	
}	
ref_drc_gain = drc_gain[ch][0][band];	
}	
ref_drc_gain = drc_gain[ch][0][0];	
}	
}	
}	
NOTE: nr_drc_bands is defined in clause 4.3.13.3.8, nr_drc_channels is defined in clause 4.3.13.7.1, and nr_drc_subframes is defined in clause 4.3.13.7.2.	

4.2.14.11 dialog_enhancement - Dialog enhancement metadata

Table 76: Syntax of dialog_enhancement()

Syntax	No. of bits
`dialog_enhancement(b_iframe)`	
`{`	
`    if (**b_de_data_present**) {`	1
`        if (b_iframe) {`	
`            **de_config()**;`	
`        } elseif (**de_config_flag**) {`	1
`            **de_config()**;`	
`        }`	
`        **de_data(de_method, de_nr_channels, b_iframe)**;`	
`    }`	
`}`	
NOTE: de_nr_channels is defined in table 165.	

4.2.14.12 de_config - Dialog enhancement configuration

Table 77: Syntax of de_config()

Syntax	No. of bits
`de_config()`	
`{`	
`    **de_method**;`	2
`    **de_max_gain**;`	2
`    **de_channel_config**;`	3
`}`	

4.2.14.13 de_data - Dialog enhancement data

Table 78: Syntax of de_data()

```
Syntax                                                                          No. of bits
de_data(de_method, de_nr_channels, b_iframe)
{
    if (de_nr_channels > 0) {
        if ((de_method == 1 || de_method == 3) && de_nr_channels > 1) {
            if (b_iframe) {
                de_keep_pos_flag = 0;
            }
            else {
                de_keep_pos_flag;                                               1
            }
            if (!de_keep_pos_flag) {
                de_mix_coef1_idx;                                               5
                if (de_nr_channels == 3) {
                    de_mix_coef2_idx;                                           5
                }
            }
        }
        if (b_iframe) {
            de_keep_data_flag = 0;
        }
        else {
            de_keep_data_flag;                                                  1
        }
        if (!de_keep_data_flag) {
            if ((de_method==0 || de_method==2) && de_nr_channels==2) {
                de_ms_proc_flag;                                                1
            }
            else {
                de_ms_proc_flag = 0;
            }
            for (ch = 0; ch < de_nr_channels - de_ms_proc_flag; ch++) {
                if (b_iframe && ch == 0) {
                    de_par[0][0] =
                        de_abs_huffman(de_method %2, de_par_code);              1...x
                    ref_val = de_par[0][0];
                    de_par_prev[0][0] = de_par[0][0];
                    for (band = 1; band < de_nr_bands; band++) {
                        de_par[0][band] = ref_val +
                            de_diff_huffman(de_method % 2, de_par_code);        1...x
                        ref_val = de_par[0][band];
                        de_par_prev[0][band] = de_par[0][band];
                    }
                }
                else {
                    for (band = 0; band < de_nr_bands; band++) {
                        if (b_iframe) {
                            de_par[ch][band] = ref_val +
                                de_diff_huffman(de_method % 2,
                                                de_par_code);                   1...x
                            ref_val = de_par[ch][band];
                        } else {
                            de_par[ch][band] = de_par_prev[ch][band] +
                                de_diff_huffman(de_method % 2,
                                                de_par_code);                   1...x
                        }
                        de_par_prev[ch][band] = de_par[ch][band];
                    }
                }
                ref_val = de_par[ch][0];
            }
            if (de_method >= 2) {
                de_signal_contribution;                                         5
            }
        }
    }
}
```

NOTE 1: de_nr_channels is defined in table 165.

NOTE 2: de_par_prev contains the corresponding parameter indices in the previous frame. If no parameter was defined (i.e. DE inactive or respective channel not used), it shall be assumed 0.

4.2.14.14 umd_payload_config - Universal metadata payload configuration

Table 79: Syntax of umd_payload_config()

```
Syntax                                                      No. of bits
umd_payload_config()
{
    if (b_smpoffst) {                                           1
        smpoffst = variable_bits(11);
    }
    if (b_duration) {                                           1
        duration = variable_bits(11);
    }
    if (b_groupid) {                                            1
        groupid = variable_bits(2);
    }
    if (b_codecdata) {                                          1
        codecdata;                                              8
    }
    if (!b_discard_unknown_payload) {                           1
        if (b_smpoffst == 0) {
            if (b_payload_frame_aligned) {                      1
                b_create_duplicate;                             1
                b_remove_duplicate;                             1
            }
        }
        if (b_smpoffst == 1 || b_payload_frame_aligned == 1) {
            priority;                                           5
            proc_allowed;                                       2
        }
    }
}
```

4.2.14.15 umd_protection - Universal metadata protection data

Table 80: Syntax of umd_protection()

```
Syntax                                                      No. of bits
umd_protection()
{
    protection_length_primary;                                  2
    protection_length_secondary;                                2
    protection_bits_primary;                                8/32/128
    protection_bits_secondary;                             0/8/32/128
}
```

4.3 Description of bitstream elements

A number of bitstream elements have values which may be transmitted, but whose meaning has been reserved. If a decoder receives a bitstream which contains reserved values, the decoder may or may not be able to decode and produce audio. In the description of bitstream elements which have reserved codes, there is an indication of what the decoder can do if the reserved code is received. In some cases, the decoder cannot decode audio. In other cases, the decoder can still decode audio by using a default value for a parameter which was indicated by a reserved code.

In many cases, the following text provides interpretations for the bits transmitted in the bitstream, by formulas or tables. When a bitstream element is referred to later in the document for an element where such an interpretation is given, the reference is understood to refer to the interpreted value, not to the bits.

EXAMPLE 1: When `wait_frames` is referred to in clauses 5 and later, `wait_frames`=4 means that a bit pattern of **011** has been transmitted if the `frame_rate_index` was 10.

If bitstreams elements with the same name and meaning are present in multiple syntactical elements, the description of this element is present only once.

EXAMPLE 2: `max_sfb[0]` is present with the same meaning in `sf_info_lfe()` and `asf_psy_info()`. The element is documented only as part of the description of the `asf_psy_info()` element.

4.3.1 raw_ac4_frame - Raw AC-4 frame

4.3.1.1 fill_area - Fill area - variable number of bits

This variable size field is used to pad the `raw_ac4_frame` to a certain frame size. Padding can occur before the first `ac4_substream_data` or after the last `ac4_substream_data`. This split of the padding bits can be beneficial for editing purposes. The start of the first `ac4_substream_data` is given by the payload base offset.

4.3.1.2 fill_bits - Byte alignment bits - variable number of bits

These bits can be used by an AC-4 encoder to achieve an ABR stream. Decoders should ignore these bits.

4.3.1.3 byte_align - Byte alignment bits - 0 to 7 bits

This bit field of size 0 to 7 bits is used for the byte alignment of the `raw_ac4_frame()` element. Byte alignment is defined relative to the start of the enclosing syntactic element.

4.3.2 variable_bits - Variable bits

The `variable_bits()` method is an extension mechanism for variable bit-length elements to use more bits for the element than the minimum element length. This method enables efficient coding of small field values with extensibility to be able to express arbitrarily large field values. Field values are split into one or more groups of *n_bits* bits, with each group followed by the 1-bit `b_read_more` field. At a minimum, coding of the value requires *n_bits* + 1 bit to be transmitted. All fields coded using `variable_bits()` shall be interpreted as unsigned integers.

If value to be encoded is between 0 and 2^{n_bits} - 1, then only one group of *n_bits* bits is required. The field value is in this case equal to the value to be encoded, and the `b_read_more` field shall be set to '0'. If the value to be encoded is larger than 2^{n_bits} - 1, the `b_read_more` field is set to '1', and a second group of *n_bits* follows in the bitstream, followed by another Boolean. If the value to be encoded falls within the range $[2^{n\_bits}, 2^{n\_bits} + 2^{2n\_bits} - 1]$, then the second Boolean is set to FALSE. Otherwise, it is set to TRUE and a third group of *n_bits* follows. This process continues until a word of sufficient length is transmitted.

4.3.2.1 read - Read bits - n_bits bits

The size of this bit field is specified by the parameter *n_bits*. The bit field value is used to determine the value of `variable_bits` as specified by the syntax box in table 3.

4.3.2.2 b_read_more - Read more flag - 1 bit

This bit indicates the presence of additional `read` bits.

4.3.3 AC-4 frame info

4.3.3.1 Purpose

The `ac4_toc` contains the configuration for an AC-4 decoder. An AC-4 decoder shall only be configured by the `ac4_toc()` (as opposed to information available on system level).

4.3.3.2 ac4_toc - AC-4 table of contents

4.3.3.2.1 bitstream_version - Bitstream version - 2 bits/variable_bits(2)

This 2-bit code which is extendable by `variable_bits()` indicates the bitstream version of the AC-4 frame. Only bitstreams with a bitstream version of 0 and 1 are decodable using this version of the AC-4 specification. Bitstreams with a bitstream version of 2 or higher are not decodable according to the present document. Encoders conforming to the present document shall use a value of 0 for the `bitstream_version`.

4.3.3.2.2 sequence_counter - Sequence counter - 10 bits

This 10-bit code contains the sequence counter value for the current AC-4 frame. Using the sequence counter, a decoder may detect uncontrolled changes in the stream source, such as might occur in splicing operation.

- "Normal operation"
  A decoder may continue its usual processing under the following three conditions:

  - Counter increase: `sequence_counter` == *sequence_counter_prev* + 1
  - Counter wrap around: `sequence_counter == 1` AND *sequence_counter_prev* == 1 020
  - Splice in previous frame: `sequence_counter` != 0 AND *sequence_counter_prev* == 0

  Here, *sequence_counter_prev* indicates the `sequence_counter` value from the previous AC-4 frame.

- "Source change detected"
  If none of the conditions above apply, the decoder detects a source change in the current frame. Although the present document does not define exact decoder behavior at source changes, a decoding system should provide continuity of audio experience in the period between recognition of a source change and the next I-Frame.

A splicing device should overwrite the `sequence_counter` value of the first frame after a splice with 0.

An encoder conforming to the present document shall not write `sequence_counter` values exceeding 1 020.

4.3.3.2.3 b_wait_frames - 1 bit

This bit, if set, indicates that `wait_frames` information shall follow in the bitstream. If not set, `wait_frames` is undefined.

4.3.3.2.4 wait_frames - 3 bits

For ABR streams, this 3-bit field indicates when decoding of this frame can commence. After having received the frame carrying `wait_frames`, a decoding system should wait for `wait_frames` frames before the decoded frame is placed into the output buffer, unless other timing information exists. Please refer to table 81 for the interpretation of the bits.

NOTE 1: This requirement makes sure that if the bitstream is received over a constant-rate channel, the decoder input buffer will not run dry before the next output frame is due.

For CBR streams where every frame carries the same number of bits, the decoder need not wait at all. For VBR streams, the delay is unknown.

NOTE 2: VBR streams are not well suited for transmission over a constant rate channel.

When the delay is unknown, the decoder should assume audio continuity, unless a source change was detected. In that case, the maximum delay of 5 (10) frames should be assumed.

Table 81: Decoding delay

wait_frames	frames to wait before output	
	frame_rate_index = [0...9, 13]	frame_rate_index = [10,11,12]
0	stream is CBR (=0)	
1	0	0
2	1	2
3	2	4
4	3	6
5	4	8
6	5	10
7	stream is VBR; no statement on wait frames can be made	

4.3.3.2.5 fs_index - Sampling frequency index - 1 bit

This bit indicates the base sampling frequency, indicated by the variable *base_samp_freq*, as shown in table 82.

Table 82: Base sampling frequency

fs_index	***base_samp_freq*** **(kHz)**
0	44,1
1	48

4.3.3.2.6 frame_rate_index - Frame rate - 4 bits

The 4-bit `frame_rate_index` indicates the rate of frames *frame_rate* in units of 1/s, the value *frame_length* indicating the length of the frame in samples (dependent on the external sample rate) and the decoder resampling ratio as shown in tables 83 and 84. The base frame length value *frame_len_base* is defined as the frame length at external sample rate 48 kHz.

Table 83: frame_rate_index for 48 kHz, 96 kHz and 192 kHz

frame_rate_index	*frame_rate* (fps)	Internal frame length (samples) *frame_length*			Decoder resampling ratio
		for external 48 kHz = *frame_len_base*	for external 96 kHz	for external 192 kHz	
0	23,976	1 920	3 840	7 680	$1001/1000 \times 25/24$
1	24	1 920	3 840	7 680	$25/24$
2	25	2 048	4 096	8 192	$15/16$
3	29,97	1 536	3 072	6 144	$1001/1000 \times 25/24$
4	30	1 536	3 072	6 144	$25/24$
5	47,95	960	1 920	3 840	$1001/1000 \times 25/24$
6	48	960	1 920	3 840	$25/24$
7	50	1 024	2 048	4 096	$15/16$
8	59,94	768	1 536	3 072	$1001/1000 \times 25/24$
9	60	768	1 536	3 072	$25/24$
10	100	512	1 024	2 048	$15/16$
11	119,88	384	768	1 536	$1001/1000 \times 25/24$
12	120	384	768	1 536	$25/24$
13	(23,44)	2 048	4 096	8 192	
14: reserved					
15: reserved					

Table 84: frame_rate_index for 44,1 kHz

frame_rate_index	*frame_rate* (fps)	Internal frame length (samples) *frame_length* for external 44,1 kHz = *frame_len_base*	Decoder resampling ratio
0…12: reserved			
13	$11025/512$	2 048	1
14: reserved			
15: reserved			

4.3.3.2.7 b_iframe_global - Global I-frame flag - 1 bit

This bit, if set, indicates that all substreams in all presentations have `b_iframe` = 1. In case `frame_rate_factor`!=1, the first `b_iframe` of a series of 2 or 4 substreams has to be 1 (true) to fulfill this.

4.3.3.2.8 b_single_presentation - Single presentation flag - 1 bit

This bit, if set, indicates the presence of a single presentation. If not set, `b_more_presentations` and additional `variable_bits` are used to derive the number of presentations contained in the AC-4 frame.

4.3.3.2.9 b_more_presentations - More presentations flag - 1 bit

This bit, if set, indicates that the number of presentations, *n_presentations*, is derived from additional `variable_bits`. If not set, no presentation is contained in the AC-4 frame.

4.3.3.2.10 b_payload_base - Payload base flag - 1 bit

This bit, if set, indicates that payload base offset information shall follow in the bitstream. If not set, the payload base offset is 0 and the first `ac4_substream_data` shall immediately follow the `ac4_toc`.

4.3.3.2.11 payload_base_minus1 - Payload base offset minus 1 - 5 bits

This 5-bit code indicates the start of the `ac4_substream_data` for substream 0 relative to the end of the byte-aligned `ac4_toc` element in bytes minus 1. An extension via `variable_bits` is done for a value of 0x1f.

4.3.3.2.12 byte_align - Byte alignment bits - 0 to 7 bits

This bit field of size 0 to 7 bits is used for the byte alignment of the `ac4_toc()` element.

4.3.3.3 ac4_presentation_info - AC-4 presentation information

4.3.3.3.1 b_single_substream - Single substream flag - 1 bit

This bit indicates that the presentation contains a single substream.

4.3.3.3.2 b_belongs_to_presentation_group - Presentation group assignment flag - 1 bit

This bit, if set, indicates that the presentation belongs to a presentation group. The presentation group identifier *presentation_group* is derived from additional `variable_bits`.

4.3.3.3.3 b_hsf_ext - high sampling frequency extension flag - 1 bit

This bit, if set, indicates that additional spectral data is available which can be used for decoding a presentation into 96 kHz or 192 kHz.

4.3.3.3.4 presentation_config - Presentation configuration - 3 bits/variable_bits(2)

This 3-bit code which is extendable by `variable_bits()`indicates the presentation configuration as shown in table 85.

Table 85: presentation_config

presentation_config	Presentation configuration
0	Dry Main + Dialog
1	Main + DE
2	Main + Associate
3	Dry Main + Dialog + Associate
4	Main + DE + Associate
5	HSF extended
≥ 6	Reserved

4.3.3.3.5 b_pre_virtualized - Pre-virtualized flag - 1 bit

This bit, if set, indicates that the audio content in the current presentation was pre-rendered by a headphone virtualizer. In this case, the receiving device/service may chose to turn off device-side content post-processing.

4.3.3.3.6 b_add_umd_substreams - Additional universal metadata substreams flag - 1 bit

This bit indicates the presence of additional universal metadata substreams.

4.3.3.3.7 n_add_umd_substreams - Number of additional universal metadata substreams - 2 bits/variable_bits(2)

This 2-bit code which is extendable by `variable_bits()` indicates the number of additional metadata substreams.

4.3.3.4 presentation_version - Presentation version information

4.3.3.4.1 b_tmp - Temporary flag - 1 bit

This bit, which might be present multiple times, is used to signal the version of the presentation. An encoder conforming to the present document shall write a presentation version value of 0. A decoder implemented in accordance to the present document shall skip the `ac4_presentation_info` if the version of the presentation is not 0.

4.3.3.5 frame_rate_multiply_info - Frame rate multiplier information

4.3.3.5.1 b_multiplier - Multiplier flag - 1 bit

This bit, if set, indicates a *frame_rate_factor* value different to 1. If not set or if not present, the *frame_rate_factor* value is 1. See clause 4.3.3.5.3.

4.3.3.5.2 multiplier_bit - Multiplier bit - 1 bit

This bit, if set, indicates a *frame_rate_factor* value of 4 and if not set, a *frame_rate_factor* value of 2. See clause 4.3.3.5.3.

4.3.3.5.3 frame_rate_factor - Frame rate factor - via table

Dependent on the *frame_rate_factor*, as specified in table 86, an `ac4_substream_info()` element refers to 1, 2 or 4 substreams. Each of those substreams shall be decoded consecutively.

Table 86: frame_rate_factor

frame_rate_index	b_multiplier	multiplier_bit	*frame_rate_factor*
2, 3, 4	0	X	1
	1	0	2
	1	1	4
0, 1, 7, 8, 9	0	X	1
	1	X	2
5, 6, 10, 11, 12, 13	X	X	1

4.3.3.6 umd_info - Universal metadata information

4.3.3.6.1 umd_version - UMD syntax version - 2 bits/variable_bits(2)

The 2-bit `umd_version` field which is extendable by `variable_bits()` indicates the syntax version that the UMD substream conforms with. For substreams that conform to the syntax defined in this version of the AC-4 specification, the `umd_version` field shall be set to '0'.

4.3.3.6.2 key_id - cryptographic key identification - 3 bits/variable_bits(3)

The value of the 3-bit `key_id` field which is extendable by `variable_bits()` identifies the cryptographic key used to calculate the value of the `protection_bits_primary` and `protection_bits_secondary` fields of the `umd_protection` element. The values of `key_id` are implementation dependent and are not defined in the present document.

4.3.3.6.3 b_umd_payloads_substream_info - UMD payloads substream info flag - 1 bit

This bit indicates the presence of a `umd_payloads_substream_info()` element.

4.3.3.7 ac4_substream_info - AC-4 substream information

4.3.3.7.1 channel_mode - Channel mode - 1, 2, 4 or 7 bits/variable_bits(2)

This variable length field (1, 2, 4 or 7 bits plus `variable_bits()`) indicates the channel mode and the variable *ch_mode* as shown in table 87.

Table 87: channel_mode

Value of channel_mode	Channel mode		*ch_mode*
0	Mono		0
10	Stereo		1
1100	3.0		2
1101	5.0		3
1110	5.1		4
1111000	7.0: 3/4/0	(L,C,R,Ls,Rs,Lrs,Rrs)	5
1111001	7.1: 3/4/0.1	(L,C,R,Ls,Rs,Lrs,Rrs,LFE)	6
1111010	7.0: 5/2/0	(L,C,R,Lw,Rw,Ls,Rs)	7
1111011	7.1: 5/2/0.1	(L,C,R,Lw,Rw,Ls,Rs,LFE)	8
1111100	7.0: 3/2/2	(L,C,R,Ls,Rs,Vhl,Vhr)	9
1111101	7.1: 3/2/2.1	(L,C,R,Ls,Rs,Vhl,Vhr,LFE)	10
1111110	Reserved		11
≥ 1111111	Reserved		12+

The "3.0" channel mode shall only be used in the following context:

- for coding of the enhancement signal for the dialog enhancement (DE) feature
- for coding of the dialog in a dry main + dialog presentation

4.3.3.7.2 b_sf_multiplier - Sampling frequency multiplier flag - 1 bit

This bit, if set, indicates that the sampling frequency of the AC-4 substream is a multiple of the base sampling frequency. If not set, the sampling frequency of the AC-4 substream is identical to the base sampling frequency.

NOTE: This bit is only available if the base sampling frequency is 48 kHz.

4.3.3.7.3 sf_multiplier - Sampling frequency multiplier bit - 1 bit

This bit indicates the sampling frequency multiplier. Since this bit is only available if the base sampling frequency is 48 kHz, the sampling frequency of the AC-4 substream is given by table 88.

Table 88: AC-4 substream sampling frequency for a base sampling frequency of 48 kHz

b_sf_multiplier	sf_multiplier	sampling frequency
0		48 kHz
1	0	96 kHz
	1	192 kHz

4.3.3.7.4 b_bitrate_info - Bitrate presence flag - 1 bit

This bit indicates whether a bitrate indicator is specified.

4.3.3.7.5 bitrate_indicator - Bitrate indicator - 3 bits or 5 bits

This 3 or 5-bit field indicates the upper average bitrate limit per channel in the substream and the variable *brate_ind* as shown in table 89.

Table 89: bitrate_indicator

Value of bitrate_indicator	Bitrate (kbit/s)	*brate_ind*
000	16	0
010	20	1
100	24	2
110	28	3
00100	32	4
00101	40	5
00110	48	6
00111	56	7
01100	64	8
01101	80	9
01110	96	10
01111	112	11
1X1XX (8 further values)	unlimited	12…19

4.3.3.7.6 add_ch_base - Additional channels coupling base - 1 bit

This 1-bit code indicates whether the A-CPL coding of the additional channels (Lw/Rw or Vhl/Vhr depending on the `channel_mode`) is based on the L/R pair (`add_ch_base`=0) or on the Ls/Rs pair (`add_ch_base`=1). See clause 5.7.7.6.3 for details.

4.3.3.7.7 b_content_type - Content type presence flag - 1 bit

This bit indicates whether a content type is specified.

4.3.3.7.8 b_iframe - I-frame flag - 1 bit

This bit indicates for each of the *frame_rate_factor* substreams whether the contained frame is an I-frame.

4.3.3.7.9 substream_index - Substream index - 2 bits/variable_bits(2)

This 2-bit code which is extendable by `variable_bits()` indicates the substream index the `ac4_substream_info()` refers to. The substream index is used as index in the `substream_index_table()` to get the offset to the `ac4_substream()`.In case *frame_rate_factor* is not 1, `substream_index` refers to the first of *frame_rate_factor* substreams in the `substream_index_table()`.

NOTE: Each of the *frame_rate_factor* substreams has an entry in the `substream_index_table()`. *frame_rate_factor* consecutive entries are stored in the `substream_index_table()` starting at `substream_index`.

4.3.3.8 content_type - Content type

4.3.3.8.1 content_classifier - Content classifier - 3 bits

This 3-bit code classifies the content as shown in table 90.

Table 90: content_classifier

Value of content_classifier	Content classification
000	Main audio service: complete main (CM)
001	Main audio service: music and effects (ME)
010	Associated service: visually impaired (VI)
011	Any associated service: hearing impaired (HI)
100	Associated service: dialog (D)
101	Any associated service: commentary (C)
110	Associated service: emergency (E)
111	Associated service: voice over (VO)

4.3.3.8.2 b_language_indicator - Programme language indicator flag - 1 bit

This bit indicates that programme language indication data is available.

4.3.3.8.3 b_serialized_language_tag - Serialized language tag flag - 1 bit

The 1-bit `b_serialized_language_tag` field is used to indicate whether the language tag is delivered using multiple payloads in a sequence of AC-4 frames, or whether the complete language tag is delivered in the current payload. A value of '1' indicates that the language tag is delivered using multiple payloads delivered in a sequence of AC-4 frames, and information describing the sequence of frames shall follow in the payload. A value of '0' indicates that the complete language tag is stored in the current payload. The value of `b_serialized_language_tag` shall be set to '1' in all programme language payloads that follow the programme language payload in which both the value of the `b_serialized_language_tag` field and the value of the `b_start_tag` field is set to '1', up to and including the payload that contains the final byte of the language tag.

4.3.3.8.4 b_start_tag - Language tag start flag - 1 bit

The 1-bit `b_start_tag` field is used to indicate the start of a multi-frame sequence of language tag data. A value of '1' indicates that this payload contains the first chunk of the language tag, and decoders shall start decoding the language tag beginning with the data in this payload. A value of '0' indicates that this payload does not contain the start of the multi-frame sequence.

4.3.3.8.5 language_tag_chunk - Language tag chunk - 16 bits

The 16-bit `language_tag_chunk` field shall contain a two byte section of a language tag that conforms to the syntax and semantics defined in IETF BCP 47 [4]. The most significant byte of the language tag shall be stored in the most significant byte of the `language_tag_chunk` in a programme language payload that has a `b_serialized_language_tag` field value of '1'. Subsequent bytes of the language tag shall be stored in the second byte of the current `language_tag_chunk` and in the `language_tag_chunk` of each of the subsequent programme language payloads required to deliver the complete language tag. If the length of the language tag does not correspond to an even number of bytes, the value of the extra byte that follows the end of the language tag shall be set to 0x0.

4.3.3.8.6 n_language_tag_bytes - Number of language tag bytes - 6 bits

The 6-bit `n_language_tag_bytes` field indicates the total length, in bytes, of the language tag. Values of 0 and 1 and values of 43 to 63 are reserved.

4.3.3.8.7 language_tag_bytes - Language tag bytes - 8 bits

The sequence of `language_tag_bytes` shall contain a language tag that conforms to the syntax and semantics defined in IETF BCP 47 [4]. The minimum sequence length shall be two bytes, supporting a two-character language tag as specified in ISO 639 [5]. The maximum supported length of the language tag shall be 336 bits or 42 bytes.

4.3.3.9 presentation_config_ext_info - Presentation configuration extended information

4.3.3.9.1 n_skip_bytes - Number of bytes to skip - 5 bits

This 5-bit code specifies the number of bytes to skip.

4.3.3.9.2 b_more_skip_bytes - More bytes to skip flag - 1 bit

This bit indicates that the number of bytes to skip, *n_skip_bytes*, is extended by the use of `variable_bits()`.

4.3.3.9.3 reserved - Reserved - 8 bits

This 8-bit field is reserved for future use and the content shall be skipped by an AC-4 decoder conforming to the present document.

4.3.3.10 ac4_hsf_ext_substream_info - AC-4 HSF extension substream information

4.3.3.10.1 substream_index - Substream index - 2 bits/variable_bits(2)

This 2-bit code which is extendable by `variable_bits()` indicates the substream index the `ac4_hsf_ext_substream_info` refers to. The substream index is used as index in the `substream_index_table()` to get the offset to the `ac4_substream()` for the high sampling frequency extension substream.

4.3.3.11 umd_payloads_substream_info - Universal metadata payloads substream information

4.3.3.11.1 substream_index - Substream index - 2 bits/variable_bits(2)

This 2-bit code which is extendable by `variable_bits()` indicates the substream index the `umd_payloads_substream_info` refers to. The substream index is used as index in the `substream_index_table()` to get the offset to the `umd_payloads_substream()`.

4.3.3.12 substream_index_table - Substream index table

4.3.3.12.1 n_substreams - Number of substreams - 2 bits/variable_bits(2)

This 2-bit code which is extendable by `variable_bits()` indicates the number of substreams available in the `substream_index_table`.

4.3.3.12.2 b_size_present - Size present flag - 1 bit

This bit indicates the presence of the substream size.

4.3.3.12.3 b_more_bits - More bits flag - 1 bit

This bit indicates that additional `variable_bits()` are used to determine the substream size.

4.3.3.12.4 substream_size - Substream size - 10 bits

This 10-bit code which is extendable by variable_bits() if b_more_bits is set indicates the substream size in bytes for the substream with index *s*.

To get the offset for the nth substream relative to the end of the ac4_toc, the substream sizes of the substreams with a substream index less than n need to be accumulated as shown in the following pseudocode:

Pseudocode

```
// get the substream offset for substream with substream index n

substream_n_offset = payload_base;

for (s = 0; s < n; s++) {
    substream_n_offset += substream_size[s];
}
```

4.3.4 ac4_substream - AC-4 substream

4.3.4.1 audio_size_value - Audio size value - 15 bits

This 15-bit field indicates, together with potential variable_bits, the *audio_size* in bytes. The *audio_size* is the size of the ac4_substream element including any following fill_bits and byte_alignment but not including the size of the metadata element.

NOTE: This enables decoders to directly access metadata without the need to parse audio data.

4.3.4.2 b_more_bits - More bits flag - 1 bit

This bit indicates the presence of additional variable_bits to be used for the *audio_size* determination.

4.3.4.3 byte_align - Byte alignment bits - 0 to 7 bits

This bit field of size 0 to 7 bits is used for the byte alignment within the ac4_substream() element. An encoder shall use this field to pad the length of an ac4_substream to an integer number of bytes.

4.3.5 Channel elements

This clause describes the bitstream elements which can be found in single_channel_element, mono_data, channel_pair_element, stereo_data, 5_X_channel_element and 7_X_channel_element.

4.3.5.1 mono_codec_mode - Mono codec mode - 1 bit

This bit indicates the mono codec mode as shown in table 91.

Table 91: Mono codec mode

mono_codec_mode	Mono codec mode
0 = SIMPLE	Simple
1 = ASPX	A-SPX

4.3.5.2 spec_frontend - Spectral frontend selection - 1 bit

This bit indicates the used spectral frontend as shown in table 92.

Table 92: Spectral frontend selection

spec_frontend	Spectral frontend
0 = ASF	Audio Spectral Frontend
1 = SSF	Speech Spectral Frontend

The bitstream elements `spec_frontend_m`, `spec_frontend_s`, `spec_frontend_l` and `spec_frontend_r` are also spectral frontend selection flags with the same meaning. The extension indicates the channel type: m=mid, s=side, l=left and r=right.

4.3.5.3 stereo_codec_mode - Stereo codec mode - 2 bits

This 2-bit code indicates the stereo codec mode as shown in table 93.

Table 93: stereo_codec_mode

stereo_codec_mode	Stereo codec mode
0 = SIMPLE	Simple
1 = ASPX	A-SPX
2 = ASPX_ACPL_1	A-SPX, A-CPL mode 1
3 = ASPX_ACPL_2	A-SPX, A-CPL mode 2

4.3.5.4 3_0_codec_mode - 3.0 codec mode - 1 bit

This bit indicates the 3.0 codec mode as shown in table 94.

Table 94: 3_0_codec_mode

mono_codec_mode	Mono codec mode
0 = SIMPLE	Simple
1 = ASPX	A-SPX

4.3.5.5 3_0_coding_config - 3.0 coding configuration - 1 bit

This 1-bit code indicates the coding configuration for the 3.0 channel mode.

4.3.5.6 5_X_codec_mode - 5.X codec mode - 3 bits

This 3-bit code indicates the 5.X codec mode as shown in table 95.

Table 95: 5_X_codec_mode

5_X_codec_mode	5.X codec mode
0 = SIMPLE	Simple
1 = ASPX	A-SPX
2 = ASPX_ACPL_1	A-SPX, A-CPL mode 1
3 = ASPX_ACPL_2	A-SPX, A-CPL mode 2
4 = ASPX_ACPL_3	A-SPX, A-CPL mode 3
5…7	Reserved

4.3.5.7 7_X_codec_mode - 7.X codec mode - 2 bits

This 2-bit code indicates the 7.X codec mode as shown in table 96.

Table 96: 7_X_codec_mode

7_X_codec_mode	7.X codec mode
0 = SIMPLE	Simple
1 = ASPX	A-SPX
2 = ASPX_ACPL_1	A-SPX, A-CPL mode 1
3 = ASPX_ACPL_2	A-SPX, A-CPL mode 2

4.3.5.8 coding_config - Coding configuration - 1 or 2 bits

This 1-bit or 2-bit code, depending on the context, indicates the coding configuration for the 5.X and the 7.X channel modes.

4.3.5.9 2ch_mode - Channel coupling mode - 2 bits

This 2-bit code indicates the way channel pairs are coupled into the output channels. See clauses 5.3.4.3 and 5.3.4.4 for details.

4.3.5.10 b_enable_mdct_stereo_proc - Enable MDCT stereo processing flag - 1 bit

This bit indicates the presence of MDCT domain stereo processing data stored in a `chparam_info` element. The `sf_info` element, which is also enabled by this bit, holds ASF information which is valid for the decoding of both `sf_data` elements that follow.

4.3.5.11 chel_matsel - Matrix selection code - 4 bits

This 4-bit code selects the matrix for multichannel coupling.

4.3.5.12 b_use_sap_add_ch - Use SAP for additional channels flag - 1 bit

This bit, if set, indicates that the SAP tool shall be used for the additional channels and that SAP data stored in two `chparam_info()` elements shall follow in the bitstream.

4.3.5.13 max_sfb_master - max_sfb indication for related channels - n_side_bits bits

This element of size *n_side_bits* indicates the *max_sfb* value to be used when decoding the two `sf_data()` elements which follow the `max_sfb_master` element and the two `chparam_info()` elements. The *max_sfb* value for a block is determined by one entry in one of the tables B.8 to B.19. The table to use depends on the largest signalled transform length which is also used for the *n_side_bits* determination as indicated in the notes to tables 25 and 33. If the transform length for a block is equal to the largest signalled transform length, the `max_sfb_master` value maps directly to the *max_sfb* value. Otherwise, the *max_sfb* value is given by the *n_sfb_side* value for the transform length of the block and the `max_sfb_master` value.

4.3.6 Audio spectral frontend

4.3.6.1 asf_transform_info - ASF transform info

4.3.6.1.1 b_long_frame - Long frame flag - 1 bit

This bit, if set, indicates that the audio frame is a long frame. A long frame contains just one (full) block which needs to be transformed. The transform length for a long frame depends on the sampling frequency and the *frame_len_base* value and is given by table 97.

Table 97: Transform length for long frames

frame_len_base	**Sampling frequency**		
	44,1 kHz, 48 kHz	**96 kHz**	**192 kHz**
2 048	2 048	4 096	8 192
1 920	1 920	3 840	7 680
1 536	1 536	3 072	6 144
1 024	1 024	2 048	4 096
960	960	1 920	3 840
768	768	1 536	3 072
512	512	1 024	2 048
384	384	768	1 536

4.3.6.1.2 transf_length[i] - Transform length index i - 2 bits

An audio frame contains several partial blocks if the audio frame is not a long frame. This 2-bit code indicates the transform length for partial blocks in case the *frame_len_base* value is greater or equal to 1 536. All partial blocks which make up the first half of a corresponding full block use index 0 and the others use index 1. Table 98 through table 100 show the transform length values for partial blocks depending on the sampling frequency and the *frame_length* value.

NOTE: The `transf_length[i]` values are only available if `b_long_frame` is not set. If `b_long_frame` is set, the transform length for the audio block equals to the *frame_length* value.

Table 98: Transform length for non-long frames, frame_len_base ≥ 1 536 and 44,1 kHz or 48 kHz

frame_length	**transf_length[i]**			
	0	**1**	**2**	**3**
2 048	128	256	512	1 024
1 920	120	240	480	960
1 536	96	192	384	768

Table 99: Transform length for non-long frames, frame_len_base ≥ 1 536 and 96 kHz

frame_length	**transf_length[i]**			
	0	**1**	**2**	**3**
4 096	256	512	1 024	2 048
3 840	240	480	960	1 920
3 072	192	384	768	1 536

Table 100: Transform length for non-long frames, frame_len_base ≥ 1 536 and 192 kHz

frame_length	**transf_length[i]**			
	0	**1**	**2**	**3**
8 192	512	1 024	2 048	4 096
7 680	480	960	1 920	3 840
6 144	384	768	1 536	3 072

4.3.6.1.3 transf_length - Transform length - 2 bits

This 2-bit code indicates the transform length for the audio block or the partial blocks in case the *frame_len_base* value is less than 1 536. Table 101 through table 103 show the transform length values depending on the sampling frequency and the *frame_length* value.

Table 101: Transform length for frame_len_base < 1 536 and 44,1 kHz or 48 kHz

frame_length	**transf_length**			
	0	**1**	**2**	**3**
1 024	128	256	512	1 024
960	120	240	480	960
768	96	192	384	768
512	128	256	512	×
384	96	192	384	×

Table 102: Transform length for frame_len_base < 1 536 and 96 kHz

frame_length	**transf_length**			
	0	**1**	**2**	**3**
2 048	256	512	1 024	2 048
1 920	240	480	960	1 920
1 536	192	384	768	1 536
1 024	256	512	1 024	×
768	192	384	768	×

Table 103: Transform length for frame_len_base < 1 536 and 192 kHz

frame_length	**transf_length**			
	0	**1**	**2**	**3**
4 096	512	1 024	2 048	4 096
3 840	480	960	1 920	3 840
3 072	384	768	1 536	3 072
2 048	512	1 024	×	×
1 536	384	768	×	×

4.3.6.2 asf_psy_info - ASF psy info

4.3.6.2.1 n_msfb_bits - Number of maxsfb bits - via table

The number of bits used for the `max_sfb[i]` and `max_sfb_side[i]` bitstream elements depends on the transform length and is specified as variable *n_msfb_bits* in table 104 through table 106. The variable *n_side_bits*, which is also specified in table 104, gives the number of bits used for the `max_sfb_side[i]` bitstream element in case *b_side_limited* = 1 or for the `max_sfb_master` element in case this is present. The number of bits for `max_sfb[0]` in the `sf_info_lfe()` element is specified as variable *n_msfbl_bits* in table 104 through table 106. In case that a high sampling frequency extension is not present in the presentation, `b_hsf_ext` = 0, only table 104 applies. If `b_hsf_ext` = 1, the transform length and the sampling frequency related to the high sampling frequency shall be used and either table 105 or 106 does apply.

Table 104: n_msfb_bits and n_side_bits for 44,1 kHz or 48 kHz

Transform length	*n_msfb_bits*	*n_side_bits*	*n_msfbl_bits*
2 048	6	5	3
1 920	6	5	3
1 536	6	5	3
1 024	6	5	2
960	6	5	2
768	6	5	2
512	6	5	2
480	6	5	n/a
384	6	4	2
256	5	4	n/a
240	5	4	n/a
192	5	3	n/a
128	4	3	n/a
120	4	3	n/a
96	4	3	n/a

Table 105: n_msfb_bits for 96 kHz

Transform length	*n_msfb_bits*	*n_msfbl_bits*
4 096	7	3
3 840	7	3
3 072	7	3
2 048	6	2
1 920	6	2
1 536	6	2
1 024	6	2
960	6	n/a
768	6	2
512	5	n/a
480	5	n/a
384	5	n/a
256	5	n/a
240	5	n/a
192	5	n/a

Table 106: n_msfb_bits for 192 kHz

Transform length	*n_msfb_bits*	*n_msfbl_bits*
8 192	7	3
7 680	7	3
6 144	7	3
4 096	7	2
3 840	7	2
3 072	6	2
2 048	6	2
1 920	6	n/a
1 536	6	2
1 024	6	n/a
960	6	n/a
768	5	n/a
512	5	n/a
480	5	n/a
384	5	n/a

4.3.6.2.2 max_sfb[i] - Number of transmitted scale factor bands for index i - n_msfb_bits bits

Depending on the value of the transform length, this is a 4-bit, 5-bit, 6-bit or 7-bit integer value indicating the number of transmitted scale factor bands. The number of bits used for `max_sfb[i]` is given in clause 4.3.6.2.1. The value will be less or equal to *num_sfb*. `max_sfb[0]` is used for a full block or all partial blocks related to `transf_length[0]` and `max_sfb[1]` is used for all partial blocks related to `transf_length[1]`.

4.3.6.2.3 max_sfb_side[i] - Number of transmitted scale factor bands for side channel and index i - 3 to 7 bits

Like `max_sfb[i]` for the first channel, this integer value indicates the number of transmitted scale factor bands for the second (i.e. side) channel. This value is typically only transmitted when the second channel has a different number of scale factor bands than the first channel. If no `max_sfb_side[i]` is transmitted, `max_sfb[i]` is also used for the second channel.

4.3.6.2.4 n_grp_bits - Number of grouping bits - via table

The number of grouping bits is 0 if `b_long_frame`=1 and the *frame_len_base* value is greater or equal to 1 536. If `b_long_frame`=0 and the *frame_len_base* value is greater or equal to 1 536, the number of grouping bits *n_grp_bits* depends on `transf_length[0]`, `transf_length[1]` and the *frame_len_base* value and is given by table 107. If the *frame_len_base* value is less than 1 536, the number of grouping bits *n_grp_bits* depends on `transf_length` and the *frame_len_base* value and is given by table 108.

Table 107: n_grp_bits for frame_len_base ≥ 1 536 and b_long_frame=0

frame_len_base	**transf_length[0]**	**transf_length[1]**	***n_grp_bits***
2 048, 1 920, 1 536	0	0	15
	0	1	10
	0	2	8
	0	3	7
	1	0	10
	1	1	7
	1	2	4
	1	3	3
	2	0	8
	2	1	4
	2	2	3
	2	3	1
	3	0	7
	3	1	3
	3	2	1
	3	3	1

Table 108: n_grp_bits for frame_len_base < 1 536

frame_len_base	**transf_length**	***n_grp_bits***
1 024, 960, 768	0	7
	1	3
	2	1
	3	0
512, 384	0	3
	1	1
	2	0

4.3.6.2.5 scale_factor_grouping_bit - Scale factor grouping bit - 1 bit

This 1-bit field, which is present *n_grp_bits* times, is used to indicate the scale factor grouping. The conversion of the *scale_factor_grouping[i]* array into `asf_psy_info` helper elements is specified in clause 4.3.6.2.6.

4.3.6.2.6 asf_psy_info helper elements

The following helper elements are derived from the `asf_psy_info` bitstream elements, especially from the *scale_factor_grouping[i]* array:

num_windows	number of (transform) windows within the current audio frame. Possible values are: 1, 2, 3, 4, 5, 6, 8, 9, 10, 12 and 16.
num_window_groups	number of window groups
window_to_group[w]	array of size *num_windows* which maps a window number *w* into a group number *g*.
num_win_in_group[g]	vector indicating the number of windows in group *g*
sect_sfb_offset[g][sfb]	array indicating the section scale factor band offset for the scale factor band *sfb* within group *g*.

NOTE: If `transf_length[1]` = `transf_length[0]`, *num_windows* = 2, 4, 8 or 16 and if `transf_length[1]` ≠ `transf_length[0]`, *num_windows* = 3, 5, 6, 9, 10 or 12. *num_windows* = 1 if `b_long_frame` = 1.

Pseudocode

```
// derive num_windows, num_window_groups and window_to_group[w]

num_windows = 1;
num_window_groups = 1;
window_to_group[0] = 0;

if (b_long_frame == 0) {
    num_windows = n_grp_bits + 1;
    if (b_different_framing) {
        /* nr of windows in first half of frame */
        num_windows_0 = (1 << (3-transf_length[0]));
        for (i = n_grp_bits; i >= num_windows_0; i--) {
            /* shift grouping bits of 2nd half by 1 */
            scale_factor_grouping[i] = scale_factor_grouping[i-1];
        }
        /* no grouping for unequal transform lengths */
        scale_factor_grouping[num_windows_0-1] = 0;
        num_windows++;
    }
    for (i = 0; i < num_windows - 1; i++) {
        if (scale_factor_grouping[i] == 0) {
            num_window_groups += 1;
        }
        window_to_group[i + 1] = num_window_groups - 1;
    }
}
```

Pseudocode

```
// derive num_win_in_group[g] and sect_sfb_offset[g][sfb]

group_offset = 0;
for (g = 0; g < num_window_groups; g++) {
    num_win_in_group[g] = 0;
    for (win = 0; win < num_windows; win++) {
        if (window_to_group[win] == g) {
            num_win_in_group[g] += 1;
        }
    }
    max_sfb = get_max_sfb(g);
    for (sfb = 0; sfb < max_sfb; sfb++) {
        // use the sfb_offset[sfb] table from Annex B which matches the used sampling
        // frequency and the transform length related to the windows within group g
        sect_sfb_offset[g][sfb] = group_offset + sfb_offset[sfb] * num_win_in_group[g];
    }
    group_offset += sfb_offset[max_sfb] * num_win_in_group[g];
}
```

4.3.6.3 asf_section_data - ASF section data

4.3.6.3.1 sect_cb[g][i] - Section codebook - 4 bits

This 4-bit field indicates the Huffman codebook to be used for section i in group g. The values 12 to 15 do not indicate a Huffman codebook and shall not be used.

4.3.6.3.2 sect_len_incr - Section length increment - 3 or 5 bits

Depending on the transform length to be used in group g, this is a 3 or 5-bit field used for determining the section length of section i in group g.

4.3.6.4 asf_spectral_data - ASF spectral data

4.3.6.4.1 asf_qspec_hcw - Huffman coded quantized spectral lines - variable bits

This variable length field holds Huffman coded quantized spectral lines. Either two or four spectral lines, indicated by the used Huffman codebook dimension, are coded with one codeword.

4.3.6.4.2 huff_decode(hcb, hcw) - Huffman decoding

This is a function which takes a Huffman codebook table and a Huffman codeword as input and returns the index of the Huffman codeword in the Huffman codebook table.

4.3.6.4.3 quad_sign_bits - Quad sign bits - 0 to 4 bits

If an unsigned Huffman codebook of dimension four is used for the quantized spectral lines, additional sign bits are transmitted for each quantized spectral line which is different to zero.

4.3.6.4.4 pair_sign_bits - Pair sign bits -0 to 2 bits

If an unsigned Huffman codebook of dimension two is used for the quantized spectral lines, additional sign bits are transmitted for each quantized spectral line which is different to zero.

4.3.6.4.5 ext_code - Extension code - 5 to 21 bits

This 5 to 21 bit field is an extension code which is used in connection with Huffman codebook 11. For details see clause 5.1.2.

4.3.6.5 asf_scalefac_data - ASF scale factor data

4.3.6.5.1 reference_scale_factor - Reference scale factor - 8 bit

This 8-bit field specifies the reference scale factor. The first scale factor used in the decoding process is equal to the reference scale factor. The remaining scale factor values are derived via delta decoding. See clause 6.2.6.4 for details.

4.3.6.5.2 asf_sf_hcw - Huffman coded scale factor delta - variable bits

This variable length field holds a Huffman coded delta value used for the *dpcm_sf[g][sfb]* determination. See clause 6.2.6.4 for details.

4.3.6.6 asf_snf_data - ASF spectral noise fill data

4.3.6.6.1 b_snf_data_exists - Spectral noise fill data exists flag - 1 bit

This bit, if set, indicates that spectral noise fill data shall follow in the bitstream.

4.3.6.6.2 asf_snf_hcw - Huffman code spectral noise fill delta - variable bits

This variable length field holds a Huffman coded delta value used for the *dpcm_snf[g][sfb]* determination. See clause 5.1.4 for details.

4.3.7 Speech spectral frontend

4.3.7.1 ssf_data - Speech spectral frontend data

4.3.7.1.1 b_ssf_iframe - SSF I-frame flag - 1 bit

This bit, if set, indicates that the first SSF granule in the `ssf_data` is an SSF-I-frame. The first SSF granule in the `ssf_data` is an SSF-I-frame if either `b_iframe` in the `ac4_substream_info` is set or `b_ssf_iframe` is set.

4.3.7.2 ssf_granule - Speech spectral frontend granule

4.3.7.2.1 stride_flag - Stride flag - 1 bit

This bit indicates the SSF coder mode and the number of SSF blocks per SSF granule, *num_blocks*, as shown in table 109. The `stride_flag` shall be set to '0' for SSF configurations which do not allow for a short stride mode as indicated by a value of '1' in the column "maximum number of SSF blocks" in tables 110 and 111.

Table 109: SSF coder mode

stride_flag	SSF coder mode	name	*num_blocks*
0	long stride mode	LONG_STRIDE	1
1	short stride mode	SHORT_STRIDE	4

4.3.7.2.2 SSF configuration via table

The SSF configuration depends on the sampling frequency and the `frame_rate_index` as shown in tables 110 and 111.

Table 110: SSF configuration for 48 kHz sampling frequency

frame_rate_index	Video frame rate (fps)	Frame length (samples) *frame_length*	SSF granule length (samples) *granule_length*	maximum number of SSF blocks	number of SSF granules
0	23,976	1 920	960	4	2
1	24	1 920	960	4	2
2	25	2 048	1 024	4	2
3	29,97	1 536	768	4	2
4	30	1 536	768	4	2
5	47,95	960	960	4	1
6	48	960	960	4	1
7	50	1 024	1 024	4	1
8	59,94	768	768	4	1
9	60	768	768	4	1
10	100	512	512	1	1
11	119,88	384	384	1	1
12	120	384	384	1	1
13	(23,44)	2 048	1 024	4	2
14+15	reserved				

Table 111: SSF configuration for 44,1 kHz sampling frequency

frame_rate_index	**Video frame rate (fps)**	**Frame length (samples) *frame_length***	**SSF granule length (samples) *granule_length***	**maximum number of SSF blocks**	**number of SSF granules**
0…12: reserved					
13	11025/512	2 048	1 024	4	2
14+15: reserved					

The SSF block length in samples, *n_mdct*, is given by:

$$n\_mdct = \frac{granule\_length}{num\_blocks}$$

4.3.7.2.3 num_bands_minus12 - Number of SSF coded bands minus 12 - 3 bits

This 3-bit code indicates the number of SSF coded bands minus 12. To get the number of SSF coded bands, *num_bands*, a value of 12 needs to be added to `num_bands_minus12`.

4.3.7.2.4 predictor_presence_flag[b] - Predictor presence flag for block b - 1 bit

This bit indicates the existence of predictor parameters in the bitstream for block b.

4.3.7.2.5 delta_flag[b] - Delta coding flag for block b - 1 bit

This bit indicates the usage of the differential coding of the prediction lag for block b.

4.3.7.3 ssf_st_data - Speech spectral frontend static data

4.3.7.3.1 env_curr_band0_bits - Signal envelope index for band 0 - 5 bits

This 5-bit code indicates the signal envelope index for band 0. The envelope decoder expects this value in ***env_idx[0]***.

4.3.7.3.2 env_startup_band0_bits - Startup envelope index for band 0 - 5 bits

This 5-bit code indicates the extra startup envelope index for band 0.

4.3.7.3.3 gain_bits[b] - Envelope gain bits for block b - 4 bits

This 4-bit code indicates a gain index for block b. The gain index is derived by adding an offset value of -8 to the gain bits value: ***gain_idx[b]*** = `gain_bits[b]` - 8.

4.3.7.3.4 predictor_lag_delta_bits[b] - Predictor lag delta for block b - 4 bits

This 4-bit code indicates a predictor lag delta to be used for the predictor lag index calculation for block b. See clause 5.2.4.

4.3.7.3.5 predictor_lag_bits[b] - Predictor lag index for block b - 9 bits

This 9-bit code indicates the predictor lag index directly for block b.

4.3.7.3.6 variance_preserving_flag[b] - Variance preserving flag for block b - 1 bit

This bit, if set, indicates that variance preserving for block b shall be used.

4.3.7.3.7 alloc_offset_bits[block] - Allocation offset bits for block b - 5 bits

This 5-bit code indicates a value to be used for calculating an allocation offset for block b. See clause 5.2.5 for usage details.

4.3.7.4 ssf_ac_data - Speech spectral frontend arithmetic coded data

4.3.7.4.1 env_curr_ac_bits - Arithmetic coded signal envelope indices - Variable bits

This variable length field contains *num_bands* - 1 arithmetic coded signal envelope indices for band indices 1 to *num_bands* - 1. The arithmetic decoded values are stored in ***env_idx[band]*** which is an input signal of the envelope decoder.

4.3.7.4.2 env_startup_ac_bits - Arithmetic coded startup envelope indices - Variable bits

This variable length field contains *num_bands* - 1 arithmetic coded extra startup envelope indices for band indices 1 to *num_bands* - 1. The startup envelope is decoded like the signal envelope and the decoded startup envelope is used as previous envelope ***env_prev[band]*** by the envelope decoder.

4.3.7.4.3 predictor_gain_ac_bits[b] - Arithmetic coded predictor gain index for block b - Variable bits

This variable length field contains one arithmetic coded predictor gain index for block b.

4.3.7.4.4 q_mdct_coefficients_ac_bits[b] - Arithmetic coded quantized MDCT coefficients for block b - Variable bits

This variable length field contains *num_bins* arithmetic coded quantized MDCT transform coefficients for block b.

4.3.7.5 SSF helper elements

The following helper elements are derived from the SSF bitstream elements:

start_bin[k]	vector indicating the coefficient index for the start of band *k*
end_bin[k]	vector indicating the coefficient index for the end of band *k*
num_bins	number of coded spectral coefficients

The following pseudocode shows how to initialize the helper elements using *num_bands*, *n_mdct* and the SSF bandwidths given in table C.1.

Pseudocode

```
// initialize start_bin[], stop_bin[] and num_bins

//band_widths is a column from table C.1 selected by n_mdct
MAX_NUM_BANDS = 19;

start_bin[0] = 0;
end_bin[0] = band_widths[0] - 1;
for (i = 1; i < MAX_NUM_BANDS; i++) {
    start_bin[i] = start_bin[i-1] + band_widths[i-1];
    end_bin[i] = start_bin[i] + band_widths[i] - 1;
}

num_bins = end_bin[num_bands - 1] + 1;
```

4.3.8 Stereo audio processing

4.3.8.1 chparam_info - Stereo information

4.3.8.1.1 sap_mode - Stereo audio processing mode - 2 bits

This 2-bit code indicates the stereo audio processing mode as shown in table 112.

Table 112: Stereo audio processing mode

sap_mode	SAP mode
0	no SAP
1	M/S processing in scale factor bands specified by ms_used[g][sfb]
2	M/S processing in all scale factor bands
3	full SAP

4.3.8.1.2 ms_used - M/S coding used - 1 bit

This bit indicates, if set, that M/S coding is used in scale factor band *sfb* in group *g*.

4.3.8.2 sap_data - Stereo audio processing data

4.3.8.2.1 sap_coeff_all - SAP coding all flag - 1 bit

This bit indicates, if set, that SAP coding is used in all scale factor bands. If not set, SAP coding is used in scale factor bands specified by `sap_coeff_used[g][sfb]`.

4.3.8.2.2 sap_coeff_used - SAP coding used - 1 bit

This bit indicates, if set, that SAP coding is used in scale factor band *sfb* in group *g*.

4.3.8.2.3 delta_code_time - Delta coding in time - 1 bit

This bit indicates delta coding in time of the *alpha_q* values. For group $g > 0$, the *alpha_q* deltas apply to the *alpha_q* values from group g-1. For group g=0, or if this bit is not set, the *alpha_q* deltas apply to the *alpha_q* values from scale factor band *sfb*-2. See clause 5.3.1 for details.

4.3.8.2.4 sap_hcw - Huffman coded alpha_q delta - variable bits

This variable length field holds a Huffman coded delta value used for the *alpha_q[g][sfb]* determination. See clause 5.3.1 for details. The same Huffman codebook as for the ASF scale factor deltas is used.

4.3.9 Companding control

This section contains control bits for the QMF domain companding tool. It is transmitted once per frame and applies to all channels.

Unless the `sync_flag` is set, the companding tool can be switched on or off separately for every channel; and if at least one channel has companding on, the "average mode" can be signalled for all channels that have it on.

If the `sync_flag` is set, the companding parameters are only transmitted once but apply to all channels in the element.

4.3.10 Advanced spectral extension - A-SPX

4.3.10.1 aspx_config - A-SPX configuration

4.3.10.1.1 aspx_quant_mode_env - 1 bit

This is a binary flag indicating the size of quantization steps used for encoded signal envelopes. Its values are given in table 113. See clause 5.7.6.3.5 for details.

Table 113: aspx_quant_mode_env

aspx_quant_mode_env	Quantization step size
0	1,5 dB
1	3,0 dB

4.3.10.1.2 aspx_start_freq - A-SPX start QMF subband - 3 bits

This is an index into the scale factor subband group table starting from the first subband moving upwards in steps of 2 subbands. An `aspx_start_freq` of 1 will hence point to subband 20 for the high bit rate table. Please see clause 5.7.6.3.1.1 for more details.

4.3.10.1.3 aspx_stop_freq - A-SPX stop QMF subband - 2 bits

This is an index into the scale factor subband group table starting from the last subband going downwards in steps of 2 subbands. An `aspx_stop_freq` of 2 will hence point to subband 50 in the high bit rate table. Please see clause 5.7.6.3.1.1 for more details.

4.3.10.1.4 aspx_master_freq_scale - A-SPX master frequency table scale - 1 bit

This is a value indicating which of the two static subband group tables is to be used to generate the master subband group table, according to table 114.

Table 114: aspx_master_scale

aspx_master_scale	Meaning
0	Low bit-rate scale factor table
1	High bit-rate scale factor table

4.3.10.1.5 aspx_interpolation - A-SPX interpolation used - 1 bit

This is a value determining how to estimate the energy of envelopes within an A-SPX interval. The estimation is performed by calculating the average of the squared complex subband samples over the time and frequency regions of the QMF time-frequency matrix. The meaning of `aspx_interpolation` is given in table 115.

Please see clause 5.7.6.4.2.1 for more details.

Table 115: aspx_interpolation

aspx_interpolation	Meaning
0	Interpolation not used
1	Interpolation used

4.3.10.1.6 aspx_preflat - A-SPX pre-flattening used - 1 bit

This value indicates whether spectral pre-flattening is used or not, according to table 116.

Table 116: aspx_preflat

aspx_preflat	Meaning
0	Pre-flattening not used
1	Pre-flattening used

4.3.10.1.7 aspx_limiter - A-SPX limiter used - 1 bit

This value indicates whether the limiter is turned on or off for a particular A-SPX interval, according to table 117.

Table 117: aspx_limiter

aspx_limiter	Meaning
0	Limiter off
1	Limiter on

4.3.10.1.8 aspx_noise_sbg - A-SPX number of noise subband groups - 2 bits

This is an input parameter to the function that calculates the number of noise subband groups, as detailed in clause 5.7.6.3.1.3.

4.3.10.1.9 aspx_num_env_bits_fixfix - A-SPX frame class FIXFIX bit count - 1 bit

This is an input parameter to the `aspx_framing` bitstream parsing block described in clause 4.3.10.4. The parameter indicates whether 1 or 2 bits are used for the transmission of the `tmp` bitstream parameter in the FIXFIX interval class. In the FIXFIX inverval class the number of signal envelopes is derived from `tmp` as `aspx_num_env[ch] = 2^tmp`. Hence, transmitting `tmp` using 1 bit only allows the use of either 1 or 2 signal envelopes, while 2 bits additionally allows the use of 4 uniformly spaced signal envelopes (8 envelopes is prohibited by the A-SPX syntax, see table 118.

Table 118: aspx_num_env_bits_fixfix

aspx_num_env_bits_fixfix	Meaning (FIXFIX class only)
0	`tmp` is transmitted using 1 bit
1	`tmp` is transmitted using 2 bits

4.3.10.1.10 aspx_freq_res_mode - A-SPX frequency resolution transmission mode - 2 bits

This is an input parameter to the `aspx_framing` bitstream parsing block described in clause 4.3.10.4 and to the function that calculates the A-SPX interval borders, as outlined in clause 5.7.6.3.3.1.

Table 119: aspx_freq_res_mode

aspx_freq_res_mode	Meaning
0	`aspx_freq_res` is signaled in `aspx_framing`
1	`aspx_freq_res` defaults to low resolution
2	`aspx_freq_res` defaults to predetermined values depending on the signal envelope duration
3	`aspx_freq_res` defaults to high resolution

4.3.10.2 aspx_data_1ch - A-SPX 1-channel data

4.3.10.2.1 aspx_xover_subband_offset - A-SPX crossover subband offset - 3 bits

This is an index into the high bit rate subband group table starting at the first subband, moving upward with a step of 1 subband.

4.3.10.3 aspx_data_2ch - A-SPX 2-channel data

4.3.10.3.1 aspx_xover_subband_offset - A-SPX crossover subband offset - 3 bits

This bitstream element has been described in clause 4.3.10.2.1.

4.3.10.3.2 aspx_balance - A-SPX balance setting - 1-bit

This value indicates whether the envelope and noise scale factors of the two channels have been coded separately or paired, according to table 120.

Table 120: aspx_balance

aspx_balance	Coding of envelope and noise scale factors
0	Separate per-channel coding
1	Paired channel coding

4.3.10.4 aspx_framing - A-SPX framing

4.3.10.4.1 aspx_int_class - A-SPX interval class - 1, 2 or 3-bits

Four different A-SPX interval classes are signaled in the bitstream element `aspx_int_class`. The element is signalled using up to three bits, as indicated in table 121.

Table 121: aspx_int_class

aspx_int_class	Value
0	FIXFIX
10	FIXVAR
110	VARFIX
111	VARVAR

For the FIXFIX interval class, the following information is transmitted in the bitstream:

- the number of uniformly spaced signal envelopes; and
- the frequency resolution (coarse or fine) for all envelopes.

For the FIXVAR interval class, the bitstream contains the following relevant information:

- the location of the trailing (right) interval border;
- the number of relative borders computed from the end of the interval;
- vectors containing relative borders associated with the trailing interval border;
- a pointer indicative of the signal envelope of transient characteristics if any; and
- the frequency resolution (coarse or fine) of each envelope.

For the VARFIX interval class, the bitstream contains the following relevant information:

- the location of the leading (left) interval border;
- the number of relative borders computed from the beginning of the interval;
- vectors containing relative borders associated with the leading interval border;
- a pointer indicative of the signal envelope of transient characteristics if any; and
- the frequency resolution (coarse or fine) of each envelope.

For the VARVAR interval class, the bitstream contains the following relevant information:

- the location of the leading (left) interval border;
- the number of relative borders computed from the beginning of the interval;
- vectors containing relative borders associated with the leading interval border;

- the location of the trailing (right) interval border;
- the number of relative borders computed from the end of the interval;
- vectors containing relative borders associated with the trailing interval border;
- a pointer indicative of the signal envelope of transient characteristics if any; and
- the frequency resolution (coarse or fine) of each envelope.

For the FIXFIX interval class, the signal envelopes are uniformly spaced within the A-SPX interval, while the FIXVAR, VARFIX and VARVAR interval classes do not generally have uniformly spaced signal envelopes.

Within one A-SPX interval there can either be one or two noise envelopes. If the number of signal envelopes is 1, the interval contains one noise envelope and if the number of signal envelopes is greater than 1, the interval contains two noise envelopes. In the latter case, the placement of the border between the two noise envelopes is a function of the variable *aspx_tsg_ptr* and the frame interval class `aspx_int_class`.

4.3.10.4.2 tmp_num_env - Temporary variable - envbits + 1 bits

A temporary variable used to calculate the variable *aspx_num_env* described in clause 4.3.10.4.11.

4.3.10.4.3 aspx_freq_res[ch][env] - Frequency resolution - 1-bit

The frequency resolution - low or high - for each channel and signal envelope, as shown in table 122.

Table 122: aspx_freq_res

aspx_freq_res	Meaning
0	Low frequency resolution
1	High frequency resolution

4.3.10.4.4 aspx_var_bord_left[ch] - Leading VAR interval envelope border - 2-bits

Indicates the position of the leading variable time slot group border for signal envelopes for the interval classes VARFIX and VARVAR.

4.3.10.4.5 aspx_var_bord_right[ch] - Trailing VAR interval envelope border - 2-bits

Indicates the position of the trailing variable time slot group border for signal envelopes for interval classes FIXVAR and VARVAR.

4.3.10.4.6 aspx_num_rel_left[ch] - Relative envelope border - 1 or 2-bits

Indicates number of relative time slot group borders for signal envelopes starting from `aspx_var_bord_0`. 2 bits are read for frame interval class VARVAR. For classes FIXVAR and VARFIX, 2 bits are read if num_aspx_timeslots are greater than 12, otherwise 1 bit is read.

4.3.10.4.7 aspx_num_rel_right[ch] - Relative envelope border - 1 or 2-bits

Indicates number of relative time slot group borders for signal envelopes starting from `aspx_var_bord_right`. 2 bits are read for frame interval class VARVAR. For classes FIXVAR and VARFIX, 2 bits are read if num_aspx_timeslots are greater than 12, otherwise 1 bit is read.

4.3.10.4.8 aspx_rel_bord_left[ch][rel] - Leading relative envelope borders - 1 or 2 bits

Array of variables indicating the offset of the signal envelope borders relative to the leading envelope border `aspx_var_bord_left`. For all frame interval classes, 2 bits are read if num_aspx_timeslots are greater than 12, otherwise 1 bit is read.

4.3.10.4.9 aspx_rel_bord_right[ch][rel] - Trailing relative envelope borders - 1 or 2 bits

Array of variables indicating the offset of the signal envelope borders relative to the trailing envelope border `aspx_var_bord_right`. For all frame interval classes, 2 bits are read if num_aspx_timeslots are greater than 12, otherwise 1 bit is read.

4.3.10.4.10 aspx_tsg_ptr[ch] - Pointer to envelope border - variable bits

Pointer to a specific A-SPX time slot group border, used to determine the position of the A-SPX time slot for placing sinusoids and noise envelope borders.

Please see clauses 5.7.6.3.3.1 and 5.7.6.4.2.1 for more details.

4.3.10.4.11 A-SPX Framing Helper Variables

The following helper elements are derived from the A-SPX Framing elements:

aspx_num_env[ch]	Variable indicating the number of signal envelopes per channel
aspx_num_noise[ch]	Variable indicating the number of noise envelopes per channel

Note that in a valid bitstream the number of signal envelopes in an A-SPX interval satisfies the conditions described in table 123.

Table 123: Restrictions for the number of envelopes (*aspx_num_env*)

	aspx_int_class		
	FIXFIX	**FIXVAR, VARFIX**	**VARVAR**
aspx_num_env	≤ 4	≤ 5	≤ 5

4.3.10.5 aspx_delta_dir - A-SPX delta coding direction

4.3.10.5.1 aspx_sig_delta_dir[ch][env] - A-SPX delta coding for signal envelopes - 1 bit

The binary array indicates whether the Huffman values for the signal envelopes are coded in time or frequency direction. The meaning of an entry is given in table 124.

Table 124: aspx_sig_delta_dir

aspx_sig_delta_dir	**Meaning**
0	Delta coding in frequency direction
1	Delta coding in time direction

4.3.10.5.2 aspx_noise_delta_dir[ch][env] - A-SPX delta coding for noise envelopes - 1 bit

The binary array indicates whether the Huffman values for the noise envelopes are coded in time or frequency direction. The meaning of an entry is given in table 125.

Table 125: aspx_noise_delta_dir

aspx_noise_delta_dir	**Meaning**
0	Delta coding in frequency direction
1	Delta coding in time direction

4.3.10.6 aspx_hfgen_iwc_1ch - A-SPX 1-channel HF generation and interleaved waveform coding

4.3.10.6.1 aspx_tna_mode[n] - A-SPX subband tonal to noise ratio adjustment mode - 2 bits

The subband tonal to noise ratio adjustment mode signaled for an A-SPX interval is one of the four values shown in table 126.

Table 126: aspx_tna_mode

aspx_tna_mode	Meaning
0	None
1	Light
2	Moderate
3	Heavy

The amount of tonal to noise ratio adjustment is proportional to a value calculated based on the values of `aspx_tna_mode` for the current A-SPX interval and the previous A-SPX interval.

For more detail on the subband tonal to noise ratio adjustment, please see clause 5.7.6.4.1.3.

4.3.10.6.2 aspx_add_harmonic[n] - A-SPX add harmonics - 1 bit

This value indicates the insertion of a sinusoid in a QMF subband where there was none present in the previous A-SPX interval, according to table 127.

Table 127: aspx_add_harmonic

aspx_add_harmonic	Meaning
0	Do not add sinusoid
1	Add sinusoid

4.3.10.6.3 aspx_fic_present - A-SPX frequency interleaved coding present - 1 bit

This value indicates the presence of frequency interleaved coding, according to table 128.

Table 128: aspx_fic_present

aspx_fic_present	Meaning
0	Frequency interleaved coding present
1	Frequency interleaved coding present

4.3.10.6.4 aspx_fic_used_in_sfb[n] - A-SPX frequency interleaved coding used in subband group - 1 bit

This value indicates the use of frequency interleaved coding in a particular subband group, according to table 129.

Table 129: aspx_fic_used_in_sfb

aspx_fic_used_in_sfb	Meaning
0	Frequency interleaved coding not used in current subband group
1	Frequency interleaved coding used in current subband group

4.3.10.6.5 aspx_tic_present - A-SPX time interleaved coding present - 1 bit

This value indicates the presence of time interleaved coding, according to table 130.

Table 130: aspx_tic_present

aspx_tic_present	Meaning
0	Time interleaved coding not present
1	Time interleaved coding present

4.3.10.6.6 aspx_tic_used_in_slot[n] - A-SPX time interleaved coding used in slot - 1 bit

This value indicates the use of time interleaved coding in an adjacent pair of QMF slots, according to table 131.

Table 131: aspx_tic_used_in_slot

aspx_tic_used_in_slot	Meaning
0	Time interleaved coding not used in current slot pair
1	Time interleaved coding used in current slot pair

4.3.10.6.7 aspx_ah_present - A-SPX add harmonics present - 1 bit

This bit, if set, indicates that `aspx_add_harmonic[n]` data shall follow in the bitstream.

4.3.10.7 aspx_hfgen_iwc_2ch - A-SPX 2-channel HF generation and interleaved waveform coding

4.3.10.7.1 aspx_tna_mode[ch][n] - A-SPX subband tonal to noise adjustment mode - 2 bits

This bitstream element has been described in clause 4.3.10.6.1, and takes an additional channel index.

4.3.10.7.2 aspx_add_harmonic[ch][n] - A-SPX add harmonics - 1 bit

This bitstream element has been described in clause 4.3.10.6.2, and takes an additional channel index.

4.3.10.7.3 aspx_fic_present - A-SPX frequency interleaved coding present - 1 bit

This bitstream element has been described in clause 4.3.10.6.3.

4.3.10.7.4 aspx_fic_left - A-SPX frequency interleaved coding in left channel - 1 bit

This value indicates the presence of frequency interleaved coding in the left channel, according to table 132.

Table 132: aspx_fic_left

aspx_fic_left	Meaning
0	Frequency interleaved coding not present in left channel
1	Frequency interleaved coding present in left channel

4.3.10.7.5 aspx_fic_right - A-SPX frequency interleaved coding in right channel - 1 bit

This value indicates the presence of frequency interleaved coding in the right channel, according to table 133.

Table 133: aspx_fic_right

aspx_fic_right	Meaning
0	Frequency interleaved coding not present in right channel
1	Frequency interleaved coding present in right channel

4.3.10.7.6 aspx_fic_used_in_sfb[ch][n] - A-SPX frequency interleaved coding used in subband group - 1 bit

This bitstream element has been described in clause 4.3.10.6.4, and takes an additional channel index.

4.3.10.7.7 aspx_tic_present - A-SPX time interleaved coding present - 1 bit

This bitstream element has been described in clause 4.3.10.6.5.

4.3.10.7.8 aspx_tic_copy - A-SPX time interleaved coding copy data - 1 bit

This value indicates whether or not the time interleaved data is copied to both left and right channels, according to table 134.

Table 134: aspx_tic_copy

aspx_tic_copy	Meaning
0	Time interleaved coding not copied
1	Time interleaved coding copied

4.3.10.7.9 aspx_tic_left - A-SPX time interleaved coding in left channel - 1 bit

This value indicates the presence of time interleaved coding in the left channel, according to table 135.

Table 135: aspx_tic_left

aspx_tic_left	Meaning
0	Time interleaved coding not present in left channel
1	Time interleaved coding present in left channel

4.3.10.7.10 aspx_tic_right - A-SPX time interleaved coding in right channel - 1 bit

This value indicates the presence of time interleaved coding in the right channel, according to table 136.

Table 136: aspx_tic_right

aspx_tic_right	Meaning
0	Time interleaved coding not present in right channel
1	Time interleaved coding present in right channel

4.3.10.7.11 aspx_tic_used_in_slot[n] - A-SPX time interleaved coding used in slots - 1 bit

This bitstream element has been described in clause 4.3.10.6.6.

4.3.10.7.12 aspx_ah_left - A-SPX add harmonics in left channel - 1 bit

This bit, if set, indicates that `aspx_add_harmonic[0][n]` data shall follow in the bitstream.

4.3.10.7.13 aspx_ah_right - A-SPX add harmonics in right channel - 1 bit

This bit, if set, indicates that `aspx_add_harmonic[1][n]` data shall follow in the bitstream.

4.3.10.8 aspx_ec_data - A-SPX Huffman data

This is a function that returns the Huffman decoded signal and noise envelope scale factors from the bitstream. It takes the following arguments:

data_type	Variable indicating the whether to decode a signal or a noise envelope
num_env	Variable indicating the number of envelopes to decode
freq_res	Variable indicating the frequency resolution of signal envelopes
quant_mode	Variable indicating the quantization mode for signal envelopes
stereo_mode	Variable indicating the whether to decode two channels separately or paired
direction	Variable indicating the direction of the delta coding

4.3.10.8.1 aspx_huff_data - Huffman decoding for A-SPX values

This is a function which returns an array of scale factors of a decoded signal or noise envelope.

It takes the following arguments:

data_type	Variable indicating the whether to decode a signal or a noise envelope
sbg	Variable indicating the number of subband groups in the current envelope
qm	Variable indicating the quantization mode for signal envelopes
sm	Variable indicating the whether to decode two channels separately or paired
dir	Variable indicating the direction of the delta coding

4.3.10.8.2 aspx_hcw - A-SPX Huffman code word - 1 to x bits

This value indicates the Huffman code word used for Huffman decoding.

4.3.10.8.3 huff_decode_diff(hcb, hcw) - Huffman decoding for differences

This is a function which takes a Huffman codebook table and a Huffman codeword as input and returns the index of the Huffman codeword in the Huffman codebook table subtracted by the Huffman codebook offset *cb_off*, which is specified together with the Huffman codebook table.

4.3.11 Advanced coupling - A-CPL

4.3.11.1 acpl_config_1ch - A-CPL 1-channel configuration

4.3.11.1.1 acpl_1ch_mode - A-CPL 1-channel mode - via table

This helper element indicates the A-CPL 1-channel mode as shown in table 137.

Table 137: acpl_1ch_mode

acpl_1ch_mode	Meaning
0	FULL
1	PARTIAL

4.3.11.1.2 acpl_num_param_bands_id - A-CPL number of parameter bands - 2 bits

A parameter band is a grouping of QMF subbands. Advanced Coupling parameters are transmitted per parameter band. This value indicates the number of parameter bands, *acpl_num_param_bands*, as shown in table 138.

Table 138: acpl_num_param_bands_id

acpl_num_param_bands_id	*acpl_num_param_bands*
0	15
1	12
2	9
3	7

4.3.11.1.3 acpl_quant_mode - A-CPL quantization mode - 1 bit

This value indicates coarse or fine quantization.

Table 139: acpl_quant_mode

acpl_quant_mode	Meaning
0	Fine
1	Coarse

4.3.11.1.4 acpl_qmf_band - A-CPL QMF band - 3 bits

This value indicates a QMF subband below which the signal is mid-side coded, and above or at which, the signal is coded using advanced coupling.

4.3.11.2 acpl_cfg_2ch - A-CPL 2-channel configuration

4.3.11.2.1 acpl_num_param_bands_id - A-CPL number of parameter bands - 2 bits

This bitstream element has been described in clause 4.3.11.1.2.

4.3.11.2.2 acpl_quant_mode_0 - A-CPL quantization mode 0 - 1 bit

This value indicates coarse or fine quantization for `acpl_alpha` and `acpl_beta` coefficients. The possible values of this bitstream element are shown in clause 4.3.11.1.3.

4.3.11.2.3 acpl_quant _mode_1 - A-CPL quantization mode 1 - 1 bit

This value indicates coarse or fine quantization for `acpl_gamma` coefficients. The possible values of this bitstream element are shown in clause 4.3.11.1.3.

4.3.11.3 acpl_data_1ch - A-CPL 1-channel data

acpl_alpha1 and *acpl_beta1* are identifiers used for Huffman table dequantization.

4.3.11.4 acpl_data_2ch - A-CPL 2-channel data

acpl_alpha1, *acpl_alpha2*, *acpl_beta1*, *acpl_beta2*, *acpl_beta3*, *acpl_gamma1*, *acpl_gamma2*, *acpl_gamma3*, *acpl_gamma4*, *acpl_gamma5*, and *acpl_gamma6* are identifiers used for Huffman table dequantization.

4.3.11.5 acpl_framing_data - A-CPL framing data

4.3.11.5.1 acpl_interpolation_type - A-CPL interpolation type - 1 bit

This value indicates the type of interpolation used.

Table 140: acpl_interpolation_type

acpl_interpolation_type	Meaning
0	smooth A-CPL interpolation
1	steep A-CPL interpolation

4.3.11.5.2 acpl_num_param_sets_cod - A-CPL number of parameter sets per frame - 1 bit

This value indicates the value *acpl_num_param_sets* which describes how many parameter sets per frame are transmitted in the bitstream.

Table 141: acpl_num_param_sets

acpl_num_param_sets_cod	*acpl_num_param_sets*
0	1
1	2

4.3.11.5.3 acpl_param_subsample - A-CPL parameter change at subsample - 5 bits

When steep interpolation is used, this value indicates the QMF subsample (0-31) at which the parameter set values change.

4.3.11.6 acpl_huff_data - A-CPL Huffman data

4.3.11.6.1 acpl_hcw - A-CPL Huffman code word - 1 to x bits

This value indicates the Huffman code word used for Huffman decoding. The following pseudocode describes how to choose the correct Huffman table for decoding the bitstream elements.

Pseudocode

```
get_acpl_hcb(data_type, quant_mode, hcb_type)
{
    // data_type = {ALPHA, BETA, BETA3, GAMMA}
    // quant_mode = {COARSE, FINE)
    // hcb_type = {F0, DF, DT}

    acpl_hcb = ACPL_HCB_<data_type>_<quant_mode>_<hcb_type>;
    // the line above expands using the inputs data_type, quant_mode and hcb_type

    // The 24 A-CPL Huffman codebooks are given in table A.34 to table A.57
    // and are named according to the schema outlined above.

    return acpl_hcb;
}
```

4.3.12 Basic and extended metadata

4.3.12.1 metadata - Metadata

4.3.12.1.1 tools_metadata_size_value - Size of tools metadata - 7 bits

This field, potentially extended with `variable_bits()`, generates *tools_metadata_size*, indicating the size in bits of the metadata related to the DRC and dialog enhancement tools.

4.3.12.1.2 b_more_bits - More bits flag - 1 bit

This bit indicates the presence of additional `variable_bits` to be used for the *tools_metadata_size* determination.

4.3.12.1.3 b_umd_payloads_substream - UMD payloads substream flag - 1 bit

This bit, if set, indicates the presence of a `umd_payloads_substream()` element.

4.3.12.2 basic_metadata - Basic metadata

4.3.12.2.1 dialnorm_bits - Input reference level - 7 bits

This 7-bit code specifies the *dialnorm* in dB_{FS}, in steps of ¼ dB_{FS}. *dialnorm* indicates the input reference level.

$$dialnorm = -0{,}25 \times dialnorm\_bits \ [\text{dB}_{\text{FS}}]$$

4.3.12.2.2 b_more_basic_metadata - More basic metadata flag - 1 bit

This bit, if set, indicates that more basic metadata is available.

4.3.12.2.3 b_further_loudness_info - Additional loudness information flag - 1 bit

This bit, if set, indicates that additional loudness information is present in a `further_loudness_info()` element.

4.3.12.2.4 b_prev_dmx_info - Previous downmix information flag - 1 bit

This bit, if set, indicates that downmix information of the signal before encoding is present.

4.3.12.2.5 pre_dmixtyp_2ch - Previous downmix to 2 channels type - 3 bits

This 3-bit code indicates what downmix was performed before encoding into 2 channels as shown in table 142.

Table 142: pre_dmixtyp_2ch

pre_dmixtyp_2ch	Downmix equation
0	unknown

1	Lo = L + (-3 dB) × C + (-3 dB) × Ls
	Ro = R + (-3 dB) × C + (-3 dB) × Rs

2	Lt = L + (-3 dB) × C - (-3 dB) × Ls - (-3 dB) × Rs
	Rt = R + (-3 dB) × C + (-3 dB) × Ls + (-3 dB) × Rs

3	Lt = L + (-3 dB) × C - (-1,2 dB) × Ls - (-6,2 dB) × Rs
	Rt = R + (-3 dB) × C + (-1,2 dB) × Rs + (-6,2 dB) × Ls

4…7	reserved

4.3.12.2.6 phase90_info_2ch - Phase 90 in 2 channels info - 2 bits

This 2-bit code indicates whether a 90° phase filtering was done before downmixing and encoding into 2 channels as shown in table 143.

Table 143: phase90_info_2ch

phase90_info_2ch	Semantics
0	not indicated
1	reserved
2	surrounds have undergone a 90° phase shift before downmixing and encoding
3	surrounds have not undergone a 90° phase shift before downmixing and encoding

4.3.12.2.7 b_dmx_coeff - Downmix coefficients present flag - 1 bit

This bit, if set, indicates that downmix coefficients are present.

4.3.12.2.8 loro_center_mixgain - LoRo center mix gain - 3 bits

This 3-bit code indicates the gain to be used for LoRo downmixing the center channel of a 5 or 7 channel signal into 2 channels as shown in table 144.

Table 144: loro_{center,surround}_mixgain

loro_center_mixgain loro_surround_mixgain	LoRo mixgain
000	1,414 (+3,0 dB)
001	1,189 (+1,5 dB)
010	1,000 (0,0 dB)
011	0,841 (-1,5 dB)
100	0,707 (-3,0 dB) // Note
101	0,595 (-4,5 dB)
110	0,500 (-6,0 dB)
111	0,000 (-inf dB)
NOTE: When no mixgains have been transmitted, a downmix gain of -3,0 dB shall be used.	

4.3.12.2.9 loro_surround_mixgain - LoRo surround mix gain - 3 bits

This 3-bit code indicates the gain to be used for LoRo downmixing the surround channels of a 5 or 7 channel signal into 2 channels as shown in table 144.

4.3.12.2.10 b_loro_dmx_loud_corr - LoRo downmix loudness correction flag - 1 bit

This bit, if set, indicates that LoRo downmix loudness correction data is present.

4.3.12.2.11 loro_dmx_loud_corr - LoRo downmix loudness correction - 5 bits

This 5-bit code indicates a LoRo downmix loudness correction. This correction factor is used to calibrate downmix loudness to original program loudness

$$\text{loro\_corr\_gain} = (15 - \text{loro\_dmx\_loud\_corr})/2\ [\text{dB}_2]$$

A value of 31 is reserved, and if present indicates a gain of 0 [dB].

4.3.12.2.12 b_ltrt_mixinfo - LtRt downmix info present - 1 bit

This bit, if set, indicates that LtRt downmix coefficients are present. If not set, the LtRt downmix coefficients are the same as the LoRo downmix coefficients.

4.3.12.2.13 ltrt_center_mixgain - LtRt center mix gain - 3 bits

This 3-bit code indicates the gain to be used for LtRt downmixing the center channel of a 5 or 7 channel signal into 2 channels as shown in table 144.

4.3.12.2.14 ltrt_surround_mixgain - LtRt surround mix gain - 3 bits

This 3-bit code indicates the gain to be used for LtRt downmixing the surround channels of a 5 or 7 channel signal into 2 channels as shown in table 144.

4.3.12.2.15 b_ltrt_dmx_loud_corr - LtRt downmix loudness correction flag - 1 bit

This bit, if set, indicates that LtRt downmix loudness correction data is present.

4.3.12.2.16 ltrt_dmx_loud_corr - LtRt downmix loudness correction - 5 bits

This field indicates a LtRt downmix loudness correction. This correction factor is used to calibrate downmix loudness to original program loudness

$$\text{ltrt\_corr\_gain} = (15 - \text{ltrt\_dmx\_loud\_corr})/2\ [\text{dB}_2]$$

A value of 31 is reserved, and if present indicates a gain of 0 [dB].

4.3.12.2.17 b_lfe_mixinfo - LFE downmix info present - 1 bit

This bit, if set, indicates that a LFE mix gain is present.

4.3.12.2.18 lfe_mixgain - LFE mix gain - 5 bits

This field indicates the gain to be used for mixing the LFE channel:

$$LFE\ mix\ gain\ in\ dB = 10\ - \text{lfe\_mixgain}$$

The LFE mix gain range is -21 dB…+10 dB.

4.3.12.2.19 preferred_dmx_method - Preferred downmix method - 2 bits

This 2-bit code indicates the preferred way of downmixing from 5 to 2 channel as shown in table 145.

Table 145: preferred_dmx_method

preferred_dmx_method	Used coefficients
0	not indicated
1	loro
2	ltrt
3	ltrt

4.3.12.2.20 b_predmxtyp_5ch - Previous downmix to 5 channels flag - 1 bit

This bit, if set, indicates that downmix information of the signal before encoding is present.

4.3.12.2.21 pre_dmixtyp_5ch - Previous downmix to 5 channels type - 3 bits

This 3-bit code indicates what downmix was performed before encoding into 5 channels as shown in table 146.

Table 146: pre_dmixtyp_5ch

pre_dmixtyp_5ch	Downmix equation

0	L = L
	R = R
	C = C
	LFE = LFE
	Ls' = Ls + (-3 dB) × Cs
	Rs' = Rs + (-3 dB) × Cs

1	L = L
	R = R
	C = C
	LFE = LFE
	Ls' = (-3 dB) × Ls + (-3 dB) × Lrs
	Rs' = (-3 dB) × Rs + (-3 dB) × Rrs

2	L = L
	R = R
	C = C
	LFE = LFE
	Ls' = Ls + (-1,2 dB) × Lrs + (-6,2 dB) × Rrs
	Rs' = Rs + (-1,2 dB) × Rrs + (-6,2 dB) × Lrs

3	L = L
	R = R
	C = C
	LFE = LFE
	Ls' = Ls - (-1,2 dB) × Lvh - (-6,2 dB) × Rvh
	Rs' = Rs + (-1,2 dB) × Rvh + (-6,2 dB) × Lvh

4…7	reserved

4.3.12.2.22 b_preupmixtyp_5ch - Previous upmix to 5 channels flag - 1 bit

This bit, if set, indicates that upmix information of the signal before encoding is present.

4.3.12.2.23 pre_upmixtyp_5ch - Previous upmix to 5 channels type - 4 bits

This 4-bit code indicates what upmix from 2 channels was performed before encoding into 5 channels as shown in table 147.

Table 147: pre_upmixtyp_5ch

pre_upmixtyp_5ch	Upmix type
0	Dolby® Pro Logic®
1	Dolby® Pro Logic® II Movie Mode
2	Dolby® Pro Logic® II Music Mode
3	Dolby® Professional Upmixer
4…15	Reserved

4.3.12.2.24 b_upmixtyp_7ch - Previous upmix to 7 channels flag - 1 bit

This bit, if set, indicates that upmix information of the signal before encoding is present.

4.3.12.2.25 pre_upmixtyp_3_4 - Previous upmix to 7 channels type - 2 bits

This 2-bit code indicates what upmix from 5 channels was performed before encoding into 7 channels (3/4) as shown in table 148.

Table 148: pre_upmixtyp_3_4

pre_upmixtyp_3_4	Upmix type
0	Dolby® Pro Logic® IIx Movie Mode
1	Dolby® Pro Logic® IIx Music Mode
2…3	Reserved

4.3.12.2.26 pre_upmixtyp_3_2_2 - Previous upmix to 7 channels type - 2 bits

This 2-bit code indicates what upmix from 5 channels was performed before encoding into 7 channels (3/2/2) as shown in table 149.

Table 149: pre_upmixtyp_3_2_2

pre_upmixtyp_3_2_2	Upmix type
0	Dolby® Pro Logic® IIz Height
1	Reserved

4.3.12.2.27 phase90_info_mc - Phase 90 in multi-channel info - 2 bits

This 2-bit code indicates whether a 90° phase filtering was done before encoding as shown in table 150.

Table 150: phase90_info_mc

phase90_info_mc	Semantics
0	not indicated
1	surrounds have undergone a 90° phase shift before encoding
2	surrounds have not undergone a 90° phase shift before encoding
3	reserved

4.3.12.2.28 b_surround_attenuation_known - Surround attenuation known flag - 1 bit

This bit should be ignored by the decoder.

4.3.12.2.29 b_lfe_attenuation_known - LFE attenuation known flag - 1 bit

This bit should be ignored by the decoder.

4.3.12.2.30 b_dc_blocking - DC blocking flag - 1 bit

This bit, if set, indicates that DC blocking information is present.

4.3.12.2.31 dc_block_on - DC blocking - 1 bit

This bit, if set, indicates that the signal has been DC filtered before encoding. The signal has not been DC filtered before encoding if this bit is not set.

4.3.12.3 further_loudness_info - Additional loudness information

4.3.12.3.1 loudness_version - Loudness version - 2 bits

This 2-bit field, which is extendable via `extended_loudness_version`, indicates the version of the loudness payload. For loudness data payloads that conform to the present document, the `loudness_version` field shall be set to '00', and thus the `extended_loudness_version` field shall not be present in the payload.

4.3.12.3.2 extended_loudness_version - Loudness version extension - 4 bits

This 4-bit field, which is only present if `loudness_version=3`, holds an extension to the `loudness_version`.

4.3.12.3.3 loud_prac_type - Loudness practice type - 4 bits

This 4-bit field indicates which recommended practice for programme loudness measurement has been followed to generate the programme loudness of the current audio substream as shown in table 151.

Table 151: loud_prac_type

loud_prac_type	Semantics
0000	Loudness regulation compliance not indicated
0001	Programme loudness measured according to ATSC A/85 [i.5]
0010	Programme loudness measured according to EBU R128 [i.6]
0011	Programme loudness measured according to ARIB TR-B32 [i.7]
0100	Programme loudness measured according to FreeTV OP-59 [i.8]
0101…1101	Reserved
1110	Manual
1111	Consumer leveller

4.3.12.3.4 b_loudcorr_dialgate - Loudness correction dialog gating flag - 1 bit

This bit, if set, indicates whether the loudness of the programme has been corrected using dialog gating.

4.3.12.3.5 dialgate_prac_type - Dialog gating practice type - 3 bits

This 3-bit field indicates which dialog gating method has been utilized for loudness correction with current audio substream as shown in table 152. The `dialgate_prac_type` parameter is only indicated if the `b_loudcorr_dialgate` parameter preceding in the bitstream is set to '1'.

Table 152: dialgate_prac_type

dialgate_prac_type	Semantics
000	Dialog gating method – Not Indicated
001	Dialog gating method – Automated Center or Left+Right Channel(s)
010	Dialog gating method – Automated Left, Center, and/or Right Channel(s)
011	Dialog gating method – Manual Selection
100…111	Reserved
NOTE 1: `dialgate_prac_type method`s '001' and '010' are described in [i.9] and a value of '011' indicates that a manual process was utilized to select representative portions of the program containing dialog for measurement. NOTE 2: For `dialgate_prac_type method` '001', dialog gating is applied to the Center channel of a multichannel program OR to the power sum of Left and Right channels in a stereo program. NOTE 3: For `dialgate_prac_type method` '010', dialog gating is applied individually to all front (main) channels of the program.	

4.3.12.3.6 b_loudcorr_type - Loudness correction type - 1 bit

If the loudness of the programme has been corrected with an infinite look-ahead (file based) loudness correction process, the value of the `b_loudcorr_type` field is set to '0'. If the loudness of the programme has been corrected using a realtime loudness measurement, the value of this field is set to '1'.

4.3.12.3.7 b_loudrelgat - Loudness value relative gated flag - 1 bit

This bit, if set, indicates that a `loudrelgat` field is present in the payload.

4.3.12.3.8 loudrelgat - Loudness value relative gated - 11 bits

This 11-bit field indicates the integrated loudness of the audio programme, measured according to Recommendation ITU-R BS.1770 [i.3] and without any gain adjustments due to dialnorm and dynamic range compression being applied.

The `loudrelgat` parameter is encoded according to:

$$\text{loudrelgat} = \lfloor loudness\_value\_relative\_gated \times 10 + 0{,}5 \rfloor + 1\,024$$

and decoded according to:

$$loudness\_value\_relative\_gated = (\text{loudrelgat} - 1\,024)/10 \text{ [LUFS]}$$

4.3.12.3.9 b_loudspchgat - Loudness value speech gated flag - 1 bit

This bit, if set, indicates that a `loudspchgat` and a `dialgate_prac_type` field is present in the payload.

4.3.12.3.10 loudspchgat - Loudness value speech gated - 11 bits

This 11-bit field indicates the integrated dialog-based loudness of the entire audio programme, measured according to formula (2) of Recommendation ITU-R BS.1770 [i.3] with dialog-gating. The `loudspchgat` value represents the dialog-based loudness without any gain adjustments due to dialnorm and dynamic range compression being applied.

The `loudspchgat` parameter is encoded according to:

$$\text{loudspchgat} = \lfloor loudness\_value\_speech\_gated \times 10 + 0{,}5 \rfloor + 1\,024$$

and decoded according to:

$$loudness\_value\_speech\_gated = (\text{loudspchgat} - 1\,024)/10 \text{ [LUFS]}$$

4.3.12.3.11 dialgate_prac_type - Dialog gating practice type - 3 bits

This 3-bit field indicates which dialog gating method has been utilized to generate the `loudspchgat` parameter value for the current audio substream as shown in table 152. The `dialgate_prac_type` parameter is only indicated if the `b_loudspchgat` parameter preceding in the bitstream is set to '1'.

4.3.12.3.12 b_loudstrm3s - Loudness values short term 3s flag - 1 bit

This bit, if set, indicates that a `loudstrm3s` field is present in the payload.

4.3.12.3.13 loudstrm3s - Loudness values short term 3s - 11 bits

This 11-bit field indicates the loudness of the preceding 3 seconds of the audio programme, measured according to Recommendation ITU-R BS.1771 [i.4] and without any gain adjustments due to dialnorm and dynamic range compression being applied.

The `loudstrm3s` parameter is encoded according to:

$$\mathrm{loudstrm3s} = \lfloor loudness\_value\_short\_term\_3s \times 10 + 0{,}5 \rfloor + 1\,024$$

and decoded according to:

$$loudness\_value\_short\_term\_3s = (\mathrm{loudstrm3s} - 1\,024)/10\ [\mathrm{LUFS}]$$

4.3.12.3.14 b_max_loudstrm3s - Max loudness value short term 3s flag - 1 bit

This bit, if set, indicates that a `max_loudstrm3s` field follows in the bitstream.

4.3.12.3.15 max_loudstrm3s - Max loudness value short term 3s - 11 bits

This 11-bit field indicates the maximum short-term loudness of the audio programme, measured according to Recommendation ITU-R BS.1771 [i.4] and without any gain adjustments due to dialnorm and dynamic range compression being applied.

The `max_loudstrm3s` parameter is encoded according to:

$$\mathrm{max\_loudstrm3s} = \lfloor max\_loudness\_value\_short\_term\_3s \times 10 + 0{,}5 \rfloor + 1\,024$$

and decoded according to:

$$max\_loudness\_value\_short\_term\_3s = (\mathrm{max\_loudstrm3s} - 1\,024)/10\ [\mathrm{LUFS}]$$

4.3.12.3.16 b_truepk - True peak flag - 1 bit

This bit, if set, indicates that a `truepk` field follows in the bitstream.

4.3.12.3.17 truepk - True peak - 11 bits

This 11-bit field indicates the true peak sample value of the programme measured since the previous value was sent in the bitstream. The `truepk` value is measured according to Annex 2 of Recommendation ITU-R BS.1770 [i.3].

The `truepk` parameter is encoded according to:

$$\mathrm{truepk} = \lfloor true\_peak\_value \times 10 + 0{,}5 \rfloor + 1\,024$$

and decoded according to:

$$true\_peak\_value = (\mathrm{truepk} - 1\,024)/10\ [\mathrm{dBTP}]$$

4.3.12.3.18 b_max_truepk - Max true peak flag - 1 bit

This bit, if set, indicates that a `max_truepk` field follows in the bitstream.

4.3.12.3.19 max_truepk - Max true peak - 11 bits

This 11-bit field indicates the maximum true peak sample value of the programme measured according to Annex 2 of Recommendation ITU-R BS.1770 [i.3].

The max_truepk parameter is encoded according to:

$$\mathrm{max\_truepk} = \lfloor max\_true\_peak\_value \times 10 + 0{,}5 \rfloor + 1\,024$$

and decoded according to:

$$max\_true\_peak\_value = (\mathrm{max\_truepk} - 1\,024)/10\ [\mathrm{dBTP}]$$

4.3.12.3.20 b_prgmbndy - Programme boundary flag - 1 bit

This bit, if set, indicates that programme boundary data is present in the payload.

4.3.12.3.21 prgmbndy_bit - Programme boundary bit - 1 bit

This bit, which can be present multiple times, is used for the determination of the programme boundary value *prgmbndy*. The value of the variable *prgmbndy* shall be equal to the number of frames between the current frame and the frame that contains the boundary between two different programmes. This data may be used to determine when to begin and end the measurement of the loudness parameters specified in this payload. An encoder shall restrict the transmitted value to the range [2…512].

4.3.12.3.22 b_end_or_start - Programme boundary end or start flag - 1 bit

If this field is set to '1', then the value of the *prgmbndy* variable indicates the number of frames between the current frame and the upcoming frame that contains the next programme boundary.

If the b_end_or_start field is '0', then the value of the *prgmbndy* variable indicates the number of frames between the current frame and the past frame that contained the previous programme boundary.

4.3.12.3.23 b_prgmbndy_offset - Programme boundary offset flag - 1 bit

This bit, if set, indicates that a prgmbndy_offset field is present in the payload.

4.3.12.3.24 prgmbndy_offset - Programme boundary offset - 11 bits

This 11-bit field indicates the offset in audio samples from the first sample of the frame indicated by the *prgmbndy* variable to the actual audio sample in that frame that corresponds to the programme boundary.

4.3.12.3.25 b_lra - Loudness range flag - 1 bit

This bit, if set, indicates that an lra field follows in the bitstream.

4.3.12.3.26 lra - Loudness range - 10 bits

This 10-bit field indicates the loudness range of the programme as specified in EBU Tech Document 3342 [i.10].

The lra parameter is encoded according to:

$$\mathrm{lra} = \lfloor loudness\_range \times 10 + 0{,}5 \rfloor$$

and decoded according to:

$$loudness\_range = \mathrm{lra}/10\ [\mathrm{LU}]$$

4.3.12.3.27 lra_prac_type – Loudness range measurement practice type - 3 bits

This 3-bit field indicates which method has been utilized to compute the loudness range with the current audio substream as shown in table 153.

Table 153: lra_prac_type

lra_prac_type	Semantics
000	Loudness Range as per EBU Tech 3342 [i.10] v1
001	Loudness Range as per EBU Tech 3342 [i.10] v2
010…111	Reserved

4.3.12.3.28 b_loudmntry - Momentary loudness flag - 1 bit

This bit, if set, indicates that a `loudmntry` field follows in the bitstream.

4.3.12.3.29 loudmntry - Momentary loudness - 11 bits

This 11-bit field indicates the momentary loudness of the programme measured since the previous value was sent in the bitstream. The `loudmntry` measurement is specified in Recommendation ITU-R BS.1771 [i.4] (or EBU Tech 3341 [i.11]) without any gain adjustments due to dialnorm and dynamic range compression applied.

The `loudmntry` parameter is encoded according to:

$$\text{loudmntry} = \lfloor momentary\_loudness \times 10 + 0{,}5 \rfloor + 1\,024$$

and decoded according to:

$$momentary\_loudness = (\text{loudmntry} - 1\,024)/10\ [\text{LUFS}]$$

4.3.12.3.30 b_max_loudmntry - Maximum momentary loudness flag - 1 bit

This bit, if set, indicates that a `max_loudmntry` field follows in the bitstream.

4.3.12.3.31 max_loudmntry - Maximum momentary loudness - 11 bits

This 11-bit field indicates the maximum momentary loudness of the programme measured as specified in Recommendation ITU-R BS.1771 [i.4] (or EBU Tech 3341 [i.11]) without any gain adjustments due to dialnorm and dynamic range compression applied.

The `max_loudmntry` parameter is encoded according to:

$$\text{max\_loudmntry} = \lfloor maximum\_momentary\_loudness \times 10 + 0{,}5 \rfloor + 1\,024$$

and decoded according to:

$$maximum\_momentary\_loudness = (\text{max\_loudmntry} - 1\,024)/10\ [\text{LUFS}]$$

4.3.12.3.32 b_extension - Extension flag - 1 bit

This bit, if set, indicates that further loudness extension data is present.

4.3.12.3.33 e_bits_size - Extension size - 5 bits

This 5-bit field, potentially extended with `variable_bits()`, indicates the size of the `extension_bits` field in bits.

4.3.12.3.34 extension_bits - Extension bits - e_bits_size bits

This field of size *e_bits_size* holds extension bits. The content of these bits is not defined in the present document.

4.3.12.4 extended_metadata

4.3.12.4.1 b_associated - Associate substream flag - parameter

This parameter is true when the containing substream is an associate substream, either indicated by the `presentation_config` or by the `content_classifier`.

4.3.12.4.2 b_dialog - Dialog substream flag - parameter

This parameter is true when the containing substream is a dialog substream, either indicated by the `presentation_config` or by the `content_classifier`.

4.3.12.4.3 b_scale_main - Scale main flag - 1 bit

This bit, if set, indicates that scale main information is present.

4.3.12.4.4 scale_main - Scale main - 8 bits

This 8-bit value represents a negative gain of 0 dB (0x00) to 76,2 dB (0xfe) in 0,3 dB steps. A value of 0xff indicates a full mute.

4.3.12.4.5 b_scale_main_center - Scale main center flag - 1 bit

This bit, if set, indicates that scale main center information is present.

4.3.12.4.6 scale_main_center - Scale main center - 8 bits

This 8-bit value represents a negative gain of 0 dB (0x00) to 76,2 dB (0xfe) in 0,3 dB steps. A value of 0xff indicates a full mute.

4.3.12.4.7 b_scale_main_front - Scale main front flag - 1 bit

This bit, if set, indicates that scale main front information is present.

4.3.12.4.8 scale_main_front - Scale main front - 8 bits

This 8-bit value represents a negative gain of 0 dB (0x00) to 76,2 dB (0xfe) in 0,3 dB steps. A value of 0xff indicates a full mute.

4.3.12.4.9 pan_associated - Associate pan data- 8 bits

This 8-bit value represents an angle from 0 (0x00) to 358,5 (0xef) degrees in 1,5 degree steps. The values 0xf0…0xff shall not be used. The angle is to be interpreted as 0 being the front direction, increasing angle seen clockwise (standard Right speaker at +30 degree (0x14)).

4.3.12.4.10 b_dialog_max_gain - Dialog max gain flag - 1 bit

This bit, if set, indicates that a `dialog_max_gain` element is present and shall follow in the bitstream.

4.3.12.4.11 dialog_max_gain - Dialog max gain - 2 bits

This 2-bit field indicates the maximum gain, *g_dialog_max*, by which a user might change the dry main / dialog balance in favour of the dialog. *g_dialog_max* is determined via

$$g\_dialog\_\max[dB] = (1 + dialog\_\max\ \_gain) \times 3$$

4.3.12.4.12 b_pan_dialog_present - Dialog pan data present flag - 1 bit

This bit, if set, indicates that dialog pan data is present and shall follow in the bitstream.

4.3.12.4.13 pan_dialog - Dialog pan data - 8 bit

This 8-bit field indicates an angle, used for panning one or two dialog signals onto the dry main track. The semantics are the same as for `pan_associated`. See clause 4.3.12.4.9.

4.3.12.4.14 reserved - Reserved - 2 bits

This 2-bit field is reserved for future use. A decoder conforming to the present document shall ignore this field.

4.3.12.4.15 b_channels_classifier - Channel classifier flag - 1 bit

This bit, if set, indicates that channel classification data is available.

4.3.12.4.16 b_{c,l,r,ls,rs,lrs,rrs,lw,rw,vhl,vhr,lfe}_active - Channel active flag - 1 bit

This bit, if set, indicates that the corresponding channel, C, L, R, Ls, Rs, Lrs, Rrs, Lw, Rw, Vhl, Vhr or LFE, is active.

4.3.12.4.17 b_{c,l,r}_has_dialog - Channel has dialog flag - 1 bit

This bit, if set, indicates that the corresponding channel, C, L or R, contains a dialog.

4.3.12.4.18 b_event_probability_present - Event probability present flag - 1 bit

This bit indicates whether an event probability value is transmitted.

4.3.12.4.19 event_probability - Event probability - 4 bits

This field indicates the probability that a congruent audio event is ongoing. A high value corresponds to a high certainty.

4.3.12.5 Channel mode query functions

4.3.12.5.1 channel_mode_contains_Lfe()

This function returns true if the channel mode contains a Low-Frequency Effects channel.

Pseudocode

```
channel_mode_contains_Lfe()
{
    if (ch_mode == 4 || ch_mode == 6 || ch_mode == 8 || ch_mode == 10) {
        return TRUE;
    }
    return FALSE;
}
```

4.3.12.5.2 channel_mode_contains_c()

This function returns true if the channel mode contains a Center channel.

Pseudocode

```
channel_mode_contains_c()
{
    if (ch_mode == 0 || (ch_mode >= 2 && ch_mode <= 10)) {
        return TRUE;
    }
    return FALSE;
}
```

4.3.12.5.3 channel_mode_contains_lr()

This function returns true if the channel mode contains a Left and a Right channel.

Pseudocode

```
channel_mode_contains_lr()
{
    if (ch_mode >= 1 && ch_mode <= 10) {
        return TRUE;
    }
    return FALSE;
}
```

4.3.12.5.4 channel_mode_contains_LsRs()

This function returns true if the channel mode contains a Left Surround and a Right Surround channel.

Pseudocode

```
channel_mode_contains_LsRs()
{
    if (ch_mode >= 3 && ch_mode <= 10) {
        return TRUE;
    }
    return FALSE;
}
```

4.3.12.5.5 channel_mode_contains_LrsRrs()

This function returns true if the channel mode contains a Left Rear Surround and a Right Rear Surround channel.

Pseudocode

```
channel_mode_contains_LrsRrs()
{
    if (ch_mode == 5 || ch_mode == 6) {
        return TRUE;
    }
    return FALSE;
}
```

4.3.12.5.6 channel_mode_contains_LwRw()

This function returns true if the channel mode contains a Left Wide and a Right Wide channel.

Pseudocode

```
channel_mode_contains_LwRw()
{
    if (ch_mode == 7 || ch_mode == 8) {
        return TRUE;
    }
    return FALSE;
}
```

4.3.12.5.7 channel_mode_contains_VhlVhr()

This function returns true if the channel mode contains a Left Vertical Height and a Right Vertical Height channel.

Pseudocode

```
channel_mode_contains_VhlVhr()
{
    if (ch_mode == 9 || ch_mode == 10) {
        return TRUE;
    }
    return FALSE;
}
```

4.3.13 Dynamic range control - DRC

4.3.13.1 drc_frame - Dynamic Range Control

4.3.13.1.1 b_drc_present - DRC present - 1 bit

This field indicates if DRC data is available in the bitstream.

4.3.13.2 drc_config - DRC configuration

4.3.13.2.1 drc_decoder_nr_modes - Number of DRC decoder modes - 3 bits

This field indicates the number of DRC decoder modes carried in `drc_frame`, which is `drc_decoder_nr_modes` + 1.

4.3.13.2.2 drc_eac3_profile - (E-)AC-3 profile - 3 bits

This field indicates which (E-)AC-3 profile to use when transcoding an AC-4 stream into (E-)AC-3. Table 154 shows how the values of this field are related to the (E-)AC-3 profile.

Table 154: (E-)AC-3 profiles

drc_eac3_profile	Profile
0	None
1	Film standard
2	Film light
3	Music standard
4	Music light
5	Speech
6	Reserved
7	Reserved

4.3.13.3 drc_decoder_mode_config - DRC decoder mode_config

4.3.13.3.1 drc_decoder_mode_id - DRC decoder mode ID - 3 bits

This 3 bit field identifies DRC decoder mode IDs as shown in table 155.

Table 155: drc_decoder_mode_id

Value of drc_decoder_mode_id	DRC decoder mode	Output level range
0	Home Theatre	-31...-27
1	Flat panel TV	-26...-17
2	Portable - Speakers	-16...0
3	Portable - Headphones	-16...0
4	Reserved	
5	Reserved	
6	Reserved	
7	Reserved	

4.3.13.3.2 drc_output_level_from - Lowest reference output level - 5 bits

This field indicates the lowest output level of the reference level (*dialnorm*, see clause 4.3.12.2.1) for the corresponding DRC decoder mode. The value of the field represents levels in dB_{FS} according to

$$L_{out,min} = -\text{drc\_output\_level\_from [dB]}$$

The output range for the default DRC decoder modes is defined in table 155.

4.3.13.3.3 drc_output_level_to - Highest reference output level - 5 bits

This field indicates the highest output level of the reference level (*dialnorm*, see clause 4.3.12.2.1) for the corresponding DRC decoder mode.

The value of the field represents levels in dB_{FS} according to:

$$L_{out,max} = -\text{drc\_output\_level\_to [dB]}$$

The output range for the default DRC decoder modes is defined in table 155.

4.3.13.3.4 drc_repeat_profile_flag - Repeat profile flag - 1 bit

This bit indicates the presence of the `drc_repeat_id`, described in clause 4.3.13.3.5.

4.3.13.3.5 drc_repeat_id - Repeat data from ID - 3 bits

When the configuration of two or more DRC decoder modes is identical, this field provides an option to duplicate the configuration without signalling it twice.

If this field is sent, this DRC decoder mode is defined by the mode with ID `drc_repeat_id`.

4.3.13.3.6 drc_default_profile_flag - Default profile flag - 1 bit

This bit indicates that the (E-)AC-3 profile [i.1], indicated by `drc_eac3_profile`, shall be used to define the compression curve and time constants. The definition of the (E-)AC-3 profiles data is given in table 156.

NOTE 1: Semantics for the compression curve parameters are given in clause 4.3.13.4.

Table 156: (E-)AC-3 profiles definition

Profile / Parameter	None	Film standard	Film light	Music standard	Music light	Speech
Compression Curve						
$L_{0,low}$ [dB]	0	0	-10	0	-10	0
$L_{0,high}$ [dB]	0	5	10	5	10	5
$G_{maxboost}$ [dB]	0	6	6	12	12	15
drc_nr_boost_sections	0	0	0	0	0	0
$L_{maxboost}$ [dB]	0	-12	-22	-24	-34	-19
G_{maxcut} [dB]	0	-24	-24	-24	-15	-24
drc_nr_cut_sections	0	1	1	1	0	1
$L_{sectioncut}$ [dB]	0	15	20	15		15
L_{maxcut} [dB]	0	35	40	35	40	35
$G_{sectioncut}$ [dB]	0	-5	-5	-5		-5
Time smoothing						
τ_{attack} [ms]	N/A	100	100	100	100	100
$\tau_{release}$ [ms]	N/A	3 000	3 000	10 000	3 000	1 000
$\tau_{attack,fast}$ [ms]	N/A	10	10	10	10	10
$\tau_{release,fast}$ [ms]	N/A	1 000	1 000	1 000	1 000	200
drc_attack_threshold	N/A	15	15	15	15	10
drc_release_threshold	N/A	20	20	20	20	10

NOTE 2: Profile **None** has a constant gain (i.e. it effectively disables the compressor). Therefore, the time constants are not applicable. Time constants for this profile can be signaled using `drc_tc_default_flag`=1.

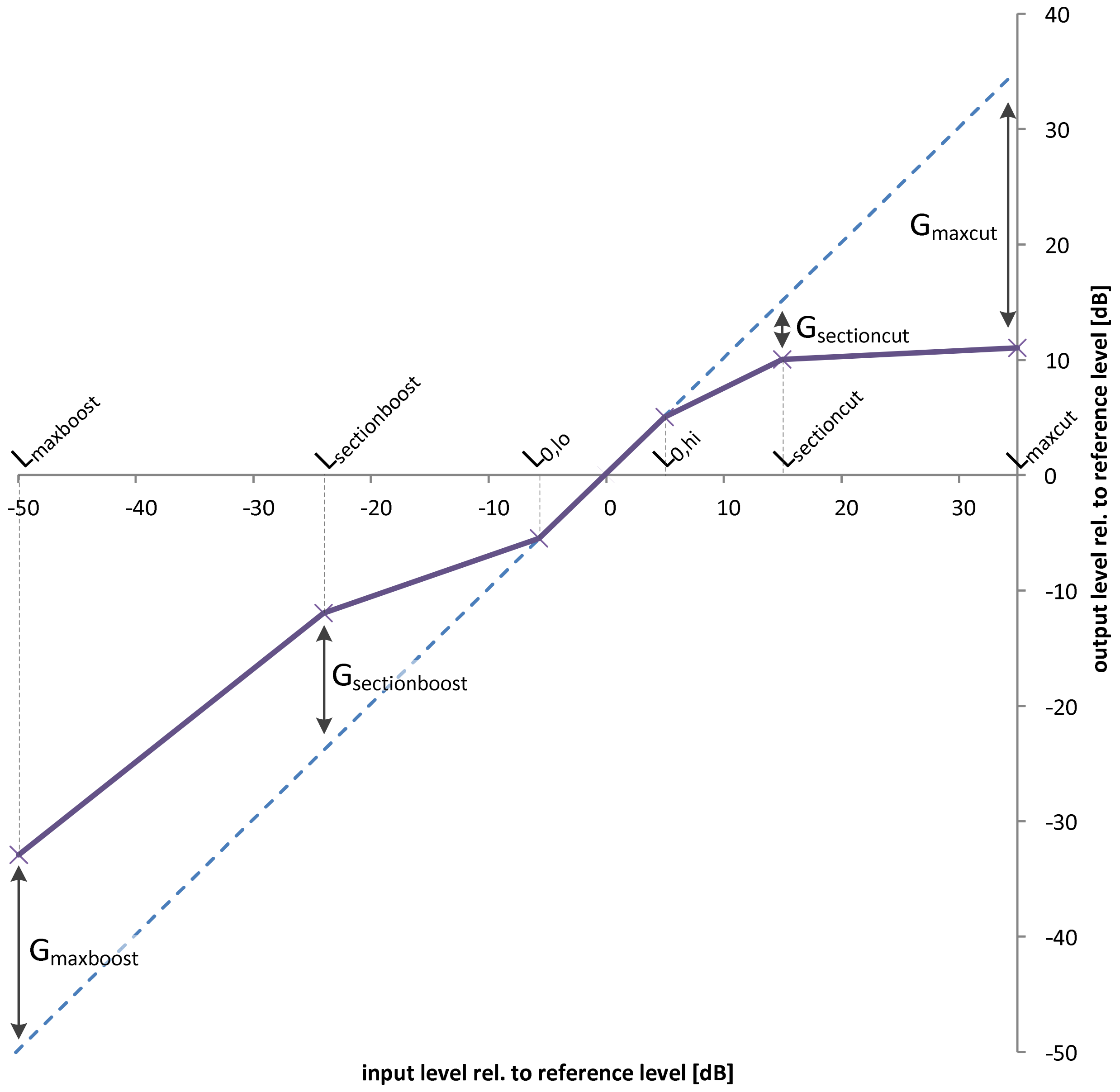

Figure 1: Parameterization of gain curve

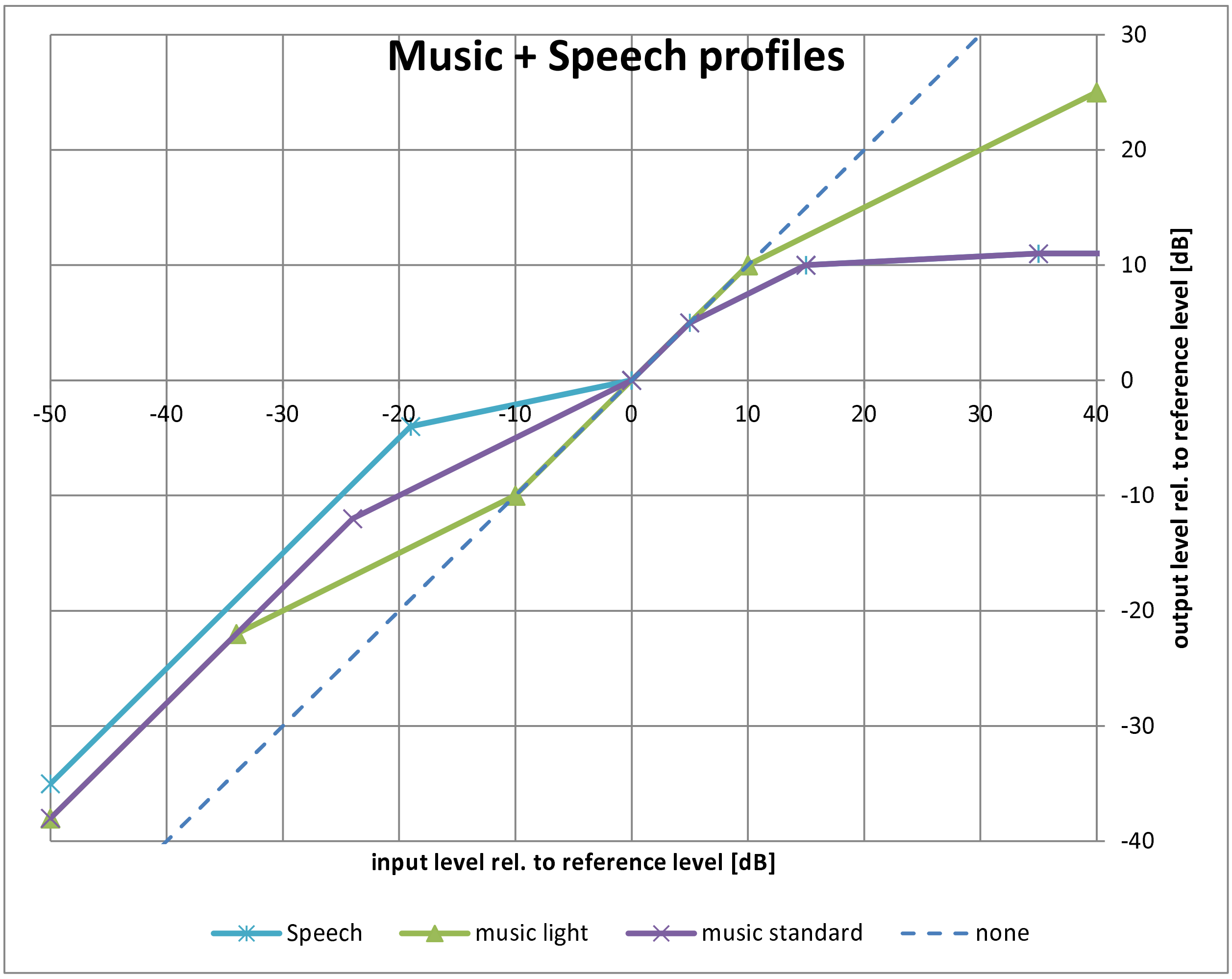

Figure 2: Music and speech compressor profiles

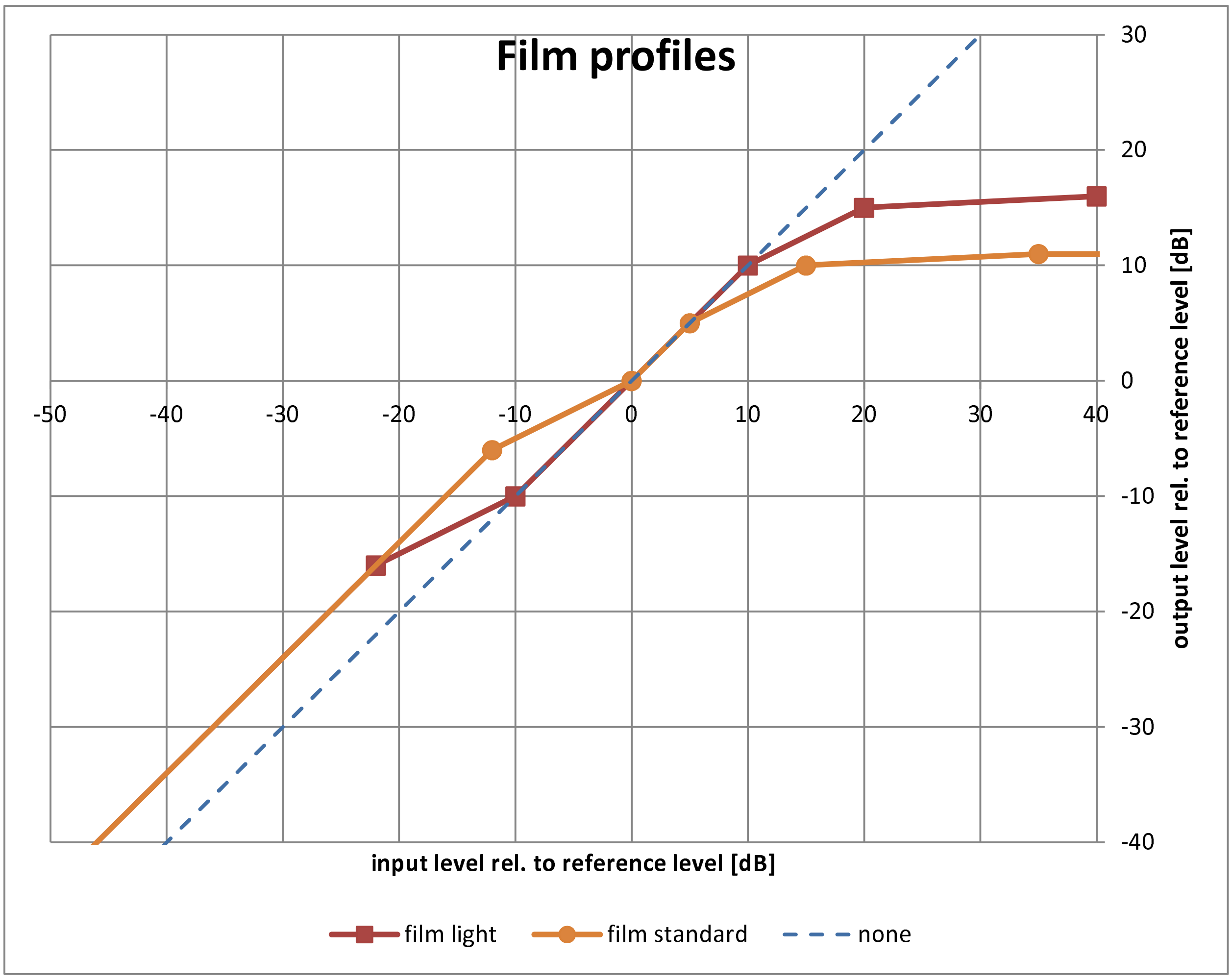

Figure 3: Film compressor profiles

4.3.13.3.7 drc_compression_curve_flag - Compression curve flag - 1 bit

This bit indicates whether a compression curve is transmitted for the current DRC decoder mode (`drc_compression_curve_flag`=1) or compression gains are transmitted by `drc_frame` (`drc_compression_curve_flag`=0).

4.3.13.3.8 drc_gains_config - DRC gains configuration - 2 bits

This field indicates the manner in which configuration DRC gains for this DRC decoder mode are transmitted, and implicitly the total number of DRC bands used in this mode. The number of parameter bands can be derived using table 157, and the definitions for configurations 2 and 3 are listed in table 158.

Table 157: Description of drc_gains_config

Value of drc_gains_config	Gains config	nr_drc_bands
0	Single wideband gain for all channels	1
1	Channel dependent gains, wideband	1
2	Channel dependent gains, 2 frequency bands	2
3	Channel dependent gains, 4 frequency bands	4
NOTE: Channel dependent DRC gains are transmitted for groups of channels. Clause 4.3.13.7.1 gives an overview of the groups per speaker configuration.		

Table 158: Parameter band definition for DRC gains

	drc_gains_config = 2		drc_gains_config = 3	
Parameter band	**QMF start band**	**QMF end band**	**QMF start band**	**QMF end band**
0	0	4	0	0
1	5	Qmax - 1	1	4
2			5	16
3			17	Qmax - 1

Qmax depends on the sampling frequency as shown in table 159.

Table 159: Qmax value

Sampling frequency	Qmax
44,1 kHz, 48 kHz	64
96 kHz	128
192 kHz	256

4.3.13.4 drc_compression_curve - Compression curve parameters

4.3.13.4.1 DRC compression curve parameterization

Figure 1 shows the parameterization of the gain curve. The gain curve is described by control points, defining sections. The control points are described by pairs (level, gain) which are given in dB_2. The level is relative to the reference level dialnorm, i.e. a level of zero is at the reference level.

The curve has boost sections, where the gain is all positive (or zero), and a cut section, where it is all negative (or zero).

NOTE: Most of the control points can be implicitly defined, as per table 160.

Table 160: Control points

	Transmitted if	Value if transmitted	Value if not transmitted
$G_{maxboost}$	Always	drc_gain_max_boost	N/A
$L_{maxboost}$	$G_{maxboost} > 0$	$L_{sectionboost}$ – (1+drc_lev_max_boost)	$L_{0,low}$
$G_{sectionboost}$	drc_nr_boost_sections=1	(1+drc_gain_section_boost)	
$L_{sectionboost}$	drc_nr_boost_sections=1	$L_{0,low}$ – (1+drc_lev_section_boost)	$L_{0,low}$
$L_{0,low}$	Always	0 – drc_lev_nullband_low	N/A
$L_{0,high}$	Always	0 + drc_lev_nullband_high	N/A
G_{maxcut}	Always	0 – drc_gain_max_cut	N/A
L_{maxcut}	$G_{maxcut} < 0$	$L_{sectioncut}$ + (1+drc_lev_max_cut)	$L_{0,high}$
$G_{sectioncut}$	drc_nr_cut_sections=1	0 – (1+drc_gain_section_cut)	
$L_{sectioncut}$	drc_nr_cut_sections=1	$L_{0,high}$+(1+drc_lev_section_cut)	$L_{0,high}$

4.3.13.4.2 drc_lev_nullband_low - Null band lower boundary - 4 bits

This field defines the lower limit of the null-band. See table 160.

4.3.13.4.3 drc_lev_nullband_high - Null band higher boundary - 4 bits

This field defines the upper limit of the null-band. See table 160.

4.3.13.4.4 drc_gain_max_boost - Maximum boost - 4 bits

This field defines the maximum boost gain. See table 160.

4.3.13.4.5 drc_lev_max_boost - Start of maximum boosting - 5 bits

This field defines the upper limit of the maximum boost section. See table 160.

4.3.13.4.6 drc_nr_boost_sections - Number of boost sections - 1 bit

This field controls the number of boost sections. See table 160.

4.3.13.4.7 drc_gain_section_boost - Extra boost section gain - 4 bits

This field defines an extra control point gain.

4.3.13.4.8 drc_lev_section_boost - Extra boost section control point level - 5 bits

This field defines an extra control point level.

4.3.13.4.9 drc_gain_max_cut - Maximum cut - 5 bits

This field defines the maximum cut gain. See table 160.

4.3.13.4.10 drc_lev_max_cut - Start of maximum cutting - 6 bits

This field defines the upper limit of the maximum cut section. See table 160.

4.3.13.4.11 drc_nr_cut_sections - Number of cut sections - 1 bit

This field controls the number of cut sections. See table 160.

4.3.13.4.12 drc_gain_section_cut - Extra cut section gain - 5 bits

This field defines an extra control point gain.

4.3.13.4.13 drc_lev_section_cut - Extra cut section control point level - 5 bits

This field defines an extra control point level.

4.3.13.4.14 drc_tc_default_flag - DRC default time constants flag - 1 bit

This bit indicates whether the default gain smoothing parameters should be used or if different time constants should be transmitted in the bitstream. If `drc_tc_default_flag` is 1 the smoothing data in table 161 is used.

Table 161: Default smoothing data

Field	Value
τ_{attack}	100 ms
$\tau_{release}$	3 000 ms
$\tau_{attack,fast}$	10 ms
$\tau_{release,fast}$	1 000 ms
drc_adaptive_smoothing_flag	0

4.3.13.4.15 drc_tc_attack - Time constant for attacks - 8 bits

This field defines the time constant for smoothing during attacks. The time constant is given by:

$$\tau_{attack} = \text{drc\_tc\_attack} \times 5[\text{ms}]$$

4.3.13.4.16 drc_tc_release - Time constant for release - 8 bits

This field defines the time constant for smoothing during signal release. The time constant is given by:

$$\tau_{release} = \text{drc\_tc\_release} \times 40[ms]$$

4.3.13.4.17 drc_tc_attack_fast - Time constant fast attacks - 8 bits

This field defines the time constant for smoothing during fast attacks. The time constant is given by:

$$\tau_{attack,fast} = \text{drc\_tc\_attack\_fast} \times 5[\text{ms}]$$

4.3.13.4.18 drc_tc_release_fast - Time constant for fast release - 8 bits

This field defines the time constant for smoothing during fast signal release. The time constant is given by:

$$\tau_{release,fast} = \text{drc\_tc\_release\_fast} \times 20[ms]$$

4.3.13.4.19 drc_adaptive_smoothing_flag - Adaptive smoothing flag - 1 bit

This bit indicates whether adaptive smoothing is supported by the profile.

4.3.13.4.20 drc_attack_threshold - Fast attack threshold - 5 bits

This field indicates the threshold in dB for deciding when to switch to the fast attack time constant.

4.3.13.4.21 drc_release_threshold - Fast release threshold - 5 bits

This field indicates the threshold in dB for deciding when to switch to the fast release time constant.

4.3.13.5 drc_data - DRC frame-based data

4.3.13.5.1 drc_gainset_size_value - Gain set data size - 6 bits

This field, potentially extended with `variable_bits`, codes *drc_gainset_size*. *drc_gainset_size* is the size in bits of the following `drc_gains` element. A decoder may use this information to skip profile data that are not applicable.

4.3.13.5.2 b_more_bits - More bits flag - 1 bit

This bit indicates the presence of additional `variable_bits` to be used for the *drc_gainset_size* determination.

4.3.13.5.3 drc_version - DRC version - 2 bits

This 2-bit field indicates the version of the DRC gainset data. For DRC gainset data that conforms to the present document, the `drc_version` field shall be set to '00', and thus the `drc2_bits` field shall not be present in the payload. A decoder implemented in accordance to the present document shall skip the `drc2_bits` field if the `drc_version` field is not set to '00'.

4.3.13.5.4 drc2_bits - DRC gainset extension bits – bits_left bits

This field of size *bits_left*, which is only present if `drc_version is not 0`, holds additional DRC gainset data. The content of this additional DRC gainset data is not specified here since bitstreams conforming to the present document shall have `drc_version`=0.

4.3.13.5.5 drc_reset_flag - DRC reset flag - 1 bit

This field signals a reset of the DRC algorithm.

4.3.13.5.6 drc_reserved - Reserved bits - 2 bits

This 2-bit field is reserved for future use and the content shall be skipped by an AC-4 decoder conforming to the present document.

4.3.13.6 drc_gains - DRC gains

4.3.13.6.1 drc_gain_val - DRC gain - 7 bits

This field contains the first DRC gain.

$$drc_{gain[0][0][0]} = (\text{drc\_gain\_val} \ - 64)[dB]$$

4.3.13.6.2 drc_gain_code - DRC gain codeword - 1…x bits

This field contains a Huffman code. The function huff_decode_diff() looks up the Huffman code using the Huffman code book DRC_HCB specified in table A.62 and generates a differential DRC gain value.

4.3.13.7 DRC helper elements

4.3.13.7.1 nr_drc_channels - Number of DRC channels

The variable *nr_drc_channels* indicates for how many channels DRC gains are transmitted. The value of this variable depends on the channel configuration for which the content is transmitted, as defined in table 162.

Table 162: nr_drc_channels

Channel config	*nr_drc_channels*	Group 1	Group 2	Group 3
Mono	1	C		
Stereo	1	L, R		
5.1	3	L, R, Lfe	C	Ls, Rs
7.1 (3/4/0)	3	L, R, Lfe	C	Ls, Rs, Lrs, Rrs
7.1 (5/2/0) `add_ch_base`=0	3	L, R, Lfe, Lw, Rw	C	Ls, Rs
7.1 (5/2/0) `add_ch_base`=1	3	L, R, Lfe	C	Ls, Rs, Lw, Rw
7.1 (3/2/2) `add_ch_base`=0	3	L, R, Lfe, Vhl, Vhr	C	Ls, Rs
7.1 (3/2/2) `add_ch_base`=1	3	L, R, Lfe	C	Ls, Rs, Vhl, Vhr

4.3.13.7.2 nr_drc_subframes - Number of DRC subframes

The variable *nr_drc_subframes* indicates the number of DRC subframes within an AC-4 frame. Table 163 lists the supported frame lengths and the corresponding number of DRC subframes.

Table 163: nr_drc_subframes

Frame length	Subframe length	*nr_drc_subframes*
384	384	1
512	256	2
768	256	3
960	320	3
1 024	256	4
1 536	256	6
1 920	320	6
2 048	256	8

4.3.14 Dialog enhancement - DE

4.3.14.1 b_de_data_present - Dialog enhancement data present flag - 1 bit

This bit indicates whether dialog enhancement data is present in the bitstream. In case an audio signal contains dialog, it is mandatory that dialog enhancement data is present, i.e. `b_de_data_present` shall be true for all frames.

4.3.14.2 de_config_flag - Dialog enhancement configuration flag - 1 bit

This bit indicates whether a frame that is not an I-frame contains dialog enhancement configuration data.

4.3.14.3 de_config - Dialog enhancement configuration

4.3.14.3.1 de_method - Dialog enhancement method - 2 bits

This field indicates the dialog enhancement method as shown in table 164.

Table 164: Description of de_method values

Value	de_method
0	Channel independent dialog enhancement
1	Cross-channel dialog enhancement
2	Waveform-parametric hybrid; channel independent enhancement and waveform channel
3	Waveform-parametric hybrid; cross-channel enhancement and waveform channel

4.3.14.3.2 de_max_gain - Maximum dialog enhancement gain - 2 bits

This field defines the maximum gain (G_{max}) allowed for boosting the dialog:

$$G_{max} = (\text{de\_max\_gain} + 1) \times 3\ [dB_2]$$

4.3.14.3.3 de_channel_config - Channel configuration - 3 bits

This field provides a three bit mask for the three front channels of a multichannel signal to indicate for which channels dialog enhancement parameters are transmitted. For mono and stereo content a subset of the values is supported. Table 165 provides an overview with 'X' indicating allowed values, and defines the corresponding *de_nr_channels*.

Table 165: Description of de_channel_config values

Value	de_channel_config	*de_nr_channels*	Mono	Stereo	Multichannel
000	'No parameters'	0	X	X	X
001	Right	1		X	X
010	Center	1	X		X
011	Center, right	2			X
100	Left	1		X	X
101	Left, right	2		X	X
110	Left, center	2			X
111	Left, center, right	3			X

4.3.14.4 de_data - Dialog enhancement data

4.3.14.4.1 de_keep_pos_flag - Keep position flag - 1 bit

This binary flag indicates whether the dialog panning information of the previous frame should be repeated for the current frame. For I-frames, this flag defaults to zero.

4.3.14.4.2 de_mix_coef1_idx, de_mix_coef2_idx - Dialog panning parameters - 5 bits

These fields define the mixing of the dialog signal onto the first, second and potentially third channel processed by the dialog enhancement tool. Up to two parameters are decoded using the quantization vector defined in table 166. The last coefficient is always derived considering an energy preserving upmix.

Table 166: Quantization vector for mixing coefficient parameters

de_mix_coefX_idx	0	1	2	3	4	5	6	7
de_mix_coefX	0	$6{,}32 \cdot 10^{-3}$	10^{-2}	$1{,}79 \cdot 10^{-2}$	$3{,}16 \cdot 10^{-2}$	$5{,}65 \cdot 10^{-2}$	$7{,}87 \cdot 10^{-2}$	0,111
de_mix_coefX_idx	8	9	10	11	12	13	14	15
de_mix_coefX	0,156	0,218	0,303	0,37	0,448	0,533	0,622	0,7071
de_mix_coefX_idx	16	17	18	19	20	21	22	23
de_mix_coefX	0,783	0,846	0,894	0,929	0,953	0,976	0,9877	0,9938
de_mix_coefX_idx	24	25	26	27	28	29	30	31
de_mix_coefX	0,9969	0,9984	0,9995	0,99984	0,99995	0,99998	1	Reserved

4.3.14.4.3 de_keep_data_flag - Keep data flag - 1 bit

This bit indicates whether the latest transmitted parametric data shall be repeated for the current frame.

4.3.14.4.4 de_ms_proc_flag - M/S processing flag - 1 bit

This bit indicates whether the parameters are related to processing on the Mid/Side representation of the signal.

4.3.14.4.5 de_par_code - Parameter code - 1...x bits

This field contains a Huffman code indicating a dialog enhancement parameter, representing either an absolute parameter quantization index or a differential value.

4.3.14.4.6 de_signal_contribution - Contribution of the signal to the enhancement - 5 bits

This field indicates the contribution of the coded signal in the hybrid mode. It indicates the ratio of the gain that should be contributed by the coded signal.

$$g_{DE,signal} = \frac{\text{de\_signal\_contribution}}{31} \cdot \left(10^{\frac{G_{DE}}{20}} - 1\right)$$

$$g_{DE,parametric} = \frac{31 - \text{de\_signal\_contribution}}{31} \cdot \left(10^{\frac{G_{DE}}{20}} - 1\right)$$

4.3.14.5 DE helper elements

4.3.14.5.1 de_nr_bands - Number of parameter bands

This constant value indicates the number of dialog enhancement parameter bands, which is always 8 bands. The QMF band grouping for each dialog enhancement parameter band is shown in table 167.

Table 167: Dialog enhancement parameter band definition

DE parameter band	First QMF band	Last QMF band
0	0	0
1	1	1
2	2	3
3	4	6
4	7	10
5	11	16
6	17	26
7	27	40

4.3.14.5.2 de_par[ ][ ] - Dialog enhancement parameter set

Variable that contains the decoded quantization indices to the dialog enhancement parameters for *de_nr_channels* and *de_nr_bands*.

4.3.14.5.3 de_par_prev[ ][ ] - Previous dialog enhancement parameter set

Variable that contains the decoded quantization indices to the dialog enhancement parameters for the previous frame.

4.3.14.5.4 de_abs_huffman(table_idx, code) - Absolute parameter Huffman decoding

This is a function that looks up Huffman codes, resulting in a dialog enhancement parameter quantization index. If *table_idx* is zero, the Huffman codebook DE_HCB_ABS_0 from table A.58 is used, otherwise DE_HCB_ABS_1 from table A.60.

Pseudocode

```
de_abs_huffman(table_idx, code)
{
    if (table_idx == 0) {
        hcb = DE_HCB_ABS_0;
    } else {
        hcb = DE_HCB_ABS_1;
    }
    cb_idx = huff_decode_diff(hcb, code);

    return cb_idx;
}
```

4.3.14.5.5 de_diff_huffman(table_idx, code) - Differential parameter Huffman decoding

This is a function that looks up Huffman codes, resulting in a differential dialog enhancement parameter quantization index. If *table_idx* is zero, the Huffman codebook DE_HCB_DIFF_0 from table A.59 is used, otherwise DE_HCB_DIFF_1 from table A.61.

Pseudocode

```
de_diff_huffman(table_idx, code)
{
    if (table_idx == 0) {
        hcb = DE_HCB_DIFF_0;
    } else {
        hcb = DE_HCB_DIFF_1;
    }
    cb_idx = huff_decode_diff(hcb, code);

    return cb_idx;
}
```

4.3.15 Universal metadata

The Universal Metadata (UMD) is an extensible structure for metadata delivery. Each payload is tagged with a specific payload identifier to provide an unambiguous indication of the type of data present in the payload. The order of payloads within the UMD substream is undefined - payloads can be stored in any order, and a parser shall be able to parse the entire UMD substream to extract relevant payloads, and ignore payloads that are either not relevant or are unsupported.

4.3.15.1 umd_payloads_substream - Universal metadata payloads substream

4.3.15.1.1 umd_payload_id - UMD payload identification - 5 bits/variable_bits(5)

The 5-bit `umd_payload_id` field which is extendable by `variable_bits()` identifies the type of payload that follows in the UMD substream. The assignment of `umd_payload_id` field values to specific payload types is specified in table 168.

Table 168: Defined UMD payload types

umd_payload_id	UMD payload type
0	UMD substream end
1…31, …	Reserved

The final payload in the UMD substream shall have a `umd_payload_id` value of 0, indicating that no additional payloads follow in the UMD substream. When the value of the `umd_payload_id` is 0, all fields in the `umd_payload_config` shall be set to 0, and the value of the `umd_payload_size` field shall be set to 0.

4.3.15.1.2 umd_payload_size - Size of UMD payload - variable_bits(8)

The `umd_payload_size` field indicates the size of the following payload. The value of the `umd_payload_size` field shall be equal to the number of bytes in the following payload, excluding the `umd_payload_size` field and all fields in `umd_payload_config`.

4.3.15.1.3 umd_payload_byte - UMD payload byte - 8 bits

The sequence of `umd_payload_size umd_payload_byte` values forms the UMD payload.

4.3.15.2 umd_payload_config - Universal metadata payload configuration

4.3.15.2.1 b_smpoffst - payload sample offset flag - 1 bit

This bit, if set, indicates that the `smpoffst` field shall follow in the bitstream. If the current payload applies to the first sample of the AC-4 frame, this field shall be set to '0'.

4.3.15.2.2 smpoffst - payload sample offset - variable_bits(11)

The `smpoffst` field indicates the offset, in units of PCM audio samples, from the beginning of the AC-4 frame to the first PCM audio sample that the data in the payload shall apply to. Note that this field may indicate a sample index that extends beyond the current AC-4 frame.

4.3.15.2.3 b_duration - payload duration flag - 1 bit

This bit, if set, indicates that the `duration` field shall follow in the bitstream. If the current payload applies to all samples up to and including the final sample of the AC-4 frame, this field shall be set to '0'.

4.3.15.2.4 duration - payload duration - variable_bits(11)

The `duration` field indicates the time period in units of PCM audio samples that the data in the payload applies to. Note that this field may indicate a sample index that extends beyond the current AC-4 frame.

4.3.15.2.5 b_groupid - payload group ID flag - 1 bit

This bit, if set, indicates that the `groupid` field shall follow in the bitstream.

4.3.15.2.6 groupid - payload group ID - variable_bits(2)

The `groupid` field provides a mechanism to indicate to a decoder that specific payloads within the UMD substream are associated with one another and that the payload data are related. For payloads that are associated with each other, the `groupid` field for each payload shall be set to the same value.

4.3.15.2.7 b_codecdata - codec specific data flag - 1 bit

The 1-bit `b_codecdata` field enables codec-specific data to be included in the payload configuration. For payload configuration data that conforms to the present document, this field shall be set to '0'.

4.3.15.2.8 codecdata - codec specific data - 8 bits

This is not specified here since bitstreams conforming to the present document shall have `b_codecdata`=0.

4.3.15.2.9 b_discard_unknown_payload - discard unknown payload during transcode flag - 1 bit

This bit indicates whether the current payload is retained or discarded during a transcoding process if the transcoder does not recognize the specific `umd_payload_id` value of the payload. If the `b_discard_unknown_payload` field is set to '1', the current payload shall be discarded during a transcoding process if the `umd_payload_id` value is unrecognized. If the `b_discard_unknown_payload` field is set to '0', the current payload shall be retained during a transcoding process unless otherwise indicated by the `proc_allowed` field.

4.3.15.2.10 b_payload_frame_aligned - payload to audio data frame alignment flag - 1 bit

The 1-bit `b_payload_frame_aligned` field is used to indicate how closely the payload data and the audio data within the AC-4 frame need to be synchronized after decoding and/or during transcoding. If the `b_payload_frame_aligned` field is set to '0', the application of the payload data to the decoded audio from the AC-4 frame is not required to be frame aligned. If the field is set to '1', the payload data shall be applied to the decoded audio from that AC-4 frame.

4.3.15.2.11 b_create_duplicate - create duplicate payload during transcode flag - 1 bit

In some transcode operations, the frame length of the output format may be shorter than the length of an AC-4 frame. The 1-bit `b_create_duplicate` field shall be set to '1' to indicate that the payload shall be duplicated in all frames of the output format that correspond to the same period of time as the current AC-4 frame. The `b_create_duplicate` field shall be set to '0' if the payload does not need to be repeated in all frames of the output format that correspond to the same period of time as the current AC-4 frame.

4.3.15.2.12 b_remove_duplicate - remove duplicate payload during transcode flag - 1 bit

The 1-bit `b_remove_duplicate` field shall be set to '1' to indicate that payloads with the same `umd_payload_id` value as the current payload need to be included only once in every output frame of the transcoding process when the output frame duration of a transcoding process is longer than the length of an AC-4 frame. Subsequent payloads carried in AC-4 frames that correspond to the same period of time as the current frame of the output format shall be discarded. The `b_remove_duplicate` field shall be set to '0' to indicate that payloads carried in all AC-4 frames that correspond to the same time period as the current output frame of the transcoding process shall be included in this output frame.

4.3.15.2.13 priority - payload priority - 5 bits

The 5-bit `priority` field is used to indicate the priority of this payload in relation to other payloads in the UMD substream. This field may be used to indicate whether a payload can be discarded during a transcoding process to a lower data rate output format that is not able to support all payloads. A `priority` value of '0' indicates that the payload has the highest priority of all payloads within the UMD substream. Higher values of the `priority` field indicate a lower payload priority. Multiple payloads within the UMD substream may have the same `priority` field value.

4.3.15.2.14 proc_allowed - processing allowed - 2 bits

The 2-bit `proc_allowed` field is used to indicate whether a payload should be retained or discarded during transcoding if processing is applied to the metadata and/or audio data decoded from the AC-4 frame during transcoding. The value of the `proc_allowed` field shall be set as shown in table 169.

Table 169: Meaning of proc_allowed values

proc_allowed	Meaning
00	Payload may be retained only if no processing and no changes to any metadata in the AC-4 frame occur during transcoding
01	Payload may be retained if no processing other than sampling frequency conversion of the PCM audio is applied during transcoding
10	Payload may be retained if one or more of the following processes, but no other processing, occur during transcoding: • sampling frequency conversion • momentary sound elements are mixed with the decoded PCM from the frame • informational metadata related to the audio data is modified • the channel configuration of the audio is changed
11	Payload may be retained regardless of any processing performed during transcoding

4.3.15.3 umd_protection - Universal metadata protection data

4.3.15.3.1 protection_length_primary - length of protection_bits_primary field - 2 bits

The 2-bit `protection_length_primary` field is used to specify the length of the `protection_bits_primary` field. The value of the `protection_length_primary` field shall be set as shown in table 170.

Table 170: Meaning of protection_length_primary values

protection_length_primary	Length of protection_bits_primary field
00	Reserved
01	8 bits
10	32 bits
11	128 bits

4.3.15.3.2 protection_length_secondary - length of protection_bits_secondary field - 2 bits

The 2-bit `protection_length_secondary` field is used to specify the length of the `protection_bits_secondary` field. The value of the `protection_length_secondary` field shall be set as shown in table 171.

Table 171: Meaning of protection_length_secondary values

protection_length_secondary	Length of protection_bits_secondary field
00	0 bits
01	8 bits
10	32 bits
11	128 bits

4.3.15.3.3 protection_bits_primary - primary UMD substream protection data - 8 to 128 bits

The `protection_bits_primary` field contains protection data that may be used to verify the integrity of the UMD substream and the payload data within the UMD substream. Calculation of the value of the `protection_bits_primary` field is implementation dependent and is not defined in the present document.

4.3.15.3.4 protection_bits_secondary - secondary UMD substream protection data - 0 to 128 bits

The `protection_bits_secondary` field, if present, contains additional optional protection data that may be used to check the integrity of the UMD substream and the payload data within the UMD substream. Calculation of the value of the `protection_bits_secondary` field is implementation dependent and is not defined in the present document.

5 Algorithmic details

5.1 Audio spectral frontend (ASF)

5.1.1 Introduction

The audio spectral frontend tool is one of two spectral frontend tools which are used to create spectral lines of audio tracks from `sf_data()` elements in an AC-4 frame. These spectral lines are typically routed through the stereo and multichannel processing tool before being routed into the IMDCT.

The ASF tool consists of three stages, namely:

- Entropy coding of spectral values
- Quantization, reconstruction and scaling
- Spectral ungrouping

The three stages are detailed in the following clauses.

Data and Control Interfaces

The input is retrieved from an `sf_data()` element in the AC-4 frame, classified to be processed by the ASF tool.

spectral_data	Huffman coded spectral data (collection of Huffman codewords and sign bits).

The output from the ASF tool is a vector of spectral lines corresponding to track *ch*:

$\mathbf{s}_{\mathrm{ASF},ch}$	vector of (scaled) spectral lines of track *ch*. This vector is named ***spec_reord*** in the spectral ungrouping tool.

NOTE: The track index *ch* is based on the occurrence of the `sf_data()` element in the frame: The 1st sf_data() element corresponds to *ch*=0, 2nd to *ch*=1, etc.

The control data needed to decode the ***spectral_data*** **is** contained in the `sf_data()` element and in the corresponding `sf_info()` element.

5.1.2 Entropy coding of spectral values

5.1.2.1 Introduction

Data and Control Interfaces

The input to the Huffman decoding tool is the `asf_spectral_data()` part of a `sf_data()` element:

spectral_data	Huffman coded spectral data (collection of Huffman codewords and sign bits). Note that the length of the spectral data is only known after the Huffman decoding process.

The output from the Huffman decoding tool is a vector of quantized spectral lines:

quant_spec	vector of quantized spectral lines.

The bitstream parameter and helper elements used by the Huffman decoding tool are:

sect_cb[g][s]	Huffman codebook number for section *s* in group *g*. This is part of the `asf_section_data()`.
CB_DIM[cb]	table indicating the Huffman codebook dimension for the Huffman codebook *cb*. See table A.14.
UNSIGNED_CB[cb]	table indicating whether Huffman codebook *cb* is a signed or unsigned codebook. See table A.15.

The Huffman codebook tables in Annex A contain more Huffman codebook specific information used by the Huffman decoder.

5.1.2.2 Decoding process

The quantized spectral lines are stored as Huffman encoded values in the bitstream. To retrieve the quantized spectral lines, Huffman decoding shall be applied.

For each Huffman codeword in the bitstream, the used Huffman codebook number `sect_cb[g][s]` is known from the section data. All the valid Huffman codebooks for the spectral lines are listed in clause A.2 through clause A.12. Each Huffman codeword `asf_qspec_hcw` can be decoded into a symbol *cb_idx* by a lookup in the corresponding Huffman codebook. The symbol is the index *cb_idx* of the codeword entry in the Huffman codebook. It determines the values of multiple quantized spectral lines. Depending on the used Huffman codebook, 2 or 4 quantized spectral lines are the result. All Huffman codewords are decoded in the order specified by the syntax of `asf_spectral_data()`, see table 40. The detailed decoding process for one symbol can be written in pseudocode:

Pseudocode

```
hcb = ASF_HCB_<sect_cb[g][s]>;
cb_idx = huff_decode(hcb, asf_qspec_hcw);

if (CB_DIM[hcb] == 4)
{
    quant_spec_1 = INT(cb_idx/cb_mod3) - cb_off;
    cb_idx -= (quant_spec_1+cb_off)*cb_mod3;
    quant_spec_2 = INT(cb_idx/cb_mod2) - cb_off;
    cb_idx -= (quant_spec_2+cb_off)*cb_mod2;
    quant_spec_3 = INT(cb_idx/cb_mod) - cb_off;
    cb_idx -= (quant_spec_3+cb_off)*cb_mod;
    quant_spec_4 = cb_idx - cb_off
}
else    // CB_DIM[hcb] == 2
{
    quant_spec_1 = INT(cb_idx/cb_mod) - cb_off;
    cb_idx -= (quant_spec_1+cb_off)*cb_mod;
    quant_spec_2 = cb_idx - cb_off;
}
```

The values *cb_mod*, *cb_mod2*, *cb_mod3* and *cb_off* are Huffman codebook specific and listed together with the Huffman codebook tables in Annex A. Huffman codewords using an unsigned Huffman codebook are followed by up to *CB_DIM[hcb]* sign bits in the bitstream. For each quantized spectral line different to 0 a sign bit is transmitted. A value of 1 indicates that the quantized spectral line is negative. No sign bits are transmitted if a signed Huffman codebook is in use.

If the Huffman codebook number *hcb* is 0 for the spectral line to be decoded, no Huffman data is stored in `asf_spectral_data` and the quantized spectral lines are set to 0.

If the Huffman codebook 11 is used and the magnitude value of at least one of the quantized spectral lines would decode to a value of 16 then additional bits are stored after the possible sign bits. For each quantized spectral line with a preliminary magnitude of 16 an extension code is used to determine the real magnitude of the corresponding quantized spectral line. First, a run-length encoded value N_{ext} in the range [0 … 8] is stored: a sequence of N_{ext} binary "1" values followed by a binary "0". Next, an unsigned extension value `ext_val` of size N_{ext}+4 bits is stored. The real magnitude of the quantized spectral line *quant_spec* is given by the following formula:

$$|\text{quant\_spec}| = 2^{N_{ext}+4} + ext\_val$$

5.1.3 Quantization reconstruction and scaling

5.1.3.1 Introduction

Data and Control Interfaces

The input to the quantization reconstruction and scaling tool is a vector of quantized spectral lines and the `asf_scalefac_data()` part of a `sf_data()` element:

quant_spec	vector of quantized spectral lines

NOTE 1: The maximum allowed value of the elements of ***quant_spec*** is 8191.

scale_factor_data	Huffman coded scale factor difference data (collection of Huffman codewords)

The output from the quantization reconstruction and scaling tool is a vector of scaled spectral lines:

scaled_spec	vector of scaled spectral lines

The bitstream parameter and helper elements used by the quantization reconstruction and scaling tool are:

reference_scale_factor	`reference_scale_factor` value as stored in `asf_scalefac_data`
sfb_cb[g][sfb]	Huffman codebook number for scale factor band *sfb* in group *g*

NOTE 2: This is a helper element derived from the `asf_section_data()`.

max_quant_idx[g][sfb]	maximum of the absolute values of the quantized spectral lines for group *g* and scale factor band *sfb*

NOTE 3: This is a helper element derived during the parsing of `asf_spectral_data()` and, if `b_hsf_ext` = 1, the spectral data part of `ac4_hsf_ext_substream()`.

max_sfb_in_group[g]	vector indicating the maximum transmitted scale factor band in group *g*

5.1.3.2 Decoding process

To convert the quantized spectral lines into scaled spectral lines a quantization reconstruction and scaling step shall be applied.

The following (non-uniform) reconstruction process shall be used to apply the inverse operation to the non-uniform quantization operation used by the encoder for the spectral lines. The reconstruction stage operates on the output of the Huffman decoder (i.e. on the quantized spectral lines). A reconstructed spectral line *rec_spec* is calculated from the quantized spectral line *quant_spec* according to the following formula:

$$rec\_spec = sign(quant\_spec) \times |quant\_spec|^{\frac{4}{3}}$$

This reconstruction process shall be done for all spectral lines.

The coded spectrum is divided into scale factor bands. Each scale factor band contains spectral lines which are scaled using a common scale factor gain *sf_gain[g][sfb]*. A scaled spectral line *scaled_spec*, which belongs to scale factor band *sfb* in group *g*, is calculated from the reconstructed spectral coefficient *rec_spec* according to the formula:

$$scaled\_spec = sf\_gain[g][sfb] \times rec\_spec$$

No scale factor is transmitted for scale factor bands that have only zero-valued spectral lines as either indicated by the use of the Huffman codebook 0 for the scale factor band or by the fact that all quantized spectral lines within the scale factor band have an absolute value of zero. The transmitted scale factor values are Huffman coded values of the difference to the previously transmitted scale factor value or to the *reference_scale_factor* for the first transmitted scale factor difference. The used Huffman codewords for the scale factor difference values are given in table A.1. To get the value of the nth transmitted scale factor sf_n, the following formula shall be used:

$$sf_n = sf_{(n-1)} + cw\_idx_n - 60$$

where:

- *cw_idx*$_n$ is the corresponding index in table A.1 to the nth codeword from the bitstream;
- *sf*$_{(n-1)}$ is the previous scale factor value (or the *reference_scale_factor* in case n = 1);
- 60 is the index offset to be used in connection with the scale factor codebook.

NOTE: Only scale factor values sf_n in the range 0 to 255 are valid scale factor values.

To get a scale factor gain k_{sf} from a scale factor value *sf*, the following formula shall be used:

$$k_{sf} = 2^{0,25 \cdot (sf-100)}$$

All decoded scale factor gain values k_{sf} are stored in *sf_gain[g][sfb]* during the decoding of the `asf_scalefac_data()` element. The following pseudocode, which takes the bitstream grouping into account, shows this.

Pseudocode

```
scale_factor = reference_scale_factor;

first_scf_found = 0;
for (g = 0; g < num_window_groups; g++)
{
    for (sfb = 0; sfb < max_sfb_in_group[g]; sfb++)
    {
        if (sfb_cb[g][sfb] != 0)
        {
            if (max_quant_idx[g][sfb] > 0) {
                if (first_scf_found == 1) {
                    dpcm_sf[g][sfb] = huff_decode(ASF_HCB_SCALEFAC, asf_sf_hcw);
                    scale_factor += dpcm_sf[g][sfb] - 60;
                else {
                    first_scf_found = 1;
                }
                sf_gain[g][sfb] = pow (2.0, 0.25 * (scale_factor - 100));
            }
        }
    }
}
```

5.1.4 Spectral noise fill

5.1.4.1 Introduction

Data and Control Interfaces

The input to and the output from the spectral noise fill tool is a vector of spectral lines in bitstream order:

scaled_spec	vector of (scaled) spectral lines in bitstream order

The bitstream parameter and helper elements used by the spectral noise fill tool are:

num_window_groups	number of window groups

NOTE 1: This is an `sf_info` helper element.

num_win_in_group[g]	vector indicating the number of windows in group *g*
sfb_offset[sfb]	vector indicating the scale factor band offset for scale factor band *sfb*

The matching *sfb_offset[sfb]* vector for the given sampling frequency and transform length related to the current window shall be used. All possible *sfb_offset[sfb]* vectors are listed as tables in Annex B.

sfb_cb[g][sfb]	Huffman codebook number for scale factor band *sfb* in group *g*

NOTE 2: This is a helper element derived from the `asf_section_data()`.

max_quant_idx[g][sfb]	maximum of the absolute values of the quantized spectral lines for group *g* and scale factor band *sfb*

NOTE 3: This is a helper element derived during the parsing of `asf_spectral_data()` and, if `b_hsf_ext` = 1, the spectral data part of `ac4_hsf_ext_substream()`.

max_sfb_in_group[g]	vector indicating the maximum transmitted scale factor band in group *g*

5.1.4.2 Decoding process

Spectral lines in ***scaled_spec*** which are 0 because either the corresponding *sfb_cb[g][sfb]* is 0 or the corresponding *max_quant_idx[g][sfb]* is 0 shall be replaced by random noise of a transmitted level if noise fill data is present as indicated by `b_snf_data_exists`=1. If `b_snf_data_exists`=0, the spectral noise fill tool is not active and ***scaled_spec*** is not modified.

The first step in decoding the spectral noise fill data is to find the initial noise fill reference level. This is the first decoded band that has at least one non-zero component. The following pseudocode shows this process.

Pseudocode

```
k = 0;
previous_rms = -1000;   /* init to a suitably large negative value */

for (g = 0; g < num_window_groups; g++) {
    for (sfb = 0; sfb < max_sfb_in_group[g]; sfb++) {
        band_rms = 0;
        for (w = 0; w < num_win_in_group[g]; w++) {
            for (l = sfb_offset[sfb]; l < sfb_offset[sfb+1]; l++) {
                band_rms += scaled_spec[k]* scaled_spec[k];
                k++;
            }
        }
        if (band_rms > 0) {
            band_rms /= num_win_in_group[g] * (sfb_offset[sfb+1] - sfb_offset[sfb]);
            previous_rms = 1.44269504 * log(band_rms);
            break;
        }
    }
    if (previous_rms != -1000)
        break;
}
```

Next, the noise fill delta Huffman codewords are read from the bitstream, decoded and the appropriate noise fill levels are computed. This decoding process and the insertion of the noise signal are shown in the following pseudocode.

Pseudocode

```
k = 0;

for (g = 0; g < num_window_groups; g++) {
    for (sfb = 0; sfb < max_sfb_in_group[g]; sfb++) {
        band_rms = 0;
        for (w = 0; w < num_win_in_group[g]; w++) {
            for (l = sfb_offset[sfb]; l < sfb_offset[sfb+1]; l++) {
                band_rms += scaled_spec[k]* scaled_spec[k];
                k++;
            }
        }
        if (band_rms > 0) {
            band_rms /= num_win_in_group[g] * (sfb_offset[sfb+1] - sfb_offset[sfb]);
            previous_rms = 1.44269504 * log(band_rms);
        }
        else {  /* noise fill band */
            dpcm_snf[g][sfb] = huff_decode(ASF_HCB_SNF, asf_snf_hcw);
            delta = dpcm_snf[g][sfb] - 17;
            if (delta != -17) {
                /* -17 is an escape for no noise fill */
                /* and the relative level is NOT updated */
                noise_rms = previous_rms + delta;
                previous_rms = noise_rms;

                noise_rms = pow(2.0, 0.5*noise_rms);
                k -= num_win_in_group[g] * (sfb_offset[sfb+1] - sfb_offset[sfb])
                for (w = 0; w < num_win_in_group[g]; w++) {
                    for (l = sfb_offset[sfb]; l < sfb_offset[sfb+1]; l++) {
                        noise = GetRandomNoiseValue(&nRndStateSnf) * noise_rms;
                        scaled_spec[k++] = noise;
                    }
                }
            }
        }
    }
}
```

The same *GetRandomNoiseValue()* function as in the Speech Spectral Frontend is used, but with an own state, *nRndStateSnf*. This function returns a normal distributed random number with unit variance and zero mean. The random number generator is initialized at the beginning of the decoding of an ASF frame, using the `sequence_counter` value, by calling the function *ResetRandGenStateSnf(&nRndStateSnf, sequence_counter)*. This function is described in the following pseudocode.

Pseudocode

```
// initialize / reset the random number generator for spectral noise fill

void ResetRandGenStateSnf(ssf_rndgen_state *psS, int sequence_counter)
{
    int stateIdx;
    int currentIdx;
    int start = 255 * (sequence_counter % 256);

    psS->uiOffsetA = start % 255;
    psS->uiOffsetB = (start / 255) % 256;
    stateIdx = (psS->uiOffsetA+1)*((psS->uiOffsetA+2)+2*psS->uiOffsetB)/2
                + 255*psS->uiOffsetB *(255+psS->uiOffsetB)/2;
    if (start % 130560 >= 65280)
        stateIdx += 128;
    psS->uiStateIdx = stateIdx % 256;
    currentIdx = psS->uiOffsetA*(psS->uiOffsetA+1)/2 + psS->uiOffsetB*32386;
    psS->uiCurrentIdx = currentIdx % 256;
}
```

5.1.5 Spectral ungrouping tool

5.1.5.1 Introduction

Data and Control Interfaces

The input to the ungrouping tool is a vector of spectral lines in bitstream order:

scaled_spec	vector of (scaled) spectral lines in bitstream order

The output from the ungrouping tool is a vector of reordered spectral lines:

spec_reord	vector of reordered (scaled) spectral lines, i.e. ordered according to the window number and ascending within a window

The bitstream parameters used by the ungrouping tool are:

num_window_groups	number of window groups

NOTE: This is an `sf_info` helper element.

num_win_in_group[g]	vector indicating the number of windows in group *g*
sfb_offset[sfb]	vector indicating the scale factor band offset for scale factor band *sfb*

The matching *sfb_offset[sfb]* vector for the given sampling frequency and transform length related to the current window shall be used. All possible *sfb_offset[sfb]* vectors are listed as tables in Annex B.

win_offset[win]	vector indicating the offset to the start of window *win* in the reordered output

5.1.5.2 Decoding process

When more than one transform block is used in an AC-4 frame, spectral lines are stored in groups. To undo this grouping, the AC-4 decoder shall apply reordering as specified in the following pseudocode:

Pseudocode

```
k = 0;
win = 0;
spec_reord[] = 0;   /* init all spectral lines with 0 */

for (g = 0; g < num_window_groups; g++) {
    max_sfb = get_max_sfb(g);
    for (sfb = 0; sfb < max_sfb; sfb++) {
        for (w = 0; w < num_win_in_group[g]; w++) {
            for (l = sfb_offset[sfb]; l < sfb_offset[sfb+1]; l++) {
                spec_reord[win_offset[win+w] + l] = scaled_spec[k++];
            }
        }
    }
    win += num_win_in_group[g];
}
```

5.2 Speech spectral frontend (SSF)

5.2.1 Introduction

The Speech Spectral Frontend (SSF) is an alternative spectral frontend to the Audio Spectral Frontend (ASF). It is designed to be used for speech content. Like the ASF, the SSF will provide blocks of spectral lines which need to be inverse transformed using the same inverse MDCT as for the ASF output. The SSF data which is part of an AC-4 frame is subdivided into one or two parts, called SSF granules.

Each SSF granule is either independently decodable and called an SSF-I-frame or the SSF granule relies on decoded data from previous SSF granules and is called a SSF-P-frame. The decoding of SSF data can only start with an I-frame.

There are two possible operation modes of the SSF decoder: the so called SHORT_STRIDE mode, where the SSF granule is subdivided into 4 blocks and the so called LONG_STRIDE mode in which the SSF granule contains just one block. One feature of the SHORT_STRIDE mode is that it includes an envelope interpolation.

Data and Control Interfaces

The input is retrieved from an `sf_data()` element in the AC-4 frame, classified to be processed by the SSF tool.

speech_data	envelope and gain indices encoded in the bitstream, as well as decoded envelope values related to the previous frame - in case of SSF-P-frames

The output from the SSF tool is a vector of spectral lines:

$\mathbf{s}_{\mathrm{SSF},ch}$	vector of *n_mdct* (scaled) spectral lines of track *ch*

NOTE: The track index *ch* is based on the occurrence of the `sf_data()` element in the frame: The 1st `sf_data()` element corresponds to *ch*=0, 2nd to *ch*=1, etc.

The control data needed to decode the ***speech_data*** is contained in the `sf_data()` element.

5.2.2 Top level structure of the SSF

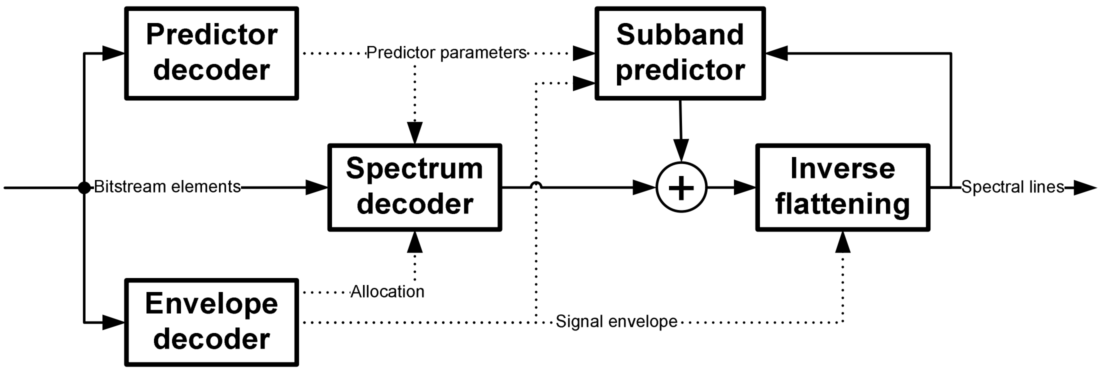

Figure 4: SSF decoder block diagram

Figure 4 shows the SSF decoder block diagram. The predictor decoder, the spectrum decoder and the envelope decoder have bitstream elements as input. The output of the spectrum decoder is combined (added) with the output of the subband predictor and sent to the inverse flattening which applies the signal envelope from the envelope decoder. The signal after the inverse flattening, filled with additional zero spectral lines to provide *n_mdct* spectral lines, is the output signal of the SSF. This signal is also the input of the subband predictor which uses buffered SSF output signals, the predictor parameters from the predictor decoder and the signal envelope from the envelope decoder to generate the subband predictor output. The envelope allocation and the predictor parameters are additional inputs of the spectrum decoder.

Each block of the SSF decoder block diagram is explained in more detail in the following clauses.

5.2.3 Envelope decoder

Data and Control Interfaces

The input to the envelope decoder is a vector of envelope indices, a vector of gain indices and a vector decoded envelope values from the previous frame:

env_idx[band]	vector of *num_bands* envelope indices

NOTE 1: The first index describes an absolutely coded value of the first band while the remaining indices describe differentially coded envelope values.

gain_idx[block]	vector of *num_blocks* gain indices
env_prev[band]	vector of *num_bands* decoded envelope values related to the previous frame

NOTE 2: For SSF-P-frames this envelope would be the ***env[band]*** envelope from the previous frame. For SSF-I-frames this will be the decoded envelope indices related to the extra startup envelope that is transmitted in the SHORT_STRIDE case.

The outputs of the envelope decoder are two arrays:

env_alloc[block][band]	*num_blocks* × *num_bands* dimensional array of envelope allocation values (integer)
f_env_signal[block][band]	*num_blocks* × *num_bands* dimensional array of signal envelope values (float)

The bitstream parameters used by the envelope decoder are:

num_bands	number of bands
num_blocks	number of blocks. *num_blocks* = 1 for the LONG_STRIDE mode and *num_blocks* = 4 for the SHORT_STRIDE mode

The following intermediate signals are used by the envelope decoder:

env[band]	vector of *num_bands* decoded envelope values
env_interp[block][band]	*num_blocks* × *num_bands* dimensional array of interpolated envelope values
gain[block]	vector of *num_blocks* decoded gain values

Decoding of *env_idx[band]* into *env[band]*

Pseudocode

```
// decode envelope
ENV_DELTA_MIN = -16;
ENV_BAND0_MIN = -28;

env[0] = env_idx[0] + ENV_BAND0_MIN;
for (band = 1; band < num_bands; band++) {
    delta = env_idx[band] + ENV_DELTA_MIN;
    env[band] = env[band-1] + delta;
}
```

NOTE: Valid env[band] entries are in the range -64,…,63.

Envelope interpolation

In case of SHORT_STRIDE envelope interpolation shall be done. In case of LONG_STRIDE no interpolation is needed and ***env_interp[0][band]*** = ***env[band]***. The envelope interpolation is described in the following pseudocode.

Pseudocode

```
// interpolate envelope (SHORT_STRIDE)

const int32 iUnit = 1024;                                     /* Q.10 */
const int32 iHalf = 512;                                      /* Q.10 */
const int32 iInvNumBlocks = 256;                              /* Q.10 */

iNumLeftBlocks = SSF_I16_MUL_I16(4U, iUnit);                  /* Q3.0*Q0.10=Q3.10  */
for (band = 0; band < num_bands; band++) {
    iLeftDelta = SSF_I16_MUL_I16((env[band] - env_prev[band]), iUnit);  /* Q7.10     */
    iLeftSlope = SSF_I16_MUL_I16(iLeftDelta, iInvNumBlocks);    /* Q7.10*Q0.10=Q7.20 */
    iLeftSlope = SSF_I32_SHIFT_RIGHT(iLeftSlope, 10);           /* Q7.10             */
    for (block = 0; block < num_blocks; block++) {
        iInterpEnv = SSF_I16_MUL_I16((1 + block), iLeftSlope);  /* Q3.0*Q7.10=Q10.10 */
        iTmp = SSF_I16_MUL_I16(env_prev[band], iUnit));
        iInterpEnv = SSF_I32_ADD_I32(iInterpEnv, iTmp);         /* Q10.10            */
        if (iInterpEnv > 0) {
            iInterpEnv = SSF_I32_ADD_I32(iInterpEnv, iHalf);
        } else {
            iInterpEnv = SSF_I32_SUB_I32(iInterpEnv, iHalf);
        }
        env_interp[block][band] = SSF_I32_SHIFT_RIGHT(iInterpEnv, 10);  /* Q.0       */
    }
}
```

NOTE: The used fixed point operators are defined in clause 5.2.8.2.

Decoding of *gain_idx[block]* into *gain[block]*

In case of SHORT_STRIDE gain indices are transmitted for each of the 4 blocks. The following pseudocode shows how the gain indices are converted into gains.

Pseudocode

```
// gain_idx to gain conversion

for (block = 0; block < 4; block++) {
    gain[block] = pow(10.0f, gain_idx[block] * 0.1f);
}
```

For LONG_STRIDE, *gain_idx[0]* = 0 and *gain[0]* = 1.

Envelope refinement

Pseudocode

```
SSF_HIGH_FREQ_GAIN_THRESHOLD = 2;
ENV_MIN = -64;
ENV_MAX = 63;

// signal envelope
for (block = 0; block < num_blocks; block++) {
    for (band = 0; band < num_bands; band++) {
        // envelope reconstruction
        f_env_signal[block][band] = pow(2.0f, 0.5f * env_interp[block][band]);
        if (band >= SSF_HIGH_FREQ_GAIN_THRESHOLD) {
            // apply gain
            f_env_signal[block][band] *= gain[block];
        }
    }
}

// allocation envelope
for (block = 0; block < num_blocks; block++) {
    for (band = 0; band < num_bands; band++) {
        env_alloc[block][band] = env_interp[block][band];
        if (band >= SSF_HIGH_FREQ_GAIN_THRESHOLD) {
            // apply gain
            env_alloc[block][band] += round(2.0f * gain_idx[block] / 3.0f);
            // limit envelope
            if (env_alloc[block][band] < ENV_MIN)
                env_alloc[block][band] = ENV_MIN;
            if (env_alloc[block][band] > ENV_MAX)
                env_alloc[block][band] = ENV_MAX;
        }
    }
}
```

5.2.4 Predictor decoder

Data and Control Interfaces

The inputs to the predictor decoder are SSF bitstream elements.

The outputs from the predictor decoder are the following predictor parameters:

f_pred_gain[block]	vector of *num_blocks* predictor gain values
f_pred_lag[block]	vector of *num_blocks* predictor lag values

The state internal to the predictor decoder is the predictor lag index:

i_prev_pred_lag_idx	predictor lag index of the previous block

Pseudocode

```
// decode predictor parameters for block index block

PRED_LAG_DELTA_MIN = -8;

if ((block >= start_block) && (block < end_block)) {
    // predictor parameters are possible
    if (predictor_presence_flag[block] == 1) {
        i_pred_gain_idx[block] = arithmetic_decode_pred();
        // reconstruct
        f_pred_gain[block] = PRED_GAIN_QUANT_TAB[i_pred_gain_idx[block]];
        if (delta_flag[block] == 1) {
            i_pred_lag_idx[block] = predictor_lag_delta_bits[block];
            i_pred_lag_idx[block] += i_prev_pred_lag_idx + PRED_LAG_DELTA_MIN;
        }
        else {
            i_pred_lag_idx[block] = predictor_lag_bits[block];
        }
    }
    else {
        f_pred_gain[block] = 0.0f;
        i_pred_lag_idx[block] = 0;
    }
}
else {
    //reset predictor parameters
    f_pred_gain[block] = 0.0f;
    i_pred_lag_idx[block] = 0;
}
// update i_prev_pred_lag_idx for next block
i_prev_pred_lag_idx = i_pred_lag_idx[block];
// reconstruct
f_pred_lag[block] = 640*pow(2,(i_pred_lag_idx[block]-509)/170);
```

NOTE 1: PRED_GAIN_QUANT_TAB is defined in table C.3.
NOTE 2: Valid i_pred_lag_idx[block] entries are in the range 0,…,509.

5.2.5 Spectrum decoder

Data and Control Interfaces

The inputs to the spectrum decoder are envelope allocation values and the predictor gain for the current block:

env_alloc[band]	vector of *num_bands* envelope allocation values (integer) for current block
f_pred_gain	predictor gain value for the current block
block	block index of the current block

The output from the spectrum decoder is a vector of residual spectral lines:

f_spec_res[bin]	vector of *num_bins* residual spectral lines

The bitstream parameters used by the spectrum decoder are:

num_bands	number of bands
num_bins	number of coded spectral lines
n_mdct	SSF block length
alloc_offset_bits	value of bitstream element `alloc_offset_bits` for the current block

First, some helper variables are calculated.

Pseudocode

```
// compute helper variables f_rfu, i_alloc_dithering_threshold, f_adaptive_noise_gain
// and f_adaptive_noise_gain_var_pres for current block

RFU_THRESHOLD = 0.75f;
ALLOC_DITHERING_THRESHOLD_SMALL = 3;
ALLOC_DITHERING_THRESHOLD_LARGE = 5;

f_gain = f_pred_gain;
if (f_gain < -1.0f)
    f_rfu = 1.0f;
else if (f_gain < 0.0f)
    f_rfu = -f_gain;
else if (f_gain < 1.0f)
    f_rfu = f_gain;
else if (f_gain < 2.0f)
    f_rfu = 2.0f - f_gain;
else // f_gain >= 2.0f
    f_rfu = 0.0f;

if (f_rfu > RFU_THRESHOLD)
    i_alloc_dithering_threshold = ALLOC_DITHERING_THRESHOLD_SMALL;
else
    i_alloc_dithering_threshold = ALLOC_DITHERING_THRESHOLD_LARGE;

if (variance_preserving_flag == 1)
    i_alloc_dithering_threshold = ALLOC_DITHERING_THRESHOLD_LARGE;
f_adaptive_noise_gain_var_pres = sqrt(1 - (f_rfu * f_rfu));
f_adaptive_noise_gain = 1 - f_rfu;
```

The heuristic scaling and the modification of the envelope allocation, which shall be done if `variance_preserving_flag` is '0' and *f_rfu*>0, are shown in the following pseudocode. If `variance_preserving_flag` is '1', *env_alloc_mod[band]* is simply set to *env_alloc[band]* and *f_gain_q[band]* is set to 1.0.

If *f_rfu*=0, the *env_alloc_mod[band]* is set to *env_alloc[band]* and *f_gain_q[band]* is set to 1.0 regardless of the value of `variance_preserving_flag`.

Pseudocode

```
// Heuristic scaling and envelope allocation modification

ENV_MIN = -64;
ENV_MAX = 63;

for (band = 0; band < num_bands; band++) {
    env_in[band] = 3 * env_alloc[band];
}

/* compute heuristic scaling */
HeuristicScaling(iRfu,              // input: rfu parameter in Qx.10
                 env_in,            // input: 1dB envelope in Qx.0
                 int_weights_dB);   // output in Qx.10

for (band = 0; band < num_bands; band++) {
    i_w_dB[band] = Q_10_to_Q0(int_weights_dB[band] / 2);    // convert result to Q.0
}

/* LF-boost */
const int lf_boost_threshold = 3;
i_w_dB[0] = (i_w_dB[0] > lf_boost_threshold) ? i_w_dB[0] - lf_boost_threshold : 0;

for (band = 0; band < num_bands; band++) {
    // store gains for Heuristic inverse scaling
    f_w_dB = FLOAT(int_weights_dB[band]);           // conversion from Qx.10 to float
    f_gain_q[band] = pow(10.0f, 1.5f / 20.0f * f_w_dB);

    // modify allocation - apply heuristic scaling
    env_alloc_mod[band] = env_alloc[band] - i_w_dB[band];

    // limit allocation envelope to [-64,63]
    if (env_alloc_mod[band] < ENV_MIN)
        env_alloc_mod[band] = ENV_MIN;
    if (env_alloc_mod[band] > ENV_MAX)
        env_alloc_mod[band] = ENV_MAX;
}
```

The function that performs the heuristic scaling based on a fix point representation of the *f_rfu* value and the allocation envelope is implemented as below.

Pseudocode

```
// HeuristicScaling()
{
    /* Inputs: iRfu in Qx.10; env_in[] in Q.0 */
    /* Output: int_weights_dB[]               */

    /* constant values in Q.10 */
    iDynThreshold = SSF_I32_SHIFT_LEFT(40, 10);
    iMaxiWdB = SSF_I32_SHIFT_LEFT(15, 10);
    iInvThree = 341;

    /* compute max{ env_in } and min{ env_in } */
    iMaxEnv = VECMAX(env_in, num_bands);/* find the maximum of the envelope vec. */
    iMinEnv = VECMIN(env_in, num_bands);/* find the minimum of the envelope vec. */
    iDynUnscaled = iMaxEnv-iMinEnv;
    iDyn = SSF_I32_SHIFT_LEFT((iMaxEnv-iMinEnv), 10);   /* result in Q.10 */

    if (iDyn > iDynThreshold) {
        iCmpFact = iDynThreshold / iDynUnscaled;        /* result in Q.10 */
        for (band = 0; band < num_bands; band++) {
            env_local[band] = SSF_I32_SUB_I32(env_in[band], iMinEnv);
            env_local[band] = SSF_I16_MUL_I16(env_local[band], iCmpFact);
        }
    } else {
        /* make a local copy of the envelope */
        for (band = 0; band < num_bands; band++) {
            env_local[band] = SSF_I32_SUB_I32(env_in[band], iMinEnv);   /* Qx.0  */
            env_local[band] = SSF_I32_SHIFT_LEFT(env_local[band], 10);  /* Q6.10 */
        }
    }
    /* sort the env_local in descending order                          */
    /* env_local_sorted is a vector with sorted envelope values (Q6.10)    */
    /* env_indices are such as env_local_sorted = env_local[env_indices]   */
    Sort(env_local, env_local_sorted, env_indices);
    /* convert the sorted envelope to a linear domain                      */
    iMtr = 0;
    for (band = 0; band < num_bands; band++) {
        /* conversion from dB domain to the log domain is based on a look-up table */
        weights_lin[band] = Map_dB_to_Lin(env_local_sorted[band]);      /* Q7.10 */
        iTmp = SSF_I16_MUL_I16(weights_lin[band],
                               band_widths[env_indices[band]);          /* Q12.10 */
        iMtr = SSF_I32_ADD_I32(iMtr, iTmp);                     /* Q13.10 + Q13.10 */
    }
    iMtr = SSF_I32_SHIFT_RIGHT(iMtr, 10);
    iMtr = SSF_I16_MUL_I16(iMtr, iRfu);
    iMtr = SSF_I32_SHIFT_RIGHT(iMtr, 7);
    iMtr = SSF_I16_MUL_I16(iMtr, iRfu);
    iMtr = SSF_I32_SHIFT_RIGHT(iMtr, 3);
    /* reverse water-filling procedure */
    iNum = 0; iMnt = 0; iBsum = 0;
    while ((iMnt < iMtr) && (band < (num_bands-1))) {
        iTCurrLev = weights_lin[band];                                  /* Q7.10  /
        while ((weights_lin[band] == iTCurrLev) && (band < (num_bands-1))) {
            iBsum = SSF_I32_ADD_I32(iBsum, band_widths[env_indices[band]);
            band = band + 1;
        }
        iTmp2 = SSF_I32_SUB_I32(iTCurrLev, weights_lin[band];           /* Q7.10  */
        iTmp2 = SSF_I16_MUL_I16(iTmp2, iBsum);          /* Q7.10 * Q10.0 = Q17.10 */
        iMnt = SSF_I32_ADD_I32(iTmp2, iMnt);                            /* Q17.10 */
    }
    if (iMnt < iMtr) iBsum = num_bins;
    iTmp = SSF_I32_SUB_I32(iMnt, iMtr);                                 /* Q17.10 */
    iTmp = SSF_I32_SHIFT_LEFT(iTmp, 4);
    iTmp2 = iTmp / iBsum;
    iTmp2 = SSF_I32_SHIFT_RIGHT(iTmp2, 4);
    iTCurrLev = SSF_I32_ADD_I32(weights_lin[band], iTmp2);

    for (band = 0; band < num_bands; band++) {
        iTmp2 = Map_Lin_to_dB(iTCurrLev);                               /* Q6.10  */
        iTmp = SSF_I32_SUB_I32(env_local[band], iTmp2);                 /* Q10.10 */
        iTmp = SSF_I16_MUL_I16(iTmp, iInvThree);                        /* Q10.20 */
        iTmp = SSF_I32_SHIFT_RIGHT(iTmp, 10);                           /* Qx.10  */
        iTmp = iTmp > 0 ? iTmp : 0;
        iTmp = iTmp < iMaxiWdB ? iTmp : iMaxiWdB;
        int_weights_dB[band] = iTmp;                        /* final result Q10.10 */
    }
}
```

The above implementation uses two dedicated functions for an approximate conversion between the linear scale and the dB scale. The conversion from dB values to linear values is implemented by the following function:

Pseudocode

```
int32 Map_dB_to_Lin(int32 iInput    // input Qx.10
                    )               // result in Qx.10
{
    int32 iRes, iInt, iIndex, iFract;

    /* this function works with input Q.4, therefore we need a shift */
    iInput = SSF_I32_SHIFT_RIGHT(iInput, 6);    /* Qx.4 */

    /* figure out interval */
    iIndex = SSF_I32_SHIFT_RIGHT(iInput, 6);    /* Qx.0 */

    if (iIndex < 10) {
        iRes = SSF_I16_MUL_I16(SLOPES_DB_TO_LIN[iIndex], iInput);   /* use LUT1 */
        /* Q8.4 * max Q11.4 = Q(8+11).8, no overflow by design of input */
        iRes = SSF_I32_SHIFT_RIGHT(iRes, 4);
        iRes = SSF_I32_ADD_I32(iRes, OFFSETS_DB_TO_LIN[iIndex]);    /* use LUT2 */
        iRes = SSF_I32_SHIFT_LEFT(iRes, 6);     /* Qx.10 */
    } else {
        /* index out of range */
        iRes = SSF_I32_SHIFT_LEFT(100U, 10);    /* Qx.10 */
    }
    return iRes;
}
```

NOTE: SLOPES_DB_TO_LIN is specified in table C.13 and OFFSETS_DB_TO_LIN in table C.14.

The conversion from linear values to dB values is implemented by the following function:

Pseudocode

```
int32 Map_Lin_to_dB(int32 iInput    // input Qx.10
                    )               // result in Q7.10
{
    int32 iRes, iQuantIn, iIndex, iFract, iInt;
    int32 iTmp1, iTmp2;
    const int32 iMaxTableSize = 50;

    iInput = SSF_I32_SHIFT_RIGHT(iInput, 2);                               // Qx.8
    iQuantIn = SSF_I32_SHIFT_RIGHT(iInput, 1);                             // Qx.8
    iIndex = SSF_I32_SHIFT_RIGHT(iQuantIn, 8);                             // Qx.0
    iInt = SSF_I32_SHIFT_LEFT(iIndex, (8 + 1));
    iFract = SSF_I32_SUB_I32(iInput, iInt);
    if (iIndex < iMaxTableSize) {
        iTmp2 = SSF_I32_SHIFT_LEFT(iIndex, 1);                             // Qx.0
        // access LUT3
        iTmp1 = SSF_I16_MUL_I16(SLOPES_LIN_TO_DB[iIndex], iTmp2);          // Q11.8
        iTmp2 = SSF_I16_MUL_I16(SLOPES_LIN_TO_DB[iIndex], iFract);         // Q11.16
        iTmp2 = SSF_I32_SHIFT_RIGHT(iTmp2, 8);
        iRes = SSF_I32_ADD_I32(iTmp1, iTmp2);
        // access LUT4
        iRes = SSF_I32_ADD_I32(iRes, OFFSETS_LIN_TO_DB[iIndex]);           // Q11.8
        iRes = SSF_I32_SHIFT_LEFT(iRes, 2);
    } else {
        iRes = SSF_I32_SHIFT_LEFT(40, 10);                                 // Qx.10
    }
    return iRes;    // actually the max here is Q6.10
}
```

NOTE: SLOPES_LIN_TO_DB is specified in table C.15 and OFFSETS_LIN_TO_DB in table C.16.

Next is the lossless decoding step.

Pseudocode

```
// Lossless decoding

MIN_ALLOC_OFFSET = -21;
ENV_MAX_2_MIN_OFFSET = 20;

i_alloc_offset = alloc_offset_bits + MIN_ALLOC_OFFSET;

// find max entry in env_alloc_mod[band]
i_max = env_alloc_mod[0];
for (band = 1; band < num_bands; band++) {
    if (env_alloc_mod[band] > i_max)
        i_max = env_alloc_mod[band];
}
i_max -= ENV_MAX_2_MIN_OFFSET;

// setup i_alloc_table[band]
for (band = 0; band < num_bands; band++) {
    i_alloc_table[band] = env_alloc_mod[band] - i_max + i_alloc_offset;
    if (i_alloc_table[band] < 0)
        i_alloc_table[band] = 0;
    if (i_alloc_table[band] > 20)
        i_alloc_table[band] = 20;
}

// setup i_dither[bin]
for (band = 0; band < num_bands; band++) {
    for (bin = start_bin[band]; bin <= end_bin[band]; bin++) {
        if ((i_alloc_table[band] != 0) &&
            (i_alloc_table[band] < i_alloc_dithering_threshold))
            i_dither[bin] = i_dither_cur[block][bin];
        else
            i_dither[bin] = 0;
    }
}

arithmetic_decode_coeffs();
// This function uses i_alloc_table[band] and i_dither[bin]
// and outputs i_quant_idx[bin]. See clause 5.2.8.2.
```

The inverse quantization step is shown in the following pseudocode.

Pseudocode

```
// Inverse quantization (for block index block)

/* important variables for bit-exact processing */
int32 i_step_size, i_mid_point, i_dither, i_quant_index;
const int32 i_model_unit = (1U << 15);

for (band = 0; band < num_bands; band++)
{
    i_alloc = i_alloc_table[band];

    /* get the step size */
    i_step_size = STEP_SIZES_Q4_15[i_alloc];

    for (bin = start_bin[band]; bin <= end_bin[band]; bin++)
    {
        if (i_alloc == 0) {
            f_spec_invq[bin] = GetRandomNoiseValue(&nRndState);
            if ((variance_preserving_flag[block] == 1) && (band > 1)) {
                f_spec_invq[bin] *= f_adaptive_noise_gain_var_pres;
            } else {
                f_spec_invq[bin] *= f_adaptive_noise_gain;
            }
        }
        else {
            i_quant_index = i_quant_idx[bin];
            if (i_alloc < i_alloc_dithering_threshold) {
                i_dither = i_dither_cur[block][bin];
                i_mid_point = Idx2Reconstruction(i_quant_index, i_dither, i_step_size);
                f_mid_point = (float)i_mid_point / (float)i_model_unit;
                f_post_gain = POST_GAIN_LUT[i_alloc-1];
                if ((variance_preserving_flag[block] == 1) && (band > 1)) {
                    f_post_gain_var_pres = sqrt(f_post_gain) *
                                           f_adaptive_noise_gain_var_pres;
                    if (f_post_gain_var_pres > f_post_gain) {
                        f_post_gain = f_post_gain_var_pres;
                    }
                }
                f_spec_invq[bin] = f_mid_point * f_post_gain;
            }
            else { // quantizers with no dither
                i_mid_point = Idx2Reconstruction(i_quant_index, 0, i_step_size);
                f_mid_point = (float)i_mid_point / (float)i_model_unit;
                f_step_size = (float)i_step_size/(float)i_model_unit;
                f_spec_invq[bin] = MmseLaplace(f_mid_point, f_step_size);
            }
        }
    }
}
```

NOTE: POST_GAIN_LUT[ ] is defined in table C.2 and STEP_SIZES_Q4_15[ ] in table C.11. GetRandomNoiseValue() is defined in clause 5.2.8.3. nRndState is the state of the random noise generator.

The used function *MmseLaplace()* is defined below.

Pseudocode

```
float MmseLaplace(float f_mid_point, float f_step_size)
{
    f_upper = f_mid_point + f_step_size / 2.0f;
    f_lower = f_mid_point - f_step_size / 2.0f;

    f_pdf_lower = sqrt(2) / 2 * exp(-fabs(f_lower) * sqrt(2));
    f_pdf_upper = sqrt(2) / 2 * exp(-fabs(f_upper) * sqrt(2));

    // Prevent numerical problems
    if (f_pdf_lower < 0.0f)
        f_pdf_lower = 0.0f;
    if (f_pdf_upper < 0.0f)
        f_pdf_upper = 0.0f;

    f_mmse_n  = f_pdf_lower*(sqrt(2)*f_lower-1)+f_pdf_upper*(sqrt(2)*f_upper+1);
    f_mmse_n /= sqrt(2)*(f_pdf_lower+f_pdf_upper)-2;

    if (f_lower > 0) {
        f_mmse_n  = f_pdf_upper*(sqrt(2)*f_upper+1.0f);
        f_mmse_n -= f_pdf_lower*(sqrt(2)*f_lower+1.0f);
        f_mmse_n /= sqrt(2)*(f_pdf_upper-f_pdf_lower);
    }

    if (f_upper < 0) {
        f_mmse_n  = f_pdf_upper*(sqrt(2)*f_upper-1.0f);
        f_mmse_n -= f_pdf_lower*(sqrt(2)*f_lower-1.0f);
        f_mmse_n /= sqrt(2)*(f_pdf_upper-f_pdf_lower);
    }

    return f_mmse_n;
}
```

Finally, the heuristic inverse scaling shall be applied if `variance_preserving_flag` is '0'.

Pseudocode

```
// Heuristic inverse scaling

for (band = 0; band < num_bands; band++) {
    f_gain_value = 1.0f / f_gain_q[band];
    for (bin = start_bin[band] ; bin <= end_bin[band]; bin++) {
        f_spec_res[bin] = f_spec_invq[bin] * f_gain_value;
    }
}
```

5.2.6 Subband predictor

Data and Control Interfaces

The input to the subband predictor is a vector of SSF output spectral lines from the previous block, a signal envelope vector for the current block and the predictor parameters from the predictor decoder for the current block:

f_spec[bin]	vector of *num_bins* spectral lines from the previous block
f_env_signal[band]	vector of *num_bands* signal envelope values (float) for the current block
f_pred_gain	predictor gain value for the current block
f_pred_lag	predictor lag value for the current block

The output of the subband predictor is a vector of predicted spectral lines:

f_spec_pred[bin]	vector of *num_bins* predicted spectral lines for the current block

The bitstream and helper parameters used by the subband predictor are:

num_bands	number of bands
n_mdct	SSF block length
num_bins	number of coded spectral lines

The constants used by the subband predictor are:

NUM_SPEC_BUF = 5	number of buffered spectra
NUM_ENV_BUF = 4	number of buffered envelopes

The state internal to the subband predictor is the subband buffer and the envelope buffer:

f_spec_buffer[buf][bin]	*NUM_SPEC_BUF* × *num_bins* dimensional array of spectral lines (float)
f_env_buffer[buf][band]	*NUM_ENV_BUF* × *num_bands* dimensional array of signal envelope values (float)

The following intermediate signal is used by the subband predictor:

f_spec_extract[bin]	vector of *num_bins* spectral lines

The input signals are stored in the internal subband and envelope buffer.

Pseudocode

```
// update of subband and envelope buffer

for (spec = NUM_SPEC_BUF-1; spec > 0; spec--)
{
    f_spec_buffer[spec] = f_spec_buffer[spec-1];
}
f_spec_buffer[0] = f_spec[];

for (env = NUM_ENV_BUF-1; env > 0; env--)
{
    f_env_buffer[env] = f_env_buffer[env-1];
}
f_env_buffer[0] = f_env_signal[];
```

The extractor used by the subband predictor is a model based extractor. The period T_0 in units of the MDCT block length is defined by:

$$T_0 = \frac{f\_pred\_lag}{n\_mdct}.$$

An integer valued shift k_s is set to:

$$k_s = \begin{cases} 0, & T_0 \leq 81/32; \\ 1, & T_0 > 81/32. \end{cases}$$

The reduced period is then given by $T = T_0 - k_s$ and a table index n_T is derived as follows:

$$n_T = \begin{cases} 0, & T \leq 9/32; \\ \lfloor 16T + 0{,}5 \rfloor - 4, & T > 9/32. \end{cases}$$

This index is used to select radii in time and frequency from tables C.4 and C.5:

$$Rt = TAB\_PRED\_RTS\_TABLE[n_T],$$

$$Rf = TAB\_PRED\_RFS\_TABLE[n_T].$$

A three-dimensional array of prediction coefficients of size $(2Rf + 1) \times 65 \times Rt$ is selected using n_T:

$$\boldsymbol{C}(v, \eta, k) = \boldsymbol{C_{all}}[n_T]$$

NOTE: The setup of $\boldsymbol{C_{all}}[\ ]$ is described in clause 5.2.8.1.

From the subband buffer, an input array is created,

$$\boldsymbol{Z}(n, k) = \boldsymbol{f\_spec\_buffer}[k + k_s][n], k = 0, \ldots, Rt - 1, n = 0, \ldots, num\_bins - 1$$

This input array is extended by zeros for $n = num\_bins, \ldots, num\_bins - 1 + Rf$ and by using even reflection $\boldsymbol{Z}(n, k) = \boldsymbol{Z}(-1 - n, k)$ for the negative values $n = -Rf, \ldots, -1$.

With these extensions, the extracted output is defined by the summation:

$$f\_spec\_extract[p] = \sum_{k=0}^{Rt-1} \sum_{v=-Rf}^{Rf} (-1)^{(k+1)p} \boldsymbol{C}(v, f(2p+v+1), k)\boldsymbol{Z}(p+v, k)\,, p = 0, \dots, num\_bins - 1$$

Here, the function *f* is a composition of two functions *f(μ)=g(φ(μ))*, where the inner function is given by:

$$\varphi(\mu) = \left\lfloor \frac{T}{4}\mu + 0{,}5 \right\rfloor - \frac{T}{4}\mu,$$

and the outer function is given by:

$$g(\varphi) = \begin{cases} 32, & \varphi > T \\ -32, & \varphi < -T \\ \left\lfloor \frac{64\varphi}{\min\{2T,1\}} + 0{,}5 \right\rfloor, & \text{otherwise.} \end{cases}$$

The operation of the extractor in pseudocode notation is given below:

Pseudocode

```
// model based prediction extraction

f_period = f_pred_lag / n_mdct;      // T_0
k_s = 0;
if (f_period > 81.0/32.0) {
    k_s = 1;
    f_period -= 1.0f;   // T (reduced period)
}

tab_idx = (f_period <= 9.0/32.0) ? 0 : floor(16*f_period + 0.5) - 4;    // n_T
Rt = PRED_RTS_TABLE[tab_idx];
Rf = PRED_RFS_TABLE[tab_idx];

C = C_all[tab_idx];

// create input array Z from subband buffer
for (k = 0; k < Rt; k++) {
    for (n = 0; n < num_bins; n++) {
        Z[n][k] = f_spec_buffer[k+k_s][n];
    }
    // extend Z
    for (n = num_bins; n < num_bins + Rf; n++) {
        Z[n][k] = 0.0f;
    }
    for (n = -Rf; n < 0; n++) {
        Z[n][k] = Z[-n-1][k];
    }
}

// calculate extracted output
for (bin = 0; bin < num_bins; bin++) {
    tmp = 0;
    for (nu = -Rf; nu <= Rf; nu++) {
        // calc f=f(2*bin+nu+1)
        mu = 2*bin+nu+1;
        phi = (f_period/4)*mu;
        phi = round(phi)-phi;
        if (phi > f_period) {
            f = 32;
        }
        else if (phi < -f_period) {
            f = -32;
        }
        else {
            min_2T = 2*f_period < 1 ? 2*f_period : 1;
            f = round(64*phi/min_2T);
        }
        for (k = 0; k < Rt; k++) {
            s = (bin % 2) ? s*(-1) : 1;
            tmp += s*C[nu][f][k]*Z[bin+nu][k];
        }
    }
    f_spec_extract[bin] = tmp;
}
```

NOTE:	PRED_RTS_TABLE[ ] and PRED_RFS_TABLE[ ] are defined in Annex C. The initialization of C_all[tab_idx] is defined in clause 5.2.8.1.

The operation of the Shaper and the application of the prediction gain is specified in the following pseudocode:

Pseudocode

```
// shaper and prediction gain (for block index block)

integer_lag = round(f_pred_lag/n_mdct);
if ((b_iframe == 1) && (integer_lag > 0)) {
    // limit integer_lag to only use available envelop buffer entries
    integer_lag = 0;
}

for (band = 0; band < num_bands; band++) {
    f_envelope = 1.0f / f_env_buffer[integer_lag][band];
    for (bin = start_bin[band] ; bin <= end_bin[band]; bin++) {
        f_spec_pred[bin] = f_spec_extract[bin] * f_envelope * f_pred_gain;
    }
}
```

5.2.7 Inverse flattening

Data and Control Interfaces

The input to the inverse flattening is the spectrum from the spectrum decoder, the spectrum from the subband predictor and the signal envelope values for the current block:

f_spec_res[bin]	vector of *num_bins* residual spectral lines from the spectrum decoder
f_spec_pred[bin]	vector of *num_bins* predicted spectral lines from the subband predictor
f_env_signal[band]	vector of *num_bands* signal envelope values (float) for the current block

The output from the inverse flattening is a vector of spectral lines:

$\mathbf{s}_{\mathrm{SSF},ch}$	vector of *num_bins* spectral lines for the current block of track *ch*

The bitstream and helper parameters used by the inverse flattening are:

num_bands	number of bands
num_bins	number of coded spectral lines

Pseudocode

```
// Inverse flattening

// compose f_spec_flat[bin]
for (bin = 0; bin < num_bins; bin++) {
    f_spec_flat[bin] = f_spec_res[bin] + f_spec_pred[bin];
}

// apply signal envelope
for (band = 0; band < num_bands; band++) {
    for (bin = start_bin[band] ; bin <= end_bin[band]; bin++) {
        f_spec[bin] = f_spec_flat[bin] * f_env_signal[band];
    }
}
```

5.2.8 Parameterization

5.2.8.1 C matrix

The extractor of the subband predictor uses a matrix ***C***, selected by *tab_idx*. The setup of all 37 possible matrices, stored in a big matrix $\boldsymbol{C}_{all}$*[tab_idx]*, is described in the following. Each matrix ***C*** is a three-dimensional array of prediction coefficients of the size (2*Rf* + 1)×65×*Rt*, where *Rt* and *Rf* are dependent on *tab_idx*:

$$Rt = PRED\_RTS\_TABLE[tab\_idx],$$

$$Rf = PRED\_RFS\_TABLE[tab\_idx].$$

For the clarity of the description, $\boldsymbol{C}$ is addressed here with both positive and negative indices:

$$\boldsymbol{C}(\nu, \eta, k), \nu = -Rf, \dots, Rf, \eta = -32, \dots, 32, k = 0, \dots, Rt - 1$$

First, the sub-array for positive η indices is initialized with the array of quantized predictor coefficients given by table PRED_COEFF_QUANT_MAT[tab_idx], which is of size $(2Rf + 1)\times 33\times Rt$. Next, the quantized prediction coefficients are replaced by the reconstructed values using:

$$coeff_{reconstruct} = 1{,}1787855 \times \frac{coeff_{quantized} - 146}{128}$$

for the reconstruction. The remaining coefficients of the prediction coefficients array are initialized using the rule:

$$\boldsymbol{C}(\nu, \eta, k) = (-1)^{k+1}\boldsymbol{C}(-\nu, -\eta, k)$$

for negative values $\eta = -32,\dots,-1$. ($\nu = -Rf,\dots,Rf$, $k = 0,\dots,Rt - 1$)

The setup written in pseudocode is given below:

Pseudocode

```
// setup of prediction coefficients arrays

for (tab_idx = 0; tab_idx <= 36; tab_idx++)
{
    Rf = PRED_RFS_TABLE[tab_idx];
    Rt = PRED_RTS_TABLE[tab_idx];

    // extract and reconstruct prediction coefficients for positive eta indices
    for (k = 0; k < Rt; k++) {
        for (eta = 0; eta < 33; eta++) {
            for (nu = -Rf; nu <= Rf; nu++) {
                C_all[tab_idx][nu][eta][k] =
                    1.1787855*(PRED_COEFF_QUANT_MAT[tab_idx][nu][eta][k] - 146)/128;
            }
        }
    }

    // initialize prediction coefficients for negative eta indices
    s = 1;
    for (k = 0; k < Rt; k++) {
        s *= -1;
        for (eta = -32; eta < 0; eta++) {
            for (nu = -Rf; nu <= Rf; nu++) {
                C_all[tab_idx][nu][eta][k] = s * C_all[tab_idx][-nu][-eta][k];
            }
        }
    }
}
```

5.2.8.2 Arithmetic coding

A single instance of an arithmetic coder is used during the arithmetic decoding of the following three parameter types within an SSF granule, each using a different statistical model:

- **Envelope indices** using a static pre-trained CDF table
- **Predictor gain indices** using a static pre-trained CDF table
- **Transform coefficients** using CDF computation based on pre-trained CDF prototype function

The arithmetic coder is initialized once per SSF granule, when the parsing has reached the `ssf_ac_data` element. The envelope quantization indices are coded first and a predefined look-up table (LUT) is used to obtain values of the cumulative distribution function (CDF) for respective indices. Next, the arithmetic coder operates in a loop of *num_blocks* blocks, where each block comprises coding of a single predictor gain index and a single set of quantization indices representing quantized MDCT coefficients. Depending on the parameter that is coded at a given time instance, the arithmetic coder uses either a LUT for the predictor gains, or computes the CDF values given dither realization and quantization step-sizes. The arithmetic decoding process is terminated only at the end of the SSF granule.

Termination bits are computed at the end of the decoding process such that an arithmetic decoder is able to provide the exact number of arithmetic decoded bits.

In the case of an I-frame, two envelopes are present instead of a single envelope.

Arithmetic coding implementation

The arithmetic decoding used within the SSF is specified using fixed point arithmetic and the two integer data types uint32 and int32. The unsigned type is used to represent the state of the arithmetic decoder and the signed type is used for counters in programme loops. The fixed point operators are defined below.

Pseudocode

```
// data types used by the arithmetic decoder

typedef unsigned int uint32;    /* ui */
typedef          int  int32;    /* i */

// fixed point operators used by the arithmetic decoder

// subtraction with conversion to signed variables
#define SSF_I32_SUB_I32(a, b)           ((int32)(a) - (int32)(b))

// addition with conversion to signed variables
#define SSF_I32_ADD_I32(a, b)           ((int32)(a) + (int32)(b))

// subtraction with a forced conversion to unsigned variables
#define SSF_UI32_SUB_UI32(a, b)         ((uint32)(a) - (uint32)(b))

// addition with a forced conversion to unsigned variables
#define SSF_UI32_ADD_UI32(a, b)         ((uint32)(a) + (uint32)(b))

// absolute value
#define SSF_I32_ABS(a)                  ((a)>=0 ? a : -a)

// multiplication in a 32 bit signed register
#define SSF_I16_MUL_I16(a, b)           ((int32)(a) * (int32)(b))

// multiplication in a 32 bit unsigned register
#define SSF_UI16_MUL_UI16(a, b)         ((uint32)(a) * (uint32)(b))

// left shift with a conversion to an unsigned integer
#define SSF_UI32_SHIFT_LEFT(a, b)       (((uint32)(a)) << (b))

// right shift with a conversion to an unsigned integer
#define SSF_UI32_SHIFT_RIGHT(a, b)      ((uint32)(a) >> (b))

// left shift of a signed integer (sign preserving)
#define SSF_I32_SHIFT_LEFT(a, b)        ((a) >= 0 ? (int32)SSF_UI32_SHIFT_LEFT(a, b) :
-(int32)SSF_UI32_SHIFT_LEFT(-(a), b))

// right shift of a signed integer (sign preserving)
#define SSF_I32_SHIFT_RIGHT(a, b)       ((a) >= 0 ? (int32)SSF_UI32_SHIFT_RIGHT(a, b) :
-(int32)SSF_UI32_SHIFT_RIGHT(-(a), b))
```

NOTE 1: Not all operators are actually used in the implementation.
NOTE 2: All variables involved in arithmetic coding are actually unsigned integers.

The implementation of the arithmetic decoder uses the following constant values.

Pseudocode

```
// constants used by the arithmetic decoder

/* Number of model bits */
#define SSF_MODEL_BITS          15  /* All CDFs come in Q0.15 */
/* Model unit for the CDF specification */
#define SSF_MODEL_UNIT          (1U<<(SSF_MODEL_BITS))

/* Number of range bits */
#define SSF_RANGE_BITS          30
/* Half of the range unit */
#define SSF_SSF_THRESHOLD_LARGE (1U<<((SSF_RANGE_BITS)-1))
/* Quarter of the range unit */
#define SSF_THRESHOLD_SMALL     (1U<<((SSF_RANGE_BITS)-2))

/* Offset bits */
#define SSF_OFFSET_BITS         14
```

The following state structure represents the state of the arithmetic decoder.

Pseudocode

```
// state of the arithmetic decoder

typedef struct _ac_state
{
    // state
    uint32 uiLow;
    uint32 uiRange;
    uint32 uiOffset;
    uint32 uiOffset2;

    // fixed scales (updated during initialization - remain constant during operation)
    uint32 uiThresholdSmall;
    uint32 uiThresholdLarge;
    uint32 uiModelUnit;

    // specification - these values are also constant
    uint32 uiRangeBits;
    uint32 uiModelBits;

    // bitstream
    bitstream *psBs;    // current position in the bitstream is stored internally

} ac_state;
```

The initialization of the arithmetic decoder sets up the variables representing the state of the arithmetic decoder and reads the beginning of the arithmetic coded data, i.e. `ssf_ac_data`.

Pseudocode

```
// initialize the arithmetic decoder

int32 AcDecoderInit(ac_state  *psS,      // input & output
                    bitstream *psBs)     // input (pointer to the bitstream structure)
{
    uint32 uiTmp;
    uint32 iBitIdx;

    /* Constant values */
    psS->uiModelBits = SSF_MODEL_BITS;
    psS->uiModelUnit = SSF_MODEL_UNIT;
    psS->uiRangeBits = SSF_RANGE_BITS;
    psS->uiThresholdLarge = SSF_THRESHOLD_LARGE;
    psS->uiThresholdSmall = SSF_THRESHOLD_SMALL;

    /* Initialization */
    psS->uiLow = 0;
    psS->uiRange = SSF_THRESHOLD_LARGE;

    /* Bitstream initialization */
    psS->psBs = psBs; // just a pointer to some bitstream structure

    /* Read bits (Here we read SSF_RANGE_BITS from the bitstream.)*/
    psS->uiOffset = (uint32)BsReadBit(psS->psBs);
    for (iBitIdx = 1; iBitIdx < psS->uiRangeBits; iBitIdx++) {
        uiTmp = (uint32)BsReadBit(psS->psBs);
        psS->uiOffset = SSF_UI32_SHIFT_LEFT(psS->uiOffset, 1);
        psS->uiOffset = SSF_UI32_ADD_UI32(psS->uiOffset, uiTmp);
    }
    psS->uiOffset2 = psS->uiOffset;

    return 0;
}
```

Decoding a single symbol involves the usage of three functions, all provided below. A function that decodes a single symbol given two CDF values is called *AcDecodeSymbolExtCdf()*. The function returns a signed quantization index and updates the state of the arithmetic decoder.

Pseudocode

```
int32 AcDecodeSymbolExtCdf(ac_state    *psS,            // input & output
                           uint32      *puiCdfTable,    // input
                           const int32  iMinSymbol,     // input
                           const int32  iMaxSymbol)     // input
{   // returns a decoded index

    uint32 uiTarget, uiCdfLow, uiCdfHigh, uiVal;
    int32  iFinalSymbol, iSymbolIdx;
    iFinalSymbol = (1U<<15);

    /* First we call AcDecodeTarget(). This is done once per symbol. */
    uiTarget = AcDecodeTarget(psS);     /* Q0.15 */

    /* ===loop over the whole codebook=== */
    /* Now we search over the entire codebook by computing the CDF values     */
    /* corresponding to the possible quantization indices.                    */
    /* Assume that we have a codebook where the possible symbols are indexed  */
    /* starting form iMinSymbol and ending at iMaxSymbol.                     */
    for (iSymbolIdx = iMinSymbol; iSymbolIdx <= iMaxSymbol; iSymbolIdx++)
    {
        if (puiCdfTable != NULL) {
            /* This is done when decoding the envelope indices and the predictor */
            /* gain indices */
            uiCdfLow  = puiCdfTable[iSymbolIdx-iMinSymbol];
            uiCdfHigh = puiCdfTable[iSymbolIdx-iMinSymbol+1];
        } else {
            /* For the transform coefficients we use the actual CDF computation. */
            /* See code below, how to derive uiCdfLow and uiCdfHigh from */
            /* iSymbolIdx, i_step_size and i_dither_val */
        }

        /* Now, if uiTarget is between uiCdfHigh and uiCdfLow we found the symbol. */
        if ((uiTarget < uiCdfHigh) && (uiTarget >= uiCdfLow))
        {
            iFinalSymbol = iSymbolIdx; /* we have just found a symbol index */

            /* Let the arithmetic decoder advance in the bitstream and */
            /* update its state                                         */
            AcDecode(uiCdfLow, uiCdfHigh, psS);
            break; /* we are done - quit the codebook search loop */
        }
    } /* end of the loop over the whole codebook */

    return iFinalSymbol;
}
```

Pseudocode

```
int32 AcDecodeTarget(ac_state *psS) // input & output
{
    uint32 uiTarget, uiRange, uiNum, uiDen, uiTmp, uiNumShifts;
    int32 iIdx;

    uiRange = SSF_UI32_SHIFT_RIGHT(psS->uiRange, psS->uiModelBits);

    uiTmp = SSF_UI32_SHIFT_LEFT(1, SSF_OFFSET_BITS);

    if (uiRange < uiTmp) {
        uiNumShifts = psS->uiModelBits;
    } else {
        uiNumShifts = psS->uiModelBits - 1;
    }

    uiNum = psS->uiOffset;
    uiDen = SSF_UI32_SHIFT_LEFT(uiRange, uiNumShifts);
    uiTarget = 0; // initialize

    for (iIdx = uiNumShifts; iIdx > 0; iIdx--)
    {
        if (uiNum >= uiDen) {
            uiNum = SSF_UI32_SUB_UI32(uiNum, uiDen);
            uiTarget = SSF_UI32_ADD_UI32(uiTarget, 1);
        }

        uiNum = SSF_UI32_SHIFT_LEFT(uiNum, 1);
        uiTarget = SSF_UI32_SHIFT_LEFT(uiTarget, 1);
    }

    if (uiNum >= uiDen) {
        uiNum = SSF_UI32_SUB_UI32(uiNum, uiDen);
        uiTarget = SSF_UI32_ADD_UI32(uiTarget, 1);
    }

    if (uiTarget >= psS->uiModelUnit) {
        uiTarget = SSF_UI32_SUB_UI32(psS->uiModelUnit, 1);
    }


    return uiTarget; /* Q0.15 by design */
}
```

Pseudocode

```
int32 AcDecode(uint32    uiCdfLow,  // input
               uint32    uiCdfHigh, // input
               ac_state *psS)       // input & output
{
    uint32 uiTmp1, uiTmp2;
    uint32 uiRange;

    uiRange = SSF_UI32_SHIFT_RIGHT(psS->uiRange, psS->uiModelBits);

    uiTmp1 = SSF_UI16_MUL_UI16(uiRange, uiCdfLow);

    psS->uiOffset = SSF_UI32_SUB_UI32(psS->uiOffset, uiTmp1);

    if (uiCdfHigh < psS->uiModelUnit) {
        uiTmp2 = SSF_UI32_SUB_UI32(uiCdfHigh, uiCdfLow);
        psS->uiRange = SSF_UI16_MUL_UI16(uiRange, uiTmp2);
    } else {
        psS->uiRange = SSF_UI32_SUB_UI32(psS->uiRange, uiTmp1);
    }

    // denormalize
    while (psS->uiRange <= psS->uiThresholdSmall)
    {
        /* Read a single bit from the bitstream */
        uiTmp1 = (uint32)BsReadBit(psS->psBs);
        psS->uiRange   = SSF_UI32_SHIFT_LEFT(psS->uiRange, 1);
        psS->uiOffset  = SSF_UI32_SHIFT_LEFT(psS->uiOffset, 1);
        psS->uiOffset  = SSF_UI32_ADD_UI32(psS->uiOffset, uiTmp1);
        psS->uiOffset2 = SSF_UI32_SHIFT_LEFT(psS->uiOffset2, 1);
        if (psS->uiOffset & 1) {
            psS->uiOffset2++;
        }
    }

    return 0;
}
```

Computing the number of termination bits happens once per granule. The function is needed so the decoder can tell how many bits have been successfully decoded from the bitstream.

Pseudocode

```
int32 AcDecodeFinish(ac_state *psS) // input & output
{
    int32  iRes, iBitIdx;
    uint32 uiConstUpFact, uiBits, uiVal;
    uint32 uiTmp1, uiTmp2, uiRevIdx;
    iRes = psS->psBs->iPos;         // iPos: bits read so far by the bitstream reader
    iRes = iRes - psS->uiRangeBits;

    /* Determine the finish bits */
    psS->uiLow = (psS->uiOffset2 & (psS->uiThresholdLarge-1));
    uiTmp1 = SSF_UI32_SUB_UI32(psS->uiThresholdLarge, psS->uiOffset);
    psS->uiLow = SSF_UI32_ADD_UI32(psS->uiLow, uiTmp1);

    for (iBitIdx = 1; iBitIdx <= psS->uiRangeBits; iBitIdx++)
    {
        uiRevIdx = psS->uiRangeBits - iBitIdx;
        uiConstUpFact = SSF_UI32_SHIFT_LEFT(1U, uiRevIdx);
        uiConstUpFact = SSF_UI32_SUB_UI32(uiConstUpFact, 1U);
        uiTmp1 = SSF_UI32_ADD_UI32(psS->uiLow, uiConstUpFact);
        uiBits = SSF_UI32_SHIFT_RIGHT(uiTmp1, uiRevIdx);
        uiVal  = SSF_UI32_SHIFT_LEFT(uiBits, uiRevIdx);
        uiTmp1 = SSF_UI32_ADD_UI32(uiVal, uiConstUpFact);
        uiTmp2 = SSF_UI32_SUB_UI32(psS->uiRange, 1U);
        uiTmp2 = SSF_UI32_ADD_UI32(uiTmp2, psS->uiLow);
        if ((psS->uiLow <= uiVal) && (uiTmp1 <= uiTmp2))
        {
            break;
        }
    }

    iRes = iRes + iBitIdx;

    return iRes; // returns number of arithmetically decoded bits
}
```

The arithmetic decoding of the envelope indices using the above mentioned function *AcDecodeSymbolExtCdf()* is given by the following pseudocode:

Pseudocode

```
// envelope indices AC decoding

for (idx = 1; idx < num_bands; idx++)
{
    env_idx[idx] = AcDecodeSymbolExtCdf(ac_state, ENVELOPE_CDF_LUT, 0, 32);
}
```

NOTE: ENVELOPE_CDF_LUT is defined in table C.8.

Similarly, the arithmetic decoding of the predictor gain indices can be described as follows.

Pseudocode

```
// predictor gain indices AC decoding

arithmetic_decode_pred()
{
    i_pred_gain_idx = AcDecodeSymbolExtCdf(ac_state, PREDICTOR_GAIN_CDF_LUT, 0, 32);

    return i_pred_gain_idx;
}
```

NOTE: PREDICTOR_GAIN_CDF_LUT is defined in table C.7.

The arithmetic decoding of the transform coefficients can be done according to the following pseudocode.

Pseudocode

```
// transform coefficients AC decoding

// input: i_alloc_table[band], i_dither[bin]
// output: i_quant_idx[bin]

arithmetic_decode_coeffs()
{
    for(band = 0; band < num_bands; band++)
    {
        i_alloc = i_alloc_table[band];
        if (i_alloc == 0)
        {
            for (bin = start_bin[band]; bin <= end_bin[band]; bin++)
            {
                i_quant_idx[bin] = 0;
            }
        }
        else // (i_alloc > 0)
        {
            i_step_size = STEP_SIZES_Q4_15[i_alloc];
            i_max_idx = AC_COEFF_MAX_INDEX[i_alloc] + 1;

            for (bin = start_bin[band]; bin <= end_bin[band]; bin++)
            {
                i_dither_val = i_dither[bin];

                i_quant_idx[bin] = AcDecodeSymbolExtCdf(ac_state, NULL, 0, i_max_idx);
            }
        }
    }
}
```

NOTE: AC_COEFF_MAX_INDEX[ ] is defined in table C.12.

The function *AcDecodeSymbolExtCdf()* is called without a CDF table to signal that the CDF needs to be calculated.

NOTE: For the CDF calculation, the values *i_step_size* and *i_dither_val* are needed by *AcDecodeSymbolExtCdf()*. These values are also passed on to this function although they are not shown as input parameters.

The calculation of the CDF is done according to the following pseudocode.

Pseudocode

```
// CDF calculation to be used during the AC decoding of the transform coefficients

// uiCdfLow and uiCdfHigh are derived from iSymbolIdx, i_step_size and i_dither_val

const int32 i_max_value = 327680;
int32 iLeft;
int32 iRight;
int32 iMidpoint;
int32 i_half_step_size;

iMidpoint = Idx2Reconstruction(iSymbolIdx, i_dither_val, i_step_size);
i_half_step_size = SSF_I32_SHIFT_RIGHT(i_step_size, 1);
iLeft =  SSF_I32_SUB_I32(iMidpoint, i_half_step_size);
iRight = SSF_I32_ADD_I32(iLeft, i_step_size);

iLeft = iLeft < -i_max_value ? -i_max_value : iLeft;
iRight = iRight > i_max_value ? i_max_value : iRight;

uiCdfLow = CdfEst(iLeft);
uiCdfHigh = CdfEst(iRight);
```

The used functions *Idx2Reconstruction()* and *CdfEst()* are given below.

Pseudocode

```
int32 Idx2Reconstruction(int32 iIndex, int32 iDitherValue, int32 iStepSize)
{
    int32 iReconstruction, iTmp1, iTmp2;

    // subtract the dither
    iTmp1 = SSF_I32_SHIFT_LEFT(iIndex, 15);
    iReconstruction = SSF_I32_SUB_I32(iTmp1, iDitherValue);

    // multiply times the step size
    iTmp2 = SSF_I32_SHIFT_RIGHT(iReconstruction, 15);
    iTmp1 = SSF_I32_SHIFT_LEFT(iTmp2, 15);
    iTmp1 = SSF_I32_SUB_I32(iReconstruction, iTmp1);
    iTmp1 = SSF_I32_SHIFT_RIGHT(iTmp1, 3);
    iReconstruction = SSF_I16_MUL_I16(iTmp1, iStepSize);
    iReconstruction = SSF_I32_SHIFT_RIGHT(iReconstruction, 12);
    iTmp1 = SSF_I16_MUL_I16(iTmp2,iStepSize);
    iReconstruction = SSF_I32_ADD_I32(iReconstruction, iTmp1);

    return iReconstruction;
}
```

Pseudocode

```
uint32 CdfEst(int32 iInVal) // input value Qx.15
{
    int32 iIdx;

    // Q4.15->Q.0 (we keep 9 MSB, result in [-352, 352] for a correctly bounded input)
    iIdx = SSF_I32_SHIFT_RIGHT(iInVal, 10); // note that it is a shift with a sign

    // iIdx is an offset from the beginning of the table [-352, 352]
    iIdx = iIdx + 352;

    return CDF_TABLE[iIdx]; // read from the CDF table
}
```

NOTE: CDF_TABLE[ ] is defined in table C.6.

5.2.8.3 Dither and random noise

Two instances of a random number generator are used during the SSF decoding - one for the dither signal setup and the other one for noise signal values. The implementation of the random number generator shall conform to the code in the following pseudocode since it is important that the SSF encoder and the SSF decoder use the same pseudo-random values. The random number generators are initialized at the beginning of the decoding of an SSF-I-frame.

The following state structure represents the state of the random number generator.

Pseudocode

```
// This structure is used to store the state of the random number generator.
// All variables in the state structure are subject to implicit modulo 256.
typedef struct _ssf_rndgen_state
{
    uint8 uiOffsetA;    // scan step size
    uint8 uiOffsetB;    // scan offset
    uint8 uiStateIdx;   // state counter 1
    uint8 uiCurrentIdx; // state counter 2 (main index)
} ssf_rndgen_state;
```

The initialization of the random number generator sets up the variables representing the state of the random number generator.

Pseudocode

```
// initialize / reset the random number generator

void ResetRandGenState(ssf_rndgen_state *psS)
{
    psS->uiOffsetA = 0;
    psS->uiOffsetB = 0;
    psS->uiStateIdx = 1;
    psS->uiCurrentIdx = 0;
}
```

The following function, which returns a random value, is used during the setup of the dither signal.

Pseudocode

```
int32 GetDitherValue(ssf_rndgen_state *psS)
{
    int32 iRes;

    // main look-up
    iRes = DITHER_TABLE[psS->uiCurrentIdx];

    // state update (using implicit modulo 256)
    psS->uiOffsetA++;
    psS->uiStateIdx = psS->uiStateIdx++;

    if (psS->uiOffsetA == 255) {
        psS->uiCurrentIdx = psS->uiCurrentIdx++;
        psS->uiOffsetB++;
        psS->uiOffsetA = 0;
    }

    psS->uiCurrentIdx = psS->uiCurrentIdx + psS->uiOffsetA;
    psS->uiStateIdx = psS->uiStateIdx + psS->uiOffsetB + psS->uiOffsetA;

    return iRes;
}
```

NOTE: DITHER_TABLE[ ] is defined in table C.9.

The following function, which returns a random value, is used by the spectrum decoder to insert noise values.

Pseudocode
(code below)

```
float32 GetRandomNoiseValue(ssf_rndgen_state *psS)
{
    float32 fRes, fRes1, fRes2;

    // main look-up
    fRes1 = RANDOM_NOISE_TABLE[psS->uiCurrentIdx];
    fRes2 = RANDOM_NOISE_TABLE[psS->uiStateIdx];

    // compute result
    fRes = fRes1 + fRes2;

    // state update (using implicit modulo 256)
    psS->uiOffsetA++;
    psS->uiStateIdx=psS->uiStateIdx++;

    if (psS->uiOffsetA == 255) {
        psS->uiCurrentIdx = psS->uiCurrentIdx++;
        psS->uiOffsetB++;
        psS->uiOffsetA = 0;
    }

    psS->uiCurrentIdx = psS->uiCurrentIdx + psS->uiOffsetA;
    psS->uiStateIdx = psS->uiStateIdx + psS->uiOffsetB + psS->uiOffsetA;

    return fRes;
}
```

NOTE: RANDOM_NOISE_TABLE[ ] is defined in table C.10.

To generate the dither signal to be used by the decoding of the current SSF granule, the above introduced function *GetDitherValue()* is employed.

Pseudocode

```
// setup of i_dither_cur[block][bin]

// ditherGenState holds the state of the random number generator used for the dither
// signal generation

for (block = 0; block < num_blocks; block++) {
    for (bin = 0; bin < num_bins; bin++) {
        i_dither_cur[block][bin] = GetDitherValue(&ditherGenState);
    }
}
```

5.3 Stereo and multichannel processing

5.3.1 Introduction

The multichannel tools take their input from one or more spectral frontend(s), receiving n-tuples of spectra with matching time/frequency layout. The multichannel processing tool applies a number time- and frequency varying matrix operation on these tuples. Between two and seven spectra are transformed at a time.

Parameters for the transformations are transmitted by `chparam_info()` elements; the various transform matrices **M** (up to 7 × 7) are built up from these parameters as described in the following paragraphs.

When the tuples have been transformed, they are passed on to the IMDCT transform, defined in clause 5.5. While the input to the multichannel processing tool does not have a "channel" meaning, the output from the tool carries channels, ordered as L, R, C, Ls, Rs, etc.

Data and Control Interfaces

The input to the stereo and multichannel processing tool are scaled spectral lines of tracks derived from decoding the `sf_data` element stored in:

- a `channel_pair_element` ($M_{SAP} = 2$),
- a `3_0_channel_element` ($M_{SAP} = 3$),
- a `5_X_channel_element` ($M_{SAP} = 5$) or
- a `7_X_channel_element` ($M_{SAP} = 7$)

with the ASF or the SSF tool:

$\mathbf{s}_{SMP,[0\|1\|\ldots]}$	Up to M_{SAP} vectors of spectral lines, each vector representing a track decoded from an `sf_data` element.

NOTE 1: The tracks are numbered according to their occurrence in the bitstream, starting from track $\mathbf{s}_{SMP,0}$.

The output from the stereo and multichannel processing tool are M_{SAP} blocks of scaled spectral lines:

$\mathbf{s}_{SMP,[L\|R\|C\|\ldots]}$	M_{SAP} vectors of *blk_len* spectral lines assigned to channels with discrete speaker locations

NOTE 2: See clause D.1 for a listing of channel abbreviations.

The bitstream and additional information used by the stereo audio processing tool is:

blk_len	block length. Equals to the number of input and output spectral lines in one channel.
N	block length. Equals to the number of input and output spectral lines in one channel.
sap_used[g][b]	array indicating the operating mode of the stereo audio processing tool for group *g* and scale factor band *sfb*.
sap_gain[g][b]	array of real valued gains for group *g* and scale factor band *sfb*.

5.3.2 Parameter extraction

All parameters for the multichannel tool are transmitted in one or more `chparam_info()` elements. The following pseudocode describes parsing the `chparam_info()` element and extracting parameters a, b, c, d:

Pseudocode

```
// initialize a[g][sfb], b[g][sfb], c[g][sfb] and d[g][sfb] after parsing of
// chparam_info()

for (g = 0; g < num_window_groups; g++) {
    for (sfb = 0; sfb < max_sfb[g]; sfb++) {
        if (sap_mode == 0 || (sap_mode == 1 && ms_used[g][sfb] == 0)) {
            a[g][sfb]=d[g][sfb]=1; b[g][sfb]=c[g][sfb]=0;
        }
        else if (sap_mode == 2 || ((sap_mode == 1 && ms_used[g][sfb] == 1)) {
            a[g][sfb]=b[g][sfb]=c[g][sfb]=1; d[g][sfb]=-1;
        }
        else {  // sap_mode == 3
            sap_used[g][sfb] = sap_coeff_used[g][sfb];
            // setup alpha_q[g][sfb]
            if (sfb % 2) {
                alpha_q[g][sfb] = alpha_q[g][sfb-1];
            }
            else {
                if (sap_used[g][sfb]) {
                    delta = dpcm_alpha_q[g][sfb] - 60;
                    code_delta = (g == 0) ? 0 : delta_code_time;
                    if (code_delta) {
                        alpha_q[g][sfb] = alpha_q[g-1][sfb] + delta;
                    }
                    else if (sfb == 0) {
                        alpha_q[g][sfb] = delta;
                    }
                    else {
                        alpha_q[g][sfb] = alpha_q[g][sfb-2] + delta;
                    }
                }
                else {
                    alpha_q[g][sfb] = 0;
                }
            }
            // inverse quantize alpha_q[g][sfb]
            sap_gain[g][sfb] = alpha_q[g][sfb] * 0.1f;
            a[g][sfb]=1+sap_gain[g][sfb]; b[g][sfb]=1;
            c[g][sfb]=1-sap_gain[g][sfb]; d[g][sfb]=-1;
        }
    }
}
```

5.3.3 Processing the channel data elements

In the various channel data elements, a number n > 1 of input tracks (I_0, …, I_n) are matrix transformed into the n output tracks (O_0, …, O_n). They share the same `sf_info` and so share the same time/frequency resolution. The matrix equations below are understood to apply to each time/frequency tile separately, with separate parameters a, b, c, d per such tile.

In clauses 5.3.3.1 through 5.3.3.5 below, "decoding" an `sf_data` element is understood to mean decoding the spectral frontend data.

5.3.3.1 Processing tracks of the mono_data element

Let I_0 be the track decoded from the `sf_data` element contained in the `mono_data` element.

The decoder shall map the input track directly to the output O_0, i.e.:

$$[O_0] = [I_0]$$

5.3.3.2 Processing tracks of the two_channel_data or stereo_data element

Let I_0 and I_1 be the tracks decoded from the two `sf_data` elements contained in the `two_channel_data` element, or of the `stereo_data` element.

If `b_enable_mdct_stereo_proc` = 0, no further processing is necessary and O_0,$=I_0$, $O_1=I_1$.

Otherwise, the decoder shall extract the parameters a, b, c, d from the contained `chparam_info` element as described in clause 5.3.2 and derive the outputs O_0, O_1 by:

$$\begin{bmatrix} O_0 \\ O_1 \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} I_0 \\ I_1 \end{bmatrix}$$

5.3.3.3 Processing tracks of the three_channel_data element

Let I_0, I_1, and I_2 be the tracks decoded from the three `sf_data` elements contained in the `three_channel_data` element.

The decoder shall extract parameters a_i, b_i, c_i, d_i (i = 0…1) from the two contained `chparam_info` elements, and from the `chel_matsel` element determine the transform matrix **M** as described in table 172.

Finally, the output tracks O_0, O_1 and O_2 shall be derived by:

$$\begin{bmatrix} O_0 \\ O_1 \\ O_2 \end{bmatrix} = \mathbf{M} \times \begin{bmatrix} I_0 \\ I_1 \\ I_2 \end{bmatrix}$$

Table 172: Determining the three_channel_data() element transform matrix

chel_matsel	M
0	$\begin{bmatrix} a_0a_1 & b_0a_1 & b_1 \\ c_0 & d_0 & 0 \\ a_0c_1 & b_0c_1 & d_1 \end{bmatrix}$
1	$\begin{bmatrix} d_0 & c_0 & 0 \\ b_0a_1 & a_0a_1 & b_1 \\ b_0c_1 & a_0c_1 & d_1 \end{bmatrix}$
2	$\begin{bmatrix} a_0a_1 & b_1 & b_0a_1 \\ a_0c_1 & d_1 & b_0c_1 \\ c_0 & 0 & d_0 \end{bmatrix}$
3	$\begin{bmatrix} a_1 & c_0b_1 & d_0b_1 \\ 0 & a_0 & b_0 \\ c_1 & c_0d_1 & d_0d_1 \end{bmatrix}$
4	$\begin{bmatrix} a_0 & 0 & b_0 \\ c_0b_1 & a_1 & d_0b_1 \\ c_0d_1 & c_1 & d_0d_1 \end{bmatrix}$
5	$\begin{bmatrix} a_1 & d_0b_1 & c_0b_1 \\ c_1 & d_0d_1 & c_0d_1 \\ 0 & b_0 & a_0 \end{bmatrix}$
6	$\begin{bmatrix} d_0d_1 & c_0d_1 & c_1 \\ b_0 & a_0 & 0 \\ d_0b_1 & c_0b_1 & a_1 \end{bmatrix}$
7	$\begin{bmatrix} a_0 & b_0 & 0 \\ c_0d_1 & d_0d_1 & c_1 \\ c_0b_1 & d_0b_1 & a_1 \end{bmatrix}$
8	$\begin{bmatrix} d_0d_1 & c_1 & c_0d_1 \\ d_0b_1 & a_1 & c_0b_1 \\ b_0 & 0 & a_0 \end{bmatrix}$
9	$\begin{bmatrix} d_1 & b_0c_1 & a_0c_1 \\ 0 & d_0 & c_0 \\ b_1 & b_0a_1 & a_0a_1 \end{bmatrix}$
10	$\begin{bmatrix} d_0 & 0 & c_0 \\ b_0c_1 & d_1 & a_0c_1 \\ b_0a_1 & b_1 & a_0a_1 \end{bmatrix}$
11	$\begin{bmatrix} d_1 & a_0c_1 & b_0c_1 \\ b_1 & a_0a_1 & b_0a_1 \\ 0 & c_0 & d_0 \end{bmatrix}$

5.3.3.4 Processing tracks of the four_channel_data element

Let $I_0 \ldots I_3$ be the tracks decoded from the four `sf_data` elements contained in the four_channel_data element.

The decoder shall extract parameters a_i, b_i, c_i, d_i (i = 0…3) from the four contained `chparam_info` elements.

Finally, the output tracks $O_0 \ldots O_3$ shall be derived by:

$$\begin{bmatrix} O_0 \\ O_1 \\ O_2 \\ O_3 \end{bmatrix} = \begin{bmatrix} a_0a_2 & b_0a_2 & a_1\,b_2 & b_1\,b_2 \\ c_0a_3 & d_0a_3 & c_1b_3 & d_1b_3 \\ a_0c_2 & b_0c_2 & a_1d_2 & b_1d_2 \\ c_0c_3 & d_0c_3 & c_1d_3 & d_1d_3 \end{bmatrix} \times \begin{bmatrix} I_0 \\ I_1 \\ I_2 \\ I_3 \end{bmatrix}$$

5.3.3.5 Processing tracks of the five_channel_data element

Let $I_0 \ldots I_4$ be the tracks decoded from the five `sf_data` elements contained in the five_channel_data element.

Next the decoder shall extract parameters a_i, b_i, c_i, d_i (i = 0…4) from the five contained `chparam_info` elements, and from the `chel_matsel` element determine the transform matrix **M** as described in table 173.

Finally, the output tracks $O_0 \ldots O_4$ shall be derived by:

$$\begin{bmatrix} O_0 \\ O_1 \\ O_2 \\ O_3 \\ O_4 \end{bmatrix} = \mathbf{M} \times \begin{bmatrix} I_0 \\ I_1 \\ I_2 \\ I_3 \\ I_4 \end{bmatrix}$$

Table 173: Determining the five_channel_data() element transform matrix

chel_matsel	M
0	$\begin{bmatrix} a_0a_1a_3 & b_0a_1a_3 & b_1a_3 & a_2b_3 & b_2b_3 \\ c_0a_4 & d_0a_4 & 0 & c_2b_4 & d_2b_4 \\ a_0c_1 & b_0c_1 & d_1 & 0 & 0 \\ a_0a_1c_3 & b_0a_1c_3 & b_1c_3 & a_2d_3 & b_2d_3 \\ c_0c_4 & d_0c_4 & 0 & c_2d_4 & d_2d_4 \end{bmatrix}$
1	$\begin{bmatrix} d_0a_3 & c_0a_3 & 0 & a_2b_3 & b_2b_3 \\ b_0a_1a_4 & a_0a_1a_4 & b_1a_4 & c_2b_4 & d_2b_4 \\ b_0c_1 & a_0c_1 & d_1 & 0 & 0 \\ d_0c_3 & c_0c_3 & 0 & a_2d_3 & b_2d_3 \\ b_0a_1c_4 & a_0a_1c_4 & b_1c_4 & c_2d_4 & d_2d_4 \end{bmatrix}$
2	$\begin{bmatrix} a_0a_1a_3 & b_1a_3 & b_0a_1a_3 & a_2b_3 & b_2b_3 \\ a_0c_1a_4 & d_1a_4 & b_0c_1a_4 & c_2b_4 & d_2b_4 \\ c_0 & 0 & d_0 & 0 & 0 \\ a_0a_1c_3 & b_1c_3 & b_0a_1c_3 & a_2d_3 & b_2d_3 \\ a_0c_1c_4 & d_1c_4 & b_0c_1c_4 & c_2d_4 & d_2d_4 \end{bmatrix}$
3	$\begin{bmatrix} a_1a_3 & c_0b_1a_3 & d_0b_1a_3 & a_2b_3 & b_2b_3 \\ 0 & a_0a_4 & b_0a_4 & c_2b_4 & d_2b_4 \\ c_1 & c_0d_1 & d_0d_1 & 0 & 0 \\ a_1c_3 & c_0b_1c_3 & d_0b_1c_3 & a_2d_3 & b_2d_3 \\ 0 & a_0c_4 & b_0c_4 & c_2d_4 & d_2d_4 \end{bmatrix}$
4	$\begin{bmatrix} a_0a_3 & 0 & b_0a_3 & a_2b_3 & b_2b_3 \\ c_0b_1a_4 & a_1a_4 & d_0b_1a_4 & c_2b_4 & d_2b_4 \\ c_0d_1 & c_1 & d_0d_1 & 0 & 0 \\ a_0c_3 & 0 & b_0c_3 & a_2d_3 & b_2d_3 \\ c_0b_1c_4 & a_1c_4 & d_0b_1c_4 & c_2d_4 & d_2d_4 \end{bmatrix}$
5	$\begin{bmatrix} a_1a_3 & d_0b_1a_3 & c_0b_1a_3 & a_2b_3 & b_2b_3 \\ c_1a_4 & d_0d_1a_4 & c_0d_1a_4 & c_2b_4 & d_2b_4 \\ 0 & b_0 & a_0 & 0 & 0 \\ a_1c_3 & d_0b_1c_3 & c_0b_1c_3 & a_2d_3 & b_2d_3 \\ c_1c_4 & d_0d_1c_4 & c_0d_1c_4 & c_2d_4 & d_2d_4 \end{bmatrix}$
6	$\begin{bmatrix} d_0d_1a_3 & c_0d_1a_3 & c_1a_3 & a_2b_3 & b_2b_3 \\ b_0a_4 & a_0a_4 & 0 & c_2b_4 & d_2b_4 \\ d_0b_1 & c_0b_1 & a_1 & 0 & 0 \\ d_0d_1c_3 & c_0d_1c_3 & c_1c_3 & a_2d_3 & b_2d_3 \\ b_0c_4 & a_0c_4 & 0 & c_2d_4 & d_2d_4 \end{bmatrix}$
7	$\begin{bmatrix} a_0a_3 & b_0a_3 & 0 & a_2b_3 & b_2b_3 \\ c_0d_1a_4 & d_0d_1a_4 & c_1a_4 & c_2b_4 & d_2b_4 \\ c_0b_1 & d_0b_1 & a_1 & 0 & 0 \\ a_0c_3 & b_0c_3 & 0 & a_2d_3 & b_2d_3 \\ c_0d_1c_4 & d_0d_1c_4 & c_1c_4 & c_2d_4 & d_2d_4 \end{bmatrix}$
8	$\begin{bmatrix} d_0d_1a_3 & c_1a_3 & c_0d_1a_3 & a_2b_3 & b_2b_3 \\ d_0b_1a_4 & a_1a_4 & c_0b_1a_4 & c_2b_4 & d_2b_4 \\ b_0 & 0 & a_0 & 0 & 0 \\ d_0d_1c_3 & c_1c_3 & c_0d_1c_3 & a_2d_3 & b_2d_3 \\ d_0b_1c_4 & a_1c_4 & c_0b_1c_4 & c_2d_4 & d_2d_4 \end{bmatrix}$
9	$\begin{bmatrix} d_1a_3 & b_0c_1a_3 & a_0c_1a_3 & a_2b_3 & b_2b_3 \\ 0 & d_0a_4 & c_0a_4 & c_2b_4 & d_2b_4 \\ b_1 & b_0a_1 & a_0a_1 & 0 & 0 \\ d_1c_3 & b_0c_1c_3 & a_0c_1c_3 & a_2d_3 & b_2d_3 \\ 0 & d_0c_4 & c_0c_4 & c_2d_4 & d_2d_4 \end{bmatrix}$

chel_matsel	M
10	$\begin{bmatrix} d_0a_3 & 0 & c_0a_3 & a_2b_3 & b_2b_3 \\ b_0c_1a_4 & d_1a_4 & a_0c_1a_4 & c_2b_4 & d_2b_4 \\ b_0a_1 & b_1 & a_0a_1 & 0 & 0 \\ d_0c_3 & 0 & c_0c_3 & a_2d_3 & b_2d_3 \\ b_0c_1c_4 & d_1c_4 & a_0c_1c_4 & c_2d_4 & d_2d_4 \end{bmatrix}$
11	$\begin{bmatrix} d_1a_3 & a_0c_1a_3 & b_0c_1a_3 & a_2b_3 & b_2b_3 \\ b_1a_4 & a_0a_1a_4 & b_0a_1a_4 & c_2b_4 & d_2b_4 \\ 0 & c_0 & d_0 & 0 & 0 \\ d_1c_3 & a_0c_1c_3 & b_0c_1c_3 & a_2d_3 & b_2d_3 \\ b_1c_4 & a_0a_1c_4 & b_0a_1c_4 & c_2d_4 & d_2d_4 \end{bmatrix}$

5.3.4 Processing the channel elements

Tracks routed into the stereo and multiprocessing tool ($s_{SMP,[0|1|2|...]}$) may originate from four different channel elements: `channel_pair_element`, `3_0_channel_element`, `5_X_channel_element` or `7_X_channel_element`.

Processing and mapping from input tracks to output channels ($s_{SMP,[L|R|C|...]}$) differs depending on the channel element, the codec mode and the coding config.

5.3.4.1 Processing tracks of a channel_pair_element

If `stereo_codec_mode` ∈ {SIMPLE,ASPX,ASPX_ACPL_1}, the 2 tracks of the `stereo_data()` element shall be processed according to clause 5.3.3.2. The mapping of input tracks and output channels shall be done as follows:

$$\begin{bmatrix} I_0 \\ I_1 \end{bmatrix} = \begin{bmatrix} s_{SMP,0} \\ s_{SMP,1} \end{bmatrix}$$

$$\begin{bmatrix} s_{SMP,L} \\ s_{SMP,R} \end{bmatrix} = \begin{bmatrix} O_0 \\ O_1 \end{bmatrix}$$

If `stereo_codec_mode` = ASPX_ACPL_2, the single track of the `sf_data()` element shall be mapped to the output channels as follows:

$$\begin{bmatrix} s_{SMP,L} \\ s_{SMP,R} \end{bmatrix} = \begin{bmatrix} 1 & 0 \end{bmatrix}\begin{bmatrix} s_{SMP,0} \end{bmatrix}$$

5.3.4.2 Processing tracks of a 3_0_channel_element

If `3_0_coding_config = 0`, the 2 tracks of the `stereo_data()` element ($s_{SMP,0}$, $s_{SMP,1}$) are processed according to clause 5.3.3.2. The mapping of input tracks and output channels shall be done as follows:

$$\begin{bmatrix} I_0 \\ I_1 \end{bmatrix} = \begin{bmatrix} s_{SMP,0} \\ s_{SMP,1} \end{bmatrix}$$

$$\begin{bmatrix} s_{SMP,L} \\ s_{SMP,R} \end{bmatrix} = \begin{bmatrix} O_0 \\ O_1 \end{bmatrix}$$

The third track of the additional mono_data() element, $s_{SMP,2}$, shall be processed according to clause 5.3.3.1. The mapping shall be done as follows:

$$\begin{bmatrix} I_0 \end{bmatrix} = \begin{bmatrix} s_{SMP,2} \end{bmatrix}$$

$$\begin{bmatrix} s_{SMP,C} \end{bmatrix} = \begin{bmatrix} O_0 \end{bmatrix}$$

If `3_0_coding_config = 1`, the three_channel_data() element is processed according to clause 5.3.3.3. The mapping of input and output tracks is done as follows:

$$\begin{bmatrix} I_0 \\ I_1 \\ I_2 \end{bmatrix} = \begin{bmatrix} s_{SMP,0} \\ s_{SMP,1} \\ s_{SMP,2} \end{bmatrix}$$

$$\begin{bmatrix} s_{\mathrm{SMP,L}} \\ s_{\mathrm{SMP,R}} \\ s_{\mathrm{SMP,C}} \end{bmatrix} = \begin{bmatrix} O_0 \\ O_1 \\ O_2 \end{bmatrix}$$

5.3.4.3 Processing tracks of a 5_X_channel_element

The up to 5 tracks acquired from a `5_X_channel_element` shall be processed according to the channel data element they originate from. Tables in this clause describe which tracks shall be taken as input, and which channels are produced as output when processing the various channel data elements according to clause 5.3.3.

To improve readability, input and output vectors are denoted as row vectors. Furthermore, numbers in the input vector elements [0, 1, …] represent the input tracks [$s_{\mathrm{SMP},0}$, $s_{\mathrm{SMP},1}$, …] according to the bitstream order of the channel data elements. Letters in the output vector elements [L, R, …] represent output channels [$s_{\mathrm{SMP,L}}$, $s_{\mathrm{SMP,R}}$, …], respectively.

5.3.4.3.1 5_X_codec_mode ∊ {SIMPLE,ASPX}

If `5_X_codec_mode` ∊ {SIMPLE,ASPX}, five audio tracks have been acquired from the bitstream. The mapping of input tracks and output channels is described in table 174. In case `coding_config` = 0, the element `2ch_mode` determines the final output channel mapping of the two `two_channel_data()` elements.

Table 174: Input and output mapping for 5_x_codec_mode {SIMPLE,ASPX}

coding_config / element	0			1		2		3	
2ch_mode	Input	Output 0	Output 1	Input	Output	Input	Output	Input	Output
mono_data	[4]	[C]	[C]	-	-	[4]	[C]	-	-
1st two_channel_data	[0,1]	[L,R]	[L,Ls]	[3,4]	[Ls,Rs]	-	-	-	-
2nd two_channel_data	[2,3]	[Ls,Rs]	[R,Rs]	-	-	-	-	-	-
three_channel_data	-	-	-	[0,1,2]	[L,R,C]	-	-	-	-
four_channel_data	-	-	-	-	-	[0,1,2,3]	[L,R,Ls,Rs]	-	-
five_channel_data	-	-	-	-	-	-	-	[0,1,2,3,4]	[L,R,C,Ls,Rs]

5.3.4.3.2 5_X_codec_mode ∊ {ASPX_ACPL_1, ASPX_ACPL_2}

If `5_X_codec_mode` ∊ {ASPX_ACPL_1, ASPX_ACPL_2}, three audio tracks have been acquired from channel data elements. The mapping of input tracks and preliminary output channels [A, B, C] is described in table 175.

Table 175: Input and output mapping for 5_x_codec_mode {ASPX_ACPL_[1|2]}

coding_config / element	0		1	
	Input	Output	Input	Output
mono_data	[2]	[C]	-	-
1st two_channel_data	[0,1]	[A,B]	-	-
three_channel_data	-	-	[0,1,2]	[A,B,C]

If `5_X_codec_mode` ∊ {ASPX_ACPL_1}, two additional audio tracks have been acquired from two single `sf_data` elements, denoted as [$s_{\mathrm{SMP},3}$, $s_{\mathrm{SMP},4}$]. Using the extracted parameters [a_0, b_0, c_0, d_0] and [a_1, b_1, c_1, d_1] from the two associated `chparam_info` elements as described in clause 5.3.2, the output channels can be generated according to:

$$\begin{bmatrix} s_{\mathrm{SMP,L}} \\ s_{\mathrm{SMP,R}} \\ s_{\mathrm{SMP,C}} \\ s_{\mathrm{SMP,Ls}} \\ s_{\mathrm{SMP,Rs}} \end{bmatrix} = \begin{bmatrix} a_0 & 0 & 0 & b_0 & 0 \\ 0 & a_1 & 0 & 0 & b_1 \\ 0 & 0 & 1 & 0 & 0 \\ c_0 & 0 & 0 & d_0 & 0 \\ 0 & c_1 & 0 & 0 & d_1 \end{bmatrix} \times \begin{bmatrix} s_{\mathrm{SMP,A}} \\ s_{\mathrm{SMP,B}} \\ s_{\mathrm{SMP,C}} \\ s_{\mathrm{SMP,3}} \\ s_{\mathrm{SMP,4}} \end{bmatrix}.$$

If `5_X_codec_mode` ∈ {ASPX_ACPL_2}, the output channel generation simplifies to:

$$\begin{bmatrix} s_{SMP,L} \\ s_{SMP,R} \\ s_{SMP,C} \\ s_{SMP,Ls} \\ s_{SMP,Rs} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} s_{SMP,A} \\ s_{SMP,B} \\ s_{SMP,C} \end{bmatrix}.$$

5.3.4.3.3 5_X_codec_mode = ASPX_ACPL_3

If `5_X_codec_mode` = ASPX_ACPL_3, the two audio tracks ($s_{SMP,0}$, $s_{SMP,1}$) are acquired from a `stereo_data` element and processed according to clause 5.3.3.2. The mapping of input tracks and output channels shall be done as follows:

$$\begin{bmatrix} I_0 \\ I_1 \end{bmatrix} = \begin{bmatrix} s_{SMP,0} \\ s_{SMP,1} \end{bmatrix}$$

$$\begin{bmatrix} s_{SMP,L} \\ s_{SMP,R} \\ s_{SMP,C} \\ s_{SMP,Ls} \\ s_{SMP,Rs} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \times \begin{bmatrix} O_0 \\ O_1 \end{bmatrix}$$

5.3.4.4 Processing tracks of the 7_X_channel_element

The `7_X_channel_element` enables the decoder for two more channels beyond L, R, C, Ls and Rs (refer to table 87 for details). Analog to the `5_X_channel_element`, the up to 7 tracks of a `7_X_channel_element` shall be processed according to the channel data element they originate from. Hence, table notations are borrowed from clause 5.3.4.3. Note that the mapping is partially done to 'preliminary' channel descriptors first, which are again mapped to the actual channels afterwards.

5.3.4.4.1 7_X_codec_mode ∈ {SIMPLE, ASPX}

If `7_X_codec_mode` ∈ {SIMPLE,ASPX}, seven audio tracks are available. The mapping of input tracks and preliminary output channel descriptors (A,B,D,E,F,G) is described in table 176. The processing for each of the channel data elements shall be done according to clause 5.3.3. Note that for `coding_config` = 0, the output channel mapping for the first two `two_channel_data` elements depends on the element `2ch_mode`.

Table 176: Input and preliminary output mapping for 7_X_codec_mode {SIMPLE,ASPX}

element \ **coding_config**	**0**			**1**		**2**		**3**	
2ch_mode	**Input**	**Output 0**	**Output 1**	**Input**	**Output**	**Input**	**Output**	**Input**	**Output**
mono_data	[6]	[C]	[C]	-	-	[6]	[C]	-	-
1st two_channel_data	[0,1]	[A,B]	[A,D]	[3,4]	[D,E]	[4,5]	[F,G]	[5,6]	[F,G]
2nd two_channel_data	[2,3]	[D,E]	[B,E]	[5,6]	[F,G]	-	-	-	-
3rd two_channel_data	[4,5]	[F,G]	[F,G]	-	-	-	-	-	-
three_channel_data	-	-		[0,1,2]	[A,B,C]	-	-	-	-
four_channel_data	-	-		-	-	[0,1,2,3]	[A,B,D,E]	-	-
five_channel_data	-	-		-	-	-	-	[0,1,2,3,4]	[A,B,C,D,E]

If `b_use_sap_add_ch` is set, the decoder shall extract parameters a_i, b_i, c_i, d_i (i = 0…1) from the two contained `chparam_info` elements. If `b_use_sap_add_ch` = 0, the decoder shall assume the following parameters: $a_i = d_i$, = 1, $b_i = c_i$, = 0. Table 177 outlines how the decoder output shall be constructed, by mapping the preliminary output channel descriptors to the actual channels.

Table 177: Final channel mapping for 7_X_codec_mode {SIMPLE,ASPX}

channel_mode	Out
3/4/0.x	$\begin{bmatrix}\mathrm{Ls}\\\mathrm{Lrs}\\\mathrm{Rs}\\\mathrm{Rrs}\end{bmatrix}=\begin{bmatrix}a_0&b_0&0&0\\c_0&d_0&0&0\\0&0&a_1&b_1\\0&0&c_1&d_1\end{bmatrix}\times\begin{bmatrix}\mathrm{D}\\\mathrm{F}\\\mathrm{E}\\\mathrm{G}\end{bmatrix}$ $\begin{bmatrix}\mathrm{L}\\\mathrm{R}\end{bmatrix}=\begin{bmatrix}\mathrm{A}\\\mathrm{B}\end{bmatrix}$
5/2/0.x	$\begin{bmatrix}\mathrm{L}\\\mathrm{Lw}\\\mathrm{R}\\\mathrm{Rw}\end{bmatrix}=\begin{bmatrix}a_0&b_0&0&0\\c_0&d_0&0&0\\0&0&a_1&b_1\\0&0&c_1&d_1\end{bmatrix}\times\begin{bmatrix}\mathrm{A}\\\mathrm{F}\\\mathrm{B}\\\mathrm{G}\end{bmatrix}$ $\begin{bmatrix}\mathrm{Ls}\\\mathrm{Rs}\end{bmatrix}=\begin{bmatrix}\mathrm{D}\\\mathrm{E}\end{bmatrix}$
3/2/2.x	$\begin{bmatrix}\mathrm{L}\\\mathrm{Vhl}\\\mathrm{R}\\\mathrm{Vhr}\end{bmatrix}=\begin{bmatrix}a_0&b_0&0&0\\c_0&d_0&0&0\\0&0&a_1&b_1\\0&0&c_1&d_1\end{bmatrix}\times\begin{bmatrix}\mathrm{A}\\\mathrm{F}\\\mathrm{B}\\\mathrm{G}\end{bmatrix}$ $\begin{bmatrix}\mathrm{Ls}\\\mathrm{Rs}\end{bmatrix}=\begin{bmatrix}\mathrm{D}\\\mathrm{E}\end{bmatrix}$

5.3.4.4.2 7_X_codec_mode ∈ {ASPX_ACPL_1}

If `7_X_codec_mode` = ASPX_ACPL_1, seven audio tracks are available. Processing is done analog to the case when `7_X_codec_mode` = SIMPLE and `b_use_sap_add_ch` = 0, with the difference that the output channels F and G are derived from two `sf_data` elements and their associated `chparam_info` elements instead of the `two_channel_element`.

5.3.4.4.3 7_X_codec_mode ∈ {ASPX_ACPL_2}

If `7_X_codec_mode` = ASPX_ACPL_2, five audio tracks are available. The mapping of input tracks and preliminary output channel descriptors (A,B,D,E,F,G) is described in table 178. The processing for each of the channel data elements shall be done according to clause 5.3.3.

Table 178: Input and preliminary output mapping for 7_X_codec_mode {ASPX_ACPL_2}

coding_config / element	0			1		2		3	
2ch_mode	Input	Output 0	Output 1	Input	Output	Input	Output	Input	Output
mono_data	[4]	[C]	[C]	-	-	[4]	[C]	-	-
1st two_channel_data	[0,1]	[A,B]	[A,D]	[3,4]	[D,E]				
2nd two_channel_data	[2,3]	[D,E]	[B,E]			-	-	-	-
three_channel_data	-	-	-	[0,1,2]	[A,B,C]	-	-	-	-
four_channel_data	-	-	-	-	-	[0,1,2,3]	[A,B,D,E]	-	-
five_channel_data	-	-	-	-	-	-	-	[0,1,2,3,4]	[A,B,C,D,E]

The final output mapping can be derived using table 179.

Table 179: Final channel mapping for 7_X_codec_mode {ASPX_ACPL_2}

channel_mode	Out
3/4/0.x	$\begin{bmatrix} \mathrm{Ls} \\ \mathrm{Lrs} \\ \mathrm{Rs} \\ \mathrm{Rrs} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} \mathrm{D} \\ \mathrm{E} \end{bmatrix}$ $\begin{bmatrix} \mathrm{L} \\ \mathrm{R} \end{bmatrix} = \begin{bmatrix} \mathrm{A} \\ \mathrm{B} \end{bmatrix}$
5/2/0.x	$\begin{bmatrix} \mathrm{L} \\ \mathrm{Lw} \\ \mathrm{R} \\ \mathrm{Rw} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} \mathrm{D} \\ \mathrm{E} \end{bmatrix}$ $\begin{bmatrix} \mathrm{Ls} \\ \mathrm{Rs} \end{bmatrix} = \begin{bmatrix} \mathrm{D} \\ \mathrm{E} \end{bmatrix}$
3/2/2.x	$\begin{bmatrix} \mathrm{L} \\ \mathrm{Vhl} \\ \mathrm{R} \\ \mathrm{Vhr} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} \mathrm{D} \\ \mathrm{E} \end{bmatrix}$ $\begin{bmatrix} \mathrm{Ls} \\ \mathrm{Rs} \end{bmatrix} = \begin{bmatrix} \mathrm{D} \\ \mathrm{E} \end{bmatrix}$

5.4 96 and 192 kHz decoding

If bitstream element `b_hsf_ext` is set, the `ac4_hsf_ext_substream` structure contains further spectral data, scalefactor data and spectral noise fill data extending the data beyond 24 kHz. If the decoder is not capable of decoding into 96 kHz or 192 kHz output, it shall ignore the additional coefficients.

If the decoder is capable of decoding into 96 kHz or 192 kHz output, it may be configured to these higher rates. In that case, it shall read the additional data up to either twice the original block length (for 96 kHz) or four times the original block length (for 192 KHZ), and continue processing at twice or four times the block length and sampling rate.

NOTE: Streams containing high sampling frequency data do not need to employ any of the QMF domain tools.

5.5 IMDCT transform equations and block switching

5.5.1 Introduction

The choice of analysis block length is fundamental to any transform-based audio coding system. A long transform length is most suitable for input signals whose spectrum remains stationary, or varies only slowly, with time. A long transform length also provides greater frequency resolution, and hence improved coding performance for such signals. On the other hand, a shorter transform length, possessing greater time resolution, is more desirable for signals which change rapidly in time. Therefore, the time vs. frequency resolution trade-off should be considered when selecting a transform block length.

AC-4 makes this tradeoff by a flexible approach, providing full blocks and multiple partial blocks, which allows the codec to adapt the frequency/time resolution of the transform depending upon spectral and temporal characteristics of the signal being processed. The possible full block lengths - 2 048, 1 920 and 1 536 - can subsequently be split into 2, 4, 8, or 16 sub-blocks.

Clauses 5.5.2 and 5.5.3 contain information about the transform used and block-switching method in AC-4 decoders.

5.5.2 Transforms

5.5.2.1 Introduction

The Inverse Modified Discrete Cosine Transform (IMDCT) applies the inverse of the frequency mapping that was carried out in the encoder. See clause 5.5.3 for a full list of supported transform lengths.

Data and Control Interfaces

The input to the IMDCT is a block of *N* scaled spectral lines:

$\mathbf{s}_{\mathrm{IMDCT},ch}$	vector of *N* spectral lines of channel *ch*

The output from the IMDCT and the overlap/add is a block of *N* time-domain reconstructed audio samples:

$\mathbf{p}_{\mathrm{IMDCT},ch}$	vector of *N* time-domain audio samples of channel *ch*

The bitstream parameters used by the IMDCT and the overlap/add are:

N	block length, equal to the number of input spectral lines and to the number of output PCM samples. The window length is $2 \times N$.
N_{full}	block length of a full block. This is equal to the *frame_length* as specified in table 83.

The state internal to the transforms is the overlap buffer:

N_{prev}	block length of previous block. This is set to *N* after finishing the processing of the current block.
overlap	vector of delayed time-domain audio samples. The delayed samples from the previous block, which have not been windowed, will be used by the overlap/add step. Contains N_{full} values from the processing of the previous block on input and provides N_{full} values for the next block on output.

5.5.2.2 Decoding process

In the following procedure, steps 1 through 4 describe the computation of the IMDCT, which will transform the *N* spectral lines into an intermediate single real data block of length *2N* using a single *N/2* point complex IFFT with simple pre- and post-phase-shift operations. Steps 5 and 6 describe the unfolding, windowing and the overlap/add steps.

Step 1: Define the MDCT transform coefficients $X[k] = = \mathbf{s}_{\mathrm{IMDCT},ch}[k]$, $k = 0, 1, \ldots N - 1$.

Step 2: Pre-IFFT complex multiply step.

Compute *N*/2-point complex multiplication product $Z[k]$=Zr[k]+j Zi[k], $k = 0, 1, \ldots N/2 - 1$:

Pseudocode

```
for(k=0; k<N/2; k++)
{
    /* Z[k] = (X[N-2*k-1] + j * X[2*k]) * (xcos1[k] + j * xsin1[k]); */
    Zr[k] = (X[N-2*k-1] * xcos1[k] - X[2*k] * xsin1[k]);
    Zi[k] = (X[2*k] * xcos1[k] + X[N-2*k-1] * xsin1[k]);
}
```

where:

xcos1[k] = $-\cos(2\pi \times (8k + 1) / 16N)$,

xsin1[k] = $-\sin(2\pi \times (8k + 1) / 16N)$,

and j is the imaginary unit.

Step 3: Complex IFFT step.

Compute *N/2*-point complex IFFT of $Z[k]$ to generate complex-valued sequence $z[n]$ (using a direct formulation instead of the fast fourier form for clarity):

Pseudocode

```
for(n=0; n<N/2; n++)
{
    z[n] = 0;
    for(k=0; k<N/2; k++)
    {
        z[n] + = Z[k] * (cos(4*pi*k*n/N) + j * sin(4*pi*k*n/N));
    }
}
```

Step 4: Post-IFFT complex multiply step.

Compute *N/2*-point complex multiplication product $y[n]$=yr[n]+j yi[n], n = 0,1,...N/2 - 1 as:

Pseudocode

```
for(n=0; n<N/2; n++)
{
    /* y[n] = z[n] * (xcos1[n] + j * xsin1[n]); */
    yr[n] = (zr[n] * xcos1[n] - zi[n] * xsin1[n]);
    yi[n] = (zi[n] * xcos1[n] + zr[n] * xsin1[n]);
}
```

where:

$zr[n] = real(z[n])$,

$zi[n] = imag(z[n])$, and

$xcos1[n]$ and $xsin1[n]$ are as defined in step 2 above.

Step 5: Unfolding, left-half windowing and de-interleaving step.

Compute left-half windowed time-domain samples $x[n]$:

Pseudocode

```
for(n=0; n<N/4; n++)
{
    x[2*n] = -yi[N/4+n] * w[2*n];
    x[2*n+1] = yr[N/4-n-1] * w[2*n+1];
    x[N/2+2*n] = -yr[n] * w[N/2+2*n];
    x[N/2+2*n+1] = yi[N/2-n-1] * w[N/2+2*n+1];
    x[N+2*n] = -yr[N/4+n];
    x[N+2*n+1] = yi[N/4-n-1];
    x[3*N/2+2*n] = yi[n];
    x[3*N/2+2*n+1] = -yr[N/2-n-1];
}
```

where:

$yr[n] = real(y[n])$,

$yi[n] = imag(y[n])$, and

$w[n]$ is the left transform window sequence which depends on N_{prev} and N:

$$w[n] = \begin{cases} 0, \ 0 \le n < N_{skip} \\ KBD\_LEFT\left(N_W, n - N_{skip}\right), \ N_{skip} \le n < N_W + N_{skip} \\ 1, \ N_W + N_{skip} \le n < N_W + 2N_{skip} \end{cases}$$

where:

$$N_W = \begin{cases} N, \ N \le N_{prev} \\ N_{prev}, \ N > N_{prev} \end{cases},$$

$$N_{skip} = (N - N_W)/2$$

and *KBD_LEFT(N,n)* is the left part of the Kaiser-Bessel derived window as specified in clause 5.5.3.

Step 6: Right-half windowing and overlap and add step.

The first half, i.e. the windowed part of the block, is overlapped with the second half of the previous block, after applying the windowing to the second half of the previous block, to produce PCM samples (the factor of 2 scaling undoes headroom scaling performed in the encoder):

Pseudocode

```
nskip = (Nfull - N)/2;

/* window second half of previous block */
nskip_prev = (Nfull - Nprev)/2;
for (n=0; n<Nprev; n++)
{
    overlap[nskip_prev+n] *= w[n];
}

/* overlap/add using first N samples from x[n] */
for (n=0; n<N; n++)
{
    overlap[nskip+n] = 2 * (x[n] + overlap[nskip+n]);
}

/* output pcm */
for (n=0; n<N; n++)
{
    pcm[n] = overlap[n];
}

/* move samples in overlap[] not stored in pcm[] */
for (n=0; n<nskip; n++)
{
    overlap[n] = overlap[N+n];
}

/* store second N samples from x[n] for next overlap/add */
for (n=0; n<N; n++)
{
    overlap[nskip+n] = x[N+n];
}
```

where:

$w[n]$ is the right transform window sequence which depends on N_{prev} and N:

$$w[n] = \begin{cases} 1, & 0 \le n < N_{skip} \\ KBD\_RIGHT(N_W, n - N_{skip}), & N_{skip} \le n < N_W + N_{skip} \\ 0, & N_W + N_{skip} \le n < N_W + 2N_{skip} \end{cases}$$

where:

$$N_W = \begin{cases} N, & N \le N_{prev} \\ N_{prev}, & N > N_{prev} \end{cases},$$

$$N_{skip} = (N_{prev} - N_W)/2$$

and *KBD_RIGHT(N,n)* is the right part of the Kaiser-Bessel derived window as specified in clause 5.5.3.

The PCM output samples for each block $\mathbf{p}_{\mathrm{IMDCT},ch}$ correspond to the content of the pseudocode array *pcm*.

Note that the arithmetic processing in the overlap/add processing, if implemented in fixed-point arithmetic, shall use saturation arithmetic to prevent overflow (wrap-around). Since the output signal consists of the original signal plus coding error, it is possible for the output signal to exceed 100 % level even though the original input signal was less than or equal to 100 % level.

5.5.3 Block Switching

The following full and partial block lengths can be employed in the AC-4 transform block-switching procedure if the sampling frequency is 44,1 kHz.

- 2 048, 1 024, 512, 256, 128

If the sampling frequency is 48 kHz, the following additional block lengths are possible:

- 1 920, 960, 480, 240, 120
- 1 536, 768, 384, 192, 96

If the sampling frequency is 96 kHz or 192 kHz the values above shall be multiplied by 2 or 4 respectively.

Kaiser-Bessel derived (KBD) windows with the appropriate alpha values are used for the inverse MDCT operation. Table 180 shows the alpha values used for the KBD windows for the various transform lengths.

Table 180: Alpha values for KBD windows for different transform lengths

	Sampling frequency									**α value for KBD window**
	44,1 kHz or 48 kHz			**96 kHz**			**192 kHz**			
Transform sizes	2 048	1 920	1 536	4 096	3 840	3 072	8 192	7 680	6 144	3
	1 024	960	768	2 048	1 920	1 536	4 096	3 840	3 072	4
	512	480	384	1 024	960	768	2 048	1 920	1 536	4,5
	256	240	192	512	480	384	1 024	960	768	5
	128	120	96	256	240	192	512	480	384	6

The Kaiser-Bessel derived (KBD) windows are defined as follows:

$$KBD\_LEFT(N,n) = \sqrt{\frac{\sum_{p=0}^{n} W(N,p,\alpha)}{\sum_{p=0}^{N} W(N,p,\alpha)}} \quad \text{for } 0 \le n < N$$

$$KBD\_RIGHT(N,n) = \sqrt{\frac{\sum_{p=0}^{2N-n-1} W(N,p,\alpha)}{\sum_{p=0}^{N} W(N,p,\alpha)}} \quad \text{for } N \le n < 2N$$

where:

W(N,n,α) is the Kaiser-Bessel kernel window function defined as:

$$W(N,n,\alpha) = \frac{I\left(\pi\alpha\sqrt{1.0-\left(\frac{2n}{N}-1\right)^2}\right)}{I(\pi\alpha)} \quad \text{for } 0 \le n < N$$

where:

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

and α is the kernel window alpha factor as specified in table 180.

Larger block lengths have better frequency resolution but reduced time resolution. When the encoder detects a transient in the signal, it switches to shorter block lengths and indicates this in the coded bitstream. The size of the new block length selected depends on the nature of the transient.

For an input block length of *N*, the number of intermediate windowed samples is 2*N*. When the current block and the preceding block are both of the same size, a constant window overlap of 50 % is used, and a history buffer stores the second half of the 2*N* samples, i.e. *N* samples. The first *N* samples of the current block are summed with windowed *N* samples of the previous block in the history buffer to produce the time-domain output samples.

When a transient occurs, a switch to a specific smaller block length is signalled in the bitstream, and the corresponding window is then applied. To preserve the time-domain aliasing cancellation properties of the MDCT and IMDCT transforms and maintain block alignment, the window shape of the right half of the preceding block is modified when the following block has a smaller length. Because the window for the right half of each block can be applied when the overlap is done with the next block, the block length data is readily available by the time the overlap is calculated.

Similarly, when switching from a smaller block length to a larger block length, the window shape of the left half of the current block is suitably modified.

The example in figure 5 shows a full block length of *N* = 2 048 samples, and the switch to a partial block length of *N*/2 = 1 024 samples and subsequent switches to *N*/4 = 512 samples and back to *N*/2 = 1 024 samples. The change in window shape at block transitions may also be seen.

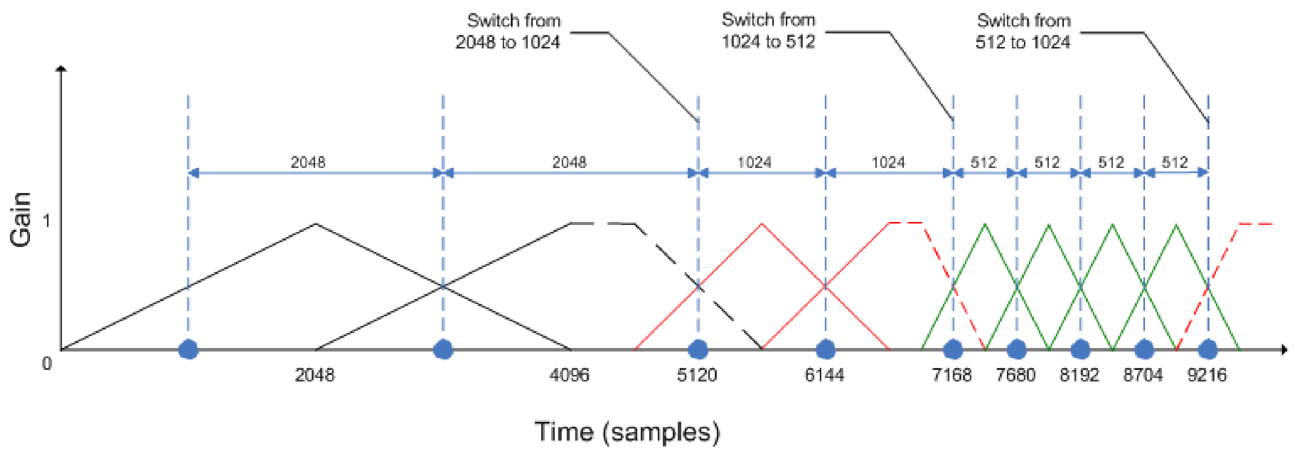

Figure 5: Block-switching during transient conditions

Partial block lengths are factors of 2, 4, 8 and 16 smaller than the full block lengths.

Two types of transitions are possible when switching from full blocks to partial blocks:

- switching from a full block to partial blocks of the same size - e.g. 2 048 to 2 x 1 024 or 2 048 to 4 x 512; and
- switching from a full block to a combination of partial block sizes - e.g. 2 048 to 1 x 1 024 + 8 x 128 or 2 048 to 2 x 512 + 1 x 1 024.

In either case, the sum total of all partial block sizes used is equal to the full block size.

Table 181 lists the various block-switching transitions permitted for initial block lengths of 2 048, 1 920 and 1 536 samples.

Table 181: Permitted block-switching transitions for full block lengths of 2 048, 1 920 and 1 536 samples

FULL BLOCK: 2 048 samples	FULL BLOCK: 1 920 samples	FULL BLOCK: 1 536 samples
1 024|1 024	960|960	768|768
1 024|2*512, 2*512|1 024	960|2*480, 2*480|960	768|2*384, 2*384|768
1 024|4*256, 4*256|1 024	960|4*240, 4*240|960	768|4*192, 4*192|768
1 024|8*128, 8*128|1 024	960|8*120, 8*120|960	768|8*96, 8*96|768
2*512|2*512	2*480|2*480	2*384|2*384
2*512|4*256, 4*256|2*512	2*480|4*240, 4*240|2*480	2*384|4*192, 4*192|2*384
2*512|8*128, 8*128|2*512	2*480|8*120, 8*120|2*480	2*384|8*96, 8*96|2*384
4*256|4*256	4*240|4*240	4*192|4*192
4*256|8*128, 8*128|4*256	4*240|8*120, 8*120|4*240	4*192|8*96, 8*96|4*192
8*256	8*240	8*192
16*128	16*120	16*96

EXAMPLE: One mono channel with *frame_length* = 1 920, consisting of 2 partial blocks of size 480 and 1 partial block of size 960 (`transf_length[0]` = 2 and `transf_length[1]` = 3). The previous frame did contain a full block:

- The first 480 spectral lines are inverse MDCT transformed. Then, the first 480 of the resulting 960 time samples are windowed using an unmodified KBD left 480 window since the preceding block did not have a shorter length. The samples in the overlap add buffer are windowed using an extended KBD right 480 window of size 1 920: the first 720 samples are unmodified, the next 480 samples are windowed using the KBD right 480 window and the last 720 samples are set to zero. Next, the 480 left windowed samples are added on a sample-by-sample basis using a factor of 2 to the 480 samples in the overlap add buffer starting at offset 720. The first 480 samples of the overlap add buffer are passed on to the PCM output buffer. The next 720 samples in the overlap add buffer are moved to the beginning of the buffer and the second 480 IMDCT output samples, the unwindowed samples, are stored in the overlap add buffer starting at offset 720.

- The second 480 spectral lines are inverse MDCT transformed. Then, the first 480 of the resulting 960 time samples are windowed using an unmodified KBD left 480 window since the preceding block did not have a shorter length. The samples in the overlap add buffer are windowed using a KBD right 480 window starting at sample 720. Next, the 480 left windowed samples are added on a sample-by-sample basis using a factor of 2 to the 480 samples in the overlap add buffer starting at offset 720. The first 480 samples of the overlap add buffer are passed on to the PCM output buffer. The next 720 samples in the overlap add buffer are moved to the beginning of the buffer and the second 480 IMDCT output samples, the unwindowed samples, are stored in the overlap add buffer starting at offset 720.

- The third block of 960 spectral lines is inverse MDCT transformed. Then, the first 960 of the resulting 1 920 time samples are windowed using an extended KBD left 480 window of size 960 since the preceding block has a shorter length: the first 240 samples are set to zero, the next 480 samples are windowed using the KBD left 480 window and the following 240 samples are left unmodified. Next, the 960 left windowed samples are added on a sample-by-sample basis using a factor of 2 to the 960 samples in the overlap add buffer starting at offset 480. The first 960 samples of the overlap add buffer are passed on to the PCM output buffer. The next 480 samples in the overlap add buffer are moved to the beginning of the buffer and the second 960 IMDCT output samples, the unwindowed samples, are stored in the overlap add buffer starting at offset 480.

In this manner, all the blocks of the current frame are processed into a composition buffer. The composition buffer needs to hold at most 4 096 samples (8 192 for a 96 kHz decoder, 16 384 for a 192 kHz decoder). After processing all blocks of the current frame, the earliest *frame_length* samples are passed on to the next tool in the processing chain, and removed from the composition buffer. Silent samples are filled in on the right.

5.6 Frame alignment

5.6.1 Introduction

AC-4 is designed to enable artefact free switching and splicing between different streams, as may happen in adaptive streaming [1]. When the input of the codec is switched to a different adaptation set, it is important to take into account the codec internal signal and metadata delays.

Because some control data apply to entire codec frames and cannot easily be delayed by fractional amounts, the PCM domain signal is delayed to make the control data delays multiples of integer frames.

Data and Control Interfaces

The input to the frame alignment tool is the time-domain signal output from the IMDCT:

$\mathbf{pin}_{\mathrm{FA},ch}$	a vector of *frame_length* time-domain audio samples of channel *ch*

The output from the frame alignment tool are delayed time-domain audio samples:

$\mathbf{pout}_{\mathrm{FA},ch}$	a vector of *frame_length* time-domain audio samples of channel *ch*

5.6.2 Decoding process

A decoder shall apply a delay between the input and output PCM samples according to:

$$out[n] = in[n - d\_pcm]$$

where *d_pcm* is defined per table 182.

Table 182: Delay in frame alignment tool

frame_rate	**PCM signal delay *d_pcm* [samples]**	**Control data delay *d_ctrl* [frames]**
23,976	288	1
24	288	1
25	352	1
29,97	96	1
30	96	1
47,952	960	2
48	960	2
50	1 056	2
59,94	672	2
60	672	2
100	1 312	4
119,88	864	4
120	864	4
21,5332 (see note 1)	352	1
23,4375 (see note 2)	352	1
NOTE 1: Music @ 44,1 kHz. NOTE 2: Music @ 48,0 kHz.		

5.7 QMF domain processing

5.7.1 Introduction

AC-4 decoders employ a complex QMF analysis/synthesis filter bank pair to enable alias suppressed frequency-domain processing. The output of the spectral frontend is transformed into a matrix of *num_qmf_subbands* complex QMF subbands as rows and *num_qmf_timeslots* time slots as columns, where *num_qmf_timeslots* is equal to (*frame_length*/*num_qmf_subbands*). Note that the entire output of a block from the spectral frontend is computed before QMF domain processing commences.

To enable tools to transition smoothly across blocks, 6 QMF time slots of history are kept in between blocks. This means that the QMF synthesis works on QMF data that is delayed by 6 QMF time slots, or 6 × *num_qmf_subbands* time domain samples.

5.7.2 QMF control data alignment

The control data of tools operating in QMF domain is at all times aligned with the corresponding spectral frontend data, contained in the `sf_data` element. This means, that within a raw AC-4 frame, the control data of any QMF domain tool shall be applied to the spectral data encoded in the very same raw AC-4 frame, regardless of the delay introduced by transforms or other tools. This is done to allow for a smoother switching behavior when decoding spliced AC-4 bitstreams.

For the decoder, this property means that it shall:

1) decode the control data of the QMF domain tool;

2) keep this control data on hold to take care of the delay introduced in the signal chain; and

3) apply it as soon as the corresponding signal data is processed by the respective QMF domain tool.

The delay introduced by the signal chain is different for different values of *frame_length*. However, due to the delay introduced in the frame alignment in clause 5.6, it always corresponds to an integer number of AC-4 frames. See value *d_ctrl* in table 182.

5.7.3 QMF analysis filterbank

5.7.3.1 Introduction

A QMF bank is used to split the time domain signal from the core decoder into *num_qmf_subbands* subband signals. The output from the filterbank, i.e. the subband samples, is complex-valued and thus oversampled by a factor two compared to a regular cosine modulated QMF bank.

Data and Control Interfaces

The input of the QMF analysis filterbank are frame-aligned time domain audio samples.

$\mathbf{pin}_{\mathrm{QMF},ch}$	a vector of *frame_length* time domain audio samples of channel *ch*

The output of the QMF analysis filterbank is a QMF matrix.

$\mathbf{Qout}_{\mathrm{QMF},ch}$	a matrix of *num_qmf_subbands* rows, and num*_qmf_timeslots* QMF time slots of channel *ch*

Each column of $\mathbf{Qout}_{\mathrm{QMF},ch}$ is vector of *num_qmf_subbands* complex values, one value for each QMF subband, representing a QMF time slot. The rows of $\mathbf{Qout}_{\mathrm{QMF,ch}}$ represent the QMF subband signals.

The bitstream and additional parameters used by the QMF analysis filterbank are:

frame_length	the frame length, equals the number of input PCM samples
num_qmf_timeslots	the number of QMF time slots, derived from *frame_length*
num_qmf_win_coef	the number of coefficients in the QMF window, 640
QWIN[num_qmf_win_coef]	the QMF window coefficients, see table D.3

The filter state is kept in:

qmf_filt	a vector of 640 delayed time-domain audio samples in reversed order, where 576 samples are saved in between calls

5.7.3.2 Decoding process

The number of QMF subbands in the analysis QMF filter bank, *num_qmf_subbands*, is always 64. The number of QMF slots is calculated as follows:

$$num\_qmf\_timeslots = \frac{frame\_length}{num\_qmf\_subbands}$$

Table 183 shows the possible values for *num_qmf_timeslots*.

Table 183: Number of QMF time slots

frame_length	2 048	1 920	1 536	1 024	960	768	384
num_qmf_timeslots	32	30	24	16	15	12	6

The filtering involves the following steps, which are also represented in the pseudocode that follows. A higher index into the vector corresponds to older samples.

- Shift the samples in the vector ***qmf_filt*** by *num_qmf_subbands* positions. The oldest *num_qmf_subbands* samples are discarded and *num_qmf_subbands* new samples are stored in positions 0 to $num\_qmf\_subbands - 1$.
- Multiply the samples of vector ***qmf_filt*** by window coefficients from *QWIN* to produce vector **z**. The window coefficients can be found in table D.3.

- Sum the samples according to the formula in the pseudocode to create the $2 \cdot num\_qmf\_subbands$ -element vector $\boldsymbol{u}$.
- Calculate 64 new subband samples by the matrix operation $\boldsymbol{M} \times \boldsymbol{u}$, where
  $$\boldsymbol{M_{k,n}} = exp\left(\frac{j \cdot \pi \cdot (k + 0.5) \cdot (2n - 1)}{2 \cdot num\_qmf\_subbands}\right), \begin{cases} 0 \le k < num\_qmf\_subbands \\ 0 \le n < 2 \cdot num\_qmf\_subbands \end{cases}$$
  In the equation, *exp()* denotes the complex exponential function and *j* is the imaginary unit.

Every loop represented in the pseudocode below produces *num_qmf_subbands* complex-valued subband samples, each representing the output from one filterbank subband. In the pseudocode ***Q***[*sb*][*ts*] corresponds to the QMF subband sample of time slot *ts* in QMF subband s*b*.

Pseudocode

```
for (ts = 0; ts < 64; ts++)
{
    /* shift time-domain input samples by 64 */
    for (sb = 639; sb >= 64; sb--)
    {
        qmf_filt[sb] = qmf_filt[sb-64];
    }

    /* feed new audio samples */
    for (sb = 64-1; sb >= 0; sb--)
    {
        qmf_filt[sb] = pcm[ts*64+63-sb];
    }

    /* multiply input samples by window coefficients */
    for (n = 0; n < 640; n++)
    {
        z[n] = qmf_filt[n] * QWIN[n];
    }

    /* sum the samples to create vector u */
    for (n = 0; n < 128; n++)
    {
        u[n] = z[n];
        for (k = 1; k <= 5; k++)
        {
            u[n] = u[n] + z[n + k*128];
        }
    }

    /* compute 64 new subband samples */
    for (sb = 0; sb < 64; sb++)
    {
        /* note that Q[sb][ts] is a complex datatype */
        Q[sb][ts] = u[0] * exp(j*(pi/128)*(sb+0.5)*(-1));
        for (n = 1; n < 128; n++)
        {
            Q[sb][ts] += u[n] * exp(j*(pi/128)*(sb+0.5)*(2*n - 1);
        }
    }
}
```

5.7.4 QMF synthesis filterbank

5.7.4.1 Introduction

Synthesis filtering of the QMF subband signals is achieved using a *num_qmf_subbands*-subband synthesis QMF bank. The output from the filterbank is real-valued time domain samples.

Data and Control Interfaces

The input to the QMF synthesis filterbank is a set of *num_qmf_subbands* complex-valued subband signals:

$\mathbf{Qin}_{\mathrm{QMF},ch}$	a matrix of *num_qmf_timeslots* QMF time slots of channel *ch*

Each column of $\mathbf{Qin}_{\mathrm{QMF},ch}$ is a vector of *num_qmf_subbands* complex values, one value for each QMF subband, representing a QMF time slot. The rows of $\mathbf{Qin}_{\mathrm{QMF},ch}$ represent the QMF subband signals.

The output of the QMF synthesis filterbank are time domain audio samples.

$\mathbf{pout}_{\mathrm{QMF},ch}$	a vector of *frame_length* time domain audio samples of channel *ch*

The bitstream and additional parameters used by the QMF synthesis filterbank are:

frame_length	the frame length, equals the number of output PCM samples
num_qmf_timeslots	the number of QMF slots, derived from *frame_length*
num_qmf_win_coef	the number of coefficients in the QMF window, 640
QWIN[num_qmf_win_coef]	the QMF window coefficients, see table D.3

The filter state is kept in:

qsyn_filt	a vector of 1 280 synthesis time domain samples, where 1 152 samples are saved in between calls.

5.7.4.2 Decoding process

The synthesis filtering comprises the following steps, where a vector ***qsyn_filt*** consisting of 1 280 samples is assumed:

- Shift the samples in the vector ***qsyn_filt*** by 128 positions. The oldest 128 samples are discarded.
- The *num_qmf_timeslots* new complex-valued subband samples are multiplied by the matrix ***N***, where:

$$\boldsymbol{N}_{n,k} = \frac{1}{num\_qmf\_timeslots} \cdot exp\left(\frac{j \cdot \pi \cdot (k+0.5) \cdot (2n - 2 * num\_qmf\_timeslots - 1)}{2 \cdot num\_qmf\_timeslots}\right), \begin{cases} 0 \le k < num\_qmf\_timeslots \\ 0 \le n < 2 \cdot num\_qmf\_timeslots \end{cases}$$

  In the equation, *exp()* denotes the complex exponential function and *j* is the imaginary unit. The real part of the output from this operation is stored in the positions 0 to *num_qmf_timeslots*-1 of vector ***qsyn_filt***.
- Extract samples from ***qsyn_filt*** to create the 10**num_qmf_timeslots*-element vector ***g***.
- Multiply the samples of vector ***g*** by window *QWIN* to produce vector ***w***. The window coefficients *QWIN* can be found in table D.3, and are the same as for the analysis filterbank.
- Calculate *num_qmf_timeslots* new output samples by summation of samples from vector ***w***.

In the pseudocode shown below, ***Q***[*sb*][*ts*] corresponds to the QMF subband sample *ts* in the QMF subband *sb*, and every new loop produces *num_qmf_timeslots* time-domain samples as output.

Pseudocode

```
for (ts = 0; ts < 64; ts++)
{
    /* shift samples by 128 */
    for (n = 1279; n >= 128; n--)
    {
        qsyn_filt[n] = qsyn_filt[n-128];
    }

    for (n = 0; n < 128; n++)
    {
        exponent = j*(pi/128)*(0.5)*(2*n - 255));
        qsyn_filt[n] = real(Q[0][ts]/64 * exp(exponent));

        for (sb = 1; sb < 64; sb++)
        {
            exponent = j*(pi/128)*(sb+0.5)*(2*n - 255);

            qsyn_filt[n] += real(Q[sb][ts]/64 * exp(exponent));
        }
    }

    for (n = 0; n <= 5; n++)
    {
        for (sb = 0; sb < 64; sb++)
        {
            g[128*n + sb]       = qsyn_filt[256*n + sb];
            g[128*n + 128 + sb] = qsyn_filt[256*n + 192 + sb];
        }
    }

    /* multiply by window coefficients */
    for (n = 0; n < 640; n++)
    {
        w[n] = g[n] * QWIN[n];
    }

    /* compute 64 new time-domain output samples */
    for (sb = 0; sb < 64; sb++)
    {
        temp = w[sb];
        for (n = 1; n <= 10; n++)
        {
            temp = temp + w[64*n + sb];
        }
        pcm[ts*64 + sb] = temp;
    }
}
```

5.7.5 Companding tool

5.7.5.1 Introduction

The companding tool is used to mitigate pre- and post-echo artefacts. It is applied on the QMF domain data. The encoder applies attenuation to signal parts with higher energy, and gain to signal parts with lower energy. The decoder applies the inverse process, which effectively shapes the coding noise by the signal energy.

Data and Control Interfaces

The input to and output from the companding tool is *M* QMF matrices, with *M* being the number of channels obtained from the channel element of the AC-4 frame. The matrices comprise exactly those QMF time slots which are part of the A-SPX interval, described in clause 5.7.6.3.3.1.

Input

$\mathbf{Qin}_{\mathrm{COMP},[a\|b\|\ldots]}$	*M* complex QMF matrices of the *M* channels to be processed by the companding tool

Output

$\mathbf{Qout}_{\mathrm{COMP},[a\|b\|\ldots]}$	*M* complex QMF matrices of the same *M* channels processed by the companding tool

In the following, the elements of the matrices $\mathbf{Qin}_{\mathrm{COMP},ch}$ and $\mathbf{Qout}_{\mathrm{COMP},ch}$ shall be denoted as $\mathrm{Qin}_{ch}(sb,ts)$ and $\mathrm{Qout}_{ch}(sb,ts)$, respectively, for each QMF subband *sb* and each QMF time slot *ts*.

The bitstream and additional information used by the companding tool is:

`b_compand_on`$_{ch}$	flag signalling whether companding is to be used for a given channel *ch*
`b_compand_avg`	flag signalling whether gains should be averaged and held constant across a frame
sync_flag	flag signalling cross-channel sync
`acpl_qmf_band`$_{ch}$	lower subband border, needed in case the codec mode is ASPX_ACPL_1
$aspx\_xover\_band_{ch}$	A-SPX crossover subband for channel *ch*

NOTE: The variable *aspx_xover_band* is derived as *sbx* in clause 5.7.6.3.1.2.

5.7.5.2 Decoding process

The QMF subband range [sb_0, sb_1] that the companding tool works on shall be determined according to:

$$sb_0 = \begin{cases} \mathrm{acpl\_qmf\_band}_{ch} & \text{in case the codec mode is ASPX\_ACPL\_1} \\ 0 & \text{otherwise} \end{cases},$$

$$sb_1 = aspx\_xover\_band_{ch}$$

Let $K = sb_1 - sb_0$ be the number of affected subbands.

Absolute sample levels $E_{ch}(sb,ts)$ shall be calculated according to:

$$E_{ch}(sb,ts) = max(|Re\{\mathrm{Qin}_{ch}(sb,ts)\}|, |Im\{\mathrm{Qin}_{ch}(sb,ts)\}|) + 0{,}5 * \ min(|Re\{\mathrm{Qin}_{ch}(sb,ts)\}|, |Im\{\mathrm{Qin}_{ch}(sb,ts)\}|)$$

A slot mean absolute level $L_{ch}(ts)$ shall be calculated according to:

$$L_{ch}(sb,ts) = 0.9105 * \frac{1}{K} \sum_{sb=sb0}^{sb1} E_{ch}(sb,ts)$$

The per-slot gain $g_{ch}(ts)$ shall be computed as:

$$g_{ch}(ts) = L_{ch}(sb,ts)^{(1-\alpha)/\alpha} \text{ where } \alpha = 0{,}65.$$

Unless `b_compand_avg` is set, the gain $\mathrm{g}_{\mathrm{ch}}(ts)$ shall be applied to all QMF samples:

$$\mathrm{Qout}_{ch}(ts,sb) \leftarrow g_{ch}(ts) \times G \times \mathrm{Qin}_{ch}(ts,sb) \text{ where } G = 0.5^{-1/\alpha}.$$

Otherwise, $g_{ch}(ts)$ shall be averaged over the entire A-SPX interval:

$$g_{avg,ch} = \frac{1}{num\_qmf\_timeslots\_in\_aspx\_int} \sum_{ts=0}^{num\_qmf\_timeslots\_in\_aspx\_int} g_{ch}(ts)$$

where *num_qmf_slots_in_aspx_int* is the number of QMF slots in the A-SPX interval, and the average gain shall be applied as:

$$\mathrm{Qout}_{ch}(ts,sb) \leftarrow g_{avg,ch} \times G \times \mathrm{Qin}_{ch}(ts,sb).$$

If multiple channels are synched (`sync_flag` = 1), then the synched per-slot gain $g_{synch}(ts)$ is computed using the per-slot gain $g_{ch}(ts)$ as:

$$g_{synch}(ts) = \frac{1}{M} \sum_{ch=0}^{M} g_{ch}(ts)$$

needs to applied on the multiple synched channels:

$$\mathrm{Qout}_{ch}(ts,sb) \leftarrow g_{synch}(ts) \times G \times \mathrm{Qin}_{ch}(ts,sb).$$

5.7.6 Advanced spectral extension tool - A-SPX

5.7.6.1 Introduction

The A-SPX tool (Advanced Spectral Processing Extension) is used to parametrically code the higher frequencies of the audio signal.

IWC (Interleaved Waveform Coding) may be employed in addition, if the parametric model cannot re-instate the correct frequency components, or to achieve a more accurate temporal reproduction of the signal.

Data and Control Interfaces

Input

$\mathbf{Qin}_{\text{ASPX},a}$	the QMF subsamples for channel *a*
$\mathbf{Qin}_{\text{ASPX},b}$	a second complex QMF matrix containing the QMF subsamples for an optional second channel *b*

Output

$\mathbf{Qout}_{\text{ASPX},a}$	a complex QMF matrix containing the A-SPX processed QMF subsamples for channel *a*
$\mathbf{Qout}_{\text{ASPX},b}$	a second complex QMF matrix containing the A-SPX processed QMF subsamples for channel *b*, in case of two input channels

The $\mathbf{Qin}_{\text{ASPX}}$ and $\mathbf{Qout}_{\text{ASPX}}$ matrices each consist of *num_qmf_timeslots* columns, and *num_qmf_subband* rows.

Control

- Dequantized control data, which are inputs to the HF generator.
- Huffman decoded and dequantized A-SPX signal and noise envelope data, which are inputs to the HF envelope adjuster.
- Signaling information used by the interleaved waveform coding block.

Figure 6 illustrates the key blocks of the A-SPX tool as well as its position in the processing chain.

The output of the IMDCT for channel *a* is frame aligned and routed into the analysis QMF bank. The filterbank generates a complex QMF matrix, containing the QMF subsamples. This matrix is subsequently processed by the companding tool, before being fed into the A-SPX tool as $\mathbf{Qin}_{\text{ASPX},a}$.

In case channel *a* is one of two channels of an `aspx_data_2ch` element, the companion channel *b* is processed the same way as channel *a*, resulting in $\mathbf{Qin}_{\text{ASPX},b}$. If not stated differently, both channels are processed likewise in the following.

In the A-SPX tool, the matrix $\mathbf{Qin}_{\text{ASPX}}$ is low band filtered at the cross over frequency to yield the matrix $\mathbf{Q}_{Low}$.

Using the parameter information transmitted in the AC-4 bitstream, and given the matrix $\mathbf{Q}_{Low}$, the HF generator performs subband tonal to noise ratio adjustment and patching operations to derive an estimate of the high frequency components of the spectrum, the matrix $\mathbf{Q}_{High}$.

The envelope adjuster then corrects the estimated spectrum $\mathbf{Q}_{High}$ by using decoded A-SPX signal and noise envelope data from the bitstream and produces matrix $\mathbf{Y}$.

The interleaved waveform coding block takes a delayed matrix $\mathbf{Qin}_{\text{ASPX}}$ and the matrix $\mathbf{Y}$ as inputs and produces an output matrix $\mathbf{Qout}_{\text{ASPX}}$, which contains the low band, spectral extension components and interleaved waveform coded components.

The matrix $\mathbf{Qout}_{\text{ASPX}}$ is finally routed into the synthesis QMF bank, unless additional processing occurs in the QMF domain.

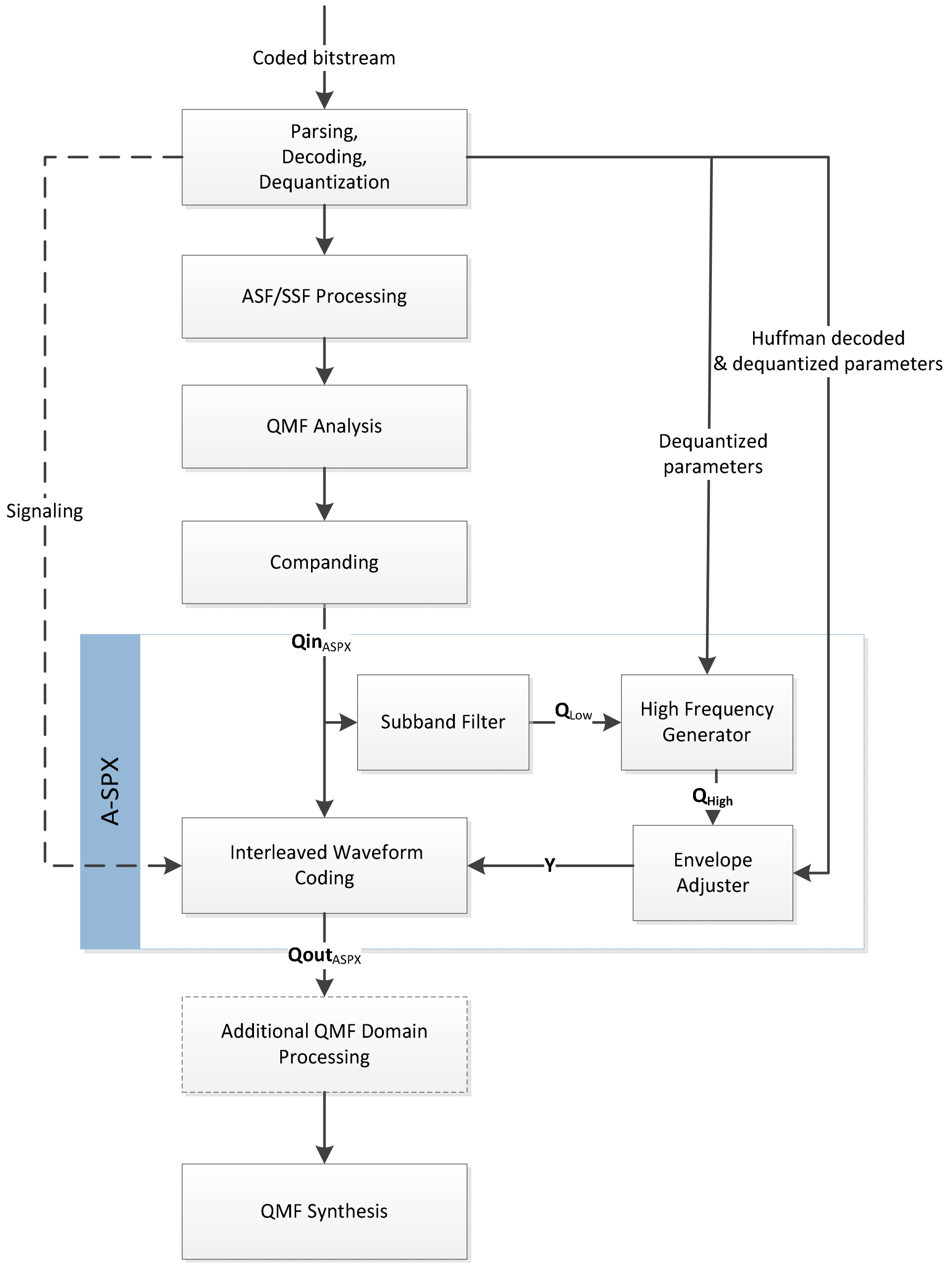

Figure 6: A-SPX block diagram

5.7.6.2 A-SPX specific variables

sbg_master	array of master QMF subband groups, of length *num_sbg_master*+1
sbg_sig_highres	array of QMF subband group borders for high frequency resolution signal envelopes, is of length *num_sbg_sig_highres*+1
sbg_sig_lowres	array of QMF subband group borders for low frequency resolution signal envelopes, is of length *num_sbg_sig_lowres*+1
sbg_lim	is of length *num_sbg_lim*+1 and contains frequency borders used by the limiter.
sbg_noise	array of QMF subband group borders for adaptive noise envelope addition, is of length *num_sbg_noise+1*
sbg_patches	array of QMF subband group borders for HF patches, is of length *num_sbg_patches+1*
sbg_sig	matrix of QMF subband group borders for low and high frequency resolution signal envelopes, contains 2 columns, column zero is of length *num_sbg_sig_lowres*+1, and column one is of length *num_sbg_sig_highres*+1
scf_sig_sbg	signal envelope scale factors, matrix of *num_atsg_sig* columns, each column *env* containing either *num_sbg_sig [env]* rows

scf_noise_sbg	noise envelope scale factors, matrix of *num_atsg_noise* columns and *num_sbg_noise* rows
sbx	the first QMF subband in the A-SPX range
sba	the first QMF subband in the *sbg_master* table
num_sb_aspx	number of QMF subbands in the A-SPX range
num_sbg_lim	number of limiter subband groups
num_sbg_master	number of subband groups in the master subband group array
num_sbg_noise	number of noise envelope subband groups
num_sbg_sig	array with 2 entries for the number of subband groups for signal envelopes, entry 0 for low resolution envelopes, entry 1 for high resolution envelopes
num_sbg_patches	number of patch subband groups
num_aspx_timeslots	number of A-SPX time slots
atsg_sig	array containing start and stop time borders for all signal envelopes in the current A-SPX interval, is of length *num_atsg_sig+1*
atsg_noise	array containing start and stop time borders for all noise envelopes in the current A-SPX interval, is of length *num_atsg_noise+1*
num_atsg_sig	number of signal envelopes in an A-SPX interval
num_atsg_noise	number of noise envelopes in an A-SPX interval
ts_offset_hfadj	time slot offset for the Envelope Adjuster
ts_offset_hfgen	time slot offset for the HF Generator
$\mathbf{Q}_{\mathbf{low}}$, *Q_low*	is the complex input QMF subband matrix to the HF generator, contains *num_qmf_timeslots* and *num_qmf_subbands* rows
$\mathbf{Q}_{\mathbf{high}}$, *Q_high*	is the complex output QMF subband matrix of the HF generator, contains *num_qmf_timeslots* and *num_sb_aspx* rows
Y	is the complex output QMF bank subband matrix from the HF envelope adjuster

5.7.6.3 Decoding A-SPX control data

5.7.6.3.1 Subband groups

QMF subband samples are combined to form subband groups, short-handed "sbg". These are described by subband group tables, which are of six types:

1) Template subband group tables, *sbg_template_[high|low]res*:
Static high and low resolution subband group tables serving as template for deriving the master table.

2) Master subband group table for signal envelopes, *sbg_master*:
This is the table from which all the other subband group tables are either directly (high and low resolution subband group tables) or indirectly (noise and limiter subband group tables) derived.

3) Subband group table for high/low resolution signal envelopes, *sbg_sig_[high|low]res*:
This table contains the subband borders of signal envelopes with high/low frequency resolution.

4) Subband group table for noise envelopes, *sbg_noise*:
This table contains the subband borders of noise envelopes.

5) Subband group table for the limiter, *sbg_lim*:
This table is contains subband borders used by the limiter.

6) Subband group table for patches, *sbg_patches*:
This table is contains subband borders used by HF patch generator.

The subband group tables contain the QMF subbands which mark the lower borders of each group. Consecutive groups lie side by side in the spectrum, hence the upper borders of the groups are implicitly defined by "lower border + 1". Additionally, the subband group tables contain the upper border of the highest subband group.

The number of subband groups in each table is described by the notation *num_[subband_group_name]*, e.g. *num_sbg_sig_highres*. The number of subband borders in each table is therefore *num_[subband_group_name]* + 1.

The parameters required by the decoder to calculate the subband group tables are transmitted in the A-SPX header, described by the bitstream element `aspx_config`. The headers are only sent with every I-frame. Hence, the parameters sent in each `aspx_config` are valid until new parameter information is transmitted in a subsequent I-frame.

If the decoder does not have this header information, e.g. in the case of an irregularly spliced bitstream where the subsequent AC-4 frame is not at an I-frame, it is unable to replicate high frequencies until a valid A-SPX header is received.

5.7.6.3.1.1 Master subband group table

Two static subband group tables are defined, one for low frequency resolutions (*sbg_template_lowres*) and one for high frequency resolutions (*sbg_template_highres*). The master table is derived from these statically pre-defined tables.

The template subband group tables are defined as:

$$sbg\_template\_lowres = [10,11,12,13,14,15,16,17,18,19,20,22,24,26,28,30,32,35,38,42,46]$$

$$sbg\_template\_highres = [18,19,20,21,22,23,24,26,28,30,32,34,36,38,40,42,44,47,50,53,56,59,62]$$

The high bit rate table supports up to 22 subband groups ranging from QMF subband 18 to subband 62. The low bit rate table starts at QMF subband 10 and goes to subband 46, having up to 20 subband groups.

The following three parameters are needed to derive the current master table from the static template tables:

1) A-SPX start frequency (`aspx_start_freq`): A 3-bit (0-7) index into the template tables starting from the first QMF subband (18 or 10) moving upwards with steps of 2. An `aspx_start_freq` of 1 will hence point to QMF subband 20 in the high resolution template table.

2) A-SPX stop frequency (`aspx_stop_freq`): A 2-bit 2 (0-3) index into the template tables starting from the last QMF subband (62 or 46) going downwards with steps of 2. An `aspx_stop_freq` of 2 will hence point to QMF subband 50 in the high resolution template table.

3) A-SPX master scale (`aspx_master_scale`): A 1-bit (0-1) value indicating which of the two static template tables is currently used. `aspx_master_scale` = 0 is the low resolution template table and `aspx_master_scale` = 1 indicates its high resolution counterpart.

Tables 184 and 185 list the possible start and stop subband for the two static template tables, and the corresponding frequencies in Hz with respect to a 48 kHz sampling frequency:

Table 184: Master table start/stop subbands for low bit rate template table

Low resolution subband group table (aspx_master_scale =0)					
aspx_start_freq	**QMF subband**	**Frequency [Hz]**	**aspx_stop_freq**	**QMF subband**	**Frequency [Hz]**
0	10	3 750	0	46	17 250
1	12	4 500	1	38	14 250
2	14	5 250	2	32	12 000
3	16	6 000	3	28	10 500
4	18	6 750			
5	20	7 500			
6	24	9 000			
7	28	10 500			

Table 185: Master table start/stop subbands for high bit rate template table

High resolution subband group table (aspx_master_scale =1)					
aspx_start_freq	**QMF subband**	**Frequency [Hz]**	**aspx_stop_freq**	**QMF subband**	**Frequency [Hz]**
0	18	6 750	0	62	23 250
1	20	7 500	1	56	21 000
2	22	8 250	2	50	18 750
3	24	9 000	3	44	16 500
4	28	10 500			
5	32	12 000			
6	36	13 500			
7	40	15 000			

The master subband group table, *sbg_master*, the number of master subband groups, *num_sbg_master*, the lower border of the first subband group, *sba*, and the upper border of the last subband group, *sbz,* are derived from the pseudocode below:

Pseudocode

```
if (master_reset == 1)
{
    if (aspx_master_scale == 1)
    {
        num_sbg_master = 22 - 2 * aspx_start_freq - 2 * aspx_stop_freq;
        for (sbg=0;sbg<=num_sbg_master;sbg++) {
            sbg_master[sbg] = sbg_template_highres(2 * aspx_start_freq + sbg);
        }
    }
    else
    {
        num_sbg_master = 20 - 2 * aspx_start_freq - 2 * aspx_stop_freq;
        for (sbg=0;sbg<=num_sbg_master;sbg++) {
            sbg_master[sbg] = sbg_template_lowres(2 * aspx_start_freq + sbg);
        }
    }
}

sba = sbg_master[0];
sbz = sbg_master[num_sbg_master];
```

wherein *master_reset* is set to 1 if any of the following parameters has changed compared to the values sent in the previous I-frame:

`aspx_master_scale`

`aspx_start_freq`

`aspx_stop_freq`

5.7.6.3.1.2 Signal envelope subband group tables

The subband group table for high frequency resolution signal envelopes, *sbg_sig_highres*, covers the upper subband groups from *sbg_master*. For that, *sbg_master* gets split in two parts using the A-SPX cross over subband offset, `aspx_xover_subband_offset`, which is a 2-bit (0-3) value. The `aspx_xover_subband_offset` is an index into the master subband group table starting at the first band, moving upward with a step of 1 subband group.

Hence, *sbg_sig_highres* is a truncated version of *sbg_master*, starting at the cross over subband, as shown in the pseudocode below:

Pseudocode

```
num_sbg_sig_highres = num_sbg_master - aspx_xover_subband_offset;

for (sbg=0; sbg<=num_sbg_sig_highres; sbg++)
    sbg_sig_highres[sbg] = sbg_master[sbg + aspx_xover_subband_offset];

sbx = sbg_sig_highres[0];

num_sb_aspx = sbg_sig_highres[num_sbg_sig_highres] - sbx;
```

The additional variables derived here are the number of subbands in A-SPX range, *num_sb_aspx,* and the cross over subband, *sbx*.

The low resolution subband group table for signal envelopes, *sbg_sig_lowres*, is always a perfect decimation of its high resolution counterpart by a factor 2. It is derived from *sbg_sig_highres* together with the number of subbands for low resolution signal envelopes, *num_sbg_sig_lowres*, as shown in the pseudocode below:

Pseudocode

```
num_sbg_sig_lowres = num_sbg_sig_highres - floor(num_sbg_sig_highres/2);

sbg_sig_lowres[0] = sbg_sig_highres[0];

if (mod(num_sbg_sig_highres, 2) == 0) {
{
    /* num_sbg_sig_highres even */
    for (sbg=1; sbg<=num_sbg_sig_lowres; sbg++)
        sbg_sig_lowres[sbg] = sbg_sig_highres[2*sbg];
} else {
    /* num_sbg_sig_highres odd */
    for (sbg=1; sbg<=num_sbg_sig_lowres; sbg++)
        sbg_sig_lowres[sbg] = sbg_sig_highres[2*sbg-1];
}

num_sbg_sig[0] = num_sbg_sig_lowres;
num_sbg_sig[1] = num_sbg_sig_highres;
```

An additional array, *num_sbg_sig*, containing both the numbers of the high and low resolution signal envelope subband groups, is derived here for convenience.

As seen from tables 184 and 185, the subband group tables always start and end on an even numbered QMF subband. For some choices of `aspx_xover_subband_offset`, the first subband group of the high and low resolution subband group table will be identical.

5.7.6.3.1.3 Noise subband group table

The noise envelope subband group table *sbg_sig_noise*, is obtained from *sbg_sig_lowres* according to:

Pseudocode

```
num_sbg_noise = max(1, floor(aspx_noise_sbg * log2(sbz/sbx) + 0.5));

idx[0] = 0;
sbg_noise[0] = sbg_sig_lowres[0];

for (sbg=1; sbg<=num_sbg_noise; sbg++)
{
    idx[sbg]  = idx[sbg-1];
    idx[sbg] += floor((num_sbg_sig_lowres - idx[sbg-1])/(num_sbg_noise + 1 - sbg));
    sbg_noise[sbg] = sbg_sig_lowres[idx[sbg]];
}
```

Note that the number of noise envelope subband groups, *num_sbg_noise*, shall satisfy $num\_sbg\_noise \leq 5$.

5.7.6.3.1.4 Patch subband group table

The patch subband group table, *num_sbg_patches*, and the number of patches, *num_sbg_patches*, as well as the derived table with the number of subbands per patch, *sbg_patch_num_sb* are created using the following pseudocode.

Note that the base sampling frequency, *base_samp_freq*, influences the patch table generation, as well.

Pseudocode

```
msb = sba;
usb = sbx;
num_sbg_patches = 0;
goal_sb = NINT( 2.048E6 / base_samp_freq );

if (aspx_master_scale == 1)
    source_band_low = 2;
else
    source_band_low = 4;

if (goal_sb < sbx + num_sb_aspx) {
    for (i = 0, sbg = 0 ; sbg_master[i] < goal_sb; i++){
        sbg = i + 1;
    }
} else {
    sbg = num_sbg_master;
}

do {
    j=sbg;
    sb = sbg_master[j];
    odd = (sb - 2 + sba) % 2;

    while (sb > ( sba - source_band_low + msb - odd )) {
        j--;
        sb = sbg_master[j];
        odd = (sb - 2 + sba) % 2;
    }

    sbg_patch_num_sb[num_sbg_patches] = max( sb - usb, 0);

    if (sbg_patch_num_sb[num_sbg_patches] > 0){
        usb = sb;
        msb = sb;
        num_sbg_patches = num_sbg_patches + 1;
    } else {
        msb = sbx;
    }
    if (sbg_master[sbg] - sb < 3) {
        sbg = num_sbg_master;
    }
} while (sb != (sbx + num_sb_aspx));

if ((sbg_patch_num_sb[num_sbg_patches-1] < 3) && (num_sbg_patches > 1)) {
    num_sbg_patches--;
}

sbg_patches[0] = sbx;
for (i = 1; i < num_sbg_patches; i++)
{
    sbg_patches[i] = sbg_patches[i-1] + sbg_patch_num_sb[sbg-1];
}
```

Note that the number of patches, *num_sbg_patches*, shall satisfy $num\_sbg\_patches \leq 5$.

5.7.6.3.1.5 Limiter subband group table

The limiter subband group table *sbg_lim*, and the number of limiter subband groups, num_sbg_lim, is derived from the low frequency resolution table, *sbg_sig_lowres*, and the patch table, *sbg_patches*, as shown in the pseudocode below. It has 2 limiter subband groups per octave.

Pseudocode

```
/* Copy sbg_sig_lowres into lower part of limiter table */
for (sbg=0; sbg<=num_sbg_sig_lowres; sbg++)
{
    sbg_lim[sbg] = sbg_sig_lowres[sbg];
}

/* Copy patch borders into higher part of limiter table */
for (sbg=1; sbg<num_sbg_patches; sbg++);
{
    sbg_lim[sbg+num_sbg_sig_lowres] = sbg_patches[sbg];
}

/* Sort patch borders + low res sbg into temporary limiter table */
sort(sbg_lim);

sbg=1;
num_sbg_lim = num_sbg_sig_lowres + num_sbg_patches - 1;

while (sbg <= num_sbg_lim)
{
    num_octaves = log2(sbg_lim[sbg] / sbg_lim[sbg-1]);

    if (num_octaves < 0.245) {
        if (sbg_lim[sbg] == sbg_lim[sbg-1]) {
            sbg_lim = remove_element(sbg_lim, num_sbg_lim, sbg);
            num_sbg_lim--;
            continue;
        } else {
            if (is_elem_of_sbg_patches(sbg_lim[sbg]) {
                if (is_element_of_sbg_patches(sbg_lim[sbg-1]) {
                    sbg++;
                    continue;
                } else {
                    sbg_lim = remove_element(sbg_lim, num_sbg_lim, sbg-1);
                    num_sbg_lim--;
                    continue;
                }
            } else {
                sbg_lim = remove_element(sbg_lim, num_sbg_lim, sbg);
                num_sbg_lim--;
                continue;
            }
        }
    } else {
        sbg++;
        continue;
    }
}
```

The functions *is_element_of_sbg_patches()* and *remove_element()* are defined as:

Pseudocode

```
is_element_of_sbg_patches(sbg_lim[sbg])
{
    for (i=0; i<=num_sbg_patches; i++) {
       if (sbg_patches[i] == sbg_lim[sbg])
           return TRUE;
    }
    return false;
}
```

and

Pseudocode

```
remove_element(sbg_lim,num_sbg_lim, sbg)
{
    for (i=sbg; i<num_sbg_lim; i++) {
       sbg_lim[i] = sbg_lim[i+1];
    }
    return sbg_lim;
}
```

5.7.6.3.2 Low band filter and QMF delay line

The QMF time-frequency samples from the analysis filterbank, stored in $\mathbf{Qin}_{\text{ASPX}}$, are truncated at the cross over subband, *sbx*, and delayed by *ts_offset_hfgen* before being fed into the HF Generator according to the following pseudocode.

Pseudocode

```
for (ts=0; ts<ts_offset_hfgen; ts++)
{
    for (sb=0; sb<num_qmf_subbands; sb++)
    {
        if (sb<sbx)
             Q_low(sb,ts) = Q_prev(sb,ts+num_qmf_timeslots-ts_offset_hfgen)
    }
}
for (ts=ts_offset_hfgen; ts<(num_qmf_timeslots+ts_offset_hfgen); ts++)
{
    for (sb=0; sb<num_qmf_subbands; sb++)
    {
        if (sb<sbx)
            Q_low(sb,ts) = Q(sb, ts-ts_offset_hfgen)
    }
}
```

Here, *Q* represents $\mathbf{Qin}_{\text{ASPX}}$, and *Q_prev* is the $\mathbf{Qin}_{\text{ASPX}}$ matrix from the previous A-SPX interval. The crossover subband *sbx* is defined in clause 5.7.6.3.1.2.

The delay in QMF time slots, *ts_offset_hfgen*, can be derived from the table 186.

Table 186: Delay in A-SPX tool

frame_length	*num_ts_in_ats*	*ts_offset_hfgen*	Delay in samples
2 048	2	6	3 * 2 * 64 = 384
1 920	2	6	3 * 2 * 64 = 384
1 536	2	6	3 * 2 * 64 = 384
1 024	1	3	3 * 1 * 64 = 192
960	1	3	3 * 1 * 64 = 192
768	1	3	3 * 1 * 64 = 192
512	1	3	3 * 1 * 64 = 192
384	1	3	3 * 1 * 64 = 192

5.7.6.3.3 Time/frequency matrix

The A-SPX time/frequency matrix is derived from the QMF matrix *Q_low*. Similarly, it has time on the horizontal axis and frequency on the vertical axis. The A-SPX matrix is generated using two methods.

- Framing, to determine the time slot borders of the signal and noise envelopes
- Tiling, to determine the subband group structure for each of the envelopes

Before the framing is applied, the QMF time slots are mapped to A-SPX time slots. Each A-SPX time slot ("ats") consists of either one or two QMF time slots ("ts"), depending on the *frame_length* of the QMF analysis, indicated by the *num_ts_in_ats* factor in table 186.

Pseudocode

```
/* Conversion from QMF timeslots into A-SPX timeslots */
num_aspx_timeslots = num_qmf_timeslots / num_ts_in_ats;
```

5.7.6.3.3.1 Framing

The temporal extent of the time/frequency grouping for A-SPX in each frame is defined as the A-SPX interval. There are four possible A-SPX interval classes, represented by the bitstream element `aspx_int_class` - FIXFIX, FIXVAR, VARFIX, and VARVAR.

The class name has two parts - the first denotes the nature of the start (left) border of the A-SPX interval, and the second denotes the nature of the stop (right) border of the A-SPX interval.

- FIX (fixed) means that either the:
  - start border of the A-SPX interval coincides with QMF time slot 0 of *Q_low;* or the
  - stop border of the A-SPX interval coincides with the QMF time slot *num_qmf_timeslots*-1 of *Q_low*.
- VAR (variable) means that the start or the stop border of the A-SPX interval does not coincide with either time slots mentioned above, but rather is variable.

Variable framing is used to allow for extending an A-SPX interval when a transient is situated at or very close to time slots representing the FIX borders. It allows a transient to be enclosed in its own envelope. For that, borders are shifted by an offset of $1 < ts\_var\_offset <= ts\_offset\_hfgen$.

If an interval class name ends with FIX, it may only be followed by a interval class whose name starts with FIX, e.g. a FIXFIX interval may be followed either by a FIXFIX or a FIXVAR interval. Similarly, an interval class name ending with VAR may only be followed by an interval class whose name starts with VAR, e.g. a FIXVAR interval may only be followed either by a VARFIX or a VARVAR interval.

The maximum size of an A-SPX interval is that of a FIXVAR interval with size *num_qmf_timeslots* + *ts_offset_hfgen* QMF time slots. The minimum size of an A-SPX interval is that of a VARFIX interval with size *num_qmf_timeslots* - *ts_offset_hfgen* QMF time slots.

Within this interval, the A-SPX time slots are grouped in A-SPX time slot groups ("atsg"), representing the signal and noise envelopes. The borders of these groups, represented by the arrays *atsg_sig* and *atsg_noise*, respectively, are derived according to the pseudocode below.

Note that for A-SPX 2-channel data channel elements with `aspx_balance` == 0, the framing needs to be calculated separately per channel. However, the channel index, *ch*, for the bitstream elements contained in `aspx_framing` is dropped in the following in order to improve readability.

Pseudocode

```
num_atsg_sig   = aspx_num_env;
num_atsg_noise = aspx_num_noise;

if(aspx_int_class == FIXFIX) {

    atsg_sig   = tab_border[num_aspx_timeslots][num_atsg_sig];
    atsg_noise = tab_border[num_aspx_timeslots][num_atsg_noise];

    atsg_freqres[0] = freq_res(atsg_sig,0,0,num_aspx_timeslots,aspx_freq_res_mode);

    for (tsg=1; tsg<num_atsg_sig; tsg++)
        atsg_freqres[tsg] = atsg_freqres[0];

} else {

    switch(aspx_int_class) {
    case FIXVAR:
        atsg_sig[0]            = 0;
        atsg_sig[num_atsg_sig] = aspx_var_bord_right + num_aspx_timeslots;

        for (tsg=0; tsg<aspx_num_rel_right; tsg++)
            atsg_sig[num_atsg_sig-tsg-1] = atsg_sig[num_atsg_sig-tsg] -
                                           aspx_rel_bord_right[tsg];
        break;

    case VARFIX:
        if(b_iframe)
            atsg_sig[0]        = aspx_var_bord_left;
        else
            atsg_sig[1]        = previous_stop_pos-num_aspx_timeslots;
        atsg_sig[num_atsg_sig] = num_aspx_timeslots;

        for (tsg=1; tsg<=aspx_num_rel_left; tsg++)
            atsg_sig[tsg] = atsg_sig[tsg-1] + aspx_rel_bord_left[tsg];
        break;

    case VARVAR:
        if(b_iframe)
            atsg_sig[0]        = aspx_var_bord_left;
        else
            atsg_sig[1]        = previous_stop_pos-num_aspx_timeslots;
        atsg_sig[num_atsg_sig] = aspx_var_bord_right + num_aspx_timeslots;

        for (tsg=1; tsg<=aspx_num_rel_left; tsg++)
            atsg_sig[tsg] = atsg_sig[tsg-1] + aspx_rel_bord_left[tsg];

        for (tsg=0; tsg<aspx_num_rel_right; tsg++)
            atsg_sig[num_atsg_sig-1-tsg] = atsg_sig[num_atsg_sig-tsg] -
                                           aspx_rel_bord_right[tsg];
        break;
    }

    atsg_noise[0]              = atsg_sig[0];
    atsg_noise[num_atsg_noise] = atsg_sig[num_atsg_sig];

    if (num_atsg_noise>1)
        atsg_noise[1] = atsg_sig[noise_mid_border[aspx_tsg_ptr][aspx_int_class]];

    for (tsg=0; tsg<num_atsg_sig; tsg++)
        atsg_freqres[tsg] = freq_res(atsg_sig,tsg,aspx_tsg_ptr,
                                     num_aspx_timeslots,aspx_freq_res_mode);
}
```

where the *tab_border* and *noise_mid_border* arrays and the function *freq_res* are defined as follows:

Table 187: Index of the middle noise border (*noise_mid_border*)

aspx_tsg_ptr	**aspx_int_class**	
	VARFIX	**FIXVAR, VARVAR**
-1	1	num_atsg_sig-1
0	num_atsg_sig-1	max(1, min(num_atsg_sig-1, aspx_tsg_ptr)

Table 188: A-SPX time slot group borders for aspx_int_class FIXFIX (*tab_border*)

num_aspx_timeslots	aspx_num_[env|noise]		
	1	2	4
6	{0, 6}	{0, 3, 6}	{0, 2, 3, 4, 6}
12	{0, 12}	{0, 6, 12}	{0, 3, 6, 9, 12}
15	{0, 15}	{0, 8, 15}	{0, 4, 8, 12, 15}
16	{0, 16}	{0, 8, 16}	{0, 4, 8, 12, 16}

Pseudocode

```
freq_res(atsg_sig, tsg, aspx_tsg_ptr, num_aspx_timeslots, aspx_freq_res_mode)
{
    switch (aspx_freq_res_mode) {
    case 0:
        freq_res = aspx_freq_res[tsg];
        break;
    case 1:
        freq_res = 0;        // FREQ_RES_LOW
        break;
    case 2:
        if( (tsg<aspx_tsg_ptr && num_aspx_timeslots>6) ||
            (atsg_sig[tsg+1]-atsg_sig[tsg])>(num_aspx_timeslots/4+1.95) )
            freq_res = 1;    // FREQ_RES_HIGH
        else
            freq_res = 0;    // FREQ_RES_LOW
        break;
    case 3:
        freq_res = 1;        // FREQ_RES_HIGH
    }
}
```

5.7.6.3.3.2 Tiling

On the frequency axis, there are two subband group tables available for signal envelopes, *sbg_sig_highres* and *sbg_sig_lowres*, and one for the noise envelopes, *sbg_noise*. They are derived in clause 5.7.6.3.1.2 and clause 5.7.6.3.1.3, respectively.

The resolution of the subband grouping for signal envelopes depends on the bitstream element `aspx_freq_res`. The subband grouping for an envelope represented by the time slot group *atsg* will be *sbg_sig_highres* for `aspx_freq_res[`*atsg*`]` equals 1, and *sbg_sig_lowres* if `aspx_freq_res[`*atsg*`]` equals 0.

In the following clauses, the subsequent mapping is used.

Pseudocode

```
/* Mapping of high and low resolutions signal envelopes to time slot groups */
for (tsg=0; tsg<num_atsg_sig; tsg++)
{
    if (aspx_freq_res[tsg]) {
        num_sbg_sig[tsg] = num_sbg_sig_highres;
        sbg_sig[tsg]     = sbg_sig_highres;
    } else {
        num_sbg_sig[tsg] = num_sbg_sig_lowres;
        sbg_sig[tsg]     = sbg_sig_lowres;
    }
}
```

The subband grouping for any noise envelope is represented by *sbg_noise*.

Together, the time slot groups and the subband groups determine the time/frequency tiling for a given A-SPX interval.

5.7.6.3.4 Decoding A-SPX signal and noise envelopes

For each A-SPX signal envelope and noise envelope, the signal and noise scale factors respectively are delta-coded either along time or frequency directions. Upon crossing an A-SPX interval boundary (along the time axis), the first envelope in the current A-SPX interval may be delta-coded using Huffman decoding with respect to the last envelope of the previous A-SPX interval. This is true for both signal and noise envelopes.

The following pseudocode describes how to choose the correct Huffman table for decoding the bitstream elements.

Pseudocode

```
get_aspx_hcb(data_type, quant_mode, stereo_mode, hcb_type)
{
    // data_type = {SIGNAL, NOISE}
    // quant_mode = {15, 30) indicating 1.5 dB or 3 dB
    // stereo_mode = {LEVEL, BALANCE} where 0 maps to LEVEL and 1 to BALANCE
    // hcb_type = {F0, DF, DT}

    if (data_type == SIGNAL ) {
        aspx_hcb = ASPX_HCB_ENV_<stereo_mode>_<quant_mode>_<hcb_type>;
        // the line above expands using the inputs stereo_mode, quant_mode and hcb_type
        // example for the expansion (stereo_mode=LEVEL, quant_mode=15 and hcb_type=DF)
        // aspx_hcb = ASPX_HCB_ENV_LEVEL_15_DF;
    }
    else {  // NOISE
        aspx_hcb = ASPX_HCB_NOISE_<stereo_mode>_<hcb_type>;
    }

    // The 18 A-SPX Huffman codebooks are given in table A.16 to table A.33
    // and are named according to the scheme outlined above.

    return aspx_hcb;
}
```

The quantized signal scale factors for each subband group, *qscf_sig_sbg,* are derived from the delta-coded signal scale factors `aspx_data_sig` as follows. Again, the channel index has been dropped from bitstream elements to improve readability.

Pseudocode

```
/* Index mapping sbg_sig_highres <-> sbg_sig_lowres */
sbg_idx_high2low[0] = 0;
sbg_idx_low2high[0] = 0;

sbg_low = 0;
for (sbg=0; sbg<num_sbg_sig_highres; sbg++) {
    if (sbg_sig_lowres[sbg_low+1] == sbg_sig_highres[sbg]) {
        sbg_low++;
        sbg_idx_low2high[sbg_low] = sbg;
    }
    sbg_idx_high2low[sbg] = sbg_low;
}

if ((ch == 1) && (aspx_balance == 1)) {
    delta = 2;
} else {
    delta = 1;
}

/* Loop over Envelopes */
for (tsg=0; tsg < num_atsg_sig; tsg++)
{
    /* Loop over scale factor subband groups */
    for (sbg=0; sbg < num_sbg_sig[tsg]); sbg++) {
        if (tsg == 0) {
            freq_res_prev[tsg]  = aspx_freq_res_prev[num_atsg_sig_prev - 1];
            qscf_prev[sbg][tsg] = qscf_sig_sbg_prev[sbg][num_atsg_sig_prev - 1];
        } else {
            freq_res_prev[tsg]  = aspx_freq_res[tsg-1];
            qscf_prev[sbg][tsg] = qscf_sig_sbg[sbg][tsg-1];
        }

        if (aspx_sig_delta_dir[tsg] == 0) {
            /* FREQ */
            qscf_sig_sbg[sbg][tsg] = 0;
            for (i = 0; i <= sbg; i++) {
                qscf_sig_sbg[sbg][tsg] += delta * aspx_data_sig[tsg][i];
            }
        } else {
            /* TIME */
            if (aspx_freq_res[tsg] == freq_res_prev[tsg]) {
                qscf_sig_sbg[sbg][tsg]  = qscf_prev[sbg][tsg]
                qscf_sig_sbg[sbg][tsg] += delta * aspx_data_sig[tsg][sbg]);
            }
            else if (aspx_freq_res[tsg] == 0) && (freq_res_prev[tsg] == 1)
            {
                qscf_sig_sbg[sbg][tsg]  = qscf_prev[sbg_idx_high2low[sbg]][tsg]
                qscf_sig_sbg[sbg][tsg] += delta * aspx_data_sig[tsg][sbg];
            }
            else if ((aspx_freq_res[tsg] == 1) && (freq_res_prev[tsg] == 0))
            {
                qscf_sig_sbg[sbg][tsg]  = qscf_prev[sbg_idx_low2high[sbg]][tsg]
                qscf_sig_sbg[sbg][tsg] += delta * aspx_data_sig[tsg][sbg];
            }
        }
    }
}
```

The signal scale factors from the last envelope of the previous A-SPX interval, *qscf_sig_sbg_prev*, are needed when delta coding in the time direction over A-SPX interval boundaries. The number of signal envelopes of the previous A-SPX interval is denoted *num_atsg_sig_prev* and is also needed in that case, as well as the frequency resolution vector of the previous A-SPX interval, denoted *aspx_freq_res_prev*.

The quantized noise envelope scale factors, *qscf_noise_sbg*, are derived from the delta coded noise envelope data `aspx_data_noise` as follows:

Pseudocode

```
if ((ch == 1) && (aspx_balance == 1)) {
    delta = 2;
} else {
    delta = 1;
}

/* Loop over envelopes */
for (tsg=0; tsg < num_atsg_noise; tsg++)
{
    /* Loop over noise subband groups */
    for (sbg=0; sbg<num_sbg_noise; sbg++)
    {
        qscf_noise_sbg[sbg][tsg] = 0;
        if (aspx_noise_delta_dir[tsg] == 0) {
            /* FREQ */
            for (i = 0; i <= sbg; i++) {
                qscf_noise_sbg[sbg][tsg] += delta * aspx_data_noise[tsg][sbg];
            }
        } else {
            /* TIME */
            if (tsg == 0) {
                qscf_noise_sbg[sbg][tsg]  = qscf_prev[sbg][num_atsg_noise_prev-1];
                qscf_noise_sbg[sbg][tsg] += delta * aspx_data_noise[tsg][sbg];
            } else {
                qscf_noise_sbg[sbg][tsg]  = qscf_noise_sbg[sbg][tsg-1];
                qscf_noise_sbg[sbg][tsg] += delta * aspx_data_noise[tsg][sbg];
            }
        }
    }
}
```

where *qscf_prev* is the noise envelope scale factors from last envelope of the previous A-SPX interval and *num_atsg_noise_prev* is the number of noise envelopes from the previous A-SPX interval.

5.7.6.3.5 Dequantization and stereo decoding

Two quantization steps are possible for the quantization of the signal scale factors:

- `aspx_quant_mode_env` = 0 corresponds to a quantization step of 1,5 dB; and
- `aspx_quant_mode_env` = 1 corresponds to a quantization step of 3,0 dB.

For a 1-channel element and for a 2-channel element with `aspx_balance` = 0, i.e. when stereo decoding is off, the quantized signal envelope scale factors, *qscf_sig_sbg*, are dequantized according to:

Pseudocode

```
if (aspx_quant_mode_env == 0)
    a = 2;
else
    a = 1;

for (tsg=0; tsg < num_atsg_sig; tsg++)
{
    for (sbg=0; sbg < num_sbg_sig[tsg]; sbg++)
    {
        scf_sig_sbg[sbg][tsg] = num_qmf_subbands * pow(2, (qscf_sig_sbg[sbg][tsg])/a)
    }
}
```

The noise envelope scale factors are dequantized as follows.

Note that when A-SPX stereo decoding is not used (`aspx_balance` = 0), the quantized noise envelope scale factors *qscf_noise_sbg* shall satisfy $scf\_noise\_sbg(sbg, tsg) \in [0{,}30]$.

Pseudocode
```
NOISE_FLOOR_OFFSET = 6;

for (tsg=0; tsg<num_atsg_noise; tsg++)
{
    for (sbg=0; sbg<num_sbg_noise; sbg++)
    {
        scf_noise_sbg[sbg][tsg] = pow(2, NOISE_FLOOR_OFFSET - qscf_noise_sbg[sbg][tsg])
    }
}
```

scf_noise_sbg and *scf_noise_sbg* are the dequantized signal and noise scale factors respectively.

Decoding of two jointly coded A-SPX channels

If `aspx_balance` = 1, a channel pair is stereo coded as a sum and balance pair and they are jointly decoded. In that case, the time envelopes, *atsg_sig* and *atsg_noise*, are identical for the channels.

In the pseudocode below, *scf_sig_sbg_a* and *scf_noise_sbg_a* are the decoded signal and noise scale factors for the sum channel *a* and *scf_sig_sbg_b* and *scf_noise_sbg_b* are the decoded signal and noise scale factors for the balance channel *b*.

As output, the dequantized scale factors for signal and noise envelopes for channels *a* and *b* are retrieved.

Pseudocode

```
pan_offset = { 24, 12 };

for (tsg=0; tsg<num_atsg_sig; tsg++)
{
    for (sbg=0; sbg<num_sbg_sig[tsg]; sbg++)
    {
        qscf_sig_sbg_a[sbg][tsg]  = pow(2, (qscf_sig_sbg_a[sbg][tsg]/a + 1)
                                    * num_qmf_subbands;
        qscf_sig_sbg_a[sbg][tsg] /= 1 + pow(2, (pan_offset[aspx_quant_mode_env]
                                    - qscf_sig_sbg_b[sbg][tsg])/a));

        qscf_sig_sbg_b[sbg][tsg]  = pow(2, (qscf_sig_sbg_a[sbg][tsg]/a + 1)
                                    * num_qmf_subbands;
        qscf_sig_sbg_b[sbg][tsg] /= 1 + pow(2, (qscf_sig_sbg_b[sbg][tsg]
                                    - pan_offset[aspx_quant_mode_env])/a));
    }
}

for (tsg=0; tsg<num_atsg_noise; tsg++)
{
    for (sbg=0; sbg<num_sbg_noise; sbg++)
    {
        scf_noise_sbg_a[sbg][tsg]  = pow(2, NOISE_FLOOR_OFFSET
                                     - qscf_noise_sbg_a[sbg][tsg] + 1)
        scf_noise_sbg_a[sbg][tsg] /= 1 + pow(2, qscf_noise_sbg_b[sbg][tsg] - 12);

        scf_noise_sbg_b[sbg][tsg]  = pow(2, NOISE_FLOOR_OFFSET
                                     - qscf_noise_sbg_a[sbg][tsg] + 1);
        scf_noise_sbg_b[sbg][tsg] /= 1 + pow(2, qscf_noise_sbg_b[sbg][tsg]) - 12);
    }
}
```

Note that when A-SPX stereo coding is used (`aspx_balance` = 1), the quantized noise and envelope scale factors of the sum channel *a* shall satisfy $qscf\_sig\_sbg\_a(sbg, tsg) \in [0,30]$ and $qscf\_sig\_sbg\_a(sbg, tsg) \in [0,24]$.

After the stereo decoding, both channels are processed likewise, but separate of each other. Hence, the channel indices are dropped in the following, and each channel's quantized noise and envelope scale factors are represented by *qscf_noise_sbg* and *qscf_sig_sbg*, respectively.

5.7.6.4 HF signal construction

5.7.6.4.1 HF generator tool

5.7.6.4.1.1 Introduction

The HF generator patches subband signals from consecutive subbands of the matrix *Q_low* to consecutive subbands of the matrix *Q_high* based on parameters transmitted in the bitstream.

The tonal to noise ratio of the subband signals *Q_high* are adjusted to the levels signaled in the bitstream by the encoder.

Prior to the subband tonal to noise ratio adjustment, a pre-flattening step is performed by which a gain value is derived from a coarse approximation of the slope of the source range, *Q_low*, used for HF generation. The inverse of this gain value is applied during the patching process.

The noise and tone generator tools are used to add adaptive noise and sinusoidals, at levels specified by the encoder, to the patched signals.

5.7.6.4.1.2 Pre-flattening control data calculation

The calculation of the pre-flattened control data involves the fitting of a third-order polynomial to the spectral envelope of the low band, *Q_low*. The fitted polynomial is a smoothed representation of the overall spectral slope of *Q_low*. The overall slope obtained is translated into a gain vector.

The pseudocode for the calculation of the pre-flattening control data is as follows:

Pseudocode

```
meanEnergy = 0;
polynomial_order = 3;

for (i=0; i<num_; i++)
{
    x[i] = i;
    slope[i] = 0;
}

/* Calculate the spectral signal envelope in dB over the current interval. */
for ( sb=0; sb<num_qmf_subbands; sb++ )
{
    pow_env[sb] = 0;
    for ( ts=atsg_sig[0]; ts<atsg_sig[num_atsg_sig]; ts++ )
    {
        pow_env[sb] += pow(Q_low_real[sb][ts],2)
        pow_env[sb] += pow(Q_low_imag[sb][ts],2)
    }
    pow_env[sb] /= atsg_sig[num_atsg_sig] - atsg_sig[0];
    pow_env[sb]  = 10*log10(pow_env[sb] + 1);
    mean_energy += pow_env[i];
}
mean_energy /= num_qmf_subbands;

poly_array = polynomial_fit(polynomial_order, num_qmf_subbands, x, pow_env);

/* Transform polynomial into slope */
for (k=polynomial_order; k>=0; k--)
{
    for(sb=0; sb< num_qmf_subbands; sb++)
    {
        slope[sb] += pow(x[sb],k)* poly_array [polynomial_order - k];
    }
}

/* Derive a gain vector from the slope */
for(sb=0; sb<num_qmf_subbands; sb++)
{
    gain_vec[sb] = pow(10,(mean_energy - slope[sb])/20);
}
```

The function *polynomial_fit* uses a least squares approach to fit to the spectral envelope of *X_low*. The result, a vector **poly_array** of length 4, contains the polynomial coefficients in a descending order of powers.

5.7.6.4.1.3 Subband tonal to noise ratio adjustment data calculation

The purpose of the per subband tonal to noise ratio adjustment is to adjust the tonal to noise ratio within the subbands of the patched signal and this is a two-step process.

The first step is to perform linear prediction within each of the QMF subband signals of $\mathbf{Q}_{Low}$. The second step is the actual tonal to noise ratio adjustment, which is again performed independently for each of the subband signals patched to $\mathbf{Q}_{High}$ by the HF generator, and is described in clause 5.7.6.4.1.4.

The subband signals are complex valued, which results in a complex covariance matrix, *cov*, for the linear prediction as well as complex filter coefficients for the filter used to control the tonal to noise ratio. The prediction filter coefficients are obtained using the covariance method. The covariance matrix elements are calculated according to the following pseudocode:

Pseudocode

```
ts_offset_hfadj = 4;

/* Create an additional delay of ts_offset_hfadj QMF time slots */
for (sb=0; sb<sba; sb++)
{
    ts_offset_prev = num_qmf_timeslots-ts_offset_hfadj;
    for (ts=0; ts<ts_hfadj; ts++)
    {
        Q_low_ext[sb][ts] = Q_low_prev[sb][ts+ts_offset_prev];
    }
    for (ts=0; ts<(num_qmf_timeslots+ts_offset_hfgen); ts++)
    {
        Q_low_ext[sb][ts-ts_offset_hfadj] = Q_low[sb][ts];
    }
}
num_ts_ext = num_qmf_timeslots+ts_offset_hfgen+ts_offset_hfadj;

/* Loop over QMF subbands */
for (sb=0; sb<sba; sb++)
{
    for (i=0; i<3; i++)
    {
        for (j=1; j<3; j++)
        {
            cov[sb][i][j] = 0;
            /* Loop over QMF time slots */
            for (ts=ts_offset_hfadj; ts<num_ts_ext; ts+=2)
            {
                cov[sb][i][j] += Q_low_ext[sb][ts-2*i]
                                 * cplx_conj(Q_low_ext[sb][ts-2*j]);
            }
        }
    }
}
```

The coefficient vectors *alpha0* and *alpha1* used to filter the subband signal are calculated as follows:

Pseudocode

```
epsilon_inv = pow(2,-20);

for (sb=0; sb<sba; sb++)
{
    denom  = cov[sb][2][2] * cov[sb][1][1];
    denom -= abs(cov[1][2]) * abs(cov[1][2]) * 1/(1+epsilon_inv);

    if (denom == 0){
        alpha1[sb]  = 0;
    } else {
        alpha1[sb]  = cov[sb][0][1] * cov[sb][1][2] - cov[sb][0][2] * cov[sb][1][1];
        alpha1[sb] /= denom;
    }

    if (cov[sb][1][1] == 0){
        alpha0[sb]  = 0;
    } else {
        alpha0[sb]  = - cov[sb][0][1] + alpha1[sb] * cplx_conj(cov[sb][1][2]);
        alpha0[sb] /= cov[sb][1][1];
    }
}
```

Note, that the pseudocode interpreter is assumed to handle complex arithmetic. The function *cplx_conj()* provides the complex conjugate of its parameter, abs() provides the magnitude of its parameter.

If either of the magnitudes of *alpha0* and *alpha1* is greater than or equal to 4, both coefficients are set to zero.

The amount of tonal to noise ratio adjustment is controlled by the values of the chirp factors, **chirp_arr**, which are calculated as shown below. Each chirp factor is used within a specific frequency range defined by the noise envelope subband group table, **sbg_noise**.

Pseudocode

```
for (sbg=0; sbg<num_sbg_noise; sbg++)
{
    new_chirp = tabNewChirp[aspx_tna_mode[sbg]][aspx_tna_mode_prev[sbg]];

    if (new_chirp < prev_chirp_array[sbg]) {
        new_chirp = 0.75000 * new_chirp + 0.25000 * prev_chirp_array[sbg];
    } else {
        new_chirp = 0.90625 * new_chirp + 0.09375 * prev_chirp_array[sbg];
    }

    if (new_chirp < 0.015625) {
        chipr_arr[sbg] = 0;
    } else {
        chipr_arr[sbg] = new_chirp;
    }
}
```

prev_chirp_array[i] contains the chirp factor values calculated in the previous A-SPX interval, and are assumed to be zero for the first A-SPX interval. *new_chirp* is a derived by a lookup function which uses *aspx_tna_mode_prev[i]* and `aspx_tna_mode`[*i*] as inputs and which can be implemented using table 189. The *aspx_tna_mode_prev[i]* values are the `aspx_tna_mode` values from the previous A-SPX interval, and are assumed to be zero for the first interval.

Table 189: Calculation of *new_chirp* values

aspx_tna_mode_prev[i]	***aspx_tna_mode[i]***			
	None	**Light**	**Moderate**	**Heavy**
None	0,0	0,6	0,9	0,98
Light	0,6	0,75	0,9	0,98
Moderate	0,0	0,75	0,9	0,98
Heavy	0,0	0,75	0,9	0,98

5.7.6.4.1.4 HF signal creation

The high frequency (HF) signal, *Q_high*, is created using the following pseudocode, which implements both the tonal to noise ratio adjustment of the lower spectrum as well as the pre-flattening.

Pseudocode

```
/* Loop over QMF time slots */
for (ts=atsg_sig[0]*num_ts_in_ats; ts<atsg_sig[num_atsg_sig]*num_ts_in_ats; ts++]
{
    sum_sb_patches = 0;
    g = 0;

    /* Loop over number of patches */
    for (i=0; i<num_sbg_patches; i++)
    {
        /* Loop over number of subbands per patch */
        for (sb=0; sb<sbg_patch_num_sb[i]; sb++)
        {
            /* Map to High QMF Subband */
            sb_high = sbx + num_sb_patches + sb;

            /* Map to current noise envelope */
            if (sbg_noise[g+1] == sb_high)
                g++;

            n = ts + ts_offset_hfadj;

            /* Current low QMF Subband */
            p = patch_start_subbands[i] + sb;

            Q_high[sb_high][ts]  = Q_low_ext[p][n];
            Q_high[sb_high][ts] += chipr_arr[g] * alpha0[p] * Q_low_ext[p][n-2];
            Q_high[sb_high][ts] += pow(chipr_arr[g],2) * alpha1[p] * Q_low_ext[p][n-4];
            Q_high[sb_high][ts] *= 1/gain_vec[p];
        }
        sum_sb_patches += sbg_patch_num_sb[i];
    }
}
```

5.7.6.4.2 HF envelope adjustment tool

The envelope adjustment process takes *Q_high* as input from the HF generator and produces as output a QMF matrix **Y**. The adjustment is performed upon the entire A-SPX range spanning the time slots of the current A-SPX interval (given by *atsg_sig*) and *num_sb_aspx* QMF subbands, starting at subband *sbx*.

5.7.6.4.2.1 Estimation of transmitted and actual envelopes in the current interval

The spectral envelopes within the current A-SPX interval are estimated depending on the bitstream element `aspx_interpolation`. The estimation is performed by taking the average of the squared complex subband samples over the time and frequency regions of the time-frequency matrix *Q_high*. The estimation can be calculated using the following pseudocode:

Pseudocode

```
/* Loop over envelopes */
for (tsg=0; tsg<num_atsg_sig; tsg++)
{
    sbg = 0;

    /* Loop over QMF subbands in A-SPX range */
    for (sb=0; sb<num_sb_aspx; sb++)
    {
        est_sig = 0;

        /* Update current subband group */
        if (sb == sbg_sig[k+1])
            sbg++;

        tsa = atsg_sig[tsg]*num_ts_in_ats + ts_offset_hfadj;
        tsz = atsg_sig[tsg+1]*num_ts_in_ats + ts_offset_hfadj;

        for (ts=tsa; ts<tsz; ts++)
        {
            if (aspx_interpolation == 0) {
                for (j=sbg_sig[sbg]; j<sbg_sig[sbg+1]; j++)
                {
                    est_sig += pow(Q_high[j][ts], 2);
                }
            } else
                est_sig += pow(Q_high[sb+sbx][ts], 2);
        }
        if (aspx_interpolation) {
            est_sig /= sbg_sig[sbg+1] - sbg_sig[sbg+1];
            est_sig /= atsg_sig[tsg+1] - atsg_sig[tsg];
        } else {
            est_sig /= atsg_sig[tsg+1] - atsg_sig[tsg];
        }
        est_sig_sb[sb][tsg] = est_sig;
    }
}
```

The frequency resolution of *est_sig_sb* equals that of the QMF filterbank. The *est_sig_sb* matrix has *num_atsg_sig* columns (one for every A-SPX envelope) and *num_sb_aspx* rows (the number of QMF subbands covered by the A-SPX range).

The Huffman decoded and dequantized signal and noise scale factors, *scf_sig_sbg* and *scf_noise_sbg*, are mapped to the resolution of the A-SPX time/frequency matrix, and the pseudo-code governing the mapping is as follows:

Pseudocode

```
tsg_noise =0;

/* Loop over Signal Envelopes */
for (tsg=0; tsg<num_atsg_sig; tsg++)
{
    /* Map Signal Envelopes from subband groups to QMF subbands */
    for (sbg=0; sbg<num_sbg_sig; sbg++)
    {
        for (sb=sbg_sig[sbg]-sbx; sb<sbg_sig[sbg+1]-sbx; sb++)
            scf_sig_sb[sb][tsg] = scf_sig_sbg[sbg][tsg];
    }

    if (atsg_sig[tsg] == atsg_noise[tsg_noise + 1])
        tsg_noise++;

    /* Map Noise Floors from subband groups to QMF subbands, and to signal envelopes */
    for (sbg=0; sbg<num_sbg_noise; sbg++)
    {
        for (sb=sbg_noise[sbg]-sbx; sb<sbg_noise[sbg+1]-sbx; sb++)
            scf_noise_sb[sb][tsg] = scf_noise_sbg[sbg][tsg_noise];
    }
}
```

The following describes how additional sinusoidal tones are added:

sine_idx_sb is a binary matrix indicating the QMF subbands in which the sinusoids are to be added.

sine_area_sb is a binary matrix indicating all QMF subbands of a subband group, into which a sinusoid is added.

The insertion of a sinusoid is signaled by the bitstream element `aspx_add_harmonic`. If a sinusoid is to be inserted in a QMF subband which, in the previous A-SPX interval, did not contain a sinusoid, the starting envelope of the inserted sinusoid is given by `aspx_tsg_ptr` in the present A-SPX interval. Further, the sinusoid will be placed in the middle of the high frequency resolution subband group. This is described by the pseudocode below:

Pseudocode

```
/* Loop over envelopes */
for (tsg=0; tsg<num_atsg_sig; tsg++)
{
    /* Loop over high resolution signal envelope subband groups */
    for (sbg=0; sbg<num_sbg_sig_highres; sbg++)
    {
        sba = sbg_sig_highres[sbg] - sbx;
        sbz = sbg_sig_highres[sbg+1] - sbx;

        sb_mid = (int) 0.5*(sbz+sba);

        /* Map sinusoid markers to QMF subbands */
        for (sb=sbg_sig_highres[sbg]-sbx; sb<sbg_sig_highres[sbg+1]-sbx; sb++)
        {
            if ((sb == sb_mid) && ((tsg >= aspx_tsg_ptr)
                || sine_idx_sb_prev[sb][num_atsg_sig_prev-1]))
            {
                sine_idx_sb[sb][tsg] = aspx_add_harmonic[sbg];
            } else {
                sine_idx_sb[sb][tsg] = 0;
            }
        }
    }
}
```

where the variables *sine_idx_sb_prev* and *num_atsg_sig_prev* are *sine_idx_sb* and *num_atsg_sig* of the previous A-SPX interval for the same subband range, respectively. If the subband range is larger for the current interval, the entries for the QMF subbands not covered by the previous *sine_idx_sb* are assumed to be zero.

The frequency resolution of signal scale factors varies and can either be coarse or fine. The frequency resolution of additional generated sinusoids is always fine. The varying frequency resolution is handled as shown in the pseudocode below:

Pseudocode

```
/* Loop over Envelopes */
for (tsg=0; tsg<num_atsg_sig; tsg++)
{
    /* Loop over subband groups */
    for (sbg=0; sbg<num_sbg_sig[tsg]; sbg++)
    {
        b_sine_present = 0;

        /* Additional sinusoid present in SF band? */
        for (sb=sbg_sig[tsg][sbg]-sbx; sb<sbg_sig[tsg][sbg+1]-sbx; sb++)
        {
            if (sine_idx_sb[sb][tsg] == 1)
                b_sine_present = 1;
        }

        /* Mark all subbands in current subband group accordingly */
        for (sb=sbg_sig[tsg][sbg]-sbx; sb<sbg_sig[tsg][sbg+1]-sbx; sb++)
        {
            sine_area_sb[sb][tsg] = b_sine_present;
        }
    }
}
```

The matrix *sine_area_sb* is hence one for all QMF subbands in the subband groups where an additional sinusoid shall be added, zero otherwise.

The resulting sinusoid and noise levels for each subband, *sine_lev_sb*, and *noise_lev_sb*, respectively, are calculated as follows:

Pseudocode

```
/* Loop over envelopes */
for (tsg=0; tsg<num_atsg_sig; tsg++)
{
    /* Loop over QMF subbands in A-SPX range */
    for (sb=0; sb<num_sb_aspx; sb++)
    {
        sine_lev_sb[sb][tsg] = sqrt(scf_sig_sb[sb][tsg] *
                                    sine_idx_sb[sb][tsg]/(1+scf_noise_sb[sb][tsg]));
        noise_lev_sb[sb][tsg] = sqrt(scf_sig_sb[sb][tsg] *
                                     scf_noise_sb[sb][tsg]/(1+scf_noise_sb[sb][tsg]));
    }
}
```

5.7.6.4.2.2 Calculation of compensatory gains

In order for the QMF subband samples signals to retain the correct envelope, compensatory gains are calculated. The level of additional sinusoids as well as the level of the additional added noise are taken into account.

The initial values for the gain, *sig_gain_sb*, are derived from the dequantized scale factors for noise and signal envelopes, as well as from the estimated actual envelope.

Pseudocode

```
if (aspx_tsg_ptr == num_atsg_sig)
    b_sine_at_end = 0;
else
    b_sine_at_end = -1;

/* Loop over envelopes */
for (tsg=0; tsg<num_atsg_sig; tsg++)
{
    /* Loop over QMF subbands in A-SPX range */
    for (sb=0; sb<num_sb_aspx; sb++)
    {
        if (sine_area_sb[sb][tsg] == 0) {
            denom    = epsilon + est_sig_sb[sb][tsg];
            if (!(tsg == aspx_tsg_ptr  || tsg == b_sine_at_end))
                denom *= (1 + scf_noise_sb[sb][tsg]);
            sig_gain_sb[sb][tsg] = sqrt(scf_sig_sb[sb][tsg] / denom);
        } else {
            denom  = epsilon + est_sig_sb[sb][tsg];
            denom *= 1 + scf_noise_sb[sb][tsg];
            sig_gain_sb[sb][tsg] = sqrt(scf_sig_sb[sb][tsg]*scf_noise_sb[sb][tsg]
                                        / denom);
        }
    }
}
```

where *aspx_tsg_ptr_prev* and *num_atsg_sig_prev* are the *aspx_tsg_ptr* and *num_atsg_sig* of the previous A-SPX interval, respectively.

The gain values are limited so as to avoid unwanted noise substitution. The pseudocode below shows the calculation of the maximum gain values used for the limiting.

Pseudocode

```
lim_gain  = 1.41254;
epsilon0 = pow(10,-12);

/* Loop over envelopes */
for (tsg=0; tsg<num_atsg_sig; tsg++)
{
    m = 0;

    /* Loop over limiter subband groups */
    for (sbg=0; sbg<num_sbg_lim; sbg++)
    {
        nom = denom = epsilon0;

        for (sb=sbg_lim[sbg]-sbx; sb<sbg_lim[sbg+1]-1-sbx; sb++)
        {
            nom   += scf_sig_sb[sb][tsg];
            denom += est_sig_sb[sb][tsg];
        }
        max_sig_gain_sbg[sbg][tsg] = sqrt(nom/denom) * lim_gain;
    }

    sbg = 0;

    /* Map to QMF subbands */
    for (sb=0; sb<num_sb_aspx; sb++)
    {
        if (sb == sbg_lim[sbg+1]-sbx)
            sbg++;

        max_sig_gain_sb[sb][tsg] = min(max_sig_gain_sbg[sbg][tsg], pow(10,5));
    }
}
```

The pseudocode below shows how the additional noise added to the HF generated signal, *noise_lev_sb*, is limited in proportion to the energy lost when the gain values are limited:

Pseudocode

```
/* Loop over envelopes */
for (tsg=0; tsg<num_atsg_sig; tsg++)
{
    /* Loop over QMF subbands */
    for (sb=0; sb<num_sb_aspx; sb++)
    {
        tmp  = noise_lev_sb[sb][tsg];
        tmp *= max_sig_gain_sb[sb][tsg]/sig_gain_sb[sb][tsg];

        noise_lev_sb_lim[sb][tsg] = min(noise_lev_sb[sb][tsg],tmp);
    }
}
```

The compensatory gain values, *sig_gain_sb*, are limited as follows:

Pseudocode

```
/* Loop over envelopes */
for (tsg=0; tsg<num_atsg_sig; tsg++)
{
    /* Loop over QMF subbands */
    for (sb=0; sb<num_sb_aspx; sb++)
    {
        sig_gain_sb_lim[sb][tsg] = min(sig_gain_sb[sb][tsg], max_sig_gain_sb[sb][tsg]);
    }
}
```

The total gain of a limiter subband group is adjusted in proportion to the energy lost during limiting. This boost factor is calculated as shown in the pseudocode below:

Pseudocode

```
epsilon0 = pow(10,-12);

/* Loop over envelopes */
for (tsg=0; tsg<num_atsg_sig; tsg++)
{
    /* Loop over limiter subband groups */
    for (sbg=0; sbg<num_sbg_lim; sbg++)
    {
        nom = denom = epsilon0;

        /* Loop over subbands */
        for (sb=sbg_lim[sbg]-sbx; sb<sbg_lim[sbg+1]-1-sbx; sb++)
        {
            nom   += scf_sig_sb[sb][tsg];

            denom += est_sig_sb[sb][tsg]*pow(sig_gain_sb_lim[sb][tsg],2);
            denom += pow(sine_lev_sb[sb][tsg],2)
            denom += pow(sine_lev_sb[sb][tsg],2)

            if (!((sine_lev_sb[sb][tsg] != 0)
                 || (tsg == aspx_tsg_ptr) || (tsg == aspx_tsg_ptr_prev)))
                denom += pow(noise_lev_sb_lim[sb][tsg],2)
        }
        boost_fact_sbg[sbg][tsg] = sqrt(nom/denom);
    }
}
```

The actual boost factor, *boost_fact_sb*, is limited in order not to get too high energy values, as follows:

Pseudocode

```
/* Loop over envelopes */
for (tsg=0; tsg<num_atsg_sig; tsg++)
{
    sbg = 0;

    /* Loop over QMF subbands */
    for (sb=0; sb<num_sb_aspx; sb++)
    {
        if (m == sbg_lim[sbg+1]-sbx)
            sbg++;

        boost_fact_sb[sb][tsg] = min(boost_fact_sbg[sbg][tsg], 1.584893192);
    }
}
```

The boost factor is then applied to the compensation gain, the noise envelope scale factors and the sinusoid levels as shown in the pseudocode below:

Pseudocode

```
/* Loop over envelopes */
for (tsg=0; tsg<num_atsg_sig; tsg++)
{
    /* Loop over QMF subbands */
    for (sb=0; sb<num_sb_aspx; sb++)
    {
        sig_gain_sb_adj[sb][tsg]  = sig_gain_sb_lim[sb][tsg]  * boost_fact_sb[sb][tsg];
        noise_lev_sb_adj[sb][tsg] = noise_lev_sb_lim[sb][tsg] * boost_fact_sb[sb][tsg];
        sine_lev_sb_adj[sb][tsg]  = sine_lev_sb[sb][tsg]      * boost_fact_sb[sb][tsg];
    }
}
```

5.7.6.4.3 Noise generator tool

Overview

The noise generator tool is used to inject noise into the recreated high-frequency spectrum in order to decrease its tonality and to more closely match the original signal.

The tool outputs a noise envelope that is subsequently added to the generated HF signal.

Data and Control Interfaces

Input

- *noise_lev_sb_adj*, a *num_sb_aspx* x *num_atsg_noise* matrix of noise levels.
- *NoiseTable*, containing 512 complex numbers with random phase and average energy of 1, as defined in table D.2.

Output

- *qmf_noise*, a *num_sb_aspx* x *num_qmf_timeslots* matrix of noise values.

Operation

The noise to be added to the recreated high-frequency subbands, *qmf_noise*, is generated according to:

Pseudocode

```
/* Loop over time slots */
tsg = 0;

for (ts=atsg_sig[0]*num_ts_in_ats; ts<atsg_sig[num_atsg_sig]*num_ts_in_ats; ts++)
{
    if (ts == atsg_sig[tsg+1]*num_ts_in_ats)
        tsg++;

    /* Loop over QMF subbands in A-SPX */
    for (sb=0; sb<num_sb_aspx; sb++)
    {
        qmf_noise[sb][ts] = noise_lev_sb_adj[sb][tsg] * NoiseTable[noise_idx(sb,ts)];
    }
}
```

The index into the noise table is calculated according to the following pseudocode:

Pseudocode

```
noise_idx(sb, ts)
{
    if (master_reset) {
        indexNoise = 0;
    } else {
        indexNoise = noise_idx_prev[sb][ts];
    }

    indexNoise += num_sb_aspx * (ts - atsg_sig[0]);
    indexNoise += sb + 1;

    return indexNoise % 512;
}
```

where *noise_idx_prev* is the last *noise_idx* from the previous A-SPX interval. The variable *master_reset* is defined in clause 5.7.6.3.1.1.

5.7.6.4.4 Tone generator tool

Overview

The tone generator tool is used to match the tonality of the original signal by adding the appropriate missing sinusoids to the recreated high band.

The tool outputs an envelope worth of values, the dimensions of which are specified in the bitstream by the encoder.

Data and Control Interfaces

Input

- *sine_lev_sb_adj*, a *num_sb_aspx* x *num_atsg_sig* matrix of boosted sine levels.

Output

- *qmf_sine*, a *num_sb_aspx* x *num_qmf_timeslots* matrix of sine tones.

Operation

The sinusoids are added at the level *sine_lev_sb_adj* for the QMF subbands. This results in the final output QMF matrix *qmf_sine*, calculated as follows:

Pseudocode

```
tsg = 0;

/* Loop over QMF time slots */
for (ts=atsg_sig[0]*num_ts_in_ats; ts<atsg_sig[num_atsg_sig]*num_ts_in_ats; ts++)
{
    if (ts == atsg_sig[tsg+1]*num_ts_in_ats)
        tsg++;

    /* Loop over QMF subbands in A-SPX */
    for (sb=0; sb<num_sb_aspx; sb++)
    {
        qmf_sine_RE[sb][ts]  = sine_lev_sb_adj[sb][tsg];
        qmf_sine_RE[sb][ts] *= SineTable_RE[sine_idx(sb,ts)];

        qmf_sine_IM[sb][ts]  = sine_lev_sb_adj[sb][tsg] * pow(-1,sb+sbx);
        qmf_sine_IM[sb][ts] *= SineTable_IM[sine_idx(sb,ts)];
    }
}
```

The index into the sine table is calculated according to the following pseudocode:

Pseudocode

```
sine_idx(sb, ts)
{
    if (first_frame) {
        index = 1;
    } else {
        index  = (sine_idx_prev[sb][ts] + 1) % 4;
    }
    index += ts - atsg_sig[0];

    return index % 4;
}
```

where *sine_idx_prev* is the last *sine_idx* from the previous A-SPX interval. The variable *first_frame* is 1 only at the codec initialization stage, 0 otherwise.

The array *SineTable* is defined in table 190.

Table 190: Sine table for tone generator tool

index	SineTable_RE(index)	SineTable_IM(index)
0	1	0
1	0	1
2	-1	0
3	0	-1

5.7.6.4.5 HF signal assembling tool

The compensated gain values are applied to the input subband matrix *Q_high*, for all signal envelopes of the current A-SPX interval as follows.

Note, in case of variable A-SPX interval borders, the previously assembled HF signals, *Y_prev* are pushed into the current output buffer *Y*.

Pseudocode

```
tsg = 0;
/* Get delayed QMF subsamples from delay buffer */
for (ts=0; ts<atsg_sig[0]*num_ts_in_ats; ts++)
    for (sb=0; sb<num_sb_aspx; sb++)
    {
        Y[sb][ts]  = Y_prev[sb][num_qmf_timeslots+ts];
    }
}

/* Loop over QMF time slots */
for (ts=atsg_sig[0]*num_ts_in_ats; ts<atsg_sig[num_atsg_sig]*num_ts_in_ats; ts++)
{
    if (ts == atsg_sig[tsg+1]*num_ts_in_ats)
        tsg++;

    /* Loop over QMF subbands */
    for (sb=0; sb<num_sb_aspx; sb++)
    {
        Y[sb][ts]  = sig_gain_sb_adj[sb][tsg];
        Y[sb][ts] *= Q_high[sb+sbx][ts+ts_offset_hfadj];
    }
}
```

The noise level, *noise_lev_sb_adj*, is passed as an input to the noise generator tool, and the generated noise *qmf_noise* is then added to the matrix **Y** as follows:

Pseudocode

```
/* Loop over time slots */
for (ts=atsg_sig[0]; ts<atsg_sig[num_atsg_sig]; ts++)
{
    /* Loop over QMF subbands */
    for (sb=0; sb<num_sb_aspx; sb++)
    {
        Y[sb][ts]  = Y[sb][ts] + qmf_noise[sb][ts];
    }
}
```

Similarly, the level of the sinusoids, *sine_lev_sb_adj*, is passed as an input to the tone generator tool and the resultant sinusoids *qmf_sine* are added to the matrix **Y** as follows:

Pseudocode

```
/* Loop over time slots */
for (ts=atsg_sig[0]; ts<atsg_sig[num_atsg_sig]; ts++)
{
    /* Loop over QMF subbands */
    for (sb=0; sb<num_sb_aspx; sb++)
    {
        Y[sb][ts]  = Y[sb][ts] + qmf_sine[sb][ts];
    }
}
```

5.7.6.5 Interleaved waveform coding

5.7.6.5.1 Introduction

A-SPX involves patching the spectral content of suitable low frequency regions to higher frequencies and adjusting them such that they perceptually match the high frequency content that was present in the original signal. The entire signal spectrum above the A-SPX crossover frequency is derived from the spectrum below the crossover frequency, and is augmented by synthetic signal components - noise and sinusoids - that are created using a parametric model.

Interleaved waveform coding is employed either when critical signal frequency components are not present below the A-SPX crossover frequency, or when more faithful reproduction of transients is needed. Waveform coded components (WCC) and spectral extension components (SEC) can be interleaved either in frequency or in time. The signal characteristics determine whether frequency interleaving or time interleaving is more beneficial.

Signals with a significant degree of tonal high frequency content will profit from waveform-coding spectral lines that contain the tonal components and interleaving them with the spectrum created by spectral extension. If frequency-interleaving is signaled in the bitstream the WCC are added to the SEC.

On the other hand, the higher frequencies of a transient event can be coded more accurately by a short segment of waveform-coding and interleaved in time with a spectrum generated by spectral extension. In the case of time interleaving, the WCC are continuous frequency regions that match the QMF subband structure. Hence the method for time interleaved coding is replacing the SEC with the corresponding WCC rather than addition. The resolution of time-interleaved waveform coding is the stride factor (*num_ts_in_ats*).

5.7.6.5.2 Signaling interleaved waveform coding

The use of frequency-interleaved waveform coding in an A-SPX interval is indicated by the bitstream element `aspx_fic_present`. The presence of a frequency-interleaved waveform component in a particular subband group is indicated by the bitstream element `aspx_fic_used_in_sfb`.

If a channel pair stereo coded, i.e. `aspx_balance` = 1, the bitstream elements `aspx_fic_left` and `aspx_fic_right` indicate whether or not frequency-interleaved waveform coding is used in the left and right channels respectively. The bitstream element `aspx_fic_used_in_sfb` takes an additional channel index.

The use of time-interleaved waveform coding in an A-SPX interval is indicated by the bitstream element `aspx_tic_present`. The presence of a time-interleaved waveform component in a particular QMF time slot is indicated by the bitstream element `aspx_tic_used_in_slot`.

If a channel pair is stereo coded, i.e. `aspx_balance` = 1, the bitstream elements `aspx_tic_left` and `aspx_tic_right` indicate whether or not time-interleaved waveform coding is used in the left and right channels respectively. The bitstream element `aspx_tic_used_in_slot` takes an additional channel index. If the bitstream element `aspx_tic_copy` is set, information regarding the presence of time-interleaved waveform components in the slots of the left channel is copied to the right channel, resulting in both channels having time-interleaved components in the same slots.

The relative priorities of frequency-interleaving, time-interleaving and sinusoid synthesis (using the tone generator tool) are in the following descending order of priority:

1) Time-interleaved coding overrides both frequency-interleaved coding and sinusoid synthesis.

2) Sinusoid synthesis overrides frequency-interleaved coding; however, the sinusoid starts after a transient if a transient is present in the A-SPX interval. In this case there may be waveform-coding followed by a synthetic sinusoid in the same band.

3) Frequency-interleaved coding.

It should be noted that frequency-interleaved WCC are not required to cover the entire QMF subband group in which they are present.

5.7.6.5.3 Interleaving WCC and SEC

The WCC components are contained in the matrix $\mathbf{Qin}_{\mathrm{ASPX}}$. The SEC are contained in the matrix $\mathbf{Y}$.

For frequency-interleaved waveform coding, the WCC are added to the SEC as follows:

$$\begin{cases} Re\{\mathbf{Qout_{ASPX}}(m,i)\} = Re\{\mathbf{Qin_{ASPX}}(m,i-\delta_{ASPX})\} + Re\{\mathbf{Y}(m,i)\} \\ Im\{\mathbf{Qout_{ASPX}}(m,i)\} = Im\{\mathbf{Qin_{ASPX}}(m,i-\delta_{ASPX})\} + Im\{\mathbf{Y}(m,i)\} \end{cases}$$

For time-interleaved waveform coding, the WCC replaces the SEC as follows:

$$\begin{cases} Re\{\mathbf{Qout_{ASPX}}(m,i)\} = Re\{\mathbf{Qin_{ASPX}}(m,i-\delta_{ASPX})\} \\ Im\{\mathbf{Qout_{ASPX}}(m,i)\} = Im\{\mathbf{Qin_{ASPX}}(m,i-\delta_{ASPX})\} \end{cases}$$

where δ_{ASPX}=*ts_offset_hfgen* is the overall delay introduced by A-SPX processing.

The output matrix, $\mathbf{Qout_{ASPX}}$, is input to the downstream QMF domain processing tool, as depicted in figure 6.

5.7.7 Advanced Coupling tool - A-CPL

5.7.7.1 Introduction

The Advanced Coupling tool is a coding tool for improved coding of signals with more than one channel.

The tool operates in one of the following configurations:

- **Advanced Coupling in the channel pair element:** For improved stereo coding. This configuration is described in clause 5.7.7.5.
- **Advanced Coupling in the multichannel element:** For improved multichannel coding. This configuration is described in clause 5.7.7.6.

Data and Control Interfaces

Input

$\mathbf{Qin}_{\mathrm{ACPL},[a|b|\ldots]}$ M_{ACPL} complex QMF matrices of M_{ACPL} channels to be processed by the A-CPL tool

Output

$\mathbf{Qout}_{\mathrm{ACPL},[a|b|\ldots]}$ M_{ACPL} complex spectrally decoupled QMF matrices corresponding to the same channels

The $\mathbf{Qin}_{\mathrm{ACPL}}$ and $\mathbf{Qout}_{\mathrm{ACPL}}$ matrices each consist of *num_qmf_timeslots* columns, and *num_qmf_subband* rows. M_{ACPL} is the number of channels contained in the channel element with the exception of the LFE.

Control

- Decoded and dequantized advanced coupling parameters.

Figure 7 illustrates the interconnection of the Advanced Coupling tool with the other AC-4 coding tools.

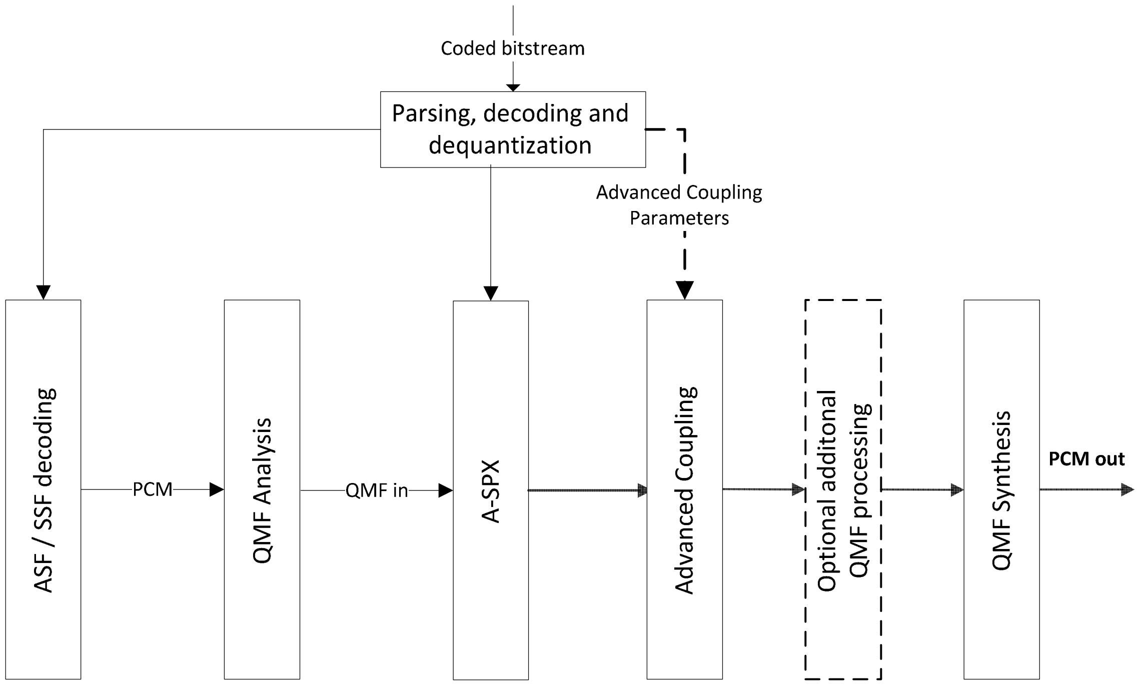
Coded bitstream
Parsing, decoding and dequantization
Advanced Coupling Parameters
ASF / SSF decoding
PCM
QMF Analysis
QMF in
A-SPX
Advanced Coupling
Optional additonal QMF processing
QMF Synthesis
PCM out

Figure 7: Advanced Coupling block diagram

The input to the Advanced Coupling tool corresponds to the output of the A-SPX tool. The Advanced coupling tool consists of one or more parallel modules that generically take two channels as input and provide one or two channels out. The two channel input to the Advanced Coupling tool is subjected to a two channel input conversion, that provides a single modified signal by means of a non-linear decorrelation operation. The two channel output from the Advanced Coupling tool module is formed from the two channel input, the modified input signal, and the corresponding bitstream elements.

5.7.7.2 Parameter band to QMF subband mapping

Advanced Coupling parameters are transmitted per parameter band. Parameter bands are groupings of QMF subbands and have lower frequency resolution than the QMF subbands. The mapping of parameter bands to QMF subbands is shown below in table 191. The number of parameter bands - 7, 9, 12, or 15 - is given by the variable *acpl_num_param_bands*.

Table 191: Mapping of parameter bands to QMF subbands

	acpl_num_param_bands			
QMF band groups	**15**	**12**	**9**	**7**
0	0	0	0	0
1	1	1	1	1
2	2	2	2	2
3	3	3	3	2
4	4	4	3	3
5	5	4	4	3
6	6	5	4	3
7	7	5	5	3
8	8	6	5	4
9 - 10	9	6	6	4
11 - 13	10	7	6	4
14 - 17	11	8	7	5
18 - 22	12	9	7	5
23 - 34	13	10	8	6
35 - 63	14	11	8	6

5.7.7.3 Interpolation

Decoded and dequantized advanced coupling parameters carried in the bitstream are time-interpolated to calculate values that are applied to the input of the decorrelator and to the ducked output of the decorrelator.

Parameter sets are transmitted either once or twice per frame, which is determined by the variable *acpl_num_param_sets*.

Two forms of interpolation - smooth and steep - are utilized to interpolate values for each QMF subsample.

When smooth interpolation is used, the values for each QMF subsample between consecutive parameter sets are linearly interpolated.

When steep interpolation is used, the QMF subsamples at which parameter set values change are indicated by the bitstream element `acpl_param_subsample`.

5.7.7.4 Decorrelator and transient ducker

5.7.7.4.1 Introduction

The decorrelation filters consist of a number of all-pass infinite impulse response (IIR) filter sections preceded by a constant frequency-dependent delay. The frequency axis of the QMF time/frequency matrix is divided into three different regions as shown in table 192, which correspond to the QMF split frequencies given in clause 5.7.7.2. The length of the delay and the length of the filter coefficients vectors are identical within each region.

Table 192: Division of QMF subbands

Subband region	Subbands	Delay	Filter length
k_0	0-6	7	7
k_1	7-22	10	4
k_2	23-63	12	2

The filtered signal is then ducked to lower the level of the reverberation tails.

5.7.7.4.2 Decorrelator IIR filtering

The decorrelator filters for the different frequency regions given in table 192 are implemented by a delay and a subsequent IIR filter.

Pseudocode

```
inputSignalModification(x)
{
    for (sb = 0; sb < num_qmf_subbands; sb++) {
        for (ts = 0; ts < num_qmf_timeslots; ts++) {
            b[0] = a[filterLength(sb)];
            y[ts][sb] = b[0]*x[n-delay(sb)][sb]/a[0];
            for (i = 1; i <= filterLength(sb); i++) {
                b[i] = a[filterLength(sb)-i];
                y[ts][sb] += (b[i]*x[ts-i-delay(sb)][sb] - a[i]*y[ts-i][sb])/a[0];
            }
        }
    }
    return y;
}
```

Three decorrelators, D_0, D_1 and D_2 are available. The coefficients *a[i]* for the different regions are given in tables 193, 194 and 195, and the coefficients *b[i]* can be derived from those using the pseudocode above.

Table 193: Coefficients a[i] for region k_0

i	D_0	D_1	D_2
0	1,0000	1,0000	1,0000
1	0,5306	-0,4178	0,4007
2	-0,4533	0,1082	0,4747
3	-0,6248	-0,2368	0,2611
4	0,0424	-0,1014	-0,1211
5	0,4237	-0,1052	-0,4248
6	0,4311	-0,3528	-0,2989
7	0,1688	0,4665	-0,1932

Table 194: Coefficients a[i] for region k_1

i	D_0	D_1	D_2
0	1,0000	1,0000	1,0000
1	0,5561	0,0425	-0,4361
2	-0,3039	0,3235	0,0345
3	-0,5024	-0,1556	0,5215
4	-0,1850	0,4958	-0,4178

Table 195: Coefficients a[i] for region k_2

i	D_0	D_1	D_2
0	1,0000	1,0000	1,0000
1	0,5773	0,2327	-0,6057
2	0,3321	-0,3901	0,3804

5.7.7.4.3 Transient ducker

To be able to handle transients and other fast time-envelopes, the output of the decorrelator all-pass filter has to be attenuated and this is performed according to the pseudocode below:

Pseudocode

```
alpha        = 0.76592833836465;
alpha_smooth = 0.25;
gamma        = 1.5;
epsilon      = 1.0e-9;

acpl_max_num_param_bands = 15;
for (pb = 0; pb < acpl_max_num_param_bands; pb++) {
    if (alpha * p_peak_decay_prev[pb] < p_energy[pb]) {
        p_peak_decay[pb] = p_energy[pb];
    }
    else {
        p_peak_decay[pb] = alpha * p_peak_decay_prev[pb];
    }

    smooth[pb]  = (1.0f - alpha_smooth) * p_smooth_prev[pb];
    smooth[pb] += alpha_smooth * p_energy[pb];

    smooth_peak_diff[pb]  = (1.0f - alpha_smooth) * smooth_peak_diff_prev[pb];
    smooth_peak_diff[pb] += alpha_smooth * (p_peak_decay - p_energy[pb]);

    if (gamma * smooth_peak_diff[pb] > smooth) {
        duck_gain[pb] = smooth / (gamma * (smooth_peak_diff[pb] + epsilon));
    }
    else {
        duck_gain[pb] = 1.0f;
    }
}
```

Here, the arrays *p_smooth_prev*, *p_peak_decay_prev* and *p_smooth_peak_diff_prev* are the *p_smooth*, *p_peak_decay* and *p_smooth_peak_diff* arrays of the previous frame, respectively, and the array *p_energy* contains the energy per parameter band of the input channel.

The transient attenuation value, duck_gain*[pb]*, is applied to the output of the decorrelator according to:

Pseudocode

```
applyTransientDucker(x) {
    for (sb = 0; sb < num_qmf_subbands; sb++)
    {
        pb = sb_to_ipar(sb);
        y[sb] = x[sb] * duck_gain[pb];
    }
    return y;
}
```

where the function *sb_to_ipar()* maps from QMF subbands to parameter bands according to table 191.

5.7.7.5 Advanced coupling in the channel pair element

For Advanced Coupling in the channel pair element, two input channels are present, left (L) and right (R). Their elements are addressed as follows:

- Left input channel $x_0(ts,sb) \in \mathbf{Qin}_{\mathrm{ACPL,L}}$
- Right input channel $x_1(ts,sb) \in \mathbf{Qin}_{\mathrm{ACPL,R}}$

The output channels are addressed as

- Left input channel $z_0(ts,sb) \in \mathbf{Qout}_{\mathrm{ACPL,L}}$
- Right input channel $z_1(ts,sb) \in \mathbf{Qout}_{\mathrm{ACPL,R}}$

They are derived given according to the pseudo-code:

Pseudocode

```
x0in = 2*x0;

u0 = inputSignalModification(x0in);
y0 = applyTransientDucker(u0);

num_pset = acpl_num_param_sets;

if (stereo_codec_mode == ASPX_ACPL_1) {
    x1in = 2*x1;

    (z0, z1) = ACplModule(acpl_alpha_1_dq, acpl_beta_1_dq, num_pset, x0in, x1in, y0);
}
else if (stereo_codec_mode == ASPX_ACPL_2) {
    acpl_qmf_band = 0;

    (z0, z1) = ACplModule(acpl_alpha_1_dq, acpl_beta_1_dq, num_pset, x0in, 0, y0);
}
```

where *inputSignalModification()* and *applyTransientDucker()* are outlined in clauses 5.7.7.4.2 and 5.7.7.4.3 respectively, and *ACplModule()* is defined as:

Pseudocode

```
ACplModule(acpl_alpha, acpl_beta, num_pset, x0, x1, y) {
    for (sb = 0; sb < num_qmf_subbands; sb++) {
        for (pset = 0; pset < num_pset; pset++) {
            start_pos = pset * num_qmf_timeslots / num_pset;
            stop_pos  = (pset + 1) * num_qmf_timeslots / num_pset - 1;
            for (ts = start_pos; ts < stop_pos; ts++) {
                interp_a = interpolate(acpl_alpha[pset][sb_to_ipar(sb)]);
                interp_b = interpolate(acpl_beta[pset][sb_to_ipar(sb)]);
                if (sb < acpl_qmf_band) {
                    z1[ts][sb]= 0.5*(x1[ts][sb] + x2[ts][sb]);
                    z2[ts][sb]= 0.5*(x1[ts][sb] - x2[ts][sb]);
                }
                else {
                    z1[ts][sb]= 0.5*(x1[ts][sb]*(1+interp_a) + y[ts][sb]*interp_b);
                    z2[ts][sb]= 0.5*(x1[ts][sb]*(1-interp_a) - y[ts][sb]*interp_b);
                }
            }
        }
    }
}
```

Here, *interpolate()* implements the interpolation outlined in clause 5.7.7.3, and *sb_to_ipar()* maps from QMF subbands to parameter bands according to table 191. The variables *acpl_alpha_1_dq* and *acpl_beta_1_dq* are the dequantized versions of the bitstream elements `acpl_alpha1` and `acpl_beta1`.

The dequantization is performed as described in clause 5.7.7.7.

5.7.7.6 Advanced coupling in the multichannel element

5.7.7.6.1 5_X_codec_mode ∈ {ASPX_ACPL_1, ASPX_ACPL_2}

For the configuration used when `5_X_codec_mode` = ASPX_ACPL_1 or ASPX_ACPL_2, two parallel Advanced Coupling modules are used.

There five input channels present. Their elements are addressed as:

- Left input channel (L) $x_0(ts,sb) \in \mathbf{Qin}_{\text{ACPL,L}}$
- Right input channel (R) $x_1(ts,sb) \in \mathbf{Qin}_{\text{ACPL,R}}$
- Center channel (C) $x_2(ts,sb) \in \mathbf{Qin}_{\text{ACPL,C}}$
- Left surround channel (Ls) $x_3(ts,sb) \in \mathbf{Qin}_{\text{ACPL,Ls}}$
- Right surround channel (Rs) $x_4(ts,sb) \in \mathbf{Qin}_{\text{ACPL,Rs}}$

The output channels are addressed as:

- Left input channel (L) $z_0(ts,sb) \in \mathbf{Qout}_{\text{ACPL,L}}$
- Right input channel (R) $z_2(ts,sb) \in \mathbf{Qout}_{\text{ACPL,R}}$
- Center channel (C) $z_4(ts,sb) \in \mathbf{Qout}_{\text{ACPL,C}}$
- Left surround channel (Ls) $z_1(ts,sb) \in \mathbf{Qout}_{\text{ACPL,Ls}}$
- Right surround channel (Rs) $z_3(ts,sb) \in \mathbf{Qout}_{\text{ACPL,Rs}}$

They are given by the pseudocode:

Pseudocode

```
x0in = 2*x0;
x1in = 2*x1;

u0 = inputSignalModification(x0in);
u1 = inputSignalModification(x1in);
y0 = applyTransientDucker(u0);
y1 = applyTransientDucker(u1);

if (5_X_codec_mode == ASPX_ACPL_1) {
    x3in = 2*x3;
    x4in = 2*x4;

    (z0, z1) = ACplModule(acpl_alpha_1_dq, acpl_beta_1_dq, num_pset_1, x0in, x3in, y0);
    (z2, z3) = ACplModule(acpl_alpha_2_dq, acpl_beta_2_dq, num_pset_2, x1in, x4in, y1);
}
else if (5_X_codec_mode == ASPX_ACPL_2) {
    acpl_qmf_band = 0;

    (z0, z1) = ACplModule(acpl_alpha_1_dq, acpl_beta_1_dq, num_pset_1, x0in, 0, y0);
    (z2, z3) = ACplModule(acpl_alpha_2_dq, acpl_beta_2_dq, num_pset_2, x1in, 0, y1);
}
z1 *= sqrt(2);
z4  = x2;
```

where *inputSignalModification()* and *applyTransientDucker()* are outlined in clauses 5.7.7.4.2 and 5.7.7.4.3 respectively, and *ACplModule()* is defined in clause 5.7.7.5.

The variables *acpl_alpha_1_dq*, *acpl_alpha_2_dq*, *acpl_beta_1_dq*, and *acpl_beta_2_dq* are the de-quantized values of `acpl_alpha1`, `acpl_alpha2`, `acpl_beta1`, and `acpl_beta2` for the two sets of the parameters in the bitstream respectively.

The variables *num_pset_1* and *num_pset_2* indicate the value *acpl_num_param_sets* of the first and the second `acpl_data_1ch()` element, respectively.

The dequantization is performed as described in clause 5.7.7.7.

5.7.7.6.2 5_X_codec_mode =ASPX_ACPL_3

For the configuration used when `5_X_codec_mode` = ASPX_ACPL_3 three parallel Advanced Coupling modules are used.

There five input channels present: Their elements are addressed as:

- Left input channel (L) $x_0(ts,sb) \in \mathbf{Qin}_{\mathrm{ACPL,L}}$
- Right input channel (R) $x_1(ts,sb) \in \mathbf{Qin}_{\mathrm{ACPL,R}}$
- Center channel (C) $x_2(ts,sb) \in \mathbf{Qin}_{\mathrm{ACPL,C}}$
- Left surround channel (Ls) $x_3(ts,sb) \in \mathbf{Qin}_{\mathrm{ACPL,Ls}}$
- Right surround channel (Rs) $x_4(ts,sb) \in \mathbf{Qin}_{\mathrm{ACPL,Rs}}$

The output channels are addressed as:

- Left input channel (L) $z_0(ts,sb) \in \mathbf{Qout}_{\mathrm{ACPL,L}}$
- Right input channel (R) $z_2(ts,sb) \in \mathbf{Qout}_{\mathrm{ACPL,R}}$
- Center channel (C) $z_4(ts,sb) \in \mathbf{Qout}_{\mathrm{ACPL,C}}$
- Left surround channel (Ls) $z_1(ts,sb) \in \mathbf{Qout}_{\mathrm{ACPL,Ls}}$

- Right surround channel (Rs) $z_3(ts,sb) \in \mathbf{Qout}_{\mathrm{ACPL,Rs}}$

They are given by the pseudo-code:

Pseudocode

```
x0in = x0*(1+2*sqrt(0.5));
x1in = x1*(1+2*sqrt(0.5));

v1 = Transform(g1_dq, g2_dq, num_pset, x0in, x1in);
v2 = Transform(g3_dq, g4_dq, num_pset, x0in, x1in);
v3 = Transform(g1_dq + g3_dq + g5_dq, g2_dq + g4_dq + g6_dq, num_pset, x0in, x1in);

u0 = inputSignalModification(v1);
u1 = inputSignalModification(v2);
u2 = inputSignalModification(v3);
y0 = applyTransientDucker(u0);
y1 = applyTransientDucker(u1);
y2 = applyTransientDucker(u2);

(z0, z1) = ACplModule2(g1_dq, g2_dq, acpl_alpha_1_dq, acpl_beta_1_dq, num_pset,
                       x0in, x1in, y0);
(z2, z3) = ACplModule2(g3_dq, g4_dq, acpl_alpha_2_dq, acpl_beta_2_dq, num_pset,
                       x0in, x1in, y1);
(z4, z5) = ACplModule2(g5_dq, g6_dq, 1, 0, num_pset, x0in, x1in, 0);

(z0, z1) = ACplModule3(acpl_beta_3_dq, acpl_alpha_1_dq, num_pset, z0, z1, y2);
(z2, z3) = ACplModule3(acpl_beta_3_dq, acpl_alpha_2_dq, num_pset, z2, z3, y2);
(z4, z5) = ACplModule3(-acpl_beta_3_dq, 1, num_pset, z4, z5, y2);

z1 *= sqrt(2);
z3 *= sqrt(2);
z4 *= sqrt(2);
```

where *inputSignalModification()* and *applyTransientDucker()* are outlined in clauses 5.7.7.4.2 and 5.7.7.4.3 respectively. The variables *g1_dq* to *g6_dq* are the dequantized versions of *acpl_gamma1* to *acpl_gamma6*, and *acpl_alpha_1_dq*, *acpl_beta_1_dq*, *acpl_alpha_2_dq*, *acpl_beta_2_dq*, *acpl_beta_3_dq* are the dequantized versions of *acpl_alpha1*, *acpl_beta1*, *acpl_alpha2*, *acpl_beta2*, and *acpl_beta3* respectively.

The dequantization is performed as described in clause 5.7.7.7.

The functions *ACplModule2()*, *ACplModule3()* and *Transform()* are given as pseudocode below.

Pseudocode

```
Transform(g1, g2, num_pset, x0, x1)
{
    for (sb = 0; sb < num_qmf_subbands; sb++) {
        ipar = sb_to_ipar(sb);
        for (pset = 0; pset < num_pset; pset++) {

            start_pos = pset * num_qmf_timeslots / num_pset;
            stop_pos  = (pset + 1) * num_qmf_timeslots / num_pset - 1;

            for (ts = start_pos; ts < stop_pos; ts++) {
                interp_g1 = interpolate(g1[pset][ipar]);
                interp_g2 = interpolate(g2[pset]]ipar]);

                v[ts][sb]  = x0[ts][sb]*interp_g1 + x1[ts][sb]*interp_g2;
            }
        }
    }
}

ACplModule2(g1, g2, a, b, num_pset, x0, x1, y)
{
    for (sb = 0; sb < num_qmf_subbands; sb++) {
        ipar = sb_to_ipar(sb);
        for (pset = 0; pset < num_pset; pset++) {

            start_pos = pset * num_qmf_timeslots / num_pset;
            stop_pos  = (pset + 1) * num_qmf_timeslots / num_pset - 1;

            for (ts = start_pos; ts < stop_pos; ts++) {
                interp_g1   = interpolate(g1[pset][ipar]);
                interp_g2   = interpolate(g2[pset]]ipar]);
                interp_g1_a = interpolate(g1[pset][ipar]*a[pset][ipar]);
                interp_g2_a = interpolate(g2[pset][ipar]*a[pset][ipar]);
                interp_b    = interpolate( b[pset][ipar]);

                z0[ts][sb]  = 0.5*(x0[ts][sb]*(interp_g1 + interp_g1_a) +
                           x1[ts][sb]*(interp_g2 + interp_g2_a) +
                           y[ts][sb]* interp_b);

                z1[ts][sb]  = 0.5*(x0[ts][sb]*(interp_g1 - interp_g1_a) +
                           x1[ts][sb]*(interp_g2 - interp_g2_a) -
                           y[ts][sb]* interp_b);
            }
        }
    }
}

ACplModule3(b3, a, num_pset, z0, z1, y2)
{
    for (sb = 0; sb < num_qmf_subbands; sb++) {
        ipar = sb_to_ipar(sb);
        for (pset = 0; pset < num_pset; pset++) {

            start_pos = pset * num_qmf_timeslots / num_pset;
            stop_pos  = (pset + 1) * num_qmf_timeslots / num_pset - 1;

            for (ts = start_pos; ts < stop_pos; ts++) {
                interp_b3   = interpolate(b3[pset][ipar]);
                interp_b3_a = interpolate(b3[pset][ipar]*a[pset][ipar]);

                z0[ts][sb]  += 0.5*y2[ts][sb]*(interp_b3 + interp_b3_a);
                z1[ts][sb]  += 0.5*y2[ts][sb]*(interp_b3 - interp_b3_a);
            }
        }
    }
}
```

Here, *interpolate()* implements the interpolation outlined in clause 5.7.7.3, and *sb_to_ipar()* maps from QMF subbands to parameter bands according to table 191. The variables *acpl_alpha_1_dq* and *acpl_beta_1_dq* are the dequantized versions of the bitstream elements `acpl_alpha1` and `acpl_beta1`.

The dequantization is performed as described in clause 5.7.7.7.

5.7.7.6.3 7_X_codec_mode ∈ {ASPX_ACPL_1, ASPX_ACPL_2}

When decoding a `7_X_channel_element`, the only codec modes utilizing the A-CPL tool are `7_X_codec_mode` = ASPX_ACPL_1 and ASPX_ACPL_2. Here, two parallel Advanced Coupling modules are used, analog to clause 5.7.7.6.1 for the `5_X_channel_element`.

There are up to eight input channels present. Depending on the channel mode and the bitstream element `add_ch_base`, defined in clause 4.3.3.7.6, the mapping of the six additional channels to ACPL input and output variables varies, as described in table 196.

Table 196: Input/Output channel mapping for 7_X_channel_element

channel_mode	**3/4/0.x**	**5/2/0.x**		**3/2/2.x**	
add_ch_base	n/a	0	1	0	1
x_0/z_0	L	L	Ls	L	Ls
x_1/z_2	R	R	Rs	R	Rs
x_2/z_4	C	C	C	C	C
x_3/z_1	Lrs	Lw	Lw	Vhl	Vhl
x_4/z_3	Rrs	Rw	Rw	Vhr	Vhr
x_6/z_6	Ls	Ls	L	Ls	L
x_7/z_7	Rs	Rs	R	Rs	R

The pseudocode for deriving the output variables is given below.

Pseudocode

```
x0in = 2*x0;
x1in = 2*x1;

u0 = inputSignalModification(x0in);
u1 = inputSignalModification(x1in);
y0 = applyTransientDucker(u0);
y1 = applyTransientDucker(u1);

if (7_X_codec_mode == ASPX_ACPL_1) {
    x3in = 2*x3;
    x4in = 2*x4;

    (z0, z1) = ACplModule(acpl_alpha_1_dq, acpl_beta_1_dq, num_pset_1, x0in, x3in, y0);
    (z2, z3) = ACplModule(acpl_alpha_2_dq, acpl_beta_2_dq, num_pset_2, x1in, x4in, y1);
}
else if (7_X_codec_mode == ASPX_ACPL_2) {
    acpl_qmf_band = 0;

    (z0, z1) = ACplModule(acpl_alpha_1_dq, acpl_beta_1_dq, num_pset_1, x0in, 0, y0);
    (z2, z3) = ACplModule(acpl_alpha_2_dq, acpl_beta_2_dq, num_pset_2, x1in, 0, y1);
}

if (add_ch_base == 1) {
  z0 *= sqrt(2);
  z2 *= sqrt(2);
}

z4 = x2;
z6 = x6;
z7 = x7;
```

where *inputSignalModification()* and *applyTransientDucker()* are outlined in clauses 5.7.7.4.2 and 5.7.7.4.3 respectively, and *ACplModule()* is defined in clause 5.7.7.5.

The variables *num_pset_1* and *num_pset_2* indicate the value *acpl_num_param_sets* of the first and the second `acpl_data_1ch()` element, respectively.

5.7.7.7 Dequantization

The dequantized values *acpl_alpha_1_dq*, *acpl_alpha_2_dq*, *acpl_beta1_dq*, and *acpl_beta2_dq* are obtained using tables 197 and 198 if the quantization mode is set to fine, and using tables 199 and 200 if the quantization mode is set to coarse.

Tables 197 and 199 are used to calculate dequantized alpha values, for fine and coarse quantization modes respectively. The value of the index i into alpha[i] is obtained as described by the equations for acpl_alpha1 in tables 61 and 62.

The index ibeta obtained from table 197 or 199 is used in table 198 or 200, for fine and coarse quantization modes respectively, to calculate the dequantized beta value.

Table 197: Alpha values - fine quantization

i	ibeta	alpha[i]
0	0	-2,000000
1	1	-1,809375
2	2	-1,637500
3	3	-1,484375
4	4	-1,350000
5	5	-1,234375
6	6	-1,137500
7	7	-1,059375
8	8	-1,000000
9	7	-0,940625
10	6	-0,862500
11	5	-0,765625
12	4	-0,650000
13	3	-0,515625
14	2	-0,362500
15	1	-0,190625
16	0	0,000000
17	1	0,190625
18	2	0,362500
19	3	0,515625
20	4	0,650000
21	5	0,765625
22	6	0,862500
23	7	0,940625
24	8	1,000000
25	7	1,059375
26	6	1,137500
27	5	1,234375
28	4	1,350000
29	3	1,484375
30	2	1,637500
31	1	1,809375
32	0	2,000000

Table 198: Beta values - fine quantization

i	beta[0][i]	beta[1][i]	beta[2][i]	beta[3][i]	beta[4][i]	beta[5][i]	beta[6][i]	beta[7][i]	beta[8][i]
0	0,0000000	0,0000000	0,0000000	0,0000000	0,0000000	0,0000000	0,0000000	0,0000000	0,0000000
1	0,2375000	0,2035449	0,1729297	0,1456543	0,1217188	0,1011230	0,0838672	0,0699512	0,0593750
2	0,5500000	0,4713672	0,4004688	0,3373047	0,2818750	0,2341797	0,1942188	0,1619922	0,1375000
3	0,9375000	0,8034668	0,6826172	0,5749512	0,4804688	0,3991699	0,3310547	0,2761230	0,2343750
4	1,4000000	1,1998440	1,0193750	0,8585938	0,7175000	0,5960938	0,4943750	0,4123438	0,3500000
5	1,9375000	1,6604980	1,4107420	1,1882319	0,9929688	0,8249512	0,6841797	0,5706543	0,4843750
6	2,5500000	2,1854300	1,8567190	1,5638670	1,3068750	1,0857420	0,9004688	0,7510547	0,6375000
7	3,2375000	2,7746389	2,3573050	1,9854980	1,6592190	1,3784670	1,1432420	0,9535449	0,8093750
8	4,0000000	3,4281249	2,9124999	2,4531250	2,0500000	1,7031250	1,4125000	1,1781250	1,0000000

Table 199: Alpha values - coarse quantization

i	ibeta	alpha[i]
0	0	-2,000000
1	1	-1,637500
2	2	-1,350000
3	3	-1,137500
4	4	-1,000000
5	3	-0,862500
6	2	-0,650000
7	1	-0,362500
8	0	0,000000
9	1	0,362500
10	2	0,650000
11	3	0,862500
12	4	1,000000
13	3	1,137500
14	2	1,350000
15	1	1,637500
16	0	2,000000

Table 200: Beta values - coarse quantization

i	beta[0][i]	beta[1][i]	beta[2][i]	beta[3][i]	beta[4][i]
0	0,0000000	0,0000000	0,0000000	0,0000000	0,0000000
1	0,5500000	0,4004688	0,2818750	0,1942188	0,1375000
2	1,4000000	1,0193750	0,7175000	0,4943750	0,3500000
3	2,5500000	1,8567190	1,3068750	0,9004688	0,6375000
4	4,0000000	2,9124999	2,0500000	1,4125000	1,0000000

Table 201: Delta values for dequantizing beta3 values

Quantization mode	delta
fine	0,125
coarse	0,25

Dequantized *acpl_beta_3_dq* values are obtained by multiplying the beta3 value by the delta factor corresponding to the quantization mode.

Table 202: Delta values for dequantizing gamma values

Quantization mode	delta
fine	1638/16384
coarse	3276/16384

Dequantized values *g1_dq* to *g6_dq* are obtained by multiplying the gamma value by the delta factor corresponding to the quantization mode.

5.7.8 Dialog Enhancement

5.7.8.1 Introduction

The Dialog Enhancement tool is a tool to increase intelligibility of the dialog in an audio scene encoded in AC-4. The underlying algorithm uses metadata encoded in the bitstream to boost the dialog in the scene. This technology is especially beneficial for the hearing impaired.

Dialog Enhancement supports enhancement of the dialog with a user-defined gain. It operates in the QMF domain and is executed before the DRC processing.

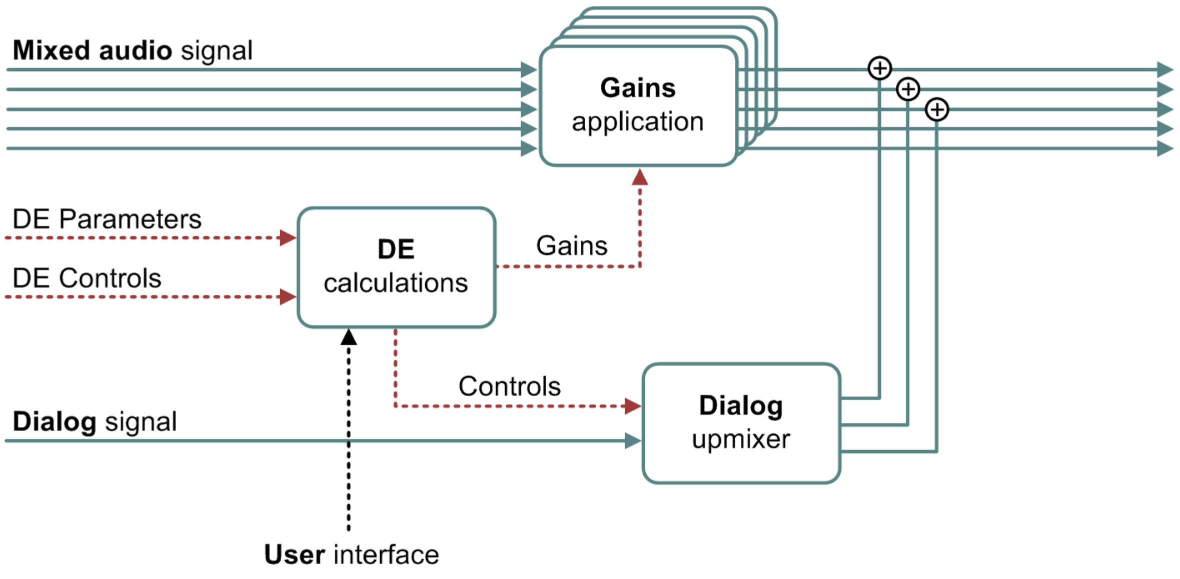

Figure 8: Dialog Enhancement in AC-4 Decoder

There are three different dialog enhancement modes available.

Parametric channel independent enhancement: In this mode a parameter subset is transmitted for each channel that contains dialog. Using these parameters any dialog contribution to the respective channels is boosted.

Parametric cross-channel enhancement: In this mode up to three channels of the content can be used to boost the dialog in one or more of these channels.

Waveform-parametric hybrid: This mode combines a waveform coded dialog signal with either parametric enhancement. Both the isolated dialog signal and the parameter set are transmitted in the bitstream. An additional parameter in the bitstream will indicate the balance between waveform and parametric enhancement. Low complexity decoders may use only the parametric data to perform dialog enhancement.
In the combination with channel independent enhancement, the dialog signal contains a channel for every transmitted parameter subset, whereas with cross-channel enhancement a single dialog signal is transmitted irrespective of the number of transmitted parameter subsets.

Data and Control Interfaces

Input

$\mathbf{Qin}_{\mathrm{DE},[a\|b\|\ldots]}$	M_{DE} complex QMF matrices of channels containing the regular audio scene
$\mathbf{Q}_{\mathrm{d}}$	a complex QMF matrix with an isolated dialog (waveform-parametric hybrid mode only)

Output

$\mathbf{Qout}_{\mathrm{DE},[a\|b\|\ldots]}$	M_{DE} complex QMF matrices containing the dialog enhanced audio scene

The $\mathbf{Qin}_{\mathrm{DE}}$, $\mathbf{Q}_{\mathrm{d}}$ and $\mathbf{Qout}_{\mathrm{DE}}$ matrices each consist of *num_qmf_timeslots* columns, and *num_qmf_subband* rows.

Control

G_{DE}	a dialog enhancement gain in dB
$\mathbf{p}$	a parameter vector, consisting of parameter subsets p_i corresponding to channel i
$\mathbf{r}$	rendering vector for spatial positioning of the dialog

5.7.8.2 Processed Channels

Each of the three decoding methods described below operates, by default, on the QMF domain time-frequency samples of a subset of the three front channels in one AC-4 frame:

$$m_{channel}(k,n) \in \mathbf{Qin}_{\mathrm{DE,channel}},\ \ 0 \leq channel \leq 3$$

As the dialog may be limited to either, all, or a combination of the three front channels, `de_channel_config` indicates the channels processed by the dialog enhancement algorithm while *de_nr_channels* indicates their absolute number. The two variables are described in clause 4.3.14.3.3.

Consequently, the input time-frequency samples to the dialog enhancement algorithm are noted in the following as:

$$m_i(k,n) \in \mathbf{Qin}_{\mathrm{DE,i}}\big|_{\mathrm{de\_channel\_config\{i\}=1}}$$

where `de_channel_config{x}` is the bit at position x in the binary representation of `de_channel_config`.

Likewise, output samples are denoted as:

$$y_i(k,n) \in \mathbf{Qout}_{\mathrm{DE,i}}\big|_{\mathrm{de\_channel\_config\{i\}=1}}$$

5.7.8.3 Dequantization

The parameter index data shall be dequantized using the quantization vectors in tables 203 and 204.

$$p'_i(b,n) = Q^{-1}\{de\_par[i][b]\},\ \ 0 < b < de\_nr\_bands$$

Table 203: Quantization vector for channel-independent enhancement mode

Parameter index	Parameter value
0	0
1	0,1
2	0,2
3	0,3
4	0,4
5	0,5
6	0,6
7	0,7
8	0,8
9	0,9
10	1,0
11	1,1
12	1,2
13	1,3
14	1,4
15	1,5
16	1,75
17	2,0
18	2,5
19	3,0
20	3,5
21	4,0
22	4,5
23	5,0
24	5,5
25	6,0
26	6,5
27	7,0
28	7,5
29	8,0
30	8,5
31	9,0

Table 204: Quantization vector for cross-channel enhancement mode

Parameter index	Parameter value
-30	-3,0
-29	-2,9
-28	-2,8
-27	-2,7
-26	-2,6
-25	-2,5
-24	-2,4
-23	-2,3
-22	-2,2
-21	-2,1
-20	-2,0
-19	-1,9
-18	-1,8
-17	-1,7
-16	-1,6
-15	-1,5
-14	-1,4
-13	-1,3
-12	-1,2
-11	-1,1
-10	-1,0
-9	-0,9
-8	-0,8
-7	-0,7
-6	-0,6
-5	-0,5
-4	-0,4
-3	-0,3
-2	-0,2
-1	-0,1
0	0
1	0,1
2	0,2
3	0,3
4	0,4
5	0,5
6	0,6
7	0,7
8	0,8
9	0,9
10	1,0
11	1,1
12	1,2
13	1,3
14	1,4
15	1,5
16	1,6
17	1,7
18	1,8
19	1,9
20	2,0
21	2,1
22	2,2
23	2,3
24	2,4
25	2,5
26	2,6
27	2,7
28	2,8
29	2,9
30	3,0

5.7.8.4 Parameter Bands

The parametric enhancement methods rely on segmenting the *num_qmf_subbands* QMF subbands into *de_nr_bands* = 8 dialog enhancement parameter bands. The mapping of QMF subbands to parameter bands is described in clause 4.3.14.5.1.

Hence, the entries of a parameter subset for channel *i:*

$$p_i'(b,n) = de\_par[i][b], \qquad 0 < b < de\_nr\_bands$$

can always be extended to a cover the full range of QMF bands by:

$$p_i(k,n) = p_i'(b,n), \qquad k \in b$$

To improve readability, the time-frequency dependency *(k,n)* is dropped in the following.

5.7.8.5 Rendering

In modes with cross-channel dialog enhancement the parametrically generated dialog enhancing signal shall be mixed into the audio scene according to the mixing parameters transmitted in the bitstream. When *de_nr_channels* indicates one involved channel, the dialog enhancement signal will be added to this channel only. That means:

$$\mathbf{r} = (1)$$

For two processed channels, the dequantized mixing coefficient determines the rendering:

$$\mathbf{r} = \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = \begin{pmatrix} \text{de\_mix\_coef1} \\ \sqrt{1 - (\text{de\_mix\_coef1})^2} \end{pmatrix}$$

For three channels:

$$\mathbf{r} = \begin{pmatrix} r_0 \\ r_1 \\ r_2 \end{pmatrix} = \begin{pmatrix} \text{de\_mix\_coef1} \\ \text{de\_mix\_coef2} \\ \sqrt{1 - (\text{de\_mix\_coef1})^2 - (\text{de\_mix\_coef2})^2} \end{pmatrix}$$

In the corresponding hybrid mode, the waveform coded dialog signal is rendered with the same rendering vector.

5.7.8.6 Interpolation

For a smooth transition between parameter sets, interpolation is applied between successive frames. Clause 5.7.8.7 through clause 5.7.8.9 describe the parameter-based processing for each frame f in terms of a matrix operation.

$$\mathbf{y} = \widehat{H}_{DE}(f) \cdot \begin{pmatrix} \mathbf{m} \\ \mathbf{d_c} \end{pmatrix}$$

The application of the dialog enhancement parameters shall facilitate changing dialog enhancement channel configurations between frames by always representing $\widehat{H}_{DE}$ as a 3×3 matrix for multichannel and a 2×2 matrix for stereo. Channels not involved in the enhancement shall have a 1 on the corresponding diagonal position and 0 in the remaining elements of the corresponding columns and rows. For example, a 2 channel enhancement on the left and right channel of a multichannel signal, shall map the dialog enhancement matrix:

$$\widehat{H}_{DE}' = \begin{pmatrix} h_{00} & h_{01} \\ h_{10} & h_{11} \end{pmatrix}$$

to its 3×3 representation:

$$\widehat{H}_{DE} = \begin{pmatrix} h_{00} & 0 & h_{01} \\ 0 & 1 & 0 \\ h_{10} & 0 & h_{11} \end{pmatrix}$$

In order to obtain the matrix $H_{DE}(k,n)$ for application in QMF time slot n within frame f, an interpolation between the enhancement matrix of the previous frame and the enhancement matrix of the current frame is performed according to:

$$H_{DE}(k,n) = \left(1 - \frac{n+0.5}{N}\right) \cdot \widehat{H}_{DE}(f-1) + \frac{n+0.5}{N} \cdot \widehat{H}_{DE}(f)$$

with N the frame length in QMF time slots, *num_qmf_timeslots*.

5.7.8.7 Parametric Channel Independent Enhancement

In this enhancement method, each channel *i* which has been indicated for processing is processed separately. A set of parameters p_i is transmitted for each. Additionally, a maximum overall gain G_{max}, defined in clause 4.3.14.3.2, limits the user-defined gain g to a range determined at encoding stage.

Each time-frequency sample m_i of each processed channel *i* is multiplied by the corresponding parameter p_i, such that the dialog-enhanced output signal for this channel can be derived as:

$$y_i = m_i + g \cdot p_i \cdot m_i$$

where:

$$g = \begin{cases} 10^{\frac{G_{DE}}{20}} - 1, & G_{DE} < G_{max} \\ 10^{\frac{G_{max,}}{20}} - 1, & otherwise \end{cases}$$

When two channels are processed in a mode with channel independent enhancement parameters ($de\_method \in \{0;\ 2\}$), an M/S flag (de_ms_proc_flag) is transmitted that indicates whether the parameters are intended for application on the Mid/Side representation of the signal. In that case only one parameter subset is transmitted for application to the Mid signal. The processing equation then becomes:

$$\begin{pmatrix} y_0 \\ y_1 \end{pmatrix} = 0.5 \cdot \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \cdot \begin{pmatrix} 1 + g \cdot p_0 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \cdot \begin{pmatrix} m_0 \\ m_1 \end{pmatrix}$$

5.7.8.8 Parametric Cross-channel Enhancement

This method focuses on enhancing the dialog using a combination of up to three input channels.

The formula for reconstructing the dialog is noted as follows:

- 1 processed channel

$$y = m + r \cdot g \cdot p \cdot m$$

- 2 processed channels

$$\begin{pmatrix} y_0 \\ y_1 \end{pmatrix} = \begin{pmatrix} m_0 \\ m_1 \end{pmatrix} + \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} \cdot g \cdot (p_0 \quad p_1) \cdot \begin{pmatrix} m_0 \\ m_1 \end{pmatrix}$$

- 3 processed channels

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} m_0 \\ m_1 \\ m_2 \end{pmatrix} + \begin{pmatrix} r_0 \\ r_1 \\ r_2 \end{pmatrix} \cdot g \cdot (p_0 \quad p_1 \quad p_2) \cdot \begin{pmatrix} m_0 \\ m_1 \\ m_2 \end{pmatrix}$$

with:

$$g = \begin{cases} 10^{\frac{G_{DE}}{20}} - 1, & G_{DE} < G_{max} \\ 10^{\frac{G_{max,}}{20}} - 1, & otherwise \end{cases}.$$

For faster processing the equation can be written into a single matrix multiplication. Hence, the above equation is represented by:

$$\mathbf{y} = (\mathbf{I} + g \cdot \mathbf{r} \cdot \mathbf{p}) \cdot \mathbf{m}$$

with **I** the identity matrix, **r** the rendering vector and **p** the dialog reconstruction vector, and **m** and **y** vectors representing the input and output time-frequency samples, respectively, one row per processed channel.

5.7.8.9 Waveform-Parametric Hybrid

The hybrid dialog enhancement modes complement the parametric enhancement parameters with transmission of up to three waveforms. The encoder also transmits a trade-off parameter, α_c, derived from `de_signal_contribution`, that indicates how the enhancement contributions should be divided over the two approaches.

The resulting reconstruction formula for channel independent enhancement can be noted as:

$$\mathbf{y} = \left(\mathbf{I} + g_{\mathrm{p}} \cdot diag(\mathbf{p})\right) \cdot \mathbf{m} + g_s \cdot \mathbf{d}_{\boldsymbol{c}}$$

or

$$\mathbf{y} = (\mathbf{I} + g_{\mathrm{p}} \cdot diag(\mathbf{p}) \quad g_s \cdot \mathbf{I}) \cdot \begin{pmatrix} \mathbf{m} \\ \mathbf{d}_{\boldsymbol{c}} \end{pmatrix}.$$

Similarly for cross-channel enhancement:

$$\mathbf{y} = (\mathbf{I} + g_{\mathrm{p}} \cdot \mathbf{r} \cdot \mathbf{p} \quad \mathbf{r} \cdot g_s) \cdot \begin{pmatrix} \mathbf{m} \\ \mathbf{d}_{c} \end{pmatrix}$$

with:

$$g_s = \begin{cases} \alpha_c \cdot \left(10^{\frac{G}{20}} - 1\right), & g < G_{max} \\ \alpha_c \cdot \left(10^{\frac{G_{max}}{20}} - 1\right), & otherwise \end{cases}$$

and

$$g_{\mathrm{p}} = \begin{cases} (1 - \alpha_c) \cdot \left(10^{\frac{G}{20}} - 1\right), & g < G_{max} \\ (1 - \alpha_c) \cdot \left(10^{\frac{G_{max}}{20}} - 1\right), & otherwise \end{cases}.$$

In the specific case when the M/S flag is set the equation is:

$$\mathbf{y} = \left(0.5 \cdot \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \cdot \begin{pmatrix} 1 + g_{\mathrm{p}} \cdot p_0 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \quad 0.5 \cdot g_{\mathrm{s}} \begin{pmatrix} 1 \\ 1 \end{pmatrix}\right) \cdot \begin{pmatrix} \mathbf{m} \\ \mathbf{d}_{\boldsymbol{c}} \end{pmatrix}$$

5.7.9 Dynamic range control (DRC) tool

5.7.9.1 Introduction

This clause describes a DRC (dynamic range control) processing unit that is configured by the AC-4 bitstream. For an explanation of how DRC can be used in the scope of an audio codec please refer to document [i.1], clause 6.7.1.1.

DRC is operated in the QMF domain and is executed before QMF synthesis.

Data and Control Interfaces

Input

$\mathbf{Qin1}_{\mathrm{DRC},[a\|b\|...]}$	***M*** complex QMF matrices for each of the channels derived from a channel element
$\mathbf{Qin2}_{\mathrm{DRC},[a\|b\|...]}$	***M*** complex QMF matrices of the same channel which have not been previously processed by the DE tool

Output

$\mathbf{Qout}_{\mathrm{DRC},[a\|b\|...]}$	*M* complex QMF matrices of the DRC processed channels

The matrices $\mathbf{Qin1}_{\mathrm{DRC},ch}$, $\mathbf{Qin2}_{\mathrm{DRC},ch}$ and $\mathbf{Qout}_{\mathrm{DRC},ch}$ for each channel *ch* consist of *num_qmf_timeslots* columns, and *num_qmf_subband* rows. Their elements are represented by $\mathrm{Qin1}_{ch,k,n}$, $\mathrm{Qin2}_{ch,k,n}$ and $\mathrm{Qout}_{ch,k,n}$, respectively.

Control

L_{in}	the *dialnorm* value
L_{out}	the reference output level for the DRC processor, supplied by the system
drc_dec_mode_config	up to eight DRC mode configurations, specifying DRC processor characteristics

5.7.9.2 DRC Profiles

Compressor operation is controlled by DRC modes. Modes are configured either through a compression curve or by direct gains to be applied, similar to [i.1], clause 6.7.1.

AC-4 supports four default modes:

- Home theatre mode
- Flat-panel mode
- Portable mode - Speakers
- Portable mode - Headphones

Additional modes may be defined and transmitted in the bitstream.

Each mode is applicable to a range $[L_{out,min}, L_{out,max}]$ of output levels. An implementation may provide listener control of choosing a specific DRC mode provided that $L_{out,min} < L_{out} < L_{out,max}$.

For the default modes, $L_{out,min}$ and $L_{out,max}$ are defined in clause 4.3.13.3.1.

For additional modes, $L_{out,min}$ and $L_{out,max}$ are transmitted in the bitstream as defined in clauses 4.3.13.3.2 and 4.3.13.3.3, respectively.

If no applicable mode configuration is available, an implementation may use the last applicable mode configuration transmitted instead.

By default, the decoder shall select the mode with the largest profile ID value for which $L_{out,min} < L_{out} < L_{out,max}$. In the case of the default portable modes, there is an additional distinction between playback over loudspeakers and playback over headphones.

5.7.9.3 Decoding process

5.7.9.3.1 Compression Curves

When the DRC profile specifies a compression curve (`drc_compression_curve_flag` == 1), the DRC processor, as depicted in figure 9, comprises the following functions:

- Level computation. This function calculates the current signal level.
- Gain computation. This function maps the current level into a gain, controlled by a DRC profile.
- Gain application. This function applies the gain to the QMF samples.

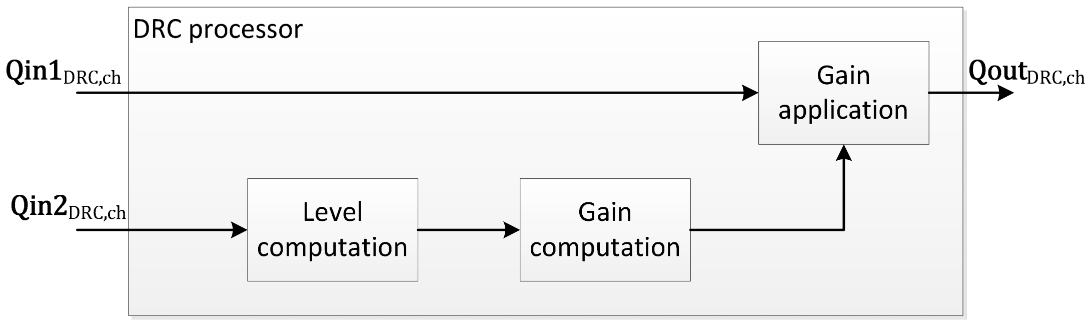

Figure 9: DRC processor

5.7.9.3.1.1 Level computation

There are several methods of level computation; an implementation may, for example, employ a level calculation as described in Recommendation ITU-R BS.1770 [i.3], or as a frequency weighted RMS computation. The level shall be expressed relative to the *dialnorm*. The result of the level calculation is an array $L_{k,n}$ of levels, one per QMF sample.

5.7.9.3.1.2 Gain computation

The gain computation stage maps the relative level $L_{k,n}$ into a gain $g_{k,n}$ through the compression curve.

The compression curve consists of seven sections.

The gain value $G_{k,n}$ should be derived from the level $L_{k,n}$ and the transmitted compression curve by a piecewise-linear interpolation:

$$G_{k,n} = \begin{cases} G_{maxBoost}, & L_{k,n} < L_{maxBoost} \\ G_1(L_{k,n}), & L_{maxBoost} < L_{k,n} \le L_{sectionboost} \\ G_2(L_{k,n}), & L_{sectionboost} < L_{k,n} \le L_{0,low} \\ 0, & L_{0,low} < L_{k,n} \le L_{0,high} \\ G_3(L_{k,n}), & L_{0,high} < L_{k,n} \le L_{sectioncut} \\ G_4(L_{k,n}), & L_{sectioncut} < L_{k,n} \le L_{maxCut} \\ G_{maxCut}, & L_{k,n} > L_{maxCut} \end{cases}$$

with:

$$G_1(\mathrm{x}) = G_{sectionboost} + (G_{maxBoost} - G_{sectionboost})\,\frac{L_{sectionboost} - \mathrm{x}}{L_{sectionboost} - L_{maxBoost}},$$

$$G_2(\mathrm{x}) = G_{sectionboost}\,\frac{L_{0,low} - \mathrm{x}}{L_{0,low} - L_{boost}},$$

$$G_3(\mathrm{x}) = G_{sectioncut}\,\frac{\mathrm{x} - L_{0,high}}{L_{sectioncut} - L_{0,high}},$$

$$G_4(\mathrm{x}) = G_{sectioncut} + (G_{maxCut} - G_{sectioncut})\,\frac{\mathrm{x} - L_{sectioncut}}{L_{maxCut} - L_{sectioncut}}$$

5.7.9.3.2 Directly transmitted DRC Gains

For profiles where DRC gains are transmitted (`drc_compression_curve_flag` == 0), there are four configurations supported for transmitting gain values in the AC-4 bitstream.

- Single wideband gain for all channels
- Wideband gain for each channel group
- Multiband for each channel group, 2 bands

- Multiband for each channel group, 4 bands

The channel-dependent gains correspond to channel groups. For a `5_X_channel_element`, for example, there are 3 channel groups *chg* ∈ [Lf, Rf, LFE], [Cf], [Ls, Rs]. The update rate of DRC gains depends on the frame length.

NOTE: The definition of the DRC gain bands, channel groups, and update rate is given in clauses 4.3.13.3.8, 4.3.13.7.1 and 4.3.13.7.2, respectively.

After differential decoding of the gains, resulting in drc_gain[chg][sf][band] per channel group, the gains for each channel *ch* are adjusted to reflect dB_2 values.

$$g_{k,n,ch} = \text{drc\_gain}[\text{chg}][\text{sf}][\text{band}]|_{k\in\text{band},n\in\text{sf},ch\in\text{chg}}$$

5.7.9.3.3 Application of gain values

As a last step, the DRC tool shall map the *dialnorm* level to the output level L_{out}, and apply the gains that have either been calculated per clause 5.7.9.3.1.2 or transmitted as per clause 5.7.9.3.2.

$$\text{Qout}_{ch,k,n} = 10^{\frac{L_{out}-L_{in}}{20}} \cdot g_{k,n,ch} \cdot \text{Qin1}_{ch,k,n}$$

5.7.9.4 Transcoding to a AC-3 or E-AC-3 format

When an AC-4 bitstream is transcoded into a format such as AC-3 or E-AC-3, no DRC processing should be applied. Instead, the encoder or transcoder should be configured with the compression curve indicated by the field `drc_eac3_transcode_curve` in table 154.

6 Decoding the AC-4 bitstream

6.1 Introduction

Clause 4 specifies the details of the AC-4 bitstream syntax and clause 5 specifies the algorithmic details of the tools used within AC-4. This clause describes the complete AC-4 decoding process and by this makes a connection between the bitstream elements and the AC-4 tools. Figure 10 shows a flow diagram of the AC-4 decoding process for one substream.

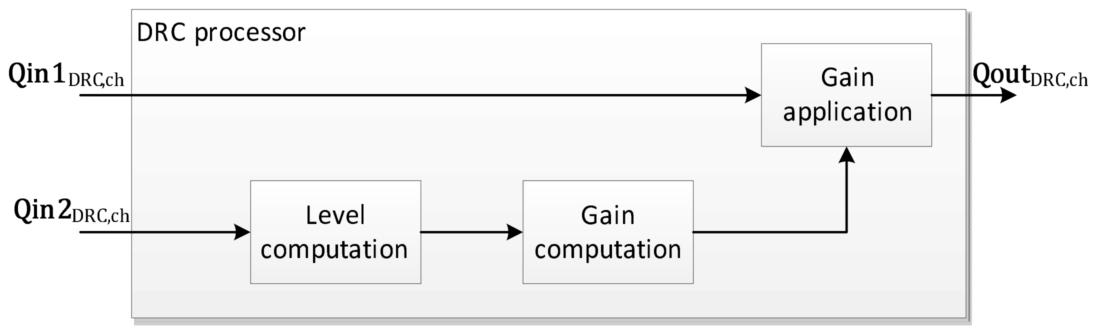

Figure 10: Flow diagram of the decoding process (one substream)

6.2 Decoding process

6.2.1 Input bitstream

The input bitstream will typically come from a transmission or storage system. To decode an AC-4 bitstream all `raw_ac4_frame` elements shall be extracted according to the used transport format and presented to the AC-4 decoder.

6.2.2 Structure of the bitstream

An AC-4 bitstream is roughly structured in several containers, namely:

- Raw AC-4 Frame
  The actual codec frame, consisting of a table of contents plus several byte aligned substreams.
- Table Of Contents (TOC)
  Signaling information on how to treat the content of the substream container of the AC-4 frame.
- Substream
  Encoded audio and metadata, accompanied by a metadata skip area and a coding paradigm.
- Presentation Information
  Details on the presentations available in the AC-4 bitstream.
- Substream Information
  Details on the substreams available in the AC-4 bitstream.
- Metadata
  Audio metadata associated with the audio data encoded in the AC-4 frame.

Details on the information contained in these structures can found in the following clauses.

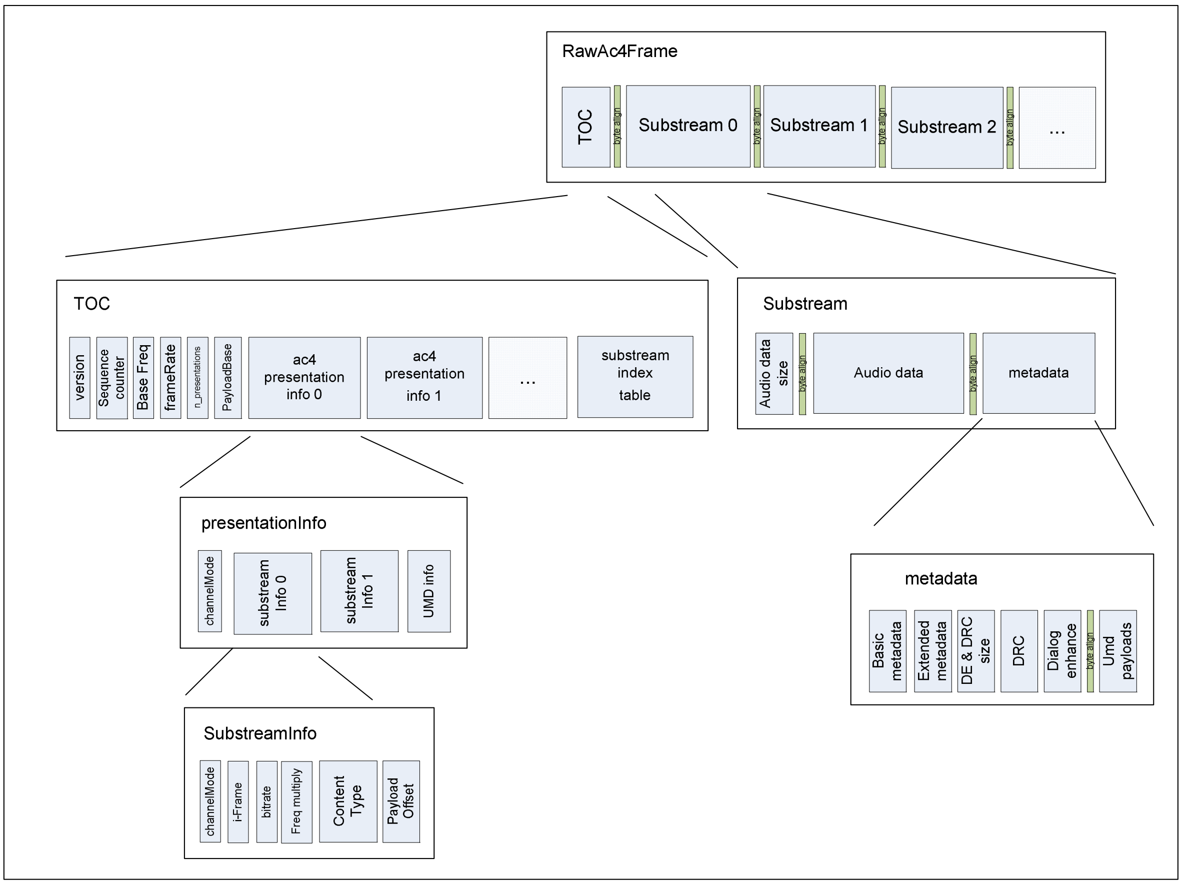

Figure 11: AC-4 container structure

6.2.2.1 Raw AC-4 Frame

Each raw AC-4 frame contains a table of contents (TOC) and at least one byte aligned substream. The TOC can be considered as the bitstream inventory where all information that is important for the overlaying system resides. Substreams are decodable units that represent a specific channel configuration (mono, stereo, 3.0, 5.1, 7.1).

In transport scenarios, a raw AC-4 frame is typically embedded in a transport container, while it would be considered an "AC-4 sample" in ISO based file formats. Details on embedding an AC-4 frame in ISO based files can be found in Annex E.

The syntax of the `raw_ac4_frame()` element is described in clause 4.3.1.

6.2.2.2 Table of Contents (TOC)

The Table of Contents contains the TOC data and at least one presentation as well as a substream index table. As the bitstream inventory the TOC provides information about and access to individual substreams.

The TOC data contains stream specific metadata, such as:

- Bitstream version
  Signals if the bitstream is decodable by a decoder compliant to this AC-4 specification.
- Sequence counter
  Can be used to detect splices in the bitstream.
- Information about buffer model such as current average bitrate and bitrate smoothing buffer level.
- Information about I-Frames usable as random access points.

- Base sampling frequency
  Serves as a common denominator for the output sampling frequency of the audio.

- Total number of presentations.

Additionally, the TOC contains the presentation information as well as a substream index. The index provides easy access to each of the substreams by specifying the absolute number and the size of each of the substreams. Further information about the presentation information can be found in clause 6.2.2.3.

Because of the fact that all the above information is in the TOC, it is not necessary to parse the complete bitstream to decode a single presentation, given more presentations are available.

The TOC is contained in the bitstream element `ac4_toc()` which is described in clause 4.3.3.2.

6.2.2.3 Presentation information

Typically, a presentation in AC-4 describes a set of substreams that is to be decoded and presented simultaneously. Mixing of decoded content will only occur within one presentation. That means only one presentation has to be decoded at a time. On the other hand, one substream can be part of several presentations.

Examples for presentations are:

- Single substream
  The common case: a single multi-channel stream in one presentation.

- Multi-Language
  A set of 'dry main' (music and effects) combined with multiple dialog language tracks.

- Main/Associate
  A set of main and associate signal, the latter of which could contain scene description or the director's comments.

- Dialog enhancement track
  A separate track used for boosting the dialog (see clause 5.7.8 for details).

The presentation information container tells the decoder which substreams to decode, where to find them, and how they shall be presented. The `ac4_presentation_info` element is described in clause 4.3.3.3.

6.2.2.4 Substream

A substream contains the actual audio data, as well as the corresponding metadata.

Additionally, the size of the audio data is given in the header of the substream, such that the metadata can be accessed without the need of parsing the audio data.

The `ac4_substream ()` element is specified in clause 4.3.4.

6.2.3 Selecting and decoding a presentation

To successfully select and decode an appropriate presentation, a decoder system has to execute the following steps.

1) Create a list of presentations available in the bitstream

   Initially, the number of presentations, *n_presentations*, has to be derived from the bitstream elements `b_single_presentation` and `b_more_presentations` in `ac4_toc()`, as described in clause 4.3.3.2.

   For each of these *n_presentations*, parse the `ac4_presentation_info()` fields. Here, "single substream" presentations as well as multi-substream presentations described by the `presentation_config` element can be found, as indicated in clause 4.3.3.3.3.

2) Select the presentation which is appropriate for the decoder and the output scenario

Information available to support this selection are mainly language type, availability of associated audio, or type of audio (multichannel for speaker rendering, or a previrtualized rendition for headphones).

A decoding system should employ a user agent to make the choice without the need to directly with the customer.

NOTE: Presentations can come and go any time in the stream.

3) Select the substreams associated with the presentation

From each of the ac4_substream_info() fields associated with the `presentation_config`, extract the presentation-specific metadata such as channel mode, bitrate information, content type and the substream index, according to clause 4.3.3.7.

The `substream_index` field indicates the substream associated with the presentation. It can be used as an index to the `substream_index_table`.

For each of the substreams, the `substream_index_table()` will provide the total size of the substream. If that is the case (`b_size_present` == 1), the offset to the substream data relative to the end of the `ac4_toc()` element in the elementary stream can be calculated as follows.

Pseudocode

```
n = substream_index;
substream_n_offset = payload_base;
For (s=0; s<n; s++) {
    substream_n_offset += substream_size[s];
}
```

NOTE: payload_base is defined in clause 4.3.3.2.11.

4) Decode the selected substream(s)

Instructions on how to decode the selected substream(s) can be found in clause 6.2.5 and the following clauses.

5) Mix the decoded substream(s)

Instructions on how to mix the substream(s) can be found in clause 6.2.16.

6) Render the mixed audio

Instructions on how to render the audio can be found in clause 6.2.17.

6.2.4 Buffer model

Decoders implemented in accordance with the present document shall provide a minimum input buffer of size N bytes where N is set according to:

$$N = v \times {R_{stream}}/{R_{frame}}$$

R_{stream} and R_{frame} are the total bitrate and the *frame_rate*, respectively, and $v = \begin{cases} 6, & R_{Frame} \leq 60 \\ 12, & else \end{cases}$.

See table 205 for typical values of N.

NOTE: VBR streams (see clause 4.3.3.2.4) may contain `raw_ac4_frames` exceeding N.

Table 205: Minimum buffer size N for typical frame rates and bitrates at samplerate 48 kHz

		frame_rate				
Bitrate [bps]		**25**	**30**	**50**	**60**	**120**
	48 000	1 440	1 200	720	600	600
	96 000	2 880	2 400	1 440	1 200	1 200
	160 000	4 800	4 000	2 400	2 000	2 000
	1 536 000	46 080	38 400	23 040	19 200	19 200

6.2.5 Decoding of a substream

As described in clause 4.2.4 there are three different types of AC-4 substreams:

- AC-4 substream
- AC-4 high sampling frequency extension substream
- Universal metadata payloads substream

6.2.5.1 Decoding of an AC-4 substream

A decoder shall decode the AC-4 substream's `audio_data()` and `metadata()`. Both bitstream elements can be accessed independently of each other by using the variable *audio_size*.

Decoding of the channel elements within `audio_data()` is described in clause 6.2.6 and following.

The bitstream elements contained in `metadata()` shall be decoded from the bitstream using the information provided in clauses 4.2.14 and 4.3.12 to clause 4.3.15. The decoded information about rendering, dialog enhancement and dynamic range control is used according to the description in clauses 5.7.8, 6.2.13 and in clause 6.2.17, respectively.

6.2.5.2 Decoding of an AC-4 HSF extension substream

Decoding of the HSF substream can be achieved using the information derived from clause 5.4. Afterwards, the SAP tool and the IMDCT described in clauses 5.3 and 5.5 need to be employed. No QMF domain processing is required.

6.2.5.3 Decoding of a UMD payloads substream

A decoder shall be able to skip a UMD substream, and may otherwise ignore its contents.

6.2.6 Spectral frontend decoding

The channel element contained in the `audio_data` element of an AC-4 substream can be one of four types, signaled by the channel_mode, as indicated in clause 4.2.6.

Within the channel elements, audio tracks are encoded as `sf_data` elements. These elements are associated with an `sf_info` element, as well as with a spectral frontend type.

Tracks with an associated spectral frontend type of ASF (`spec_frontend`=0) shall be processed by the Audio Spectral Frontend (ASF) for the decoding of spectral lines, as described in clause 6.2.6.4. Tracks associated with the type SSF (`spec_frontend`=1) shall be processed by the Speech Spectral Frontend (SSF), as described in clause 6.2.6.5. The output of both spectral frontends is a vector of spectral lines for each track, as well as associated information about the subsequent windowing.

6.2.6.1 Mono decoding

In case that `channel_mode` is mono, there is just one track in the bitstream stored as `sf_data` element within a `mono_data` element. The decoding of the `sf_data` element depends on the spectral frontend type and on the `sf_info` element and is done in 3 steps:

1) Parsing of `sf_info` which includes transform and psy information for the spectral frontend type.

2) Parsing of spectral frontend data depending on spectral frontend type.

3) Decoding of spectral frontend data depending on spectral frontend type.

The resulting mono track shall be considered the center channel (C), and shall be directly routed to the IMDCT stage.

6.2.6.2 Stereo decoding

In case that `channel_mode` is stereo, there are up to two audio tracks in the bitstream, stored as `sf_data` elements within a `channel_pair_element`. Similar to the mono decoding, the decoding of the `sf_data` element is done in 3 steps, see clause 6.2.6.1. If the `b_enable_mdct_stereo_proc` value is '1', a common `sf_info` element is present which shall be used for the decoding of both `sf_data` elements.

Subsequently, the decoded audio tracks are routed into the stereo and multichannel processing tool, described in clause 5.3. Depending on the `stereo_codec_mode`, the tool assigns dedicated speaker locations to the tracks, and by that turning them into dedicated channels. The spectral lines of these channels shall subsequently be IMDCT processed, as described in clause 6.2.7.

6.2.6.3 Multichannel audio decoding

Similar to stereo decoding, first all the `sf_data` elements are decoded. followed by routing the decoded tracks into the stereo and multichannel processing tool. This tool returns dedicated audio channels. The spectral lines of these channels shall subsequently be IMDCT processed.

In case the channel mode is "5.1" or "7.1", the audio track acquired from the first `sf_data` element shall be mapped to the LFE channel, and shall be directly routed to the IMDCT stage, i.e. is not passed into the stereo and multichannel processing tool.

6.2.6.4 Audio spectral frontend (ASF)

The `asf_section_data` and `asf_scalefac_data` elements of the `sf_data` element are parsed and helper elements are set up. See clauses 4.3.6.3 and 4.3.6.5. The `asf_spectral_data` is Huffman decoded using the helper elements. See clause 5.1.2 for details of the Huffman decoding process for the quantized spectral lines. Next, the quantized spectral lines shall be reconstructed and scaled as described in clause 5.1.3. If the scaled spectral lines do not belong to a full block, a reordering of the spectral lines shall be performed as described in clause 5.1.5.

6.2.6.5 Speech spectral frontend (SSF)

The `sf_data` element contains just the `ssf_data` element if the `spec_frontend` value is '1'. Either one or two `ssf_granule` elements are present in the `ssf_data` element. Each `ssf_granule` element is decoded into a vector of spectral lines as specified in clause 5.2.

6.2.7 Inverse transform (IMDCT) and window overlap/add

The decoding steps described above will result in a set of *frame_length* (reordered) spectral lines for each spectral frontend channel. The inverse MDCT transform converts the blocks of spectral lines into blocks of time samples.

The individual blocks of time samples shall be windowed, and adjacent blocks shall be overlapped and added together in order to reconstruct a continuous time PCM audio signal for each spectral frontend channel. The window and overlap/add steps are described along with the inverse MDCT transform in clause 5.5.

6.2.8 QMF analysis

The time domain samples of each channel are transformed individually into the QMF domain by the use of a QMF analysis filter as described in clause 5.7.3. Depending on the *frame_length*, 32, 30, 24, 15, 12 or 6 QMF slots are the result of the QMF analysis. Each QMF slot is the result of one QMF filtering step with *num_qmf_subbands* time domain samples as input. The QMF analysis step is needed for the tools which operate in the QMF domain.

6.2.9 Companding

The companding tool mitigates pre- and post-echo artefacts. The encoder applies attenuation to signal parts with higher energy, and gain to signal parts with lower energy. The decoder applies the inverse process, which effectively shapes the coding noise by the signal energy.

Whether or not a channel is processed by the companding tool is dependent on the channel element and the codec mode. Table 206 lists the channels that shall be processed by the companding tool, while channels not listed shall not be processed.

Table 206: Channels processed by companding tool

Channel element	Codec mode	Channels in Companding
single_channel_element	1	C
channel_pair_element	1	L, R
channel_pair_element	2,3	L
3_0_channel_element	1	L,R,C
5_X_channel_element	1	L,R,C,Ls,Rs
5_X_channel_element	2,3	L,R,C
5_X_channel_element	4	L,R
7_X_channel_element	2,3	L,R,C,Ls,Rs

The companding tool is described in clause 5.7.5.

6.2.10 Advanced spectral extension - A-SPX

Spectral extension includes translating the spectral content of suitable low frequency regions of the signal to higher frequencies and adjusting them such that they perceptually match the high frequency content that is present in the original signal. Waveform coding may be employed in addition, either when critical signal frequency components are not present in the low frequency regions, or to reproduce transients more accurately than with spectral extension alone.

An AC-4 decoder shall process the QMF domain channels except for the LFE channel with the A-SPX tool after the companding tool has been applied. A-SPX shall be active for all codec modes except for the SIMPLE codec mode (0).

Table 207 lists the input A-SPX channels for each channel element and codec mode. Individually processed channels (as $Qin_{ASPX,a}$) are denoted as single letters, channels processed simultaneously (as $Qin_{ASPX,a}$ and $Qin_{ASPX,b}$) are combined by surrounding parenthesis. The order of the listing indicates the order of corresponding `aspx_data` elements in the bitstream, `aspx_data_1ch` for individually processed channels, `aspx_data_2ch` for simultaneously processed channels.

Table 207: Channels processed by A-SPX tool

Channel element	Codec mode	Channel mode	Channels in A-SPX
single_channel_element	1		C
channel_pair_element	1		(L, R)
channel_pair_element	2,3		L
3_0_channel_element	1		(L,R),C
5_X_channel_element	1		(L,R),(Ls,Rs),C
5_X_channel_element	2,3		(L,R),C
5_X_channel_element	4		(L,R)
7_X_channel_element	1	3/4/0.x	(L,R),(Ls,Rs),C,(Lrs,Rrs)
7_X_channel_element	1	5/2/0.x	(L,R),(Lw,Rw),C,(Ls,Rs)
7_X_channel_element	1	3/2/2.x	(L,R),(Ls,Rs),C,(Lth,Rth)
7_X_channel_element	2,3		(L,R),(Ls,Rs),C

The A-SPX tool is described in clause 5.7.6.

6.2.11 Advanced coupling - A-CPL

Advanced coupling extracts multichannel content from a downmix into fewer channels, using encoder controlled side information. As a special case, a stereo signal can be generated from a mono downmix.

An AC-4 decoder shall process the QMF domain channels with the A-CPL tool after the A-SPX tool has been applied. A-CPL shall be active for all codec modes except for the SIMPLE and ASPX codec modes (0,1). Table 208 indicates which channels shall be processed by the A-CPL tool.

Table 208: Channels processed by A-CPL tool

Channel element	Codec mode	Channels in A-CPL
channel_pair_element	2,3	L,R
5_X_channel_element	2,3,4	L,R,Ls,Rs,C
7_X_channel_element	2,3	all channels except LFE

The A-CPL tool is described in clause 5.7.7.

6.2.12 Dialog Enhancement - DE

Dialog Enhancement control metadata can be specified in the bitstream. When the decoder is configured to enable dialog enhancement, the dialog enhancement tool uses this metadata to apply a gain on the dialog in the signal according to a system-defined gain.

Table 209 lists the input channels for the DE tool.

Table 209: Channels processed by DE tool

Channel element	Channels in DE
single_channel_element	C
channel_pair_element	L,R
3_0_channel_element	L,R,C
5_X_channel_element	L,R,C
7_X_channel_element	L,R,C

The Dialog Enhancement tool is described in clause 5.7.8.

6.2.13 Dynamic range control - DRC

Dynamic range control metadata can be specified in the bitstream. The DRC tool uses this metadata to apply gains to the channels in the QMF domain in order to adjust the dynamic range as specified in clause 5.7.9.

The DRC tool takes two inputs per channel: the audio signal which is output by a preceding tool (usually dialog enhancement) and the signal that is used to measure levels and drive the side chain. The signal driving the side chain shall be identical to the input of dialog enhancement (i.e. it does not include the dialog enhancements).

6.2.14 QMF synthesis

A QMF synthesis filter shall transform each audio channel from the QMF domain back to the time domain as specified in clause 5.7.4.

6.2.15 Sampling rate converter

For all values of the `frame_rate_index`, the decoder may be operated at external sampling frequencies of 48 kHz, 96 kHz, or 192 kHz. For `frame_rate_index` = 13, 44,1 kHz is also a permissible external sampling frequency.

To adjust the sampling frequency to the external sampling frequency a sampling rate converter shall be used. The decoder resampling ratio shall be as specified in clause 4.3.3.2.

The sampling rate converter should use high-quality anti-aliasing filters.

6.2.16 Mixing substream outputs

This clause describes how to mix several substreams contained within one presentation. This includes:

- Mixing main + associated audio
- Mixing dry main + dialog

Dialog enhancement is not part of this clause.

The decoding of a presentation which contains multiple substreams, as indicated by a `b_single_substream` value of '0', and where `presentation_config` has a value of '0', '2' or '3' shall involve a mixing of the substream outputs as specified in this clause. The channel configuration of the mixer output matches the channel configuration of the main or the dry main substream. No additional channels are allowed for the dialog or associate substream, except for a mono channel.

The mixing metadata for main / associate mixing and for dry main / dialog mixing are not necessarily sent each frame, and not necessarily each I-frame, although this is encouraged. The mixing metadata values remain valid until new ones are transmitted in the same stream or until a splice is detected.

Before mixing the substreams, they need to be a the same reference level. Before mixing, the decoder shall apply a gain to the substream:

$$\boldsymbol{pout}_{ch}(\mathrm{n}) = 10^{\frac{L_{out}-dialnorm}{20}} \cdot \boldsymbol{pin}_{ch}(n)$$

Here, $\boldsymbol{pin}_{ch}$ and $\boldsymbol{pout}_{ch}$ refer to the input and output samples of each channel *ch*, respectively. L_{out} is the reference output level, supplied by the system.

When mixing is followed by a DRC stage, L_{out} replaces dialnorm after this operation.

6.2.16.1 Mixing dry main and dialog

Before mixing the dialog channels, a single user agent provided gain *g_dialog* ∈ [-∞, *g_dialog_max*] dB should be applied to all dialog channels. This allows for a user-controlled adjustment of the relative dialog level. The user agent default value for *g_dialog* shall be 0 dB.

Each channel which is only present in the dry main substream output is left untouched. All other channels are mixed with the corresponding channel from the dialog substream output:

$$Y\_mix_i = X\_dry\_main_i + X\_dialog_i,$$

where Y_mix_i is a mixer output sample and $X\_dry\_main_i$ and X_dialog_i are the corresponding mixer input samples from the dry main channel and the dialog channel involved in the mixing process.

6.2.16.2 Mixing main and associate

Before mixing the associate channels, a single user agent provided gain *g_assoc* ∈ [-∞, 0] dB should be applied to all associate channels. This allows for a user-controlled adjustment of the relative associate level. The user agent default value for *g_assoc* shall be 0 dB.

The `extended_metadata` part of the associate substream shall provide gain values for a gain-adjustment of the channels in the main substream if an adjustment needs to be done. If no gain values are given, a default of 0 dB shall be used. First, the center channel of the main substream, if available, is gain-adjusted using `scale_main_center`. Next, the front channels of the main substream are gain adjusted using `scale_main_front`. And finally, all channels of the main substream are gain-adjusted using `scale_main`.

If the `channel_mode` of the associate substream is 'mono', a `pan_associated` value is specified in the `extended_metadata`. The panning value in °, or a default of 0°, shall be used to pan the mono signal to all channels available in the main substream. This is achieved by converting the panning value into an array of panning gains, one entry for each channel available in the main substream. The mixer output then is given by:

$$Y\_mix_{i,ch} = X\_main\_adj_{i,ch} + X\_associate_{i,mono} \times pan\_gain[ch],$$

where $Y\_mix_{i,ch}$ is a mixer output sample for channel *ch* and $X\_main\_adj_{i,ch}$ is the corresponding mixer input sample for channel *ch* form the gain-adjusted main channel. $X\_associate_{i,mono}$ is the corresponding mixer input sample from the mono associate channel and *pan_gain[ch]* is the panning gain for channel *ch*.

If the `channel_mode` of the associate substream is not 'mono', all channels which are only present in the main substream output are replaced by the gain-adjusted signal. Channels available in both, the main and associate substream, are mixed using the gain-adjusted main signal:

$$Y\_mix_i = X\_main\_adj_i + X\_associate_i,$$

where Y_mix_i is a mixer output sample and $X\_main\_adj_i$ and $X\_associate_i$ are the corresponding mixer input samples from the gain-adjusted main channel and the associate channel involved in the mixing process.

6.2.16.3 Mixing dry main, dialog and associate

The mixing of dry main, dialog and associate substream outputs is done in two stages. First, the dry main and the dialog channels are mixed as specified in clause 6.2.16.1. The resulting main signal is mixed with the associate signal as described in clause 6.2.16.2.

6.2.17 Rendering a presentation

When the number of loudspeakers does not match the number of encoded audio channels either downmixing or upmixing is required to render the complete audio programme. It is important that the equations for downmixing and upmixing be standardized, so that programme providers can be confident of how their programme will be rendered over systems with a varying number of loudspeakers. Using standardized rendering equations programme producers can monitor how the rendered version will sound and make any alterations necessary so that acceptable results are achieved for all listeners.

The source channel modes for downmixing are 7.X, 5.X, 3.0, and Stereo. The destination channel modes for downmixing are 5.X, Stereo and Mono.

The source channel modes for upmixing are Mono, Stereo, 3.0 and 5.X. The destination channel modes for upmixing are Stereo, 3.0, 5.X and 7.X.

The downmix and upmix operations are cascaded. For example, to downmix from seven channels to two channels, the seven channels are first downmixed to five channels and then five channels to two channels. Similarly, to upmix from one channel to five channels, the mono channel is upmixed to two channels, two channels to three channels, and three channels to five channels.

6.2.17.1 Generalized rendering matrix and equation

The generalized rendering matrix, **R**, is as shown below.

$$\begin{bmatrix} r_{0,0} & r_{0,1} & \cdots & r_{0,n-1} \\ r_{1,0} & r_{1,1}, & \cdots & r_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ r_{n-1,0} & r_{n-1,1} & \cdots & r_{n-1,n-1} \end{bmatrix}$$

where *n* is the number of channels listed in table D.1.

The elements $r_{0,0}$ to $r_{n-1,n-1}$ are defined for each combination of available input and output channels.

The rendering equation is as follows:

$$\begin{bmatrix} y\_out_0 \\ y\_out_1 \\ \vdots \\ y\_out_{n-1} \end{bmatrix} = \begin{bmatrix} r_{0,0} & r_{0,1} & \cdots & r_{0,n-1} \\ r_{1,0} & r_{1,1}, & \cdots & r_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ r_{n-1,0} & r_{n-1,1} & \cdots & r_{n-1,n-1} \end{bmatrix} \begin{bmatrix} x\_in_0 \\ x\_in_1 \\ \vdots \\ x\_in_{n-1} \end{bmatrix}$$

where **x_in** is a matrix of sample values of input channels, **y_out** is a matrix of sample values for output channels.

6.2.17.2 Downmixing from two channels into one channel

The rendering matrix $\mathbf{R}_{2:1}$ used to downmix two channels (L,R) into one channel (C) has the elements $r_{1,0}$ and $r_{1,2}$ set to 1 and all other elements of the matrix set to 0.

6.2.17.3 Downmixing from three channels into two channels

The rendering matrix $\mathbf{R}_{3:2}$ used to downmix from three channels into two channels is dependent on the bitstream element `preferred_dmx_method`, described in clause 4.3.12.2.19.

The non-zero entries of the rendering matrix $\mathbf{R}_{3:2}$ are given in table 210.

Table 210: Matrix entries for 3- to 2-channel downmix matrix

preferred_dmx_method	$r_{0,0}$ $r_{2,2}$	$r_{0,1}$ $r_{0,2}$
1	1	*loro_cmg*
2	1	*ltrt_cmg*
3	1	*ltrt_cmg*

where *loro_cmg* is derived from the bitstream element `loro_center_mixgain`, as described in clause 4.3.12.2.8, and *ltrt_cmg* is derived from the bitstream element `ltrt_center_mixgain`, described in clause 4.3.12.2.13.

All other elements of the rendering matrix are set to 0.

Additionally, *loro_corr_gain* (`preferred_dmx_method` = 1) or *ltrt_corr_gain* (`preferred_dmx_method` = 2,3) shall be applied to the downmixed signal if available in the bitstream, as described in clauses 4.3.12.2.11 and 4.3.12.2.16.

6.2.17.4 Downmixing from five channels into two channels

The rendering matrix $\mathbf{R}_{5:2}$ used to downmix from five channels into two channels is dependent on the bitstream element `preferred_dmx_method`, described in clause 4.3.12.2.19.

The non-zero entries of the rendering matrix $\mathbf{R}_{5:3}$ are given in table 211.

Table 211: Matrix entries for 5- to 2-channel downmix matrix

preferred_dmx_method	$r_{0,0}$ $r_{2,2}$	$r_{0,1}$ $r_{2,1}$	$r_{0,3}$	$r_{2,4}$	$r_{0,4}$	$r_{2,3}$
1	1	*loro_cmg*	*loro_smg*	*loro_smg*	0	0
2	1	*ltrt_cmg*	*-ltrt_smg*	*ltrt_smg*	*-ltrt_smg*	*ltrt_smg*
3	1	*ltrt_cmg*	*-ltrt_smg* × (+1,8 dB)	*ltrt_smg* × (+1,8 dB)	*-ltrt_smg* × (-3,2 dB)	*ltrt_smg* × (-3,2 dB)

where *loro_cmg* and *loro_smg* are derived from the bitstream elements `loro_center_mixgain` and `loro_surround_mixgain`, respectively, as described in clause 4.3.12.2.8. Likewise, *ltrt_cmg* and *ltrt_smg* are derived from the bitstream elements `ltrt_center_mixgain` and `ltrt_surround_mixgain`, respectively, described in clause 4.3.12.2.13.

All other elements of the rendering matrix are set to 0.

Additionally, *loro_corr_gain* (`preferred_dmx_method` = 1) or *ltrt_corr_gain* (`preferred_dmx_method` = 2,3) shall be applied to the downmixed signal if available in the bitstream, as described in clauses 4.3.12.2.11 and 4.3.12.2.16.

6.2.17.5 Downmixing from seven channels into five channels

The rendering matrix $\mathbf{R}_{7:5}$ used to downmix seven channels into five channels is dependent on the channel mode, as well as on the bitstream element `add_ch_base`.

Center channel ($r_{1,1} = 1$) and LFE (if present) are routed through ($r_{18,18} = 1$).

The remaining non-zero entries of the rendering matrix $\mathbf{R}_{7:5}$ are given in table 212.

Table 212: Additional matrix entries for 7-to 5-channel downmix matrix $r_{7\text{-}5}$

channel mode	add_ch_base	$r_{0,13}$ $r_{2,14}$	$r_{0,15}$ $r_{2,16}$	$r_{3,3}$ $r_{4,4}$	$r_{3,7}$ $r_{4,8}$	$r_{3,15}$ $r_{4,16}$
3/4/0.x	n/a	0	0	0,707	0,707	0
5/2/0.x	0	0,707	0	1	0	0
	1	0	0	0,707	0,707	0
3/2/2.x	0	0	0,707	1	0	0
	1	0	0	0,707	0	0,707

6.2.17.6 Upmixing from one channel into two channels

The rendering matrix $\mathbf{R}_{1:2}$ used to upmix from a mono channel to stereo has the elements $r_{1,0}$ and $r_{1,2}$ set to 0,707 and all other elements of the matrix set to 0.

6.2.17.7 Upmixing from two channels into three channels

The rendering matrix $\mathbf{R}_{2:3}$ used to upmix from two channels to three channels has the elements $r_{0,0}$, and $r_{2,2}$ set to 1 and all other elements of the matrix set to 0.

6.2.17.8 Upmixing from three channels into five channels

The rendering matrix $\mathbf{R}_{3:5}$ used to upmix from three channels to five channels has the elements $r_{0,0}$, $r_{1,1}$, and $r_{2,2}$ set to 1and all other elements of the matrix set to 0.

6.2.17.9 Upmixing from five channels into seven channels

The rendering matrix $\mathbf{R}_{5:7}$ used to upmix from five channels into seven channels has the elements $r_{0,0}$, $r_{1,1}$, $r_{2,2}$, $r_{3,3}$, and $r_{4,4}$ set to 1. The LFE channel (if present) is routed through ($r_{18,18} = 1$) as well. All other elements of the matrix are set to 0.

6.2.18 Decoding audio in sync with video

In order to achieve A/V sync, a synchronization mechanism, usually employing presentation timestamps, needs to be defined in application standards referencing the present document. If such a timestamp is defined and applies to a certain audio frame, it should indicate the presentation time of the sample at the reference point at sample position *frame_length*/2 of the composition buffer.

6.2.19 Switching streams while decoding

AC-4 has been designed to allow for a seamless switching of AC-4 streams during the run-time of an AC-4 decoder. When the switch happens at an I-frame boundary, which is the case when the first frame of the new stream is an I-frame, the AC-4 decoder shall produce a flawless output. When the switch does not happen at an I-frame boundary, the AC-4 decoder shall restore the audio output before or latest with the reception of the next I-frame. One aspect that allows for a seamless switching is the alignment of the frame signal with the control signal as described in clause 5.6. The seamless switching capability of an AC-4 decoder facilitates several use cases.

In an adaptive streaming application, the bitrate can be seamlessly adjusted according to the available transmission channel bandwidth. The server, which for example can be a MPEG-DASH [1] server, just needs to make sure that the AC-4 stream to be switched to contains an I-frame at the switching point. This information is easily available to the server without much parsing effort since the `b_iframe_global` flag is part of the `ac4_toc()`, the first element of a `raw_ac4_frame()`. When the switch is done as described on an I-frame boundary, the AC-4 decoder is able to produce a flawless output signal.

When AC-4 is used in a TV application, a user initiated switch of streams can be detected by evaluating the `sequence_counter` as described in clause 4.3.3.2.2. No external signalling of a stream switch to the AC-4 decoder is needed. Since the switch might occur at a point in time when no I-frame is present in the new stream, it is possible that the audio output signal has a reduced coverage for a short period of time until the next I-frame is received and the full output signal is available again.

A switch of streams at the broadcast side, often referred to as splicing, can be either a controlled or uncontrolled splice. A controlled splice is a splice which happens on an I-frame boundary and should be done similar to the adaptive streaming use case described above. In addition to selecting an I-frame as the first AC-4 frame after the splice point, the `sequence_counter` of this AC-4 frame should be set to '0' to indicate the splice. If all this is done, the AC-4 decoder is able to produce a seamless output signal. If a controlled splice is not possible, the `sequence_counter` of the first AC-4 frame of the new stream should be set to '0' to indicate the splice to the decoder. An AC-4 decoder shall use this splice indication to not use any information from previous frames when decoding the first frame after a splice.

Annex A (normative): Huffman codebook tables

All Huffman codebook tables are available in the accompanying file ts_103190_tables.c. Each Huffman table consists of a sub-table containing length values and another sub-table containing the codeword. The indexing starts with index 0 and ends with index *codebook_length*-1.

A.1 ASF Huffman codebook tables

Table A.1: ASF scale factor Huffman codebook

Codebook name	ASF_HCB_SCALEFAC
Codebook length table	ASF_HCB_SCALEFAC_LEN
Codebook codeword table	ASF_HCB_SCALEFAC_CW
codebook_length	121

Table A.2: ASF spectrum Huffman codebook 1

Codebook name	ASF_HCB_1
Codebook length table	ASF_HCB_1_LEN
Codebook codeword table	ASF_HCB_1_CW
codebook_length	81
cb_mod	3
cb_mod2	9
cb_mod3	27
cb_off	1

Table A.3: ASF spectrum Huffman codebook 2

Codebook name	ASF_HCB_2
Codebook length table	ASF_HCB_2_LEN
Codebook codeword table	ASF_HCB_2_CW
codebook_length	81
cb_mod	3
cb_mod2	9
cb_mod3	27
cb_off	1

Table A.4: ASF spectrum Huffman codebook 3

Codebook name	ASF_HCB_3
Codebook length table	ASF_HCB_3_LEN
Codebook codeword table	ASF_HCB_3_CW
codebook_length	81
cb_mod	3
cb_mod2	9
cb_mod3	27
cb_off	0

Table A.5: ASF spectrum Huffman codebook 4

Codebook name	ASF_HCB_4
Codebook length table	ASF_HCB_4_LEN
Codebook codeword table	ASF_HCB_4_CW
codebook_length	81
cb_mod	3
cb_mod2	9
cb_mod3	27
cb_off	0

Table A.6: ASF spectrum Huffman codebook 5

Codebook name	ASF_HCB_5
Codebook length table	ASF_HCB_5_LEN
Codebook codeword table	ASF_HCB_5_CW
codebook_length	81
cb_mod	9
cb_off	4

Table A.7: ASF spectrum Huffman codebook 6

Codebook name	ASF_HCB_6
Codebook length table	ASF_HCB_6_LEN
Codebook codeword table	ASF_HCB_6_CW
codebook_length	81
cb_mod	9
cb_off	4

Table A.8: ASF spectrum Huffman codebook 7

Codebook name	ASF_HCB_7
Codebook length table	ASF_HCB_7_LEN
Codebook codeword table	ASF_HCB_7_CW
codebook_length	64
cb_mod	8
cb_off	0

Table A.9: ASF spectrum Huffman codebook 8

Codebook name	ASF_HCB_8
Codebook length table	ASF_HCB_8_LEN
Codebook codeword table	ASF_HCB_8_CW
codebook_length	64
cb_mod	8
cb_off	0

Table A.10: ASF spectrum Huffman codebook 9

Codebook name	ASF_HCB_9
Codebook length table	ASF_HCB_9_LEN
Codebook codeword table	ASF_HCB_9_CW
codebook_length	169
cb_mod	13
cb_off	0

Table A.11: ASF spectrum Huffman codebook 10

Codebook name	ASF_HCB_10
Codebook length table	ASF_HCB_10_LEN
Codebook codeword table	ASF_HCB_10_CW
codebook_length	169
cb_mod	13
cb_off	0

Table A.12: ASF spectrum Huffman codebook 11

Codebook name	ASF_HCB_11
Codebook length table	ASF_HCB_11_LEN
Codebook codeword table	ASF_HCB_11_CW
codebook_length	289
cb_mod	17
cb_off	0

Table A.13: ASF spectral noise fill Huffman codebook

Codebook name	ASF_HCB_SNF
Codebook length table	ASF_HCB_SNF_LEN
Codebook codeword table	ASF_HCB_SNF_CW
codebook_length	22

Table A.14: CB_DIM

Codebook number	1	2	3	4	5	6	7	8	9	10	11
CB_DIM	4	4	4	4	2	2	2	2	2	2	2

Table A.15: UNSIGNED_CB

Codebook number	1	2	3	4	5	6	7	8	9	10	11
UNSIGNED_CB	false	false	true	true	false	false	true	true	true	true	true

A.2 A-SPX Huffman codebook tables

Table A.16: A-SPX Huffman codebook ASPX_HCB_ENV_LEVEL_15_F0

Codebook name	ASPX_HCB_ENV_LEVEL_15_F0
Codebook length table	ASPX_HCB_ENV_LEVEL_15_F0_LEN
Codebook codeword table	ASPX_HCB_ENV_LEVEL_15_F0_CW
codebook_length	71

Table A.17: A-SPX Huffman codebook ASPX_HCB_ENV_LEVEL_15_DF

Codebook name	ASPX_HCB_ENV_LEVEL_15_DF
Codebook length table	ASPX_HCB_ENV_LEVEL_15_DF_LEN
Codebook codeword table	ASPX_HCB_ENV_LEVEL_15_DF_CW
codebook_length	141
cb_off	70

Table A.18: A-SPX Huffman codebook ASPX_HCB_ENV_LEVEL_15_DT

Codebook name	ASPX_HCB_ENV_LEVEL_15_DT
Codebook length table	ASPX_HCB_ENV_LEVEL_15_DT_LEN
Codebook codeword table	ASPX_HCB_ENV_LEVEL_15_DT_CW
codebook_length	141
cb_off	70

Table A.19: A-SPX Huffman codebook ASPX_HCB_ENV_BALANCE_15_F0

Codebook name	ASPX_HCB_ENV_BALANCE_15_F0
Codebook length table	ASPX_HCB_ENV_BALANCE_15_F0_LEN
Codebook codeword table	ASPX_HCB_ENV_BALANCE_15_F0_CW
codebook_length	25

Table A.20: A-SPX Huffman codebook ASPX_HCB_ENV_BALANCE_15_DF

Codebook name	ASPX_HCB_ENV_BALANCE_15_DF
Codebook length table	ASPX_HCB_ENV_BALANCE_15_DF_LEN
Codebook codeword table	ASPX_HCB_ENV_BALANCE_15_DF_CW
codebook_length	49
cb_off	24

Table A.21: A-SPX Huffman codebook ASPX_HCB_ENV_BALANCE_15_DT

Codebook name	ASPX_HCB_ENV_BALANCE_15_DT
Codebook length table	ASPX_HCB_ENV_BALANCE_15_DT_LEN
Codebook codeword table	ASPX_HCB_ENV_BALANCE_15_DT_CW
codebook_length	49
cb_off	24

Table A.22: A-SPX Huffman codebook ASPX_HCB_ENV_LEVEL_30_F0

Codebook name	ASPX_HCB_ENV_LEVEL_30_F0
Codebook length table	ASPX_HCB_ENV_LEVEL_30_F0_LEN
Codebook codeword table	ASPX_HCB_ENV_LEVEL_30_F0_CW
codebook_length	36

Table A.23: A-SPX Huffman codebook ASPX_HCB_ENV_LEVEL_30_DF

Codebook name	ASPX_HCB_ENV_LEVEL_30_DF
Codebook length table	ASPX_HCB_ENV_LEVEL_30_DF_LEN
Codebook codeword table	ASPX_HCB_ENV_LEVEL_30_DF_CW
codebook_length	71
cb_off	35

Table A.24: A-SPX Huffman codebook ASPX_HCB_ENV_LEVEL_30_DT

Codebook name	ASPX_HCB_ENV_LEVEL_30_DT
Codebook length table	ASPX_HCB_ENV_LEVEL_30_DT_LEN
Codebook codeword table	ASPX_HCB_ENV_LEVEL_30_DT_CW
codebook_length	71
cb_off	35

Table A.25: A-SPX Huffman codebook ASPX_HCB_ENV_BALANCE_30_F0

Codebook name	ASPX_HCB_ENV_BALANCE_30_F0
Codebook length table	ASPX_HCB_ENV_BALANCE_30_F0_LEN
Codebook codeword table	ASPX_HCB_ENV_BALANCE_30_F0_CW
codebook_length	13

Table A.26: A-SPX Huffman codebook ASPX_HCB_ENV_BALANCE_30_DF

Codebook name	ASPX_HCB_ENV_BALANCE_30_DF
Codebook length table	ASPX_HCB_ENV_BALANCE_30_DF_LEN
Codebook codeword table	ASPX_HCB_ENV_BALANCE_30_DF_CW
codebook_length	25
cb_off	12

Table A.27: A-SPX Huffman codebook ASPX_HCB_ENV_BALANCE_30_DT

Codebook name	ASPX_HCB_ENV_BALANCE_30_DT
Codebook length table	ASPX_HCB_ENV_BALANCE_30_DT_LEN
Codebook codeword table	ASPX_HCB_ENV_BALANCE_30_DT_CW
codebook_length	25
cb_off	12

Table A.28: A-SPX Huffman codebook ASPX_HCB_NOISE_LEVEL_F0

Codebook name	ASPX_HCB_NOISE_LEVEL_F0
Codebook length table	ASPX_HCB_NOISE_LEVEL_F0_LEN
Codebook codeword table	ASPX_HCB_NOISE_LEVEL_F0_CW
codebook_length	30

Table A.29: A-SPX Huffman codebook ASPX_HCB_NOISE_LEVEL_DF

Codebook name	ASPX_HCB_NOISE_LEVEL_DF
Codebook length table	ASPX_HCB_NOISE_LEVEL_DF_LEN
Codebook codeword table	ASPX_HCB_NOISE_LEVEL_DF_CW
codebook_length	59
cb_off	29

Table A.30: A-SPX Huffman codebook ASPX_HCB_NOISE_LEVEL_DT

Codebook name	ASPX_HCB_NOISE_LEVEL_DT
Codebook length table	ASPX_HCB_NOISE_LEVEL_DT_LEN
Codebook codeword table	ASPX_HCB_NOISE_LEVEL_DT_CW
codebook_length	59
cb_off	29

Table A.31: A-SPX Huffman codebook ASPX_HCB_NOISE_BALANCE_F0

Codebook name	ASPX_HCB_NOISE_BALANCE_F0
Codebook length table	ASPX_HCB_NOISE_BALANCE_F0_LEN
Codebook codeword table	ASPX_HCB_NOISE_BALANCE_F0_CW
codebook_length	13

Table A.32: A-SPX Huffman codebook ASPX_HCB_NOISE_BALANCE_DF

Codebook name	ASPX_HCB_NOISE_BALANCE_DF
Codebook length table	ASPX_HCB_NOISE_BALANCE_DF_LEN
Codebook codeword table	ASPX_HCB_NOISE_BALANCE_DF_CW
codebook_length	25
cb_off	12

Table A.33: A-SPX Huffman codebook ASPX_HCB_NOISE_BALANCE_DT

Codebook name	ASPX_HCB_NOISE_BALANCE_DT
Codebook length table	ASPX_HCB_NOISE_BALANCE_DT_LEN
Codebook codeword table	ASPX_HCB_NOISE_BALANCE_DT_CW
codebook_length	25
cb_off	12

A.3 A-CPL Huffman codebook tables

Table A.34: A-CPL Huffman codebook ACPL_HCB_ALPHA_COARSE_F0

Codebook name	ACPL_HCB_ALPHA_COARSE_F0
Codebook length table	ACPL_HCB_ALPHA_COARSE_F0_LEN
Codebook codeword table	ACPL_HCB_ALPHA_COARSE_F0_CW
codebook_length	17

Table A.35: A-CPL Huffman codebook ACPL_HCB_ALPHA_FINE_F0

Codebook name	ACPL_HCB_ALPHA_FINE_F0
Codebook length table	ACPL_HCB_ALPHA_FINE_F0_LEN
Codebook codeword table	ACPL_HCB_ALPHA_FINE_F0_CW
codebook_length	33

Table A.36: A-CPL Huffman codebook ACPL_HCB_ALPHA_COARSE_DF

Codebook name	ACPL_HCB_ALPHA_COARSE_DF
Codebook length table	ACPL_HCB_ALPHA_COARSE_DF_LEN
Codebook codeword table	ACPL_HCB_ALPHA_COARSE_DF_CW
codebook_length	33
cb_off	16

Table A.37: A-CPL Huffman codebook ACPL_HCB_ALPHA_FINE_DF

Codebook name	ACPL_HCB_ALPHA_FINE_DF
Codebook length table	ACPL_HCB_ALPHA_FINE_DF_LEN
Codebook codeword table	ACPL_HCB_ALPHA_FINE_DF_CW
codebook_length	64
cb_off	32

Table A.38: A-CPL Huffman codebook ACPL_HCB_ALPHA_COARSE_DT

Codebook name	ACPL_HCB_ALPHA_COARSE_DT
Codebook length table	ACPL_HCB_ALPHA_COARSE_DT_LEN
Codebook codeword table	ACPL_HCB_ALPHA_COARSE_DT_CW
codebook_length	33
cb_off	16

Table A.39: A-CPL Huffman codebook ACPL_HCB_ALPHA_FINE_DT

Codebook name	ACPL_HCB_ALPHA_FINE_DT
Codebook length table	ACPL_HCB_ALPHA_FINE_DT_LEN
Codebook codeword table	ACPL_HCB_ALPHA_FINE_DT_CW
codebook_length	65
cb_off	32

Table A.40: A-CPL Huffman codebook ACPL_HCB_BETA_COARSE_F0

Codebook name	ACPL_HCB_BETA_COARSE_F0
Codebook length table	ACPL_HCB_BETA_COARSE_F0_LEN
Codebook codeword table	ACPL_HCB_BETA_COARSE_F0_CW
codebook_length	5

Table A.41: A-CPL Huffman codebook ACPL_HCB_BETA_FINE_F0

Codebook name	ACPL_HCB_BETA_FINE_F0
Codebook length table	ACPL_HCB_BETA_FINE_F0_LEN
Codebook codeword table	ACPL_HCB_BETA_FINE_F0_CW
codebook_length	9

Table A.42: A-CPL Huffman codebook ACPL_HCB_BETA_COARSE_DF

Codebook name	ACPL_HCB_BETA_COARSE_DF
Codebook length table	ACPL_HCB_BETA_COARSE_DF_LEN
Codebook codeword table	ACPL_HCB_BETA_COARSE_DF_CW
codebook_length	9
cb_off	4

Table A.43: A-CPL Huffman codebook ACPL_HCB_BETA_FINE_DF

Codebook name	ACPL_HCB_BETA_FINE_DF
Codebook length table	ACPL_HCB_BETA_FINE_DF_LEN
Codebook codeword table	ACPL_HCB_BETA_FINE_DF_CW
codebook_length	17
cb_off	8

Table A.44: A-CPL Huffman codebook ACPL_HCB_BETA_COARSE_DT

Codebook name	ACPL_HCB_BETA_COARSE_DT
Codebook length table	ACPL_HCB_BETA_COARSE_DT_LEN
Codebook codeword table	ACPL_HCB_BETA_COARSE_DT_CW
codebook_length	9
cb_off	4

Table A.45: A-CPL Huffman codebook ACPL_HCB_BETA_FINE_DT

Codebook name	ACPL_HCB_BETA_FINE_DT
Codebook length table	ACPL_HCB_BETA_FINE_DT_LEN
Codebook codeword table	ACPL_HCB_BETA_FINE_DT_CW
codebook_length	17
cb_off	8

Table A.46: A-CPL Huffman codebook ACPL_HCB_BETA3_COARSE_F0

Codebook name	ACPL_HCB_BETA3_COARSE_F0
Codebook length table	ACPL_HCB_BETA3_COARSE_F0_LEN
Codebook codeword table	ACPL_HCB_BETA3_COARSE_F0_CW
codebook_length	9

Table A.47: A-CPL Huffman codebook ACPL_HCB_BETA3_FINE_F0

Codebook name	ACPL_HCB_BETA3_FINE_F0
Codebook length table	ACPL_HCB_BETA3_FINE_F0_LEN
Codebook codeword table	ACPL_HCB_BETA3_FINE_F0_CW
codebook_length	17

Table A.48: A-CPL Huffman codebook ACPL_HCB_BETA3_COARSE_DF

Codebook name	ACPL_HCB_BETA3_COARSE_DF
Codebook length table	ACPL_HCB_BETA3_COARSE_DF_LEN
Codebook codeword table	ACPL_HCB_BETA3_COARSE_DF_CW
codebook_length	17
cb_off	8

Table A.49: A-CPL Huffman codebook ACPL_HCB_BETA3_FINE_DF

Codebook name	ACPL_HCB_BETA3_FINE_DF
Codebook length table	ACPL_HCB_BETA3_FINE_DF_LEN
Codebook codeword table	ACPL_HCB_BETA3_FINE_DF_CW
codebook_length	33
cb_off	16

Table A.50: A-CPL Huffman codebook ACPL_HCB_BETA3_COARSE_DT

Codebook name	ACPL_HCB_BETA3_COARSE_DT
Codebook length table	ACPL_HCB_BETA3_COARSE_DT_LEN
Codebook codeword table	ACPL_HCB_BETA3_COARSE_DT_CW
codebook_length	17
cb_off	8

Table A.51: A-CPL Huffman codebook ACPL_HCB_BETA3_FINE_DT

Codebook name	ACPL_HCB_BETA3_FINE_DT
Codebook length table	ACPL_HCB_BETA3_FINE_DT_LEN
Codebook codeword table	ACPL_HCB_BETA3_FINE_DT_CW
codebook_length	33
cb_off	16

Table A.52: A-CPL Huffman codebook ACPL_HCB_GAMMA_COARSE_F0

Codebook name	ACPL_HCB_GAMMA_COARSE_F0
Codebook length table	ACPL_HCB_GAMMA_COARSE_F0_LEN
Codebook codeword table	ACPL_HCB_GAMMA_COARSE_F0_CW
codebook_length	21
cb_off	10

Table A.53: A-CPL Huffman codebook ACPL_HCB_GAMMA_FINE_F0

Codebook name	ACPL_HCB_GAMMA_FINE_F0
Codebook length table	ACPL_HCB_GAMMA_FINE_F0_LEN
Codebook codeword table	ACPL_HCB_GAMMA_FINE_F0_CW
codebook_length	41
cb_off	20

Table A.54: A-CPL Huffman codebook ACPL_HCB_GAMMA_COARSE_DF

Codebook name	ACPL_HCB_GAMMA_COARSE_DF
Codebook length table	ACPL_HCB_GAMMA_COARSE_DF_LEN
Codebook codeword table	ACPL_HCB_GAMMA_COARSE_DF_CW
codebook_length	41
cb_off	20

Table A.55: A-CPL Huffman codebook ACPL_HCB_GAMMA_FINE_DF

Codebook name	ACPL_HCB_GAMMA_FINE_DF
Codebook length table	ACPL_HCB_GAMMA_FINE_DF_LEN
Codebook codeword table	ACPL_HCB_GAMMA_FINE_DF_CW
codebook_length	81
cb_off	40

Table A.56: A-CPL Huffman codebook ACPL_HCB_GAMMA_COARSE_DT

Codebook name	ACPL_HCB_GAMMA_COARSE_DT
Codebook length table	ACPL_HCB_GAMMA_COARSE_DT_LEN
Codebook codeword table	ACPL_HCB_GAMMA_COARSE_DT_CW
codebook_length	41
cb_off	20

Table A.57: A-CPL Huffman codebook ACPL_HCB_GAMMA_FINE_DT

Codebook name	ACPL_HCB_GAMMA_FINE_DT
Codebook length table	ACPL_HCB_GAMMA_FINE_DT_LEN
Codebook codeword table	ACPL_HCB_GAMMA_FINE_DT_CW
codebook_length	81
cb_off	40

A.4 DE Huffman codebook tables

Table A.58: DE Huffman codebook DE_HCB_ABS_0

Codebook name	DE_HCB_ABS_0
Codebook length table	DE_HCB_ABS_0_LEN
Codebook codeword table	DE_HCB_ABS_0_CW
codebook_length	32
cb_off	0

Table A.59: DE Huffman codebook DE_HCB_DIFF_0

Codebook name	DE_HCB_DIFF_0
Codebook length table	DE_HCB_DIFF_0_LEN
Codebook codeword table	DE_HCB_DIFF_0_CW
codebook_length	63
cb_off	31

Table A.60: DE Huffman codebook DE_HCB_ABS_1

Codebook name	DE_HCB_ABS_1
Codebook length table	DE_HCB_ABS_1_LEN
Codebook codeword table	DE_HCB_ABS_1_CW
codebook_length	61
cb_off	30

Table A.61: DE Huffman codebook DE_HCB_DIFF_1

Codebook name	DE_HCB_DIFF_1
Codebook length table	DE_HCB_DIFF_1_LEN
Codebook codeword table	DE_HCB_DIFF_1_CW
codebook_length	121
cb_off	60

A.5 DRC Huffman codebook table

Table A.62: DRC Huffman codebook DRC_HCB

Codebook name	DRC_HCB
Codebook length table	DRC_HCB_LEN
Codebook codeword table	DRC_HCB_CW
codebook_length	255
cb_off	127

Annex B (normative): ASF scale factor band tables

Table B.1: Number of scale factor bands for 44,1 kHz or 48 kHz sampling frequency

transform length	2 048	1 920	1 536	1 024	960	768	512	480	384	256	240	192	128	120	96
num_sfb	63	61	55	49	49	43	36	36	33	20	20	18	14	14	12

Table B.2: Number of scale factor bands for 96 kHz sampling frequency

transform length	4 096	3 840	3 072	2 048	1 920	1 536	1 024	920	768	512	480	384	256	240	192
num_sfb	79	76	67	57	57	49	44	44	39	28	28	24	22	22	18

Table B.3: Number of scale factor bands for 192 kHz sampling frequency

transform length	8 192	7 680	6 144	4 096	3 840	3 072	2 048	1 920	1 536	1 024	960	768	512	480	384
num_sfb	111	106	91	73	72	61	60	59	51	36	36	30	30	30	24

Table B.4: Scale factor band offsets for sampling frequency 44,1 kHz or 48 kHz and transform length 2 048, 1 920 or 1 536 or for sampling frequency 96 kHz and transform length 4 096, 3 840 or 3 072 or for sampling frequency 192 kHz and transform length 8 192, 7 680 or 6 144

sfb	sfb_offset		
	2 048@44,1 2 048@48 4 096@96 8 192@192	1 920@48 3 840@96 7 680@192	1 536@48 3 072@96 6 144@192
0	0	0	0
1	4	4	4
2	8	8	8
3	12	12	12
4	16	16	16
5	20	20	20
6	24	24	24
7	28	28	28
8	32	32	32
9	36	36	36
10	40	40	40
11	44	44	44
12	52	52	52
13	60	60	60
14	68	68	68
15	76	76	76
16	84	84	84
17	92	92	92
18	100	100	100
19	108	108	108
20	116	116	116
21	124	124	124
22	136	136	136
23	148	148	148
24	160	160	160
25	172	172	172
26	188	188	188
27	204	204	204
28	220	220	220
29	240	240	240
30	260	260	260
31	284	284	284
32	308	308	308
33	336	336	336
34	364	364	364
35	396	396	396
36	432	432	432
37	468	468	468
38	508	508	508
39	552	552	552
40	600	600	600
41	652	652	652
42	704	704	704
43	768	768	768
44	832	832	832
45	896	896	896
46	960	960	960
47	1 024	1 024	1 024
48	1 088	1 088	1 088
49	1 152	1 152	1 152
50	1 216	1 216	1 216
51	1 280	1 280	1 280
52	1 344	1 344	1 344
53	1 408	1 408	1 408
54	1 472	1 472	1 472
55	1 536	1 536	1 536
56	1 600	1 600	1 664
57	1 664	1 664	1 792
58	1 728	1 728	1 920
59	1 792	1 792	2 048
60	1 856	1 856	2 176
61	1 920	1 920	2 304
62	1 984	2 048	2 432
63	2 048	2 176	2 560
64	2 176	2 304	2 688
65	2 304	2 432	2 816
66	2 432	2 560	2 944
67	2 560	2 688	3 072
68	2 688	2 816	3 200
69	2 816	2 944	3 328
70	2 944	3 072	3 456
71	3 072	3 200	3 584
72	3 200	3 328	3 712
73	3 328	3 456	3 840
74	3 456	3 584	3 968
75	3 584	3 712	4 096
76	3 712	3 840	4 224
77	3 840	3 968	4 352
78	3 968	4 096	4 480
79	4 096	4 224	4 608
80	4 224	4 352	4 736
81	4 352	4 480	4 864
82	4 480	4 608	4 992
83	4 608	4 736	5 120
84	4 736	4 864	5 248
85	4 864	4 992	5 376
86	4 992	5 120	5 504
87	5 120	5 248	5 632
88	5 248	5 376	5 760
89	5 376	5 504	5 888
90	5 504	5 632	6 016
91	5 632	5 760	6 144
92	5 760	5 888	
93	5 888	6 016	
94	6 016	6 144	
95	6 144	6 272	
96	6 272	6 400	
97	6 400	6 528	
98	6 528	6 656	
99	6 656	6 784	
100	6 784	6 912	
101	6 912	7 040	
102	7 040	7 168	
103	7 168	7 296	
104	7 296	7 424	
105	7 424	7 552	
106	7 552	7 680	
107	7 680		
108	7 808		
109	7 936		
110	8 064		
111	8 192		

Table B.5: Scale factor band offsets for sampling frequency 44,1 kHz or 48 kHz and transform length 1 024, 960 or 768 or for sampling frequency 96 kHz and transform length 2 048, 1 920 or 1 536 or for sampling frequency 192 kHz and transform length 4 096, 3 840 or 3 072

sfb	sfb_offset		
	1 024@44,1 1 024@48 2 048@96 4 096@192	960@48 1 920@96 3 840@192	768@48 1 536@96 3 072@192
0	0	0	0
1	4	4	4
2	8	8	8
3	12	12	12
4	16	16	16
5	20	20	20
6	24	24	24
7	28	28	28
8	32	32	32
9	36	36	36
10	40	40	40
11	48	48	48
12	56	56	56
13	64	64	64
14	72	72	72
15	80	80	80
16	88	88	88
17	96	96	96
18	108	108	108
19	120	120	120
20	132	132	132
21	144	144	144
22	160	160	160
23	176	176	176
24	196	196	196
25	216	216	216
26	240	240	240
27	264	264	264
28	292	292	292
29	320	320	320
30	352	352	352
31	384	384	384
32	416	416	416
33	448	448	448
34	480	480	480
35	512	512	512
36	544	544	544
37	576	576	576
38	608	608	608
39	640	640	640
40	672	672	672
41	704	704	704
42	736	736	736
43	768	768	768
44	800	800	896
45	832	832	1 024
46	864	864	1 152
47	896	896	1 280
48	928	928	1 408
49	1 024	960	1 536
50	1 152	1 024	1 664
51	1 280	1 152	1 792
52	1 408	1 280	1 920
53	1 536	1 408	2 048
54	1 664	1 536	2 176
55	1 792	1 664	2 304
56	1 920	1 792	2 432
57	2 048	1 920	2 560
58	2 176	2 048	2 688
59	2 304	2 176	2 816
60	2 432	2 304	2 944
61	2 560	2 432	3 072
62	2 688	2 560	
63	2 816	2 688	
64	2 944	2 816	
65	3 072	2 944	
66	3 200	3 072	
67	3 328	3 200	
68	3 456	3 328	
69	3 584	3 456	
70	3 712	3 584	
71	3 840	3 712	
72	3 968	3 840	
73	4 096		

Table B.6: Scale factor band offsets for sampling frequency 44,1 kHz or 48 kHz and transform length 512, 480 or 384 or for sampling frequency 96 kHz and transform length 1 024, 960 or 768 or for sampling frequency 192 kHz and transform length 2 048, 1 920 or 1 536

sfb	sfb_offset		
	512@44,1 512@48 1 024@96 2 048@192	480@48 960@96 1 920@192	384@48 768@96 1 536@192
0	0	0	0
1	4	4	4
2	8	8	8
3	12	12	12
4	16	16	16
5	20	20	20
6	24	24	24
7	28	28	28
8	32	32	32
9	36	36	36
10	40	40	40
11	44	44	44
12	48	48	48
13	52	52	52
14	56	56	56
15	60	60	60
16	68	68	68
17	76	76	76
18	84	84	84
19	92	92	92
20	100	100	100
21	112	112	112
22	124	124	124
23	136	136	136
24	148	148	148
25	164	164	164
26	184	184	184
27	208	208	208
28	236	236	236
29	268	268	268
30	300	300	300
31	332	332	332
32	364	364	364
33	396	396	384
34	428	428	448
35	460	460	512
36	512	480	576
37	576	512	640
38	640	576	704
39	704	640	768
40	768	704	832
41	832	768	896
42	896	832	960
43	960	896	1 024
44	1 024	960	1 088
45	1 088	1 024	1 152
46	1 152	1 088	1 216
47	1 216	1 152	1 280
48	1 280	1 216	1 344
49	1 344	1 280	1 408
50	1 408	1 344	1 472
51	1 472	1 408	1 536
52	1 536	1 472	
53	1 600	1 536	
54	1 664	1 600	
55	1 728	1 664	
56	1 792	1 728	
57	1 856	1 792	
58	1 920	1 856	
59	1 984	1 920	
60	2 048		

Table B.7: Scale factor band offsets for sampling frequency 44,1 kHz or 48 kHz and transform length 256, 240, 192, 128, 120 or 96 or for sampling frequency 96 kHz and transform length 512, 480, 384, 256, 240 or 192 or for sampling frequency 192 kHz and transform length 1 024, 960, 768, 512, 480 or 384

sfb	sfb_offset					
	256@44,1 256@48 512@96 1 024@192	240@48 480@96 960@192	192@48 384@96 768@192	128@44,1 128@48 256@96 512@192	120@48 240@96 480@192	96@48 192@96 384@192
0	0	0	0	0	0	0
1	4	4	4	4	4	4
2	8	8	8	8	8	8
3	12	12	12	12	12	12
4	16	16	16	16	16	16
5	20	20	20	20	20	20
6	24	24	24	28	28	28
7	28	28	28	36	36	36
8	36	36	36	44	44	44
9	44	44	44	56	56	56
10	52	52	52	68	68	68
11	64	64	64	80	80	80
12	76	76	76	96	96	96
13	92	92	92	112	112	112
14	108	108	108	128	120	128
15	128	128	128	144	128	144
16	148	148	148	160	144	160
17	172	172	172	176	160	176
18	196	196	192	192	176	192
19	224	224	224	208	192	224
20	256	240	256	224	208	256
21	288	256	288	240	224	288
22	320	288	320	256	240	320
23	352	320	352	288	256	352
24	384	352	384	320	288	384
25	416	384	448	352	320	
26	448	416	512	384	352	
27	480	448	576	416	384	
28	512	480	640	448	416	
29	576	512	704	480	448	
30	640	576	768	512	480	
31	704	640				
32	768	704				
33	832	768				
34	896	832				
35	960	896				
36	1 024	960				

Table B.8: Mapping from max_sfb_master from transform length 2 048 to different transform lengths

max_sfb_master [2 048]	n_sfb_side [1 024]	n_sfb_side [512]	n_sfb_side [256]	n_sfb_side [128]
0	0	0	0	0
1	1	1	1	1
2	1	1	1	1
3	2	1	1	1
4	2	1	1	1
5	3	2	1	1
6	3	2	1	1
7	4	2	1	1
8	4	2	1	1
9	5	3	2	1
10	5	3	2	1
11	6	3	2	1
12	7	4	2	1
13	8	4	2	1
14	9	5	3	2
15	10	5	3	2
16	11	6	3	2
17	11	6	3	2
18	12	7	4	2
19	12	7	4	2
20	13	8	4	2
21	13	8	4	2
22	14	9	5	3
23	15	10	5	3
24	15	10	5	3
25	16	11	6	3
26	17	12	6	3
27	18	13	7	4
28	19	14	7	4
29	19	15	8	4
30	20	16	8	5
31	21	17	8	5

Table B.9: Mapping from max_sfb_master from transform length 1 024 to different transform lengths

max_sfb_master [1 024]	n_sfb_side [512]	n_sfb_side [256]	n_sfb_side [128]
0	0	0	0
1	1	1	1
2	1	1	1
3	2	1	1
4	2	1	1
5	3	2	1
6	3	2	1
7	4	2	1
8	4	2	1
9	5	3	2
10	5	3	2
11	6	3	2
12	7	4	2
13	8	4	2
14	9	5	3
15	10	5	3
16	11	6	3
17	12	6	3
18	14	7	4
19	15	8	4
20	16	8	5
21	17	8	5
22	18	9	5
23	19	9	6
24	20	10	6
25	21	11	6
26	22	11	7
27	23	12	7
28	24	12	8
29	25	13	8
30	26	13	8
31	27	14	9

Table B.10: Mapping from max_sfb_master from transform length 512

max_sfb_master [512]	n_sfb_side [256]	n_sfb_side [128]
0	0	0
1	1	1
2	1	1
3	2	1
4	2	1
5	3	2
6	3	2
7	4	2
8	4	2
9	5	3
10	5	3
11	6	3
12	6	3
13	7	4
14	7	4
15	8	4
16	8	5
17	9	5
18	9	6
19	10	6
20	10	6
21	11	6
22	11	7
23	12	7
24	12	8
25	13	8
26	13	9
27	14	9
28	15	10
29	16	10
30	17	11
31	17	12

Table B.11: Mapping from max_sfb_master from transform length 256

max_sfb_master [256]	n_sfb_side [128]
0	0
1	1
2	1
3	2
4	2
5	3
6	3
7	4
8	5
9	6
10	6
11	7
12	8
13	9
14	9
15	10

Table B.12: Mapping from max_sfb_master from transform length 1 920 to different transform lengths

max_sfb_master [1 920]	n_sfb_side [960]	n_sfb_side [480]	n_sfb_side [240]	n_sfb_side [120]
0	0	0	0	0
1	1	1	1	1
2	1	1	1	1
3	2	1	1	1
4	2	1	1	1
5	3	2	1	1
6	3	2	1	1
7	4	2	1	1
8	4	2	1	1
9	5	3	2	1
10	5	3	2	1
11	6	3	2	1
12	7	4	2	1
13	8	4	2	1
14	9	5	3	2
15	10	5	3	2
16	11	6	3	2
17	11	6	3	2
18	12	7	4	2
19	12	7	4	2
20	13	8	4	2
21	13	8	4	2
22	14	9	5	3
23	15	10	5	3
24	15	10	5	3
25	16	11	6	3
26	17	12	6	3
27	18	13	7	4
28	19	14	7	4
29	19	15	8	4
30	20	16	8	5
31	21	17	8	5

Table B.13: Mapping from max_sfb_master from transform length 960 to different transform lengths

max_sfb_master [960]	n_sfb_side [480]	n_sfb_side [240]	n_sfb_side [120]
0	0	0	0
1	1	1	1
2	1	1	1
3	2	1	1
4	2	1	1
5	3	2	1
6	3	2	1
7	4	2	1
8	4	2	1
9	5	3	2
10	5	3	2
11	6	3	2
12	7	4	2
13	8	4	2
14	9	5	3
15	10	5	3
16	11	6	3
17	12	6	3
18	14	7	4
19	15	8	4
20	16	8	5
21	17	8	5
22	18	9	5
23	19	9	6
24	20	10	6
25	21	11	6
26	22	11	7
27	23	12	7
28	24	12	8
29	25	13	8
30	26	13	8
31	27	14	9

Table B.14: Mapping from max_sfb_master from transform length 480

max_sfb_master [480]	n_sfb_side [240]	n_sfb_side [120]
0	0	0
1	1	1
2	1	1
3	2	1
4	2	1
5	3	2
6	3	2
7	4	2
8	4	2
9	5	3
10	5	3
11	6	3
12	6	3
13	7	4
14	7	4
15	8	4
16	8	5
17	9	5
18	9	6
19	10	6
20	10	6
21	11	6
22	11	7
23	12	7
24	12	8
25	13	8
26	13	9
27	14	9
28	15	10
29	16	10
30	17	11
31	17	12

Table B.15: Mapping from max_sfb_master from transform length 240

max_sfb_master [240]	n_sfb_side [120]
0	0
1	1
2	1
3	2
4	2
5	3
6	3
7	4
8	5
9	6
10	6
11	7
12	8
13	9
14	9
15	10

Table B.16: Mapping from max_sfb_master from transform length 1 536 to different transform lengths

max_sfb_master [1 536]	n_sfb_side [768]	n_sfb_side [384]	n_sfb_side [192]	n_sfb_side [96]
0	0	0	0	0
1	1	1	1	1
2	1	1	1	1
3	2	1	1	1
4	2	1	1	1
5	3	2	1	1
6	3	2	1	1
7	4	2	1	1
8	4	2	1	1
9	5	3	2	1
10	5	3	2	1
11	6	3	2	1
12	7	4	2	1
13	8	4	2	1
14	9	5	3	2
15	10	5	3	2
16	11	6	3	2
17	11	6	3	2
18	12	7	4	2
19	12	7	4	2
20	13	8	4	2
21	13	8	4	2
22	14	9	5	3
23	15	10	5	3
24	15	10	5	3
25	16	11	6	3
26	17	12	6	3
27	18	13	7	4
28	19	14	7	4
29	19	15	8	4
30	20	16	8	5
31	21	17	8	5

Table B.17: Mapping from max_sfb_master from transform length 768 to different transform lengths

max_sfb_master [768]	n_sfb_side [384]	n_sfb_side [192]	n_sfb_side [96]
0	0	0	0
1	1	1	1
2	1	1	1
3	2	1	1
4	2	1	1
5	3	2	1
6	3	2	1
7	4	2	1
8	4	2	1
9	5	3	2
10	5	3	2
11	6	3	2
12	7	4	2
13	8	4	2
14	9	5	3
15	10	5	3
16	11	6	3
17	12	6	3
18	14	7	4
19	15	8	4
20	16	8	5
21	17	8	5
22	18	9	5
23	19	9	6
24	20	10	6
25	21	11	6
26	22	11	7
27	23	12	7
28	24	12	8
29	25	13	8
30	26	13	8
31	27	14	9

Table B.18: Mapping from max_sfb_master from transform length 384

max_sfb_master [384]	n_sfb_side [192]	n_sfb_side [96]
0	0	0
1	1	1
2	1	1
3	2	1
4	2	1
5	3	2
6	3	2
7	4	2
8	4	2
9	5	3
10	5	3
11	6	3
12	6	3
13	7	4
14	7	4
15	8	4

Table B.19: Mapping from max_sfb_master from transform length 192

max_sfb_master [192]	**n_sfb_side [96]**
0	0
1	1
2	1
3	2
4	2
5	3
6	3
7	4

Annex C (normative): Speech Spectral Frontend tables

C.1 SSF bandwidths

Table C.1 specifies the SSF bandwidths (number of spectral lines per band) for the different SSF block lengths.

Table C.1: SSF bandwidths

Band index	Number of bins							
	Block length 192	Block length 240	Block length 256	Block length 384	Block length 512	Block length 768	Block length 960	Block length 1 024
0	2	3	3	5	6	9	11	12
1	2	3	3	5	6	9	11	12
2	2	3	3	5	6	9	11	12
3	2	3	3	5	6	9	11	12
4	2	3	3	5	6	9	11	12
5	2	3	3	5	6	9	11	12
6	2	3	3	5	6	9	11	12
7	2	3	3	5	6	9	11	12
8	2	3	3	5	6	9	11	12
9	2	3	3	5	6	9	11	12
10	3	4	4	6	8	12	15	16
11	3	4	4	6	8	12	15	16
12	3	4	4	6	8	12	15	16
13	4	5	5	8	10	15	19	20
14	4	5	5	8	10	15	19	20
15	5	6	6	9	12	18	23	24
16	5	7	7	11	14	21	26	28
17	5	7	7	11	14	21	26	28
18	6	8	8	12	16	24	30	32

C.2 POST_GAIN_LUT

Table C.2: POST_GAIN_LUT

Table name	POST_GAIN_LUT
table_length	20

C.3 PRED_GAIN_QUANT_TAB

Table C.3: PRED_GAIN_QUANT_TAB[ ]

Table name	PRED_GAIN_QUANT_TAB
table_length	32

C.4 PRED_RFS_TABLE

Table C.4: PRED_RFS_TABLE[ ]

Table name	PRED_RFS_TABLE
table_length	37

C.5 PRED_RTS_TABLE

Table C.5: PRED_RTS_TABLE[ ]

Table name	PRED_RTS_TABLE
table_length	37

C.6 Quantized prediction coefficients

The quantized prediction coefficients are stored in 37 tables. These tables are available in the accompanying file ts_103190_tables.c and are named ssf_pred_coeff_mat0[ ], …, ssf_pred_coeff_mat36[ ]. The mapping to the table PRED_COEFF_QUANT_MAT[tab_idx][ν][η][k], which is used in clause 5.2.8.1, is given by the following pseudocode:

Pseudocode

```
// Mapping of ssf_pred_coeff_mat<tab_idx> to PRED_COEFF_QUANT_MAT[tab_idx][ν][η][k]

rfs = PRED_RFS_TABLE[tab_idx];
rts = PRED_RTS_TABLE[tab_idx];

table_index = (ν + rfs)*rts*33 + k*33 + η;

PRED_COEFF_QUANT_MAT[tab_idx][ν][η][k] = ssf_pred_coeff_mat<tab_idx>[table_index];
```

C.7 CDF_TABLE

Table C.6: CDF_TABLE

Table name	CDF_TABLE
table_length	705

C.8 PREDICTOR_GAIN_CDF_LUT

Table C.7: PREDICTOR_GAIN_CDF_LUT

Table name	PREDICTOR_GAIN_CDF_LUT
table_length	33

C.9 ENVELOPE_CDF_LUT

Table C.8: ENVELOPE_CDF_LUT

Table name	ENVELOPE_CDF_LUT
table_length	33

C.10 DITHER_TABLE

Table C.9: DITHER_TABLE

Table name	DITHER_TABLE
table_length	256

C.11 RANDOM_NOISE_TABLE

Table C.10: RANDOM_NOISE_TABLE

Table name	RANDOM_NOISE_TABLE
table_length	256

C.12 STEP_SIZES_Q4_15

Table C.11: STEP_SIZES_Q4_15

Table name	STEP_SIZES_Q4_15
table_length	21

C.13 AC_COEFF_MAX_INDEX

Table C.12: AC_COEFF_MAX_INDEX

Table name	AC_COEFF_MAX_INDEX
table_length	21

C.14 dB conversion tables

Table C.13: SLOPES_DB_TO_LIN

Table name	SLOPES_DB_TO_LIN
table_length	10

Table C.14: OFFSETS_DB_TO_LIN

Table name	OFFSETS_DB_TO_LIN
table_length	10

Table C.15: SLOPES_LIN_TO_DB

Table name	SLOPES_LIN_TO_DB
table_length	50

Table C.16: OFFSETS_LIN_TO_DB

Table name	OFFSETS_LIN_TO_DB
table_length	50

Annex D (normative): Other tables

D.1 Channel names

Table D.1 describes channel abbreviations used in the present document.

Table D.1: Channel abbreviations

Channel name	Abbreviation
Left	L
Center	C
Right	R
Left Surround	Ls
Right Surround	Rs
Left Center	Lc
Right Center	Rc
Left Rear Surround	Lrs
Right Rear Surround	Rrs
Center Surround	Cs
Top Surround	Ts
Left Surround Direct	Lsd
Right Surround Direct	Rsd
Left Wide	Lw
Right Wide	Rw
Left Vertical Height	Vhl
Right Vertical Height	Vhr
Center Vertical Height	Vhc
Low-Frequency Effects	LFE
Low-Frequency Effects 2	LFE2

D.2 A-SPX noise table

The A-SPX noise table referenced in table D.2 is available in the accompanying file ts_103190_tables.c. It consists of *num_rows* = 512 complex values, real and complex values distributed to the *num_columns* = 2 columns.

Table D.2: A-SPX noise table

Table name	ASPX_NOISE
num_columns	2
num_rows	512

D.3 QMF filter coefficients

The QMF window coefficients referenced in table D.3 are available in the accompanying file ts_103190_tables.c.

Table D.3: QWIN

Table name	QWIN
table_length	640

Annex E (normative): AC-4 Bitstream Storage in the ISO Base Media File Format

Scope

This annex defines the necessary structures for the integration of AC-4 coded bitstreams in a file format that is compliant with the ISO Base Media File Format [2]. Examples of file formats that are derived from the ISO Base Media File Format include the MP4 file format and the 3GPP file format.

This annex additionally covers:

- the steps required to properly packetize an AC-4 bitstream for multiplexing and storage in an MPEG-DASH-compliant ISO base media file format file; and
- the steps required to demultiplex an AC-4 bitstream from an MPEG DASH-compliant ISO base media file format file.

E.1 AC-4 Track definition

In the terminology of the ISO Base Media File Format (ISOBMFF) specification (ISO/IEC 14496-12 [2]), AC-4 tracks are audio tracks. It therefore follows that these rules apply to the media box in the AC-4 tracks:

- In the Handler Reference Box, the handler_type field shall be set to "soun".
- The Media Information Header Box shall contain a Sound Media Header Box.
- The Sample Description Box shall contain a box derived from AudioSampleEntry. This box is called AC4SampleEntry and is defined in clause E.3.

The value of the timescale parameter in the Media Header Box depends on *frame_rate* and *base_samp_freq*. The time scale shall be set according to table E.1.

NOTE: For the definition of samples, see clause E.2.

The Sample Table Box ('stbl') of an AC-4 audio track shall contain a Sync Sample Box ('stss'), unless all samples are sync samples. The Sync Sample Box shall reference all sync samples part of that track. The first AC-4 sample in every chunk shall be a sync sample. The `sequence_counter` of the first sample should be set to zero.

Table E.1: Timescale for Media Header Box

base_samp_freq **[kHz]**	**frame_rate_index**	***frame_rate*** **[fps]**	**Media Time Scale [1/sec]**	**'sample_delta' [units of media time scale]**
48	0	23,976	48 000	2 002
	1	24	48 000	2 000
	2	25	48 000	1 920
	3	29,97	240 000	8 008
	4	30	48 000	1 600
	5	47,95	48 000	1 001
	6	48	48 000	1 000
	7	50	48 000	960
	8	59,94	240 000	4 004
	9	60	48 000	800
	10	100	48 000	480
	11	119,88	240 000	2 002
	12	120	48 000	400
	13	(23,44)	48 000	2 048
	14	reserved		
	15	reserved		
44,1	0...12	reserved		
	13	(21,53)	44 100	2 048
	14, 15	reserved		

E.2 AC-4 Sample definition

For the purpose of carrying AC-4 in ISOBMFF, an AC-4 Sample corresponds to one `raw_ac4_frame`, as defined in clause 4.2.1.

NOTE: All samples in one substream share the same duration and timestamps.

Sync samples are defined as samples that have the `b_iframe_global` flag set in the `ac4_toc`.

E.3 AC4SampleEntry Box

The box type of the AC4SampleEntry Box shall be 'ac-4'.

The AC4SampleEntry Box is defined by table E.2. Box entry values shall be set according to the values given in the table, except where left empty.

Table E.2: AC4SampleEntry Box definition

Syntax	**No. of bits**	**Value**
`AC4SampleEntry()`		
`{`		
`    BoxHeader.Size;`	32	Note 3
`    BoxHeader.Type;`	32	'ac-4'
`    Reserved[6];`	8	0
`    DataReferenceIndex;`	16	Note 3
`    Reserved[2];`	32	0
`    ChannelCount;`	16	Note 1, 2
`    SampleSize;`	16	16
`    Reserved;`	32	0
`    SamplingFrequency;`	16	Note 2
`    Reserved;`	16	0
`    Ac4SpecificBox();`		
`}`		

NOTE 1: The ChannelCount field should be set to the total number of audio output channels of the default presentation of that track, if not defined differently by an application standard.
NOTE 2: The values of the ChannelCount and SamplingFrequency fields within the AC4SampleEntry Box shall be ignored on decoding.
NOTE 3: The values shall be set according to the sampleEntry definition in [i.2].

The layout of the AC4SampleEntry box is identical to that of AudioSampleEntry defined in ISO/IEC 14496-12 [2] (including the reserved fields and their values), except that AC4SampleEntry ends with a box containing AC-4 bitstream information called AC4SpecificBox. The AC4SpecificBox field structure for AC-4 is defined in clause E.4.

E.4 AC4SpecificBox

The AC4SpecificBox is defined table E.3. Box entry values shall be set according to the values given in the table, except where left empty.

Table E.3: AC4SpecificBox definition

Syntax	No. of bits	Value
`AC4SpecificBox()`		
`{`		
`    BoxHeader.Size;`	32	
`    BoxHeader.Type;`	32	'dac4'
`    ac4_dsi();`		
`}`		

The AC4SpecificBox() shall contain the 'ac4_dsi()' as specified in table E.4.

Table E.4: AC-4 decoder specific information

Syntax	No. of bits
ac4_dsi()	
{	
ac4_dsi_version	3
bitstream_version	7
fs_index	1
frame_rate_index	4
n_presentations	9
for (i=0; i<n_presentations; i++) {	
b_single_substream	1
presentation_config	5
presentation_version	5
if (b_single_substream !=1 && presentation_config ==6) {	
b_add_umd_substreams=1	
} else {	
b_mdcompat	1
if (**b_belongs_to_presentation_group**) {	1
presentation_group	5
}	
dsi_**frame_rate_multiply_info**	2
umd_version	5
key_id	10
if (b_single_substream==1) {	
ac4_substream_dsi()	
} else {	
b_hsf_ext	1
switch(presentation_config) {	
case 0:	
case 1:	
case 2:	
ac4_substream_dsi()	
ac4_substream_dsi()	
break;	
case 3:	
case 4:	
ac4_substream_dsi()	
ac4_substream_dsi()	
ac4_substream_dsi()	
break;	
case 5:	
ac4_substream_dsi()	
break;	
default:	
n_skip_bytes	7
skip_bits	**n_skip_bytes * 8**
break;	
}	
b_pre_virtualized	1
b_add_umd_substreams	1
}	
}	
if (b_add_umd_substreams)	
n_add_umd_substreams	7
for (j = 0; j < n_add_umd_substreams; j++) {	
umd_version	5
key_id	10
}	
}	
byte_align	0...7
}	
}	
NOTE: The number of bits in byte_align shall pad the number of bits, counted from the start of ac4_dsi, to an integer number of bytes.	

Table E.5: AC-4 substream decoder specific information

Syntax	No. of bits
`ac4_substream_dsi()`	
`{`	
`channel_mode`	5
`dsi_sf_multiplier`	2
`if (`**`b_bitrate_indicator`**`) {`	1
`bitrate_indicator`	5
`}`	
`if (ch_mode == [7...10]) {`	
`add_ch_base`	1
`}`	
`if (`**`b_content_type`**`) {`	1
`content_classifier`	3
`if (`**`b_language_indicator`**`) {`	1
`n_language_tag_bytes`	6
`for (I = 0; I < n_language_tag_bytes; i++) {`	
`language_tag_bytes`	8
`}`	
`}`	
`}`	
`}`	

Semantics:

ac4_dsi_version - 3 bits

This field indicates the version of the DSI. For a DSI that conforms to the present document, the `ac4_dsi_version` field shall be set to '000'.

bitstream_version - 7 bits

This field shall contain the bitstream version as described in clause 4.3.3.2.1. Its value shall be the same as read from the `ac4_toc`.

fs_index - 1 bit

This field shall contain the sampling frequency index as described in clause 4.3.3.2.5. Its value shall be Its value shall be the same as read from the `ac4_toc`.

frame_rate_index - 4 bits

This field shall contain the frame rate index as described in clause 4.3.3.2.6. Its value shall be the same as read from the `ac4_toc`.

n_presentations - 9 bits

This field shall contain the number of presentations contained in the corresponding ac-4 frame. Its value shall be the same as read from the `ac4_toc`.

b_single_substream - 1 bit

This bit indicates that the presentation contains a single substream. Its value shall be the same as the respective value from the respective `ac4_presentation_info`.

presentation_config - 5 bits

This field shall contain the presentation config as described in clause 4.3.3.3.4. Its value shall be the same as the respective value read from the respective `ac4_presentation_info`.

presentation_version - 5 bits

This field shall contain the presentation version as described in clause 4.3.3.4. Its value shall be the same as the respective value read from the respective `ac4_presentation_info`.

b_belongs_to_presentation_group - 1 bit

This bit indicates that the containing presentation belongs to a group of presentations. Its value shall be the same as the respective value from the respective `ac4_presentation_info`.

presentation_group - 5 bits

This field shall contain a presentation group. Its value shall be the same as the respective value read from the respective `ac4_presentation_info`.

dsi_frame_rate_multiply_info - 2 bits

This field shall signal the `frame_rate_multiply_info` as described in clause 4.3.3.5. Its value shall correspond to the respective value read from the respective `ac4_presentation_inf` as follows:

frame_rate_index	b_multiplier	multiplier_bit	dsi_frame_rate_multiply_info
2, 3, 4	0	X	00
	1	0	01
	1	1	10
0, 1, 7, 8, 9	0	X	00
	1	X	01
5, 6, 10, 11, 12, 13	X	X	00

umd_version - 5 bits

This field shall contain the umd syntax version as described in clause 4.3.3.6.1. Its value shall be the same as the respective value read from the `umd_info` field in the respective `ac4_presentation_info`.

key_id - 10 bits

This field shall contain the cryptographic key ID as described in clause 4.3.3.6.2. Its value shall be the same as the respective value read from the `umd_info` field in the respective `ac4_presentation_info`.

b_hsf_ext - 1 bit

This bit shall indicate the availability of spectral data for high sampling frequencies as described in clause 4.3.3.3.3. Its value shall be the same as the respective value read from the respective `ac4_presentation_info`.

n_skip_bytes - 7 bits

This field indicates a number of subsequent bytes to skip.

skip_bits

This field indicates n_skip_bytes × 8 bits to skip.

b_pre_virtualized - 1 bit

This bit indicates pre-rendering as described in clause 4.3.3.3.5. Its value shall be the same as the respective value read from the respective `ac4_presentation_info`.

b_add_umd_substreams - 1 bit

This bit indicates presence of additional umd containers as described in clause 4.3.3.3.6. Its value shall be the same as the respective value read from the respective `ac4_presentation_info`.

n_add_umd_substreams - 7 bits

This field indicates presence of additional umd containers as described in clause 4.3.3.3.7. Its value shall be the same as the respective value read from the respective `ac4_presentation_info`.

umd_version - 5 bits

This field shall contain the umd syntax version as described in clause 4.3.3.6.1. Its value shall be the same as the respective value read from the `umd_info` field in the respective `n_add_umd_substreams`() loop in the respective `ac4_presentation_info`.

key_id - 10 bits

This field shall contain the cryptographic key ID as described in clause 4.3.3.6.2. Its value shall be the same as the respective value read from the `umd_info` field in the respective `n_add_umd_substreams`() loop in the respective `ac4_presentation_info`.

byte_align – 0 to 7 bits

This bit is used for the byte alignment of each presentation within the `ac4_dsi` element. Byte alignment is defined relative to the start of the enclosing syntactic element.

channel_mode - 5 bits

This field shall contain the channel mode as described in clause 4.3.3.7.1. Its value shall correspond to the respective value read from the respective ac4_substream_info in the respective `ac4_presentation_info` and is expressed either through the ch_mode parameter from table 87 (if the channel_mode bitfield is not 1111111) or through the value "12+variable_bits(2)" (if the channel_mode bitfield is 1111111).

dsi_sf_multiplier - 2 bits

This field shall signal the sf_multiplier as described in clause 4.3.3.7.3. Its value shall correspond to the respective value read from the respective `ac4_substream_info` in the respective `ac4_presentation_info` as follows:

sampling frequency	b_sf_multiplier	sf_multiplier	dsi_sf_multiplier
48 kHz	0		00
96 kHz	1	0	01
192 kHz		1	10

b_bitrate_indicator - 1 bit

This bit indicates presence of the bitrate indicator as described in 4.3.3.7.5.

bitrate_indicator - 5 bits

This field shall contain a bitrate indication as described in 4.3.3.7.5. The value shall correspond to the respective value read from the respective `ac4_substream_info` in the respective `ac4_presentation_info` and is expressed through the `brate_ind` parameter from table 89.

add_ch_base - 1 bit

This bit shall contain the Additional Channels Coupling base as described in 4.3.3.7.6. This field is present only if the ch_mode value according to table 87 is in the range [7...10].

b_content_type - 1 bit

This bit indicates the presence of content_type information as described in clause 4.3.3.7.7.

content_classifier - 3 bits

This field shall contain the content classifier as described in 4.3.3.8.1. The value shall correspond to the respective value read from the respective `content_type` field in the respective `ac4_substream_info` in the respective `ac4_presentation_info`.

b_language_indicator - 1 bit

This bit indicates presence of programme language indication as described in clause 4.3.3.8.2.

n_language_tag_bytes - 6 bits

This field shall contain the number of subsequent language tags bytes as described in clause 4.3.3.8.6.

language_tag_bytes - 8 bits

The sequence of `langage_tag_bytes` shall contain a language tag as described in clause 4.3.3.8.7. For the respective `ac4_substream_info` in the respective `ac4_presentation_info`, these values shall correspond:

- to the values of the respective `language_tag_bytes` values in the `content_type` field of the respective `ac4_substream_info` in the respective `ac4_presentation_info`, if `b_serialized_language_tag` is false;
- to the concatenation of `language_tag_chunk` fields in the `content_type` field of the respective ac4_substream_info in the respective ac4_presentation_info from consecutive frames, if `b_serialized_language_tag` is true.

The `ac4_dsi` shall not be used to configure the AC-4 decoder. The AC-4 decoder shall obtain its configuration only from the `ac4_toc` that is part of every sample.

E.5 AC-4 audio tracks in fragmented isomedia files

The first AC-4 sample in a track fragment run shall be a sync sample. Therefore, the first AC-4 sample in every Movie Fragment will be a sync sample.

The Track Fragment Header Box ('tfhd') should set 'default-sample-duration-present' flag and provide the (constant) sample duration, as given in the sample_duration field of table E.1.

The Track Fragment Run Box ('trun') shall set either 'first-sample-flags-present' or 'sample-flags-present', unless all samples in the Track Fragment Run Box are sync samples.

The 'sample_flags' of the Track Fragment Run Box ('trun') shall set the 'sample_is_non_sync_sample' bit to 0 for all sync samples, and 1 otherwise.

Annex F (normative): AC-4 Transport in MPEG-DASH

This annex describes the requirements and recommendations for delivering AC-4 streams using the MPEG Dynamic Adaptive Streaming over HTTP (DASH) standard in conjunction with the ISO base media file format, specifically referencing AC-4 audio streams within the MPEG-DASH media presentation description (MPD) file.

F.1 Media Presentation Description (MPD)

F.1.1 Overview

The MPD is an XML document. The DASH client uses the information in the MPD for constructing the HTTP URLs that then allow it to access segments containing the actual audio and video content.

F.1.2 General MPD requirements relating to AC-4

Although the syntax of the MPD is capable of using common XML elements to describe almost any media format, the encoding type and the configuration of an AC-4 elementary stream that is part of a content presentation constrains the parameter values of some of these elements. This clause defines the values that enable an MPD to properly describe an AC-4 elementary stream.

The MPD formats described here support the following scenarios:

- Media presentations that consist of a single AC-4 elementary stream.
- Media presentations that consist of multiple AC-4 elementary streams, with each elementary stream stored in a separate MP4 file or segment file.

It is possible for the MPD to describe multiple audio services delivered using one or more multiple AC-4 elementary streams (for example, a main audio service and an associated audio service that are intended to be decoded and then mixed together).

F.1.2.1 Adaptation sets

An adaptation set describes the overall media presentation. The adaptation set typically consists of multiple instances (representations) of the same audio, video, or audio/video content, with each instance encoded at a different data rate. A representation describes the parameters of each individual encoding of an adaptation set, as follows:

- The *codecs* attribute is required. It specifies the codecs used to encode all representations within the adaptation set. For AC-4 elementary streams, the *value* element of the *codecs* attribute shall include the entry ac-4.
- The *mimeType* attribute describes the encapsulation format used to store the AC-4 elementary streams present in the adaptation set. For adaptation sets that conform to ISO/IEC 14496-12 [2], the *type* element of the *mimeType* attribute shall be set to one of the following values:
  - audio/mp4 (for ISO base media files that contain an AC-4 audio track but no accompanying video track);
  - video/mp4 (for ISO base media files that contain AC-4 audio tracks and one or more video tracks).

In some applications, multiple AC-4 elementary streams may be used to simultaneously deliver different audio elements of the overall media presentation. For example, one elementary stream carries a main audio service (the main audio), and a second elementary stream carries an associated audio service (such as commentary) intended to be mixed with the main audio service before presentation to the listener.

If the content provider wishes to enable user-defined selection of specific combinations of elementary streams in the playback device (allowing different renditions of the overall media presentation to be selected and delivered), separate adaptation sets may be defined for each elementary stream. For example, one adaptation set is used to describe the main audio service on its own, and a second adaptation set describes the associated audio service that will be simultaneously delivered with the main audio service to the playback device, where both adaptation sets will be decoded and mixed together. Refer to clause F.1.3 for more details.

F.1.2.2 Representations

Each adaptation set carries one or more representations. All representations in an adaptation set shall be perceptually identical, meaning that the bit rate is the only major parameter that may differ across the AC-4 elementary streams in one adaptation set.

F.1.2.3 AudioChannelConfiguration descriptor

The representation element shall include an AudioChannelConfiguration descriptor, which unambiguously describes the channel configuration of the referenced AC-4 elementary stream. Refer to clause F.1.4.1 for details.

F.1.2.4 Accessibility descriptor

If the adaptation set provides for enhanced accessibility, the AdaptationSet may include an accessibility descriptor that describes the type of accessible audio service being provided. The required attribute *schemeIdUri* should be set to urn:tva:metadata:cs:AudioPurposeCS:2007, as defined in clause B.1 of [3], signaling the namespace for the accessibility descriptor.

The audio purpose classification scheme (AudioPurposeCS), which is used to describe the type of accessible audio service that is being delivered, is defined in clause A.15 of [3]. The value of the *termID* attribute should be set to match the type of accessible audio service carried in the AC-4 elementary stream, which is indicated by the value of the `content_classifier` parameter in the default presentation of the AC-4 stream, or in the AC4SpecificBox of the AC-4 audio track. The corresponding values of the termID attribute and `content_classifier` parameter are listed in table F.1.

Table F.1: Corresponding termID Attribute and content_classifier Parameter Values

termID Attribute Value	AudioPurposeCS Name	content_classifier Parameter Value
1	Audio description for the visually impaired	010
2	Audio description for the hearing impaired	011
3	Supplemental commentary	101
4	Director's commentary	101
5	Educational notes	101
6	Main programme audio	000
7	Clean feed (no effects mix)	100

F.1.3 MPD with associated audio services using AC-4

It is useful in some scenarios to simultaneously deliver two audio services (one main and one associated) to a decoder. This can be achieved, as follows:

- The main audio service is a self-contained presentation that can be decoded on its own.
- The associated audio service contains supplementary audio programme elements intended to be decoded and mixed with the main audio service (for example, a director's commentary or a description of the programme for a visually impaired listener).

The main and associated audio streams are stored in separate ISO base media files that are described in separate adaptation sets within the MPD. The *accessibility* and *role* descriptors describe the purpose of the audio streams.

F.1.3.1 Role descriptor

As defined in the MPEG-DASH role scheme (urn:mpeg:dash:role:2011), the value attribute of the role descriptor shall be set to describe the purpose of each adaptation set in the overall presentation, as follows:

- If the adaptation set is delivering a full audio service intended for direct presentation to the listener, the value attribute shall be main or alternate.
- If the adaptation set is delivering an audio service intended to be mixed with a full audio service delivered in a different adaptation set before presentation to the listener (sometimes referred to as a receiver-mix service), the value attribute shall be set to commentary.
- If the adaptation set is delivering a full audio service intended for direct presentation to the listener, but this audio service is intended as an alternative presentation to the main audio service (for example, when delivering a service that contains premixed main audio and audio elements for visually impaired listeners, sometimes referred to as a broadcast-mix service), the value attribute shall be set to alternate.

F.1.3.2 dependencyID

An adaptation set that is delivering an associated audio service shall not be decoded and presented to the listener on its own, but shall always be mixed with the decoded audio from the adaptation set that is delivering the corresponding main audio service. Therefore, the adaptation set that is delivering the associated audio service should include a *dependencyID* descriptor. This descriptor indicates the relationship of the associated audio service with the main audio service that it will be mixed with after decoding.

F.1.4 Descriptors specific to AC-4 elementary streams

The following descriptors are specific to AC-4 elementary streams.

F.1.4.1 AudioChannelConfiguration descriptor

For AC-4 elementary streams, the audio channel configuration descriptor shall use the AudioChannelConfiguration scheme described in the schemeIdUri urn:dolby:dash:audio_channel_configuration:2011.

The *value* element, if used, shall contain a four-digit hexadecimal representation of the 16-bit bit field, which describes the channel assignment of the referenced AC-4 elementary stream according to table F.2.

Table F.2: AudioChannelConfiguration Descriptor

Bit	Speaker location
0 (MSB)	L
1	C
2	R
3	Ls
4	Rs
5	Lc/Rc pair
6	Lrs/Rrs pair
7	Cs
8	Ts
9	Lsd/Rsd pair
10	Lw/Rw pair
11	Vhl/Vhr pair
12	Vhc
13	LFE2
14	LFE
15	reserved

NOTE 1: Bit 0, which indicates the presence of the L channel, is the MSB of the AudioChannelConfiguration descriptor. For example, to indicate that the channel configuration of the AC-4 elementary stream is L, C, R, Ls, Rs, LFE, the value element would contain the value F801 (the hexadecimal equivalent of the binary value 1111 1000 0000 0001).
NOTE 2: Please see table D.1 for an explanation of speaker location acronyms.

F.1.5 MPD manifest file examples

This clause contains example MPD files for different media presentations.

F.1.5.1 MPD for a single video component and single audio component

The following MPD example describes a simple media presentation that consists of a single video component with a single 5.1-channel (L, C, R, Ls, Rs, LFE) AC-4 audio component. Three representations of the video content and three representations of the audio content are provided, each at a different data rate.

The media presentation complies with the ISO base media file format live profile, as defined in [1].

```xml
<?xml version="1.0" encoding="utf-8"?>
<MPD xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:dolby="http://www.dolby.com/ns/online/DASH" xmlns="urn:mpeg:DASH:schema:MPD:2011"
xsi:schemaLocation="urn:mpeg:DASH:schema:MPD:2011"
type="static"
minimumUpdatePeriod="PT2S"
timeShiftBufferDepth="PT30M"
availabilityStartTime="2011-12-25T12:30:01"
minBufferTime="PT4S"
profiles="urn:mpeg:dash:profile:isoff-live:2011">
  <BaseURL>http://cdn1.example.com/</BaseURL>
  <BaseURL>http://cdn2.example.com/</BaseURL>
  <Period>
    <!-- Video -->
    <AdaptationSet mimeType="video/mp4" codecs="avc1.4D401F" frameRate="30000/1001"
    segmentAlignment="true" startWithSAP="1">
      <BaseURL>video/</BaseURL>
      <SegmentTemplate timescale="90000" media="$Bandwidth$/$Index$.m4s"
      initialization="$Bandwidth$/0.mp4">
        <SegmentTimeline>
          <S t="0" d="180180" r="10"/>
        </SegmentTimeline>
      </SegmentTemplate>
      <Representation id="v0" width="320" height="240" bandwidth="250000" />
      <Representation id="v1" width="640" height="480" bandwidth="500000" />
      <Representation id="v2" width="960" height="720" bandwidth="1000000" />
    </AdaptationSet>
    <!-- 5.1 channel English Audio -->
    <AdaptationSet mimeType="audio/mp4" codecs="ac-4" lang="en"
    segmentAlignment="true" startWithSAP="1">
      <SegmentTemplate timescale="48000" media="audio/en/$Bandwidth$/$Index$.m4s"
      initialization="audio/en/$Bandwidth$/0.mp4">
        <SegmentTimeline>
          <S t="0" d="96768" r="10"/>
        </SegmentTimeline>
      </SegmentTemplate>
      <AudioChannelConfiguration
schemeIdUri="urn:dolby:dash:audio_channel_configuration:2011" value="F801"/>
      <Representation id="a0" bandwidth="192000" />
      <Representation id="a1" bandwidth="256000" />
      <Representation id="a2" bandwidth="384000" />
    </AdaptationSet>
  </Period>
</MPD>
```

F.1.5.2 MPD for main and associated audio services delivered in separate files

This is a simple example of a dynamic presentation, with multiple languages and multiple base URLs. The following MPD document describes content available from two sources (cdn1 and cdn2) with audio available in two different English language presentations: main audio service only, or a visually impaired receiver-mix service. The visually impaired service is enabled by simultaneously delivering the AC-4 bitstream containing the main audio service and an additional AC-4 bitstream containing the associated audio service for visually impaired listeners.

Three versions of the video are provided at bit rates between 250 kbps and 1 Mbps in different spatial resolutions.

The media presentation complies with the ISO base media file format live profile, as defined in [1].

```
<?xml version="1.0" encoding="utf-8"?>
<MPD xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:dolby="http://www.dolby.com/ns/online/DASH" xmlns="urn:mpeg:DASH:schema:MPD:2011"
xsi:schemaLocation="urn:mpeg:DASH:schema:MPD:2011"
type="dynamic"
minimumUpdatePeriod="PT2S"
timeShiftBufferDepth="PT30M"
availabilityStartTime="2011-12-25T12:30:00"
minBufferTime="PT4S"
profiles="urn:mpeg:dash:profile:isoff-live:2011">
  <BaseURL>http://cdn1.example.com/</BaseURL>
  <BaseURL>http://cdn2.example.com/</BaseURL>
  <Period>
    <!-- Video -->
    <AdaptationSet mimeType="video/mp4" codecs="avc1.4D401F" frameRate="30000/1001"
    segmentAlignment="true" startWithSAP="1">
      <BaseURL>video/</BaseURL>
      <SegmentTemplate timescale="90000" media="$Bandwidth$/$Index$.m4s"
      initialization="$Bandwidth$/0.mp4">
        <SegmentTimeline>
          <S t="0" d="180180" r="12"/>
        </SegmentTimeline>
      </SegmentTemplate>
      <Representation id="v0" width="320" height="240" bandwidth="250000" />
      <Representation id="v1" width="640" height="480" bandwidth="500000" />
      <Representation id="v2" width="960" height="720" bandwidth="1000000" />
    </AdaptationSet>
    <!-- English Audio -->
    <AdaptationSet mimeType="audio/mp4" codecs="ac-4" lang="en"
    segmentAlignment="0" startWithSAP="1">
       <Role schemeIdUri="urn:mpeg:dash:role:2011" value="main" />
       <SegmentTemplate timescale="48000" media="audio/en_main/$Bandwidth$/$Index$.m4s"
       initialization="audio/en_main/$Bandwidth$/0.mp4">
          <SegmentTimeline>
             <S t="0" d="96000" r="11"/>
          </SegmentTimeline>
       </SegmentTemplate>
       <Representation id="a0" bandwidth="256000">
          <AudioChannelConfiguration
schemeIdUri="urn:dolby:audio_channel_configuration:2011" value="F801"/>
       </Representation>
    </AdaptationSet>
    <!-- English Audio for visually impaired listeners -->
    <AdaptationSet mimeType="audio/mp4" codecs="ac-4" lang="en"
    segmentAlignment="true" startWithSAP="1">
       <Accessibility schemeIdUri="urn:tva:metadata:cs:AudioPurposeCS:2007" value="1"/>
       <Role schemeIdUri="urn:mpeg:dash:role:2011" value="commentary" />
       <SegmentTemplate timescale="48000" media="audio/en_vi/$Bandwidth$/$Index$.m4s"
       initialization="audio/en_vi/$Bandwidth$/0.mp4">
          <SegmentTimeline>
             <S t="0" d="96000" r="11"/>
          </SegmentTimeline>
       </SegmentTemplate>
       <Representation id="a1" dependencyId="a0" bandwidth="64000">
          <AudioChannelConfiguration
schemeIdUri="urn:dolby:audio_channel_configuration:2011" value="4000"/>
       </Representation>
    </AdaptationSet>
    <!-- French Audio -->
    <AdaptationSet mimeType="audio/mp4" codecs="ac-4" lang="fr" segmentAlignment="0"
startWithSAP="1">
       <SegmentTemplate timescale="48000" media="audio/fr/$Bandwidth$/$Index$.m4s"
       initialization="audio/fr/$Bandwidth$/0.mp4">
         <SegmentTimeline>
             <S t="0" d="96000" r="11"/>
          </SegmentTimeline>
       </SegmentTemplate>
       <Representation id="a2" bandwidth="192000">
          <AudioChannelConfiguration
schemeIdUri="urn:dolby:audio_channel_configuration:2011" value="F801"/>
       </Representation>
    </AdaptationSet>
  </Period>
</MPD>
```

Annex G (informative): Bibliography

- Charles Q. Robinson, Kenneth Gundry: "Dynamic Range Control via Metadata", Preprint 5028, 107th AES Convention New York, Sept. 1999.
- Davidson, G. A.: "The Digital Signal Processing Handbook", Madisetti, V. K. and Williams, D. B. Eds. (CRC Press LLC, 1997), pp. 41-1 - 41-21.
- Princen J., Bradley A.: "Analysis/synthesis filter bank design based on time domain aliasing cancellation", IEEE Trans. Acoust. Speech and Signal Processing, vol. ASSP-34, pp. 1153-1161, Oct. 1986.
- R. Rao, P. Yip: "Discrete Cosine Transform", Academic Press, Boston 1990, pp. 11.
- Cover, T. M., Thomas, J. A.: "Elements of Information Theory", Wiley Series in Telecommunications, New York, 1991, pp. 13.
- Gersho A., Gray R. M.: "Vector Quantization and Signal Compression", Kluwer Academic Publisher, Boston, 1992, pp. 309.
- A. Moffat, R. Neal and I. H. Witten, "Arithmetic Coding Revisited", in Proc. IEEE Data Compression Conference, Snowbird, Utah, March 1995.

List of tables

List of figures

History

Document history		
V1.1.1	April 2014	Publication