
**Information technology — Generic coding
of moving pictures and associated audio
information —**

Part 7:
Advanced Audio Coding (AAC)

*Technologies de l'information — Codage générique des images animées et
du son associé —*

Partie 7: Codage du son avancé (AAC)



Contents

1. Scope	1
2. Normative references	3
3. Definitions	5
4. Symbols and abbreviations	7
5. Method of describing bit stream syntax	9
6. Syntax	10
7. Profiles	18
8. General information	20
9. Noiseless coding	39
10. Quantization	43
11. Scalefactors	43
12. Joint Coding	45
13. Prediction	50
14. Temporal Noise Shaping (TNS)	56
15. Filterbank and block switching	58
16. Gain Control	63
Annex A (normative) Huffman codebook tables	70
Annex B (informative) MSDL and encoder	86
Annex C (informative) Patent holders	142
Annex D (informative) Registration procedure	144
Annex E (informative) Registration application form	146
Annex F (informative) Registration Authority Diagram of administration structure	147

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

International Standard ISO/IEC 13818-7 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

ISO/IEC 13818 consists of the following parts, under the general title *Information technology — Generic coding of moving pictures and associated audio information*:

- *Part 1: Systems*
- *Part 2: Video*
- *Part 3: Audio*
- *Part 4: Compliance testing*
- *Part 5: Software simulation*
- *Part 6: Extensions for DSM-CC*
- *Part 7: Advanced Audio Coding (AAC)*
- *Part 9: Extension for real time interface for systems decoders*
- *Part 10: Conformance extensions for DSM-CC*

Annex A forms an integral part of this part of ISO/IEC 13818. Annexes B to F are for information only.

Introduction

The standardization body ISO/IEC JTC 1/SC 29/WG 11, also known as the Moving Pictures Expert Group (MPEG), was established in 1988 to specify digital video and audio coding schemes at low data rates. MPEG completed its first phase of audio specifications (MPEG-1) in November 1992, ISO/IEC 11172-3 [1]. In its second phase of development, the MPEG Audio subgroup defined a multichannel extension to MPEG-1 audio that is backwards compatible with existing MPEG-1 systems (MPEG-2 BC) and defined an audio coding standard at lower sampling frequencies than MPEG-1, ISO/IEC 13818-3 [2].

Information technology — Generic coding of moving pictures and associated audio information —

Part 7:

Advanced Audio Coding (AAC)

1 Scope

This part of ISO/IEC 13818 describes the MPEG-2 audio non-backwards compatible standard called MPEG-2 Advanced Audio Coding, AAC [3], a higher quality multichannel standard than achievable while requiring MPEG-1 backwards compatibility. This MPEG-2 AAC audio standard allows for ITU-R ‘indistinguishable’ quality according to [4] at data rates of 320 kb/s for five full-bandwidth channel audio signals.

The AAC decoding process makes use of a number of required tools and a number of optional tools. Table 1 lists the tools and their status as required or optional. Required tools are mandatory in any possible profile. Optional tools may not be required in some profiles.

Table 1

Tool Name	Required / Optional
Bitstream Formatter	Required
Noiseless Decoding	Required
Inverse Quantizer	Required
Scalefactors	Required
M/S	Optional
Prediction	Optional
Intensity/Coupling	Optional
TNS	Optional
Filterbank	Required
Gain Control	Optional

MPEG-2 AAC Tools Overview

The basic structure of the MPEG-2 AAC system is shown in Figures 1.1 and 1.2. As is shown in Table 1, there are both required and optional tools in the decoder, Figure 1.2. The data flow in this diagram is from left to right, top to bottom. The functions of the decoder are to find the description of the quantized audio spectra in the bitstream, decode the quantized values and other reconstruction information, reconstruct the quantized spectra, process the reconstructed spectra through whatever tools are active in the bitstream in order to arrive at the actual signal spectra as described by the input bitstream, and finally convert the frequency domain spectra to the time domain, with or without an optional gain control tool. Following the initial reconstruction and scaling of the spectrum reconstruction, there are many optional tools that modify one or more of the spectra in order to provide more efficient coding. For each of the optional tools that operate in the spectral domain, the option to “pass through” is retained, and in all cases where a spectral operation is omitted, the spectra at its input are passed directly through the tool without modification.

The input to the bitstream demultiplexer tool is the MPEG-2 AAC bitstream. The demultiplexer separates the parts of the MPEG-AAC data stream into the parts for each tool, and provides each of the tools with the bitstream information related to that tool.

The outputs from the bitstream demultiplexer tool are:

- The sectioning information for the noiselessly coded spectra
- The noiselessly coded spectra
- The M/S decision information (optional)
- The predictor state information (optional)
- The intensity stereo control information and coupling channel control information (both optional)
- The temporal noise shaping (TNS) information (optional)
- The filterbank control information
- The gain control information (optional)

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

The inputs to the noiseless decoding tool are:

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

The outputs of the Noiseless Decoding tool are:

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

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

The input to the Inverse Quantizer tool is:

- The quantized values for the spectra

The output of the inverse quantizer tool is:

- The un-scaled, inversely quantized spectra

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

The inputs to the scalefactors tool are:

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

The output from the scalefactors tool is:

- The scaled, inversely quantized spectra

The M/S tool converts spectra pairs from Mid/Side to Left/Right under control of the M/S decision information in order to improve coding efficiency.

The inputs to the M/S tool are:

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

The output from the M/S tool is:

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

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

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

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

The output from the prediction tool is:

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

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

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

The inputs to the intensity stereo / coupling tool are:

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

The output from the intensity stereo / coupling tool is:

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

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

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

The inputs to the TNS tool are:

- The inversely quantized spectra
- The TNS information

The output from the TNS block is:

- The inversely quantized spectra

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

The **filterbank tool** applies the inverse of the frequency mapping that was carried out in the encoder, as indicated by the filterbank control information and the presence or absence of gain control information. An inverse modified discrete cosine transform (IMDCT) is used for the filterbank tool. If the gain control tool is not used, the IMDCT input consists of either 1024 or 128 spectral coefficients, depending of the value of `window_sequence` (see 6.3, Table 6.11). If the gain control tool is used, the filterbank tool is configured to use four sets of either 256 or 32 coefficients, depending of the value of `window_sequence`.

The inputs to the filterbank tool are:

- The inversely quantized spectra
- The filterbank control information

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

- The time domain reconstructed audio signal(s).

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

The inputs to the gain control tool are:

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

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

- The time domain reconstructed audio signal(s)

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

2 Normative references

The following standards contain provisions which, through reference in this text, constitute provisions of this part of ISO/IEC 13818. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this part of ISO/IEC 13818 are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

- [1] ISO/IEC 11172-3:1993, *Information technology — Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s — Part 3: Audio*.
- [2] ISO/IEC 13818-3:1997, *Information technology — Generic coding of moving pictures and associated audio information — Part 3: Audio*.
- [3] M. Bosi, K. Brandenburg, S. Quackenbush, L. Fielder, K. Akagiri, H. Fuchs, M. Dietz, J. Herre, G. Davidson, Y. Oikawa, "ISO/IEC MPEG-2 Advanced Audio Coding", *Journal of the Audio Engineering Society*, Vol. 45, no. 10, pp. 789-814, October 1997.
- [4] ITU-R Document TG10-2/3- E only, *Basic Audio Quality Requirements for Digital Audio Bit-Rate Reduction Systems for Broadcast Emission and Primary Distribution*, 28 October 1991.
- [5] F. J. Harris, *On the Use of Windows For Harmonic Analysis of the Discrete Fourier Transform*, *Proc. of the IEEE*, Vol. 66, pp. 51- 83, January 1975.
- [6] ISO/IEC 13818-1:1996, *Information technology — Generic coding of moving pictures and associated audio information: Systems*.

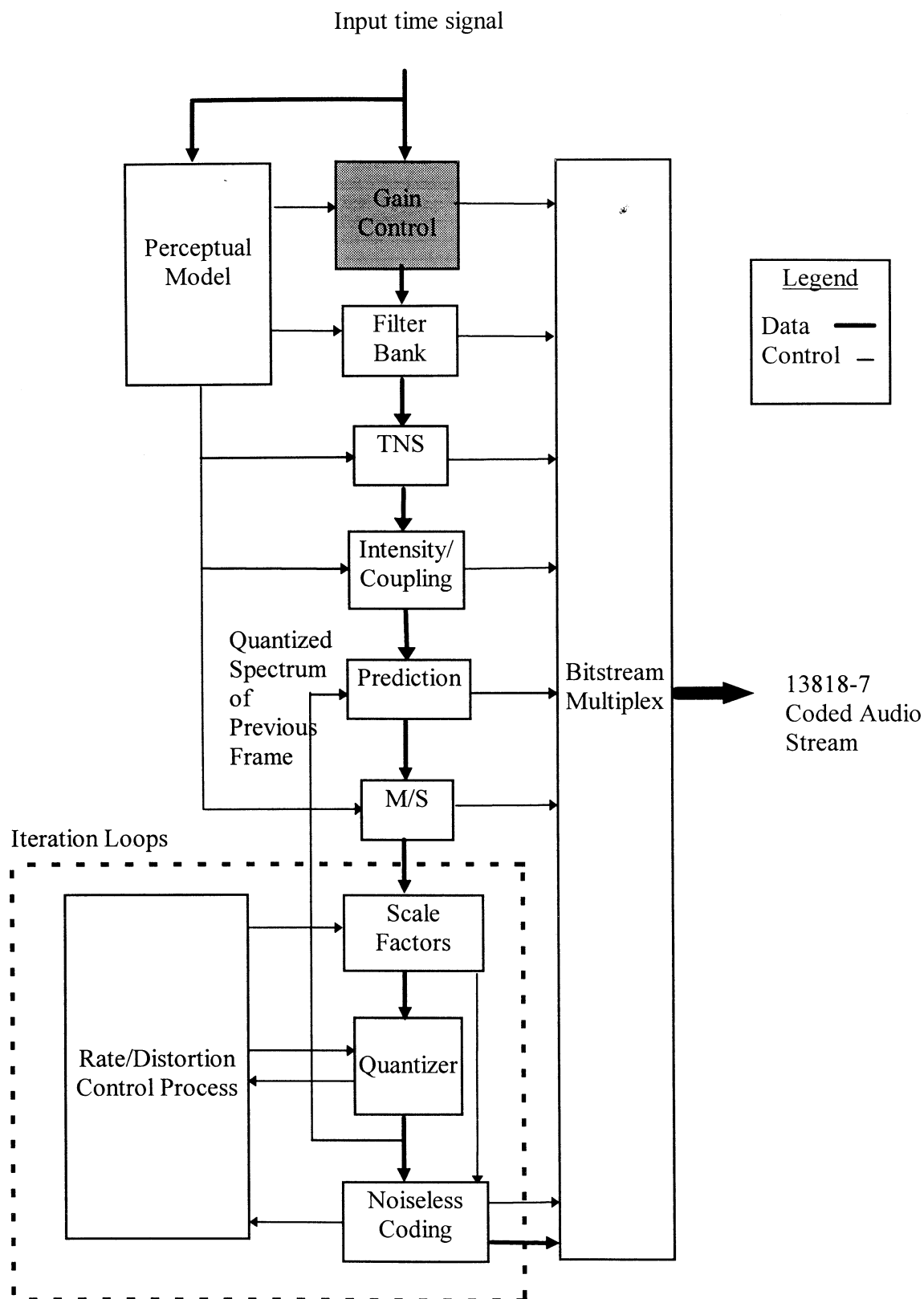


Figure – 1.1 MPEG-2 AAC Encoder Block Diagram

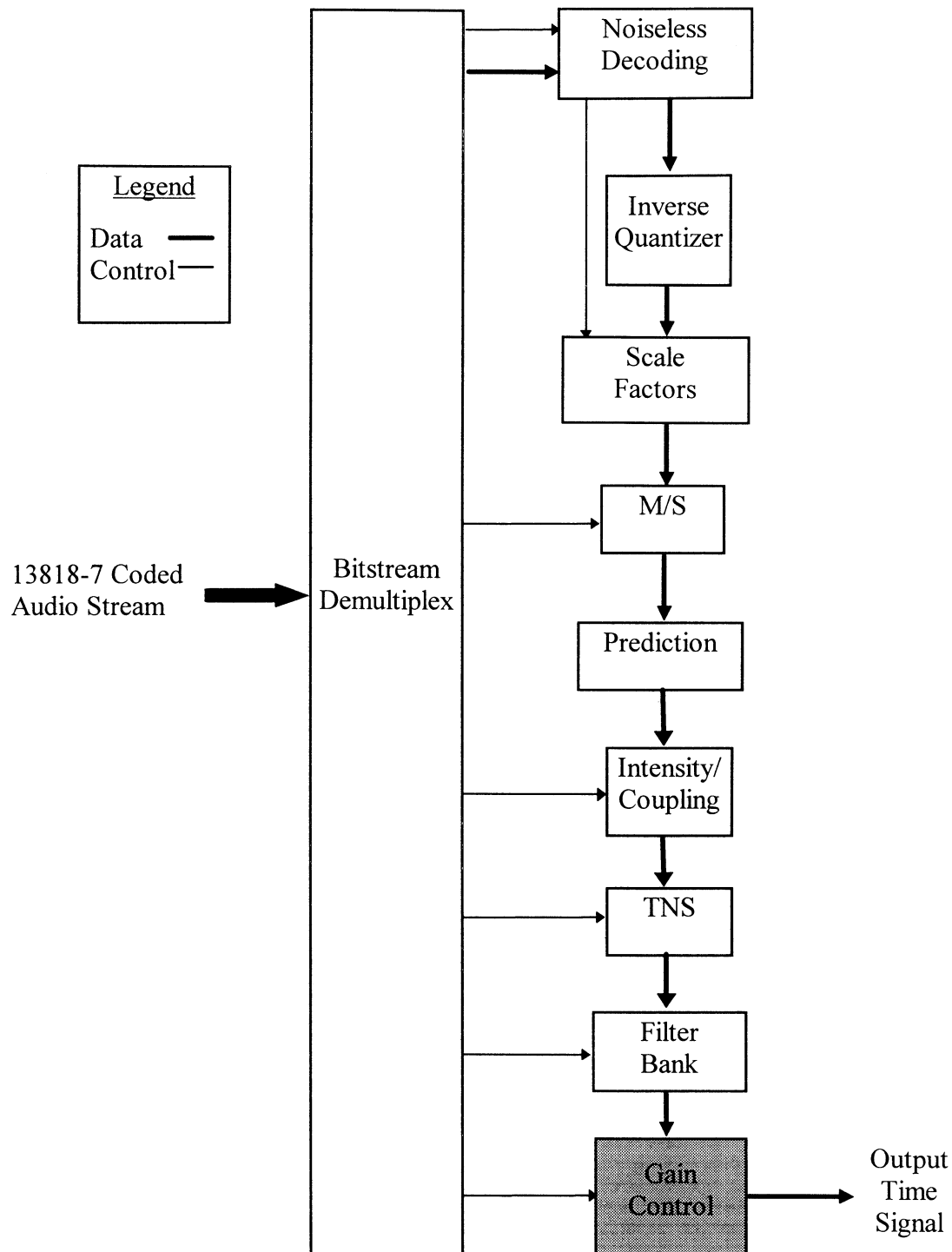


Figure 1.2 – MPEG-2 AAC Decoder Block Diagram

3 Definitions

For the purposes of this part of ISO/IEC 13818, the following definitions apply.

3.1. alias: Mirrored signal component resulting from sampling.

3.2. analysis filterbank: Filterbank in the encoder that transforms a broadband PCM audio signal into a set of spectral coefficients.

3.3. ancillary data: Part of the bitstream that might be used for transmission of ancillary data.

3.4. audio buffer: A buffer in the system target decoder (see ISO/IEC 13818-1) for storage of compressed audio data.

3.5. Bark: The Bark is the standard unit corresponding to one critical band width of human hearing.

3.6. backward compatibility: A newer coding standard is backward compatible with an older coding standard if decoders designed to operate with the older coding standard are able to continue to operate by decoding all or part of a bitstream produced according to the newer coding standard.

- 3.7. bitrate:** The rate at which the compressed bitstream is delivered to the input of a decoder.
- 3.8. bitstream; stream:** An ordered series of bits that forms the coded representation of the data.
- 3.9. bitstream verifier:** A process by which it is possible to test and verify that all the requirements specified in ISO/IEC 13818-7 are met by the bitstream.
- 3.10. block companding:** Normalising of the digital representation of an audio signal within a certain time period.
- 3.11. byte aligned:** A bit in a coded bitstream is byte-aligned if its position is a multiple of 8-bits from either the first bit in the stream for the Audio_Data_Interchange_Format (see 6.1) or the first bit in the syncword for the Audio_Data_Transport_Stream Format (see 6.2).
- 3.12. byte:** Sequence of 8-bits.
- 3.13. centre channel:** An audio presentation channel used to stabilise the central component of the frontal stereo image.
- 3.14. channel:** A sequence of data representing an audio signal intended to be reproduced at one listening position.
- 3.15. coded audio bitstream:** A coded representation of an audio signal.
- 3.16. coded representation:** A data element as represented in its encoded form.
- 3.17. compression:** Reduction in the number of bits used to represent an item of data.
- 3.18. constant bitrate:** Operation where the bitrate is constant from start to finish of the coded bitstream.
- 3.19. CRC:** The Cyclic Redundancy Check to verify the correctness of data.
- 3.20. critical band:** This unit of bandwidth represents the standard unit of bandwidth expressed in human auditory terms, corresponding to a fixed length on the human cochlea. It is approximately equal to 100 Hz at low frequencies and 1/3 octave at higher frequencies, above approximately 700 Hz.
- 3.21. data element:** An item of data as represented before encoding and after decoding.
- 3.22. de-emphasis:** Filtering applied to an audio signal after storage or transmission to undo a linear distortion due to emphasis.
- 3.23. decoded stream:** The decoded reconstruction of a compressed bitstream.
- 3.24. decoder:** An embodiment of a decoding process.
- 3.25. decoding (process):** The process defined in This part of ISO/IEC 13818 that reads an input coded bitstream and outputs decoded audio samples.
- 3.26. digital storage media; DSM:** A digital storage or transmission device or system.
- 3.27. discrete cosine transform; DCT:** Either the forward discrete cosine transform or the inverse discrete cosine transform. The DCT is an invertible, discrete orthogonal transformation.
- 3.28. downmix:** A matrixing of n channels to obtain less than n channels.
- 3.29. editing:** The process by which one or more coded bitstreams are manipulated to produce a new coded bitstream. Conforming edited bitstreams must meet the requirements defined in this part of ISO/IEC 13818.
- 3.30. emphasis:** Filtering applied to an audio signal before storage or transmission to improve the signal-to-noise ratio at high frequencies.
- 3.31. encoder:** An embodiment of an encoding process.
- 3.32. encoding (process):** A process, not specified in ISO/IEC 13818, that reads a stream of input audio samples and produces a valid coded bitstream as defined in this part of ISO/IEC 13818.
- 3.33. entropy coding:** Variable length lossless coding of the digital representation of a signal to reduce statistical redundancy.
- 3.34. FFT:** Fast Fourier Transformation. A fast algorithm for performing a discrete Fourier transform (an orthogonal transform).
- 3.35. filterbank:** A set of band-pass filters covering the entire audio frequency range.
- 3.36. flag:** A variable which can take one of only the two values defined in this specification.
- 3.37. forward compatibility:** A newer coding standard is forward compatible with an older coding standard if decoders designed to operate with the newer coding standard are able to decode bitstreams of the older coding standard.
- 3.38. Fs:** Sampling frequency.
- 3.39. Hann window:** A time function applied sample-by-sample to a block of audio samples before Fourier transformation.
- 3.40. Huffman coding:** A specific method for entropy coding.
- 3.41. hybrid filterbank:** A serial combination of subband filterbank and MDCT.
- 3.42. IDCT:** Inverse Discrete Cosine Transform.
- 3.43. IMDCT:** Inverse Modified Discrete Cosine Transform.
- 3.44. intensity stereo:** A method of exploiting stereo irrelevance or redundancy in stereophonic audio programmes based on retaining at high frequencies only the energy envelope of the right and left channels.
- 3.45. joint stereo coding:** Any method that exploits stereophonic irrelevance or stereophonic redundancy.
- 3.46. joint stereo mode:** A mode of the audio coding algorithm using joint stereo coding.
- 3.47. low frequency enhancement (LFE) channel:** A limited bandwidth channel for low frequency audio effects in a multichannel system.
- 3.48. main audio channels:** All single_channel_elements (see 8.2.1) or channel_pair_elements (see 8.2.1) in one program.
- 3.49. mapping:** Conversion of an audio signal from time to frequency domain by subband filtering and/or by MDCT.
- 3.50. masking:** A property of the human auditory system by which an audio signal cannot be perceived in the presence of another audio signal.
- 3.51. masking threshold:** A function in frequency and time below which an audio signal cannot be perceived by the human auditory system.
- 3.52. modified discrete cosine transform (MDCT):** A transform which has the property of time domain aliasing cancellation. An analytical expression for the MDCT can be found in B.2.3.1.2.

- 3.53. M/S stereo:** A method of removing imaging artefacts as well as exploiting stereo irrelevance or redundancy in stereophonic audio programmes based on coding the sum and difference signal instead of the left and right channels.
- 3.54. multichannel:** A combination of audio channels used to create a spatial sound field.
- 3.55. multilingual:** A presentation of dialogue in more than one language.
- 3.56. non-tonal component:** A noise-like component of an audio signal.
- 3.57. Nyquist sampling:** Sampling at or above twice the maximum bandwidth of a signal.
- 3.58. padding:** A method to adjust the average length of an audio frame in time to the duration of the corresponding PCM samples, by conditionally adding a slot to the audio frame.
- 3.59. parameter:** A variable within the syntax of this specification which may take one of a range of values. A variable which can take one of only two values is a flag or indicator and not a parameter.
- 3.60. parser:** Functional stage of a decoder which extracts from a coded bitstream a series of bits representing coded elements.
- 3.61. polyphase filterbank:** A set of equal bandwidth filters with special phase interrelationships, allowing for an efficient implementation of the filterbank.
- 3.62. prediction error:** The difference between the actual value of a sample or data element and its predictor.
- 3.63. prediction:** The use of a predictor to provide an estimate of the sample value or data element currently being decoded.
- 3.64. predictor:** A linear combination of previously decoded sample values or data elements.
- 3.65. presentation channel:** An audio channel at the output of the decoder.
- 3.66. program:** A set of main audio channels, coupling_channel_elements (see 8.2.1), lfe_channel_elements (see 8.2.1), and associated data streams intended to be decoded and played back simultaneously. A program may be defined by default (see 8.5.1) or specifically by a program_configuration_element (see 8.2.1). A given single_channel_element (see 8.2.1), channel_pair_element (see 8.2.1), coupling_channel_element, lfe_channel_element or data channel may accompany one or more programs in any given bitstream.
- 3.67. psychoacoustic model:** A mathematical model of the masking behaviour of the human auditory system.
- 3.68. random access:** The process of beginning to read and decode the coded bitstream at an arbitrary point.
- 3.69. reserved:** The term "reserved" when used in the clauses defining the coded bitstream indicates that the value may be used in the future for ISO/IEC defined extensions.
- 3.70. Sampling Frequency (Fs):** Defines the rate in Hertz which is used to digitise an audio signal during the sampling process.
- 3.71. scalefactor:** Factor by which a set of values is scaled before quantization.
- 3.72. scalefactor band:** A set of spectral coefficients which are scaled by one scalefactor.
- 3.73. scalefactor index:** A numerical code for a scalefactor.
- 3.74. side information:** Information in the bitstream necessary for controlling the decoder.
- 3.75. spectral coefficients:** Discrete frequency domain data output from the analysis filterbank.
- 3.76. spreading function:** A function that describes the frequency spread of masking effects.
- 3.77. stereo-irrelevant:** A portion of a stereophonic audio signal which does not contribute to spatial perception.
- 3.78. stuffing (bits); stuffing (bytes):** Code-words that may be inserted at particular locations in the coded bitstream that are discarded in the decoding process. Their purpose is to increase the bitrate of the stream which would otherwise be lower than the desired bitrate.
- 3.79. surround channel:** An audio presentation channel added to the front channels (L and R or L, R, and C) to enhance the spatial perception.
- 3.80. syncword:** A 12-bit code embedded in the audio bitstream that identifies the start of a adts_frame() (see 6.2, Table 6.4).
- 3.81. synthesis filterbank:** Filterbank in the decoder that reconstructs a PCM audio signal from subband samples.
- 3.82. tonal component:** A sinusoid-like component of an audio signal.
- 3.83. variable bitrate:** Operation where the bitrate varies with time during the decoding of a coded bitstream.
- 3.84. variable length coding:** A reversible procedure for coding that assigns shorter code-words to frequent symbols and longer code-words to less frequent symbols.
- 3.85. variable length code (VLC):** A code word assigned by variable length encoder (See variable length coding)
- 3.86. variable length decoder:** A procedure to obtain the symbols encoded with a variable length coding technique.
- 3.87. variable length encoder:** A procedure to assign variable length codewords to symbols.

4 Symbols and abbreviations

The mathematical operators used to describe this Recommendation | International Standard are similar to those used in the C programming language. However, integer division with truncation and rounding are specifically defined. The bitwise operators are defined assuming twos-complement representation of integers. Numbering and counting loops generally begin from zero.

4.1 Arithmetic operators

- + Addition.
- − Subtraction (as a binary operator) or negation (as a unary operator).
- ++ Increment.
- Decrement.
- * Multiplication.
- ^ Power.

/	Integer division with truncation of the result toward zero. For example, $7/4$ and $-7/-4$ are truncated to 1 and $-7/4$ and $7/-4$ are truncated to -1.
//	Integer division with rounding to the nearest integer. Half-integer values are rounded away from zero unless otherwise specified. For example $3//2$ is rounded to 2, and $-3//2$ is rounded to -2.
DIV	Integer division with truncation of the result towards $-\infty$.
	Absolute value. $ x = x$ when $x > 0$ $ x = 0$ when $x == 0$ $ x = -x$ when $x < 0$
%	Modulus operator. Defined only for positive numbers.
Sign()	Sign. $\text{Sign}(x) = 1$ when $x > 0$ $\text{Sign}(x) = 0$ when $x == 0$ $\text{Sign}(x) = -1$ when $x < 0$
INT ()	Truncation to integer operator. Returns the integer part of the real-valued argument.
NINT ()	Nearest integer operator. Returns the nearest integer value to the real-valued argument. Half-integer values are rounded away from zero.
sin	Sine.
cos	Cosine.
exp	Exponential.
$\sqrt{\quad}$	Square root.
log ₁₀	Logarithm to base ten.
log _e	Logarithm to base e.
log ₂	Logarithm to base 2.

4.2 Logical operators

	Logical OR.
&&	Logical AND.
!	Logical NOT

4.3 Relational operators

>	Greater than.
>=	Greater than or equal to.
<	Less than.
<=	Less than or equal to.
==	Equal to.
!=	Not equal to.
max [,...]	the maximum value in the argument list.
min [,...]	the minimum value in the argument list.

4.4 Bitwise operators

A two's complement number representation is assumed where the bitwise operators are used.

&	AND
	OR
>>	Shift right with sign extension.
<<	Shift left with zero fill.

4.5 Assignment

=	Assignment operator.
---	----------------------

4.6 Mnemonics

The following mnemonics are defined to describe the different data types used in the coded bit stream.

bslbf	Bit string, left bit first, where "left" is the order in which bit strings are written in ISO/IEC 11172. Bit strings are written as a string of 1s and 0s within single quote marks, e.g. '1000 0001'. Blanks within a bit string are for ease of reading and have no significance.
L, C, R, LS, RS	Left, center, right, left surround and right surround audio signals
rpchof	Remainder polynomial coefficients, highest order first. (Audio)
uimsbf	Unsigned integer, most significant bit first.
vlc1bf	Variable length code, left bit first, where "left" refers to the order in which the VLC codes are written.
window	Number of the actual time slot in case of <code>block_type==2</code> , $0 \leq \text{window} \leq 2$. (Audio)
The byte order of multi-byte words is most significant byte first.	

4.7 Constants

π 3,14159265358...
e 2,71828182845...

5 Method of describing bit stream syntax

The bit stream retrieved by the decoder is described in clause 6 (Syntax). Each data item in the bit stream is in bold type. It is described by its name, its length in bits, and a mnemonic for its type and order of transmission.

The action caused by a decoded data element in a bit stream depends on the value of that data element and on data elements previously decoded. The decoding of the data elements and the definition of the state variables used in their decoding are described in 8.3. The following constructs are used to express the conditions when data elements are present, and are in normal type:

Note this syntax uses the 'C'-code convention that a variable or expression evaluating to a non-zero value is equivalent to a condition that is true.

while (condition) { If the condition is true, then the group of data elements occurs next in the data stream. This repeats until the condition is not true.

data_element
...
}

do { The data element always occurs at least once. The data element is repeated until the condition is not true.

data_element

...
} while (condition)

if (condition) { If the condition is true, then the first group of data elements occurs next in the data stream

data_element

...
}

else { If the condition is not true, then the second group of data elements occurs next in the data stream.

data_element

...
}

for (expr1; expr2; expr3) { Expr1 is an expression specifying the initialisation of the loop. Normally it specifies the initial state of the counter. Expr2 is a condition specifying a test made before each iteration of the loop. The loop terminates when the condition is not true. Expr3 is an expression that is performed at the end of each iteration of the loop, normally it increments a counter.

data_element

...
}

Note that the most common usage of this construct is as follows:

for (i = 0; i < n; i++) { The group of data elements occurs n times. Conditional constructs within the group of data elements may depend on the value of the loop control variable i, which is set to zero for the first occurrence, incremented to one for the second occurrence, and so forth.

data_element

...
}

As noted, the group of data elements may contain nested conditional constructs. For compactness, the {} may be omitted when only one data element follows.

data_element [] data_element [] is an array of data. The number of data elements is indicated by the context.

data_element [n] data_element [n] is the n+1th element of an array of data.

data_element [m][n] data_element [m][n] is the m+1,n+1 th element of a two-dimensional array of data.

data_element [l][m][n] data_element [l][m][n] is the l+1,m+1,n+1 th element of a three-dimensional array of data.

data_element [m..n] data_element [m..n] is the inclusive range of bits between bit m and bit n in the data_element.

While the syntax is expressed in procedural terms, it should not be assumed that clause x.x.x implements a satisfactory decoding procedure. In particular, it defines a correct and error-free input bit stream. Actual decoders must include a means to look for start codes in order to begin decoding correctly.

Definition of bytealigned function

The function bytealigned () returns 1 if the current position is on a byte boundary, that is the next bit in the bit stream is the first bit in a byte. Otherwise it returns 0.

Definition of nextbits function

The function nextbits () permits comparison of a bit string with the next bits to be decoded in the bit stream.

6 Syntax

6.1 Audio Data Interchange Format, ADIF

Table 6.1 – Syntax of `adif sequence()`

Syntax	No. of bits	Mnemonic
<pre> adif_sequence() { adif_header() byte_alignment() raw_data_stream() } </pre>		

Table 6.2 – Syntax of `adif_header()`

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

6.2 Audio Data Transport Stream frame, ADTS

Table 6.3 – Syntax of adts sequence()

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

Table 6.4 – Syntax of adts frame()

Syntax	No. of bits	Mnemonic
<pre> adts_frame() { adts_fixed_header() adts_variable_header() adts_error_check() byte_alignment() for(i=0; i<number_of_raw_data_blocks_in_frame+1; i++) { raw_data_block() byte_alignment() } } </pre>		

6.2.1 Fixed header of ADTS

Table 6.5 – Syntax of `adts_fixed_header()`

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

6.2.2 Variable header of ADTS

Table 6.6 – Syntax of `adts_variable_header()`

Syntax	No. of bits	Mnemonic
<code>adts_variable_header()</code>		
{		
copyright_identification_bit	1	bslbf
copyright_identification_start	1	bslbf
frame_length	13	bslbf
adts_buffer_fullness	11	bslbf
number_of_raw_data_blocks_in_frame	2	uimsbf
}		

6.2.3 Error detection

Table 6.7 – Syntax of `adts_error_check()`

Syntax	No. of bits	Mnemonic
<code>adts_error_check()</code>		
{		
if (<code>protection_absent == '0'</code>)		
crc_check	16	rpchof
}		

6.3 Raw data

This is the `raw_data_stream`. It can be decoded directly or put into the Audio_Data_Transport stream using a variable rate header and a buffer fullness measure, with all the data of one frame contained between two occurrences of the header.

Table 6.7a – Syntax of `raw_data_stream()`

Syntax	No. of bits	Mnemonic
<code>raw_data_stream()</code>		
{		
while (<code>data_available()</code>) {		
<code>raw_data_block()</code>		
<code>byte_alignment()</code>		
}		
}		

Table 6.8 – Syntax of `raw_data_block()`

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

<pre> break; case ID_CCE: coupling_channel_element() break; case ID_LFE: lfe_channel_element() break; case ID_DSE: data_stream_element() break; case ID_PCE: program_config_element() break; case ID_FIL: fill_element() break; } } } </pre>

Table 6.9 – Syntax of single_channel_element()

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

Table 6.10 – Syntax of channel_pair_element()

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

Table 6.11 – Syntax of ics_info()

Syntax	No. of bits	Mnemonic
ics_info()		
{		
ics_reserved_bit	1	bslbf
window_sequence	2	uimsbf
window_shape	1	uimsbf
if(window_sequence == EIGHT_SHORT_SEQUENCE) {		
max_sfb	4	uimsbf
scale_factor_grouping	7	uimsbf
}		
else {		
max_sfb	6	uimsbf
predictor_data_present	1	uimsbf
if(predictor_data_present) {		
predictor_reset	1	uimsbf
if(predictor_reset) {		
predictor_reset_group_number	5	uimsbf
}		
}		
}		

<pre> for (sfb=0; sfb<min(max_sfb, PRED_SFB_MAX); sfb++) { prediction_used[sfb] } } } </pre>	1	uimbsf
---	---	--------

Table 6.12 – Syntax of individual_channel_stream()

Syntax	No. of bits	Mnemonic
individual_channel_stream(common_window)		
{		
global_gain	8	uimbsf
if(!common_window)		
ics_info()		
section_data()		
scale_factor_data()		
pulse_data_present	1	uimbsf
if(pulse_data_present) {		
pulse_data()		
}		
tns_data_present	1	uimbsf
if(tns_data_present)		
tns_data()		
gain_control_data_present	1	uimbsf
if(gain_control_data_present)		
gain_control_data()		
spectral_data()		
}		

Table 6.13 – Syntax of section_data()

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

Table 6.14 – Syntax of `scale_factor_data()`

Syntax	No. of bits	Mnemonic
<pre> scale_factor_data() { for (g=0; g<num_window_groups; g++) { for (sfb=0; sfb<max_sfb; sfb++) { if (sfb_cb[g][sfb] != ZERO_HCB) { if (is_intensity(g,sfb)) hcod_sf[dpcm_is_position[g][sfb]] else hcod_sf[dpcm_sf[g][sfb]] } } } } </pre>	1..19 1..19	bslbf bslbf

Table 6.15 – Syntax of `tns_data()`

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

Table 6.16 – Syntax of `spectral_data()`

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

<pre> } } } } } </pre>	hcod_esc_z	1..31	bslbf
--	-------------------	--------------	--------------

Table 6.17 – Syntax of pulse_data()

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

Table 6.18 – Syntax of coupling_channel_element()

Syntax	No. of bits	Mnemonic
coupling_channel_element()		
{		
element_instance_tag	4	uimsbf
ind_sw_cce_flag	1	uimsbf
num_coupled_elements	3	uimsbf
num_gain_element_lists = 0		
for (c=0; c<num_coupled_elements+1; c++) {		
num_gain_element_lists++		
cc_target_is_cpe[c]	1	uimsbf
cc_target_tag_select[c]	4	uimsbf
if (cc_target_is_cpe[c]) {		
cc_l[c]	1	uimsbf
cc_r[c]	1	uimsbf
if (cc_l[c] && cc_r[c])		
num_gain_element_lists++		
}		
}		
cc_domain	1	uimsbf
gain_element_sign	1	uimsbf
gain_element_scale	2	uimsbf
individual_channel_stream(0)		
for (c=1; c<num_gain_element_lists; c++) {		
if (ind_sw_cce_flag) {		
cge = 1		
} else {		
common_gain_element_present[c]	1	uimsbf
cge = common_gain_element_present[c]		
}		
if (cge)		
hcod_sf[common_gain_element[c]]	1..19	bslbf
else {		
for (g=0; g<num_window_groups) {		
for (sfb=0; sfb<max_sfb; sfb++) {		
if (sfb_cb[g][sfb] != ZERO_HCB)		
hcod_sf[dpcm_gain_element[c][g][sfb]]	1..19	bslbf
}		
}		
}		
}		
}		

Table 6.19 – Syntax of life channel element()

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

Table 6.20 – Syntax of data stream element()

Syntax	No. of bits	Mnemonic
data_stream_element()		
{		
element_instance_tag	4	uimbsbf
data_byte_align_flag	1	uimbsbf
cnt = count	8	uimbsbf
if (cnt == 255)		
cnt += esc_count ;	8	uimbsbf
if (data_byte_align_flag)		
byte_alignment()		
for (i=0; i<cnt; i++)		
data_stream_byte [element_instance_tag][i];	8	uimbsbf
}		

Table 6.21 – Syntax of program config element()

Syntax	No. of bits	Mnemonic
program_config_element() {		
element_instance_tag	4	uimbsbf
profile	2	uimbsbf
sampling_frequency_index	4	uimbsbf
num_front_channel_elements	4	uimbsbf
num_side_channel_elements	4	uimbsbf
num_back_channel_elements	4	uimbsbf
num_lfe_channel_elements	2	uimbsbf
num_assoc_data_elements	3	uimbsbf
num_valid_cc_elements	4	uimbsbf
mono_mixdown_present	1	uimbsbf
if (mono_mixdown_present == 1)		
mono_mixdown_element_number	4	uimbsbf
stereo_mixdown_present	1	uimbsbf
if (stereo_mixdown_present == 1)		
stereo_mixdown_element_number	4	uimbsbf
matrix_mixdown_idx_present	1	uimbsbf
if (matrix_mixdown_idx_present == 1) {		
matrix_mixdown_idx	2	uimbsbf
pseudo_surround_enable	1	uimbsbf
}		
for (i = 0; i < num_front_channel_elements; i++) {		
front_element_is_cpe[i];	1	bslbf
front_element_tag_select[i];	4	uimbsbf
}		
for (i = 0; i < num_side_channel_elements; i++) {		
side_element_is_cpe[i];	1	bslbf
side_element_tag_select[i];	4	uimbsbf
}		
for (i = 0; i < num_back_channel_elements; i++) {		
back_element_is_cpe[i];	1	bslbf
back_element_tag_select[i];	4	uimbsbf
}		
for (i = 0; i < num_lfe_channel_elements; i++)		
lfe_element_tag_select[i];	4	uimbsbf
for (i = 0; i < num_assoc_data_elements; i++)		
assoc data element tag select[i];	4	uimbsbf

for (i = 0; i < num_valid_cc_elements; i++) {		
cc_element_is_ind_sw [i];	1	uimsbf
valid_cc_element_tag_select [i];	4	uimsbf
}		
byte_alignment()		
comment_field_bytes	8	uimsbf
for (i = 0; i < comment_field_bytes; i++)		
comment_field_data [i];	8	uimsbf
}		

Table 6.22 – Syntax of fill_element()

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

Table 6.23 – Syntax of gain_control_data()

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

```

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

```

7 Profiles

7.1 Profiles

There are three profiles identified in the MPEG-2 AAC standard:

- Main Profile
- Low Complexity Profile
- Scaleable Sampling Rate Profile

In the program configuration element and ATDS0 fixed header, a two bit field indicates the profile in use:

Table 7.1 – Profiles

index	profile
0	Main profile
1	Low Complexity profile (LC)
2	Scaleable Sampling Rate profile (SSR)
3	(reserved)

Main Profile

The main profile is used when memory cost is not significant, and when there is substantial processing power available. With the exception of the gain control tool, all parts of the tools may be used in order to provide the best data compression possible.

Low Complexity

The low complexity profile is used when RAM usage, processing power, and compression requirements are all present. In the low complexity profile, prediction, and gain control tool are not permitted and TNS order is limited.

Scaleable Sampling Rate

In the scaleable sampling rate profile, the gain-control tool is required. Prediction and coupling channels are not permitted, and TNS order and bandwidth are limited. Gain control is not used in the lowest of the 4 PQF subbands. In the case of a reduced audio bandwidth, the SSR profile will scale accordingly in complexity.

Naming Convention for MPEG-AAC Decoders and Bitstreams

A decoder or bitstream may be specified as an A.L.I.D Channel <Profile Name> Profile MPEG-2 AAC Decoder, where A is replaced by the number of main audio channels, L by the number of LFE channels, I by the number of independently switched coupling channels, D by the number of dependently switched coupling channels, and Profile Name by the actual profile name. An example would be a 5.1.1.1 Channel Main Profile MPEG-2 AAC Decoder, indicating a decoder capable of decoding 5 main audio channels, one LFE channel, and one each independently and dependently switched CCE, with each of the channels using the profile specified. This can be abbreviated as M.5.1.1.1 where the "M" indicates a main profile decoder. Similarly, a Low Complexity decoder can be specified by a leading "L", and an SSR profile by an "S".

Minimum Decoder Capability for Specified Number of Main Audio Channels and Profile

To insure some level of interoperability the following minimum decoder capabilities for decoders of a given profile and number of main audio channels are specified.

Number of Main Audio Channels	Main Profile Capability	Low Complexity Profile Capability	SSR Profile Capability
1	1.0.0.0	1.0.0.0	1.0.0.0
2	2.0.0.0	2.0.0.0	2.0.0.0
3	3.0.1.0	3.0.0.1	3.0.0.0
4	4.0.1.0	4.0.0.1	4.0.0.0
5	5.1.1.1	5.1.0.1	5.1.0.0
7	7.1.1.2	7.1.0.2	7.1.0.0

7.1.1 Profile dependent tool parameters

Maximum TNS order and bandwidth:

According to the profile in use, the value for the constant TNS_MAX_ORDER is set as follows for long windows: For the main profile the constant TNS_MAX_ORDER is 20, for the low complexity profile and the scaleable sampling rate profile the constant TNS_MAX_ORDER is 12. For short windows, the constant TNS_MAX_ORDER is 7 for all profiles.

According to the sampling rate and profile in use, the value for the constant TNS_MAX_BANDS is set as follows:

Sampling Rate [Hz]	Low Complexity / Main Profile (long windows)	Low Complexity / Main Profile (short windows)	Scaleable Sampling Rate Profile (long windows)	Scaleable Sampling Rate Profile (short windows)
96000	31	9	28	7
88200	31	9	28	7
64000	34	10	27	7
48000	40	14	26	6
44100	42	14	26	6
32000	51	14	26	6
24000	46	14	29	7
22050	46	14	29	7
16000	42	14	23	8
12000	42	14	23	8
11025	42	14	23	8
8000	39	14	19	7

7.2 Profile interoperability

Interoperability of bitstreams and decoders

Any bitstream of a given profile (see Table 7.3) whose number of main audio channels, LFE channels, independent coupling channels, and dependent coupling channels is less than or equal to the corresponding number of channels supported by a decoder of the same profile can be decoded by that decoder.

Table 7.3 describes the interoperability of the three profiles.

Table 7.3 – Profile Interoperability

Decoder Profile	Encoder Profile		
	Main Profile	LC Profile	SSR Profile
Main Profile	yes	yes	no *
LC Profile	no	yes	no *
SSR Profile	no	no **	yes

*In Table 7.3, these entries can be decoded if the main or LC profile decoder is able to parse, but not decode, the gain control information, but the reconstructed audio will have a limited bandwidth.

**In Table 7.3, this entry can be decoded, but the bandwidth of the decoded signal will be limited to approximately 5 kHz, corresponding to the nonaliased portion of the first PQMF filter band.

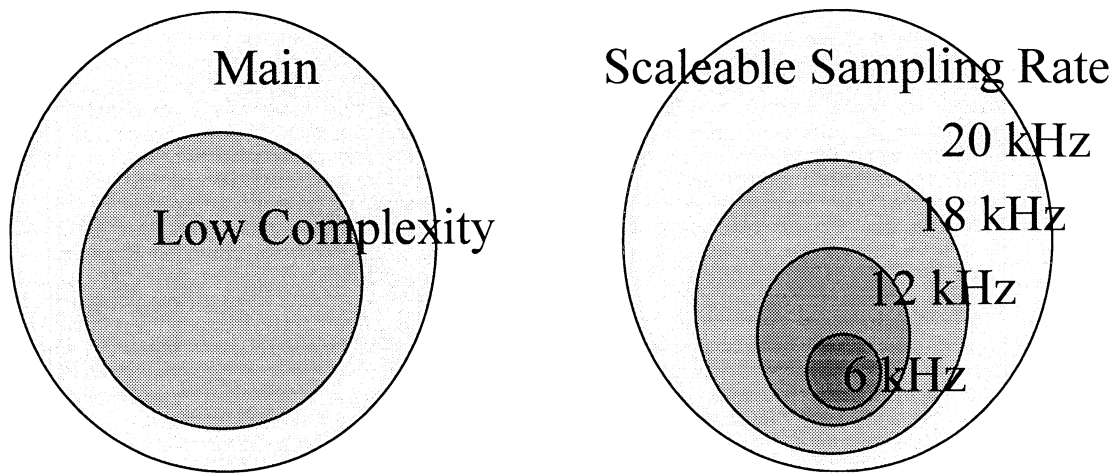


Figure 7.1 – Profile Interoperability

8 General information

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

8.1.1 Definitions

Bit stream elements:

<code>adif_sequence()</code>	a sequence according to the Audio_Data_Interchange_Format (Table 6.1)
<code>adif_header()</code>	header of the Audio_Data_Interchange_Format located at the beginning of an <code>adif_sequence</code> (Table 6.2)
<code>raw_data_block()</code>	see subclause 8.2.1 and Table 6.8
<code>adif_id</code>	ID that indicates the Audio_Data_Interchange_Format. Its value is 0x41444946 (most significant bit first), the ASCII representation of the string „ADIF“ (Table 6.2)
<code>copyright_id_present</code>	indicates whether <code>copyright_id</code> is present or not (Table 6.2)
<code>copyright_id</code>	The field consists of an 8-bit <code>copyright_identifier</code> , followed by a 64-bit <code>copyright_number</code> (Table 6.2). The <code>copyright_identifier</code> is given by a Registration Authority as designated by SC 29. The <code>copyright_number</code> is a value which identifies uniquely the copyrighted material. See ISO/IEC 13818-3, subclause 2.5.2.13.
<code>original_copy</code>	see ISO/IEC 11172-3, subclause 2.4.2.3 (Table 6.2)
<code>home</code>	see ISO/IEC 11172-3, subclause 2.4.2.3 (Table 6.2)
<code>bitstream_type</code>	a flag indicating the type of a bitstream (Table 6.2): ‘0’ constant rate bitstream. This bitstream may be transmitted via a channel with constant rate ‘1’ variable rate bitstream. This bitstream is not designed for transmission via constant rate channels
<code>bitrate</code>	a 23 bit unsigned integer indicating either the bitrate of the bitstream in bits/sec in case of constant rate bitstream or the maximum peak bitrate (measured per frame) in case of variable rate bitstreams. A value of 0 indicates that the bitrate is not known (Table 6.2)
<code>num_program_config_element</code>	number of program config elements specified for this <code>adif_sequence()</code> (Table 6.2)
<code>adif_buffer_fullness</code>	number of bits remaining in the encoder buffer after encoding the first <code>raw_data_block</code> in the <code>adif_sequence</code> (Table 6.2)
<code>program_config_element()</code>	contains information about the configuration for one program (Table 6.2). See subclause 8.5.
<code>adts_sequence()</code>	a sequence according to Audio_Data_Transport_Stream ADTS (Table 6.3)
<code>adts_frame()</code>	a ADTS frame, consisting of a fixed header, a variable header, an optional error check and a specified number of <code>raw_data_blocks()</code> (Table 6.4)
<code>adts_fixed_header()</code>	fixed header of ADTS. The information in this header does not change from frame to frame. It is repeated every frame to allow random access into a bitstream (Table 6.5)
<code>adts_variable_header()</code>	variable header of ADTS. This header is transmitted every frame as well as the fixed header, but contains data that changes from frame to frame (Table 6.6)
<code>adts_error_check()</code>	CRC error detection data generated as described in ISO/IEC 11172-3, subclause 2.4.3.1 (Table 6.7)

The following bits are protected and fed into the CRC algorithm in order of their appearance:

- all bits of the headers
- first 192 bits of any
 - single_channel_element (SCE)
 - channel_pair_element (CPE)
 - coupling_channel_element (CCE)
 - low frequency enhancement channel (LFE)

In addition, the first 128 bits of the second individual_channel_stream in the channel_pair_element must be protected. All information in any program configuration element or data element must be protected.

For any element where the specified protection length of 128 or 192 bits exceeds its actual length, the element is zero padded to the specified protection length for CRC calculation. if called from within a raw_data_block then align with respect to the first bit of the raw_data_block, else align with respect to the first bit of the header.

byte_alignment()

syncword

ID

layer

protection_absent

profile

sampling_frequency_index

The bit string '1111 1111 1111'. See ISO/IEC 11172-3, subclause 2.4.2.3 (Table 6.5)

MPEG identifier, set to '1'. See ISO/IEC 11172-3, subclause 2.4.2.3 (Table 6.5)

Indicates which layer is used. Set to '00'. See ISO/IEC 11172-3, subclause 2.4.2.3 (Table 6.5)

Indicates whether error_check() data is present or not. Same as syntax element 'protection_bit' in ISO/IEC 11172-3, subclause 2.4.1 and 2.4.2 (Table 6.5)

profile used. See clause 2. (Table 6.5)

indicates the sampling frequency used according to the following table:
(Table 6.5)

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

private_bit

channel_configuration

see ISO/IEC 11172-3, subclause 2.4.2.3 (Table 6.5)

indicates the channel configuration used. If channel_configuration is greater than 0, the channel configuration is given by the 'Default bitstream index number' in Table 8.1, see subclause 8.5. If channel_configuration equals 0, the channel configuration is not specified in the header and must be given by a program_config_element following as first bitstream element in the first raw_data_block after the header, or by the implicit configuration (see subclause 8.5) or must be known in the application. (Table 6.5)

emphasis

copyright_identification_bit

see ISO/IEC 11172-3, subclause 2.4.2.3 (Table 6.5)

One the bits of the 72-bit copyright identification field (see copyright_id above). The bits of this field are transmitted frame by frame; the first bit is indicated by the copyright_identification_start bit set to '1'. The field consists of an 8-bit copyright_identifier, followed by a 64-bit copyright_number. The copyright identifier is given by a Registration Authority as designated by SC29. The copyright_number is a value which identifies uniquely the copyrighted material. See ISO/IEC 13818-3, subclause 2.5.2.13 (Table 6.6)

copyright_identification_start

One bit to indicate that the copyright_identification_bit in this audio frame is the first bit of the 72-bit copyright identification. If no copyright identification is transmitted, this bit should be kept '0'.

'0' no start of copyright identification in this audio frame

'1' start of copyright identification in this audio frame

See ISO/IEC 13818-3, subclause 2.5.2.13 (Table 6.6)

frame_length

length of the frame including headers and error_check (Table 6.5) in bytes

adts_buffer_fullness number of 32 bit words remaining in the encoder buffer after encoding the first raw_data_block in the ADTS frame (Table 6.6). A value of hexadecimal 7FF signals that the bitstream is a variable rate bitstream. In this case, buffer fullness is not applicable.

number_of_raw_data_blocks_in_frame number of raw_data_blocks in the ADTS frame (Table 6.6)

Help elements:

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

8.1.2 Overview

The raw_data_block() contains all data which belongs to the audio (including ancillary data). Beyond that, additional information like sampling_frequency is needed to fully describe an audio sequence. The Audio_Data_Interchange_Format (ADIF) contains all elements that are necessary to describe a bitstream according to this standard.

For specific applications some or all of the syntax elements like those specified in the header of the ADIF, e.g. sampling_rate, may be known to the decoder by other means and hence do not appear in the bitstream. Furthermore, additional information that varies from block to block (e.g. to enhance the parsability or error resilience) may be required. Therefore transport streams may be designed for a specific application and are not specified in this standard. However, one possible transport stream called Audio_Data_Transport_Stream (ADTS) is given, which may be used for applications.

8.1.3 Audio_Data_Interchange_Format ADIF

The Audio_Data_Interchange_Format (ADIF) contains one header at the start of the sequence followed by a raw_data_stream(). The raw_data_stream() may not contain any further channel_configuration_elements.

As such, the ADIF is useful only for systems with a defined start and no need to start decoding from within the audio data stream, such as decoding from disk file. It can be used as an interchange format in that it contains all information necessary to decode and play the audio data.

8.1.4 Audio_Data_Transport_Stream ADTS

The Audio_Data_Transport_Stream (ADTS) is similar to syntax used in ISO/IEC 11172-3 and 13818-3. This will be recognized by ISO/IEC 11172-3 decoders as a "Layer 4" bit-stream.

The fixed header of the ADTS contains the syncword plus all parts of the header which are necessary for decoding and which do not change from frame to frame. The variable header of the ADTS contains header data which changes from frame to frame. The ADTS only supports a raw_data_stream() with only one program. The program may have up to 7 channels plus an independently switched coupling channel.

8.2 Decoding of raw data

8.2.1 Definitions

raw_data_stream()
raw_data_block()

sequence of raw_data_block()'s

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

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

Syntactic Element	ID name	encoding	Abbreviation
single channel element	ID_SCE	0x0	SCE
channel pair element	ID_CPE	0x1	CPE
coupling channel element	ID_CCE	0x2	CCE
lfe channel element	ID_LFE	0x3	LFE

data_stream_element	ID_DSE	0x4	DSE
program_config_element	ID_PCE	0x5	PCE
fill_element	ID_FIL	0x6	FIL
terminator	ID_END	0x7	TERM

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

8.2.2 Buffer requirements

Minimum decoder input buffer:

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

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

Bit reservoir:

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

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

Maximum bit rate:

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

$$\frac{6144 \frac{\text{bit}}{\text{block}}}{1024 \frac{\text{samples}}{\text{block}}} \cdot \text{sampling_frequency}$$

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

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

8.2.3 Decoding process

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

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

Examples of the simplest possible bitstreams are

bitstream segment

<SCE><TERM><SCE><TERM>...
 <CPE><TERM><CPE><TERM>...
 <SCE><CPE><CPE><LFE><TERM><SCE><CPE><CPE><LFE><TERM>...

output signal

mono signal
 stereo signal
 5.1 channel signal

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

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

<CPE><FIL><TERM><CPE><FIL><TERM>...

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

<CPE><DSE><FIL><TERM><CPE><DSE><FIL><TERM>...

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

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

8.3.1 Definitions

Bit stream elements:

<code>individual_channel_stream()</code>	contains data necessary to decode one channel (Table 6.12)
<code>ics_info()</code>	contains side information necessary to decode an individual_channel_stream. The individual_channel_streams of a channel_pair_element may share one common ics_info (Table 6.11)
<code>common_window</code>	a flag indicating whether the two individual_channel_streams share a common ics_info or not. In case of sharing, the ics_info is part of the channel_pair_element and must be used for both channels. Otherwise, the ics_info is part of each individual_channel_stream (Table 6.10)
<code>ics_reserved_bit</code>	bit reserved for future use
<code>window_sequence</code>	indicates the sequence of windows as defined in Table 8.3 (Table 6.11)
<code>window_shape</code>	A 1 bit field that determines what window is used for the trailing part of this analysis window (Table 6.11)
<code>max_sfb</code>	number of scalefactor bands transmitted per group (Table 6.11)
<code>scale_factor_grouping</code>	A bit field that contains information about grouping of short spectral data (Table 6.11)

Help elements:

<i>scalefactor window band</i>	term for scalefactor bands within a window, given in table Table 8.4 to Table 8.6
<i>scalefactor band</i>	term for scalefactor band within a group. In case of EIGHT_SHORT_SEQUENCE and grouping a scalefactor band may contain several scalefactor window bands of corresponding frequency. For all other window_sequences scalefactor bands and scalefactor window bands are identical.
<i>g</i>	group index
<i>win</i>	window index within group
<i>sfb</i>	scalefactor band index within group
<i>swb</i>	scalefactor window band index within window
<i>bin</i>	coefficient index
<i>num_window_groups</i>	number of groups of windows which share one set of scalefactors
<i>window_group_length[g]</i>	number of windows in each group.
<i>bit_set(bit_field, bit_num)</i>	function that returns the value of bit number bit_num of a bit_field (most right bit is bit 0)
<i>num_windows</i>	number of windows of the actual window sequence
<i>num_swb_long_window</i>	number of scalefactor bands for long windows. This number has to be selected depending on the sampling frequency. See 8.8.
<i>num_swb_short_window</i>	number of scalefactor window bands for short windows. This number has to be selected depending on the sampling frequency. See 8.8.
<i>num_swb</i>	number of scalefactor window bands for shortwindows in case of EIGHT_SHORT_SEQUENCE, number of scalefactor window bands for long windows otherwise
<i>swb_offset_long_window[swb]</i>	table containing the index of the lowest spectral coefficient of scalefactor band sfb for long windows. This table has to be selected depending on the sampling frequency. See 8.8.
<i>swb_offset_short_window[swb]</i>	table containing the index of the lowest spectral coefficient of scalefactor band sfb for short windows. This table has to be selected depending on the sampling frequency. See 8.8.
<i>swb_offset[swb]</i>	table containing the index of the lowest spectral coefficient of scalefactor band sfb for short windows in case of EIGHT_SHORT_SEQUENCE, otherwise for long windows
<i>sect_sfb_offset[g][section]</i>	table that gives the number of the start coefficient for the section_data() within a group. This offset depends on the window_sequence and scale_factor_grouping.
<i>sampling_frequency_index</i>	see 8.1.1

8.3.2 Decoding process

Single_channel_element and channel_pair_element

A single_channel_element is composed of an element_instance_tag and an individual_channel_stream. In this case ics_info is always located in the individual_channel_stream.

A channel_pair_element begins with an element_instance_tag and common_window flag. If the common_window equals '1', then ics_info is shared amongst the two individual_channel_stream elements and the MS information is transmitted. If common_window equals '0', then there is an ics_info within each individual_channel_stream and there is no MS information.

Decoding an individual_channel_stream (ICS)

In the individual_channel_stream, the order of decoding is:

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

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

Recovering ics_info

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

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

Recovering Sectioning Data

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

sect_cb The codebook for the section

and

sect_len The length of the section. This length is recovered by reading the bitstream sequentially for a section length, adding the escape value to the total length of the section until a non-escape value is found, which is added to establish the total length of the section. This process is clearly explained in the C-like syntax description. Note that within each group the sections must delineate the scalefactor bands from zero to **max_sfb** so that the first section within each group starts at bands zero and the last section within each group ends at **max_sfb**.

The sectioning data describes the codebook, and then the length of the section using that codebook, starting from the first scalefactor band and continuing until the total number of scalefactor bands is reached.

After this description is provided, all scalefactors and spectral data corresponding to codebook zero are zeroed, and no values corresponding to these scalefactors or spectral data will be transmitted. When scanning for scale-factor data it is important to note that scalefactors for any scalefactor bands whose Huffman codebook is zero will be omitted. Similarly, all spectral data associated with Huffman codebook zero are omitted (see clauses 9 and 11).

In addition spectral data associated with the scalefactor bands that have an intensity codebook will not be transmitted, but intensity steering coefficients will be transmitted in place of the scalefactors, as described in 12.2

Scalefactor Data Parsing and Decoding

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

Spectral Data Parsing and Decoding

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

8.3.3 Windows and window sequences

Quantization and coding is done in the frequency domain. For this purpose, the time signal is mapped into the frequency domain in the encoder. The decoder performs the inverse mapping as described in clause 15. Depending on the signal, the coder may change the time/frequency resolution by using two different windows: LONG_WINDOW and SHORT_WINDOW. To switch between windows, the transition windows LONG_START_WINDOW and LONG_STOP_WINDOW are used. Table 8.2 lists the windows, specifies the corresponding transform length and shows the shape of the windows schematically. Two transform lengths are used: 1024 (referred to as long transform) and 128 coefficients (referred to as short transform). Window sequences are composed of windows in a way that a raw_data_block always contains data representing 1024 output samples. The bitstream element **window_sequence** indicates the window sequence that is actually used. Table 8.3 lists how the window sequences are composed of individual windows. Refer to clause 15 for more detailed information about the transform and the windows.

8.3.4 Scalefactor bands and grouping

Many tools of the decoder perform operations on groups of consecutive spectral values called scalefactor bands (abbreviation 'sfb'). The width of the scalefactor bands is built in imitation of the critical bands of the human auditory system. For that reason the number of scalefactor bands in a spectrum and their width depend on the transform length and the sampling frequency. Table 8.4 to Table 8.6 list the offset to the beginning of each scalefactor band for the transform lengths 1024 and 128 and the different sampling frequencies, respectively.

To reduce the amount of side information in case of sequences which contain SHORT_WINDOWS, consecutive SHORT_WINDOWS may be grouped (see figure 8.1). The information about the grouping is contained in the **scale_factor_grouping** bitstream element. Grouping means that only one set of scalefactors is transmitted for all grouped windows as if there was only one window. The scalefactors are then applied to the corresponding spectral data in all grouped windows. To increase the efficiency of the noiseless coding (see clause 9), the spectral data of a group is transmitted in an interleaved order given in 8.3.5. The interleaving is done on a scalefactor band by scalefactor band basis, so that the spectral data can be grouped to form a virtual scalefactor band to which the common scalefactor can be applied. Within this document the expression 'scalefactor band' (abbreviation 'sfb') denotes these virtual scalefactor bands. If the scalefactor bands of the single windows are referred to, the expression 'scalefactor window band' (abbreviation 'swb') is used. Due to its influence on the scalefactor bands, grouping affects the meaning of section_data (see clause 9), the order of spectral data (see 8.3.5), and the total number of scalefactor bands. For a LONG_WINDOW scalefactor bands and scalefactor window bands are identical since there is only one group with only one window.

To reduce the amount of information needed for the transmission of side information specific to each scalefactor band, the bitstream element **max_sfb** is transmitted. Its value is one greater than the highest active scalefactor band in all groups. **max_sfb** has influence on the interpretation of section data (see clause 9), the transmission of scalefactors (see clause 9 and 11), the transmission of predictor data (see clause 13) and the transmission of the ms_mask (see 12.1).

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

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

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

```

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

```

8.3.5 Order of spectral coefficients in spectral_data

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

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

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

8.3.6 Output word length

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

8.3.7 Use of emphasis

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

8.3.8 Matrix-mixdown method

8.3.8.1 Description

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

8.3.8.2 Definitions

matrix_mixdown_idx_present	One bit indicating that a stereo matrix coefficient index is present (see Table 6.21). For all configurations other than the 3/2 format this bit must be zero.
matrix_mixdown_idx	A two bit field that indicates that the coefficient to be used in the 5-channel to 2-channel matrix-mixdown. Possible matrix coefficients are listed in 8.3.8.5.
pseudo_surround_enable	One bit indicating that pseudo surround decoding is possible.

8.3.8.3 Matrix-mixdown process

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

Set 1:

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

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

Set 2:

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

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

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

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

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

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

8.3.8.4 Advisory

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

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

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

8.3.8.5 Tables

Matrix-mixdown coefficients

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

8.4 Low Frequency Enhancement Channel (LFE)

8.4.1 General

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

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

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

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

8.5 Program Config Element (PCE)

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

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

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

8.5.1 Implicit and defined channel configurations

The MPEG-2 AAC audio syntax provides two ways to convey the mapping of channels within a set of syntactic elements to physical locations of speakers. The first way is a default mapping based on the specific set of syntactic elements received and the order in which they are received. The most common mappings are further defined in Table 8.1. If a mapping shown in Table 8.1 is not used, the following methods describe the default determination of channel mapping:

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

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

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

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

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

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

If reliable mapping of channel set to speaker geometry is a concern, then it is recommended that an implicit mapping from Table 8.1 or a program configuration element be used.

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

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

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

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

Other elements are also specified. A list of one or more LFE's is specified for application to this program. A list of one or more CCE's (profile-dependent) is also provided, in order to allow for dialog management as well as different intensity

coupling streams for different channels using the same main channels. A list of data streams associated with the program can also associate one or more data streams with a program. The program configuration element also allows for the specification of one monophonic and one stereophonic simulcast mixdown channel for a program.

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

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

8.6 Data stream element (DSE)

Bitstream elements:

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

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

Decoding process:

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

8.7 Fill element (FIL)

Bitstream elements:

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

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

Decoding process:

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

8.8 Tables

Table 8.1 – Implicit speaker mapping

Default bitstream index number	number of speakers	audio syntactic elements, listed in order received	default element to speaker mapping
1	1	single_channel_element	center front speaker
2	2	channel_pair_element	left, right front speakers
3	3	single_channel_element, channel_pair_element	center front speaker left, right front speakers
4	4	single_channel_element, channel_pair_element, single_channel_element	center front speaker left, right center front speakers, rear surround
5	5	single_channel_element, channel_pair_element,	center front speaker left, right front speakers,

		channel pair element	left surround, right surround rear speakers
6	5+1	single_channel_element, channel_pair_element, channel_pair_element, lfe_element	center front speaker left, right front speakers, left surround, right surround rear speakers, front low frequency effects speaker
7	7+1	single_channel_element, channel_pair_element, channel_pair_element, channel_pair_element, lfe_element	center front speaker left, right center front speakers, left, right outside front speakers, left surround, right surround rear speakers, front low frequency effects speaker

Table 8.2 – Transform windows (for 48 kHz)

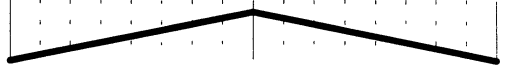
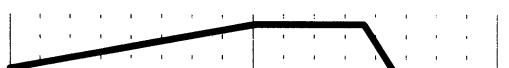
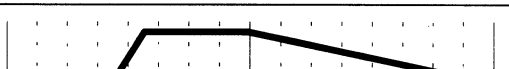
window	num_swb	#coeffs	looks like
LONG_WINDOW	49	1024	
SHORT_WINDOW	14	128	
LONG_START_WINDOW	49	1024	
LONG_STOP_WINDOW	49	1024	

Table 8.3 – Window Sequences

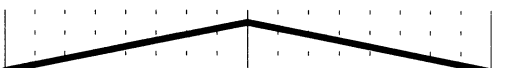
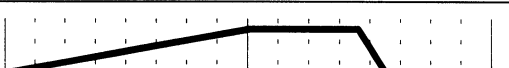
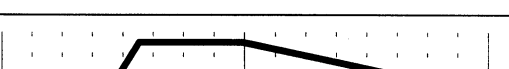
value	window_sequence	num_windows	looks like
0	ONLY_LONG_SEQUENCE = LONG_WINDOW	1	
1	LONG_START_SEQUENCE = LONG_START_WINDOW	1	
2	EIGHT_SHORT_SEQUENCE = 8 * SHORT_WINDOW	8	
3	LONG_STOP_SEQUENCE = LONG_STOP_WINDOW	1	

Table 8.4 – scalefactor bands for
LONG_WINDOW, LONG_START_WINDOW, LONG_STOP_WINDOW at 44.1 and 48 kHz

fs [kHz]	44.1, 48
num_swb_long_window	49
swb	swb_offset_long_window
0	0
1	4
2	8
3	12
4	16
5	20
6	24
7	28

swb	swb_offset_long_window
25	216
26	240
27	264
28	292
29	320
30	352
31	384
32	416

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

**Table 8.5 – scalefactor bands for SHORT_WINDOW
at 32, 44.1 and 48 kHz**

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

**Table 8.6 – scalefactor bands for
LONG_WINDOW, LONG_START_WINDOW, LONG_STOP_WINDOW
at 32 kHz**

fs [kHz]	32		
num_swb_long_window	51	swb	swb_offset_long_window
swb	swb_offset_long_window		
0	0	26	240
1	4	27	264
2	8	28	292
3	12	29	320
4	16	30	352
5	20	31	384
6	24	32	416
7	28	33	448
8	32	34	480
9	36	35	512
10	40	36	544
11	48	37	576
12	56	38	608
13	64	39	640
14	72	40	672
15	80	41	704
16	88	42	736
17	96	43	768
18	108	44	800
19	120	45	832
20	132	46	864
21	144	47	896
22	160	48	928
23	176	49	960
24	196	50	992
25	216		1024

**Table 8.7 – scalefactor bands for
LONG_WINDOW, LONG_START_WINDOW, LONG_STOP_WINDOW at 8 kHz**

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

Table 8.8 – scalefactor bands for SHORT_WINDOW at 8 kHz

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

**Table 8.9 – scalefactor bands for
LONG_WINDOW, LONG_START_WINDOW, LONG_STOP_WINDOW at 11.025, 12 and 16 kHz**

fs [kHz]	11.025, 12, 16		
num_swb_long_window	43		
swb	swb_offset_long_window	swb	swb_offset_long_window
0	0	22	228
1	8	23	244
2	16	24	260
3	24	25	280
4	32	26	300
5	40	27	320
6	48	28	344
7	56	29	368
8	64	30	396

9	72	31	424
10	80	32	456
11	88	33	492
12	100	34	532
13	112	35	572
14	124	36	616
15	136	37	664
16	148	38	716
17	160	39	772
18	172	40	832
19	184	41	896
20	196	42	960
21	212		1024

Table 8.10 – scalefactor bands for SHORT_WINDOW at 11.025, 12 and 16 kHz

fs [kHz]	11.025, 12, 16		
num_swb_short_window	15		
swb	swb_offset_short_window	swb	swb_offset_short_window
0	0	8	32
1	4	9	40
2	8	10	48
3	12	11	60
4	16	12	72
5	20	13	88
6	24	14	108
7	28		128

Table 8.11 – scalefactor bands for LONG_WINDOW, LONG_START_WINDOW, LONG_STOP_WINDOW at 22.05 and 24 kHz

fs [kHz]	22.05 and 24		
num_swb_long_window	47		
swb	swb_offset_long_window	swb	swb_offset_long_window
0	0	24	160
1	4	25	172
2	8	26	188
3	12	27	204
4	16	28	220
5	20	29	240
6	24	30	260
7	28	31	284
8	32	32	308
9	36	33	336
10	40	34	364
11	44	35	396
12	52	36	432
13	60	37	468
14	68	38	508
15	76	39	552
16	84	40	600
17	92	41	652
18	100	42	704
19	108	43	768
20	116	44	832
21	124	45	896
22	136	46	960
23	148		1024

Table 8.12 – scalefactor bands for SHORT_WINDOW at 22.05 and 24 kHz

fs [kHz]	22.05 and 24		
num_swb_short_window	15		
swb	swb_offset_short_window	swb	swb_offset_short_window
0	0	8	36
1	4	9	44
2	8	10	52
3	12	11	64
4	16	12	76
5	20	13	92
6	24	14	108
7	28		128

Table 8.13 – scalefactor bands for LONG_WINDOW, LONG_START_WINDOW, LONG_STOP_WINDOW at 64 kHz

fs [kHz]	64		
num_swb_long_window	47		
swb	swb_offset_long_window	swb	swb_offset_long_window
0	0	24	172
1	4	25	192
2	8	26	216
3	12	27	240
4	16	28	268
5	20	29	304
6	24	30	344
7	28	31	384
8	32	32	424
9	36	33	464
10	40	34	504
11	44	35	544
12	48	36	584
13	52	37	624
14	56	38	664
15	64	39	704
16	72	40	744
17	80	41	784
18	88	42	824
19	100	43	864
20	112	44	904
21	124	45	944
22	140	46	984
23	156		1024

Table 8.14 – scalefactor bands for SHORT_WINDOW at 64 kHz

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

**Table 8.15 – scalefactor bands for
LONG_WINDOW, LONG_START_WINDOW, LONG_STOP_WINDOW at 88.2 and 96 kHz**

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

Table 8.16 – scalefactor bands for SHORT_WINDOW at 88.2 and 96 kHz

fs [kHz]	88.2 and 96		
num_swb_short_window	12		
swb	swb_offset_short_window	swb	swb_offset_short_window
0	0	7	32
1	4	8	40
2	8	9	48
3	12	10	64
4	16	11	92
5	20		128
6	24		

8.9 Figures

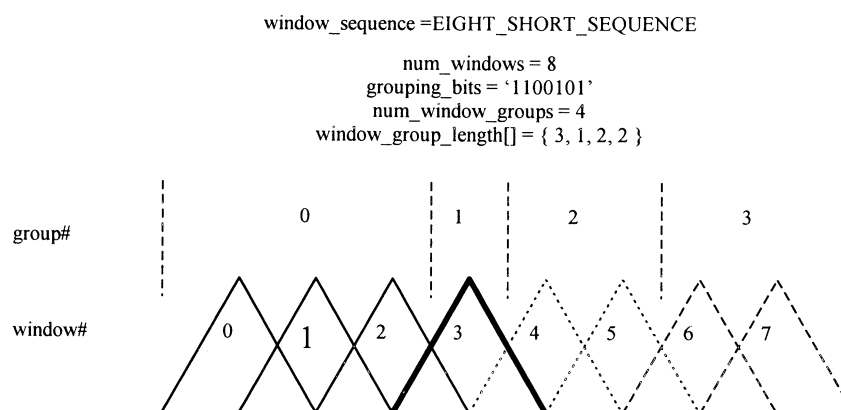
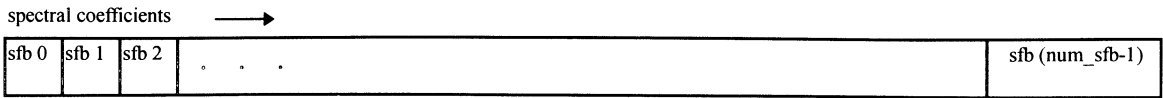
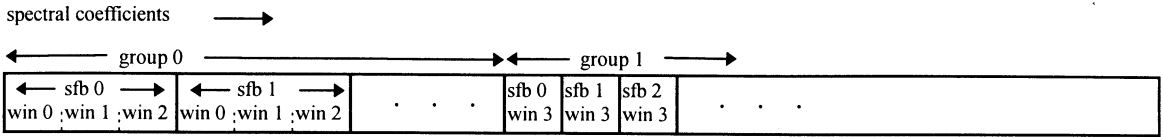


Figure 8.1 – Example for short window grouping



Order of scalefactor bands for ONLY_LONG_SEQUENCE

Figure 8.2 – Spectral order of scalefactor bands in case of ONLY_LONG_SEQUENCE



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

Figure 8.3 – Spectral order of scalefactor bands in case of EIGHT_SHORT_SEQUENCE

9 Noiseless coding

9.1 Tool description

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

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

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

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

As indicated in Table 9.2, spectrum Huffman codebooks can represent signed or unsigned n-tuples of coefficients. For unsigned codebooks, sign bits for every non-zero coefficient in the n-tuple immediately follow the associated codeword.

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

9.2 Definitions

<code>sect_cb[g][i]</code>	spectrum Huffman codebook used for section <code>i</code> in group <code>g</code> (see 6.3, Table 6.13).
<code>sect_len_incr</code>	used to compute the length of a section, measures number of scalefactor bands from start of section. The length of <code>sect_len_incr</code> is 3 bits if <code>window_sequence</code> is <code>EIGHT_SHORT_SEQUENCE</code> and 5 bits otherwise (see 6.3, Table 6.13).
<code>global_gain</code>	global gain of the quantized spectrum, sent as unsigned integer value (see 6.3, Table 6.12).
<code>hcod_sf[i]</code>	Huffman codeword from the Huffman code table used for coding of scalefactors (see 6.3, Table 6.14).

hcod[sect_cb[g][i]][w][x][y][z]	Huffman codeword from codebook sect_cb[g][i] that encodes the next 4-tuple (w, x, y, z) of spectral coefficients, where w, x, y, z are quantized spectral coefficients. Within an n-tuple, w, x, y, z are ordered as described in 8.3.5. so that $x_quant[group][win][sfb][bin] = w$, $x_quant[group][win][sfb][bin+1] = x$, $x_quant[group][win][sfb][bin+2] = y$ and $x_quant[group][win][sfb][bin+3] = z$. N-tuples progress from low to high frequency within the current section (see 6.3, Table 6.16).
hcod[sect_cb[g][i]][y][z]	Huffman codeword from codebook sect_cb[g][i] that encodes the next 2-tuple (y, z) of spectral coefficients, where y, z are quantized spectral coefficients. Within an n-tuple, y, z are ordered as described in 8.3.5 so that $x_quant[group][win][sfb][bin] = y$ and $x_quant[group][win][sfb][bin+1] = z$. N-tuples progress from low to high frequency within the current section (see 6.3, Table 6.16).
quad_sign_bits	sign bits for non-zero coefficients in the spectral 4-tuple. A '1' indicates a negative coefficient, a '0' a positive one. Bits associated with lower frequency coefficients are sent first (see 6.3, Table 6.16).
pair_sign_bits	sign bits for non-zero coefficients in the spectral 2-tuple. A '1' indicates a negative coefficient, a '0' a positive one. Bits associated with lower frequency coefficients are sent first (see 6.3, Table 6.16).
hcod_esc_y	escape sequence for quantized spectral coefficient y of 2-tuple (y,z) associated with the preceeding Huffman codeword (see 6.3, Table 6.16).
hcod_esc_z	escape sequence for quantized spectral coefficient z of 2-tuple (y,z) associated with the preceeding Huffman codeword (see 6.3, Table 6.16).
pulse_data_present	1 bit indicating whether the pulse escape is used (1) or not (0) (see 6.3, Table 6.17). Note that pulse_data_present must be 0 for an EIGHT_SHORT_SEQUENCE.
number_pulse	2 bits indicating how many pulse escapes are used. The number of pulse escapes is from 1 to 4 (see 6.3, Table 6.17).
pulse_start_sfb	6 bits indicating the index of the lowest scalefactor band where the pulse escape is achieved (see 6.3, Table 6.17).
pulse_offset[i]	5 bits indicating the offset (see 6.3, Table 6.17).
pulse_amp[i]	4 bits indicating the unsigned magnitude of the pulse (see 6.3, Table 6.17).
sect_start[g][i]	offset to first scalefactor band in section i of group g (see 6.3, Table 6.13).
sect_end[g][i]	offset to one higher than last scalefactor band in section i of group g (see 6.3, Table 6.13).
num_sec[g]	number of sections in group g (see 6.3, Table 6.13).
escape_flag	the value of 16 in the ESC Huffman codebook
escape_prefix	the bit sequence of N 1's
escape_separator	one 0 bit
escape_word	an N+4 bit unsigned integer word, msb first
escape_sequence	the sequence of <i>escape_prefix</i> , <i>escape_separator</i> and <i>escape_word</i>
escape_code	$2^{(N+4)} + \text{escape_word}$
x_quant[g][win][sfb][bin]	Huffman decoded value for group g, window win, scalefactor band sfb, coefficient bin
spec[w][k]	de-interleaved spectrum. w ranges from 0 to num_windows-1 and k ranges from 0 to swb_offset[num_swb]-1.

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

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

9.3 Decoding process

Four-tuples or 2-tuples of quantized spectral coefficients are Huffman coded and transmitted starting from the lowest-frequency coefficient and progressing to the highest-frequency coefficient. For the case of multiple windows per block (EIGHT_SHORT_SEQUENCE), the grouped and interleaved set of spectral coefficients is treated as a single set of coefficients that progress from low to high. The set of coefficients may need to be de-interleaved after they are decoded (see 8.3.5). Coefficients are stored in the array $x_quant[g][win][sfb][bin]$, and the order of transmission of the Huffman codewords is such that when they are decoded in the order received and stored in the array, *bin* is the most rapidly incrementing index and *g* is the most slowly incrementing index. Within a codeword, for those associated with spectral four-tuples, the order of decoding is w, x, y, z; for codewords associated with spectral two-tuples, the order of decoding is y, z. The set of coefficients

is divided into sections and the sectioning information is transmitted starting from the lowest frequency section and progressing to the highest frequency section. The spectral information for sections that are coded with the “zero” codebook is not sent as this spectral information is zero. Similarly, spectral information for sections coded with the “intensity” codebooks is not sent. The spectral information for all scalefactor bands at and above **max_sfb**, for which there is no section data, is zero.

There is a single differential scalefactor codebook which represents a range of values as shown in Table 9.1. The differential scalefactor codebook is shown in Table A.1. There are eleven Huffman codebooks for the spectral data, as shown in Table 9.2. The codebooks are shown in Tables A.2 through A.12. There are three other “codebooks” above and beyond the actual Huffman codebooks, specifically the “zero” codebook, indicating that neither scalefactors nor quantized data will be transmitted, and the “intensity” codebooks indicating that this individual channel is part of a channel pair, and that the data that would normally be scalefactors is instead steering data for intensity stereo. In this case, no quantized spectral data are transmitted. Codebook indices 12 and 13 are reserved.

The spectrum Huffman codebooks encode 2- or 4-tuples of signed or unsigned quantized spectral coefficients, as shown in Table 9.2. This table also indicates the largest absolute value (LAV) able to be encoded by each codebook and defines a boolean helper variable array, `unsigned_cb[]`, that is 1 if the codebook is unsigned and 0 if signed.

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

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

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

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

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

hcod[7][y][z]

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

pair_sign_bits

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

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

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

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

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

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

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

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

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

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

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

9.4 Tables

Table 9.1 – Scalefactor Huffman codebook parameters

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

Table 9.2 – Spectrum Huffman codebooks parameters

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

10 Quantization

10.1 Tool description

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

10.2 Definitions

Help elements:

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

10.3 Decoding process

The inverse quantization is described by the following formula:

$$x_invquant = Sign(x_quant) \cdot |x_quant|^{\frac{4}{3}} \quad \forall \quad k$$

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

```
for( g=0; g<num_window_groups; g++ ) {
  for( sfb=0; sfb < max_sfb; sfb++ ) {
    width = (swb_offset[g][sfb+1] - swb_offset[g][sfb]);
    for( win = 0; win < window_group_len[g]; win++ ) {
      for( bin=0; bin<width; bin++ ) {
        x_invquant[g][win][sfb][bin] = sign(x_quant[g][win][sfb][bin]) *
          abs(x_quant[g][win][sfb][bin])^(4/3);
      }
    }
  }
}
```