

IEEE Standard for Advanced Audio Coding

IEEE Computer Society

Sponsored by the
Standards Activities Board

IEEE
3 Park Avenue
New York, NY 10016-5997
USA

IEEE Std 1857.2™-2013

IEEE Standard for Advanced Audio Coding

Sponsor

Standards Activities Board
of the
IEEE Computer Society

Approved 23 August 2013

IEEE-SA Standards Board

Abstract: A set of tools to support specific audio coding functions including general audio coding and lossless audio coding are defined in this standard. The tool set defined in combination provides regular high-quality and efficient coding tool sets for compression, decompression, processing, and representing of audio data to save bandwidth for transmission; to save space for storage; to speed up indexing and multimedia search; and to enhance performance when using mixed media for virtual reality and other applications that demand high bandwidth. The target applications and services include but are not limited to audio transmission, audio recording, Internet streaming, and other video/audio-enabled services and applications.

Keywords: audio coding, bitplane coding, entropy coding, IEEE 1857.2™, lossless coding, LPC, PQ-SPSC, scalable coding, vector quantization

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

Copyright © 2013 by The Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 12 November 2013. Printed in the United States of America.

IEEE is a registered trademark in the U.S. Patent & Trademark Office, owned by The Institute of Electrical and Electronics Engineers, Incorporated.

PDF: ISBN 978-0-7381-8558-3 STD98328
Print: ISBN 978-0-7381-8559-0 STDPD98328

IEEE prohibits discrimination, harassment, and bullying.

For more information, visit <http://www.ieee.org/web/aboutus/whatis/policies/p9-26.html>.

No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.

Important Notices and Disclaimers Concerning IEEE Standards Documents

IEEE documents are made available for use subject to important notices and legal disclaimers. These notices and disclaimers, or a reference to this page, appear in all standards and may be found under the heading “Important Notice” or “Important Notices and Disclaimers Concerning IEEE Standards Documents.”

Notice and Disclaimer of Liability Concerning the Use of IEEE Standards Documents

IEEE Standards documents (standards, recommended practices, and guides), both full-use and trial-use, are developed within IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (“IEEE-SA”) Standards Board. IEEE (“the Institute”) develops its standards through a consensus development process, approved by the American National Standards Institute (“ANSI”), which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of the Institute and participate without compensation from IEEE. While IEEE administers the process and establishes rules to promote fairness in the consensus development process, IEEE does not independently evaluate, test, or verify the accuracy of any of the information or the soundness of any judgments contained in its standards.

IEEE does not warrant or represent the accuracy or content of the material contained in its standards, and expressly disclaims all warranties (express, implied and statutory) not included in this or any other document relating to the standard, including, but not limited to, the warranties of: merchantability; fitness for a particular purpose; non-infringement; and quality, accuracy, effectiveness, currency, or completeness of material. In addition, IEEE disclaims any and all conditions relating to: results; and workmanlike effort. IEEE standards documents are supplied “AS IS” and “WITH ALL FAULTS.”

Use of an IEEE standard is wholly voluntary. The existence of an IEEE standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard.

In publishing and making its standards available, IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity nor is IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing any IEEE Standards document, should rely upon his or her own independent judgment in the exercise of reasonable care in any given circumstances or, as appropriate, seek the advice of a competent professional in determining the appropriateness of a given IEEE standard.

IN NO EVENT SHALL IEEE BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO: PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE PUBLICATION, USE OF, OR RELIANCE UPON ANY STANDARD, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE AND REGARDLESS OF WHETHER SUCH DAMAGE WAS FORESEEABLE.

Translations

The IEEE consensus development process involves the review of documents in English only. In the event that an IEEE standard is translated, only the English version published by IEEE should be considered the approved IEEE standard.

Official statements

A statement, written or oral, that is not processed in accordance with the IEEE-SA Standards Board Operations Manual shall not be considered or inferred to be the official position of IEEE or any of its committees and shall not be considered to be, or be relied upon as, a formal position of IEEE. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position of IEEE.

Comments on standards

Comments for revision of IEEE Standards documents are welcome from any interested party, regardless of membership affiliation with IEEE. However, IEEE does not provide consulting information or advice pertaining to IEEE Standards documents. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Since IEEE standards represent a consensus of concerned interests, it is important that any responses to comments and questions also receive the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to comments or questions except in those cases where the matter has previously been addressed. For the same reason, IEEE does not respond to interpretation requests. Any person who would like to participate in revisions to an IEEE standard is welcome to join the relevant IEEE working group.

Comments on standards should be submitted to the following address:

Secretary, IEEE-SA Standards Board
445 Hoes Lane
Piscataway, NJ 08854 USA

Laws and regulations

Users of IEEE Standards documents should consult all applicable laws and regulations. Compliance with the provisions of any IEEE Standards document does not imply compliance to any applicable regulatory requirements. Implementers of the standard are responsible for observing or referring to the applicable regulatory requirements. IEEE does not, by the publication of its standards, intend to urge action that is not in compliance with applicable laws, and these documents may not be construed as doing so.

Copyrights

IEEE draft and approved standards are copyrighted by IEEE under U.S. and international copyright laws. They are made available by IEEE and are adopted for a wide variety of both public and private uses. These include both use, by reference, in laws and regulations, and use in private self-regulation, standardization, and the promotion of engineering practices and methods. By making these documents available for use and adoption by public authorities and private users, IEEE does not waive any rights in copyright to the documents.

Photocopies

Subject to payment of the appropriate fee, IEEE will grant users a limited, non-exclusive license to photocopy portions of any individual standard for company or organizational internal use or individual, non-commercial use only. To arrange for payment of licensing fees, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; +1 978 750 8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Updating of IEEE Standards documents

Users of IEEE Standards documents should be aware that these documents may be superseded at any time by the issuance of new editions or may be amended from time to time through the issuance of amendments, corrigenda, or errata. An official IEEE document at any point in time consists of the current edition of the document together with any amendments, corrigenda, or errata then in effect.

Every IEEE standard is subjected to review at least every ten years. When a document is more than ten years old and has not undergone a revision process, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE standard.

In order to determine whether a given document is the current edition and whether it has been amended through the issuance of amendments, corrigenda, or errata, visit the IEEE-SA Website at <http://ieeexplore.ieee.org/xpl/standards.jsp> or contact IEEE at the address listed previously. For more information about the IEEE SA or IEEE's standards development process, visit the IEEE-SA Website at <http://standards.ieee.org>.

Errata

Errata, if any, for all IEEE standards can be accessed on the IEEE-SA Website at the following URL: <http://standards.ieee.org/findstds/errata/index.html>. Users are encouraged to check this URL for errata periodically.

Patents

Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken by the IEEE with respect to the existence or validity of any patent rights in connection therewith. If a patent holder or patent applicant has filed a statement of assurance via an Accepted Letter of Assurance, then the statement is listed on the IEEE-SA Website at <http://standards.ieee.org/about/sasb/patcom/patents.html>. Letters of Assurance may indicate whether the Submitter is willing or unwilling to grant licenses under patent rights without compensation or under reasonable rates, with reasonable terms and conditions that are demonstrably free of any unfair discrimination to applicants desiring to obtain such licenses.

Essential Patent Claims may exist for which a Letter of Assurance has not been received. The IEEE is not responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patents Claims, or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this standard are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility. Further information may be obtained from the IEEE Standards Association.

Participants

At the time this IEEE standard was completed, the Audio Video Coding Working Group had the following membership:

Wen Gao, *Chair*
Cliff Reader, *Vice Chair*

Weibei Dou
Haibin Huang
Yichao Huang
Jing Li

Min Lu
Xingde Pan
Tianshu Qu

Haiyan Shu
Xinhui Yang
Rongshan Yu
Tao Zhang

The following members of the individual balloting committee voted on this standard. Balloters may have voted for approval, disapproval, or abstention.

Oscar Au
Jianfei Cai
Juan Carreon
Jianwen Chen
Keith Chow
John Cole
Weibei Dou
P. Eastman
Wen Gao
Matthew Goldman
Randall C. Groves
Zongming Guo
Werner Hoelzl
Ruimin Hu
Haibin Huang
Tiejun Huang

Longshe Huo
Noriyuki Ikeuchi
Akio Iso
Euee Seon Jang
Huizhu Jia
Ming Jiang
Piotr Karocki
Thomas Kurihara
Jia Li
Weiping Li
Fan Liang
Weisi Lin
Jiaying Liu
Wei Ma
Luntian Mou
Michael S. Newman

Ulrich Pohl
Bartien Sayogo
Haiyan Shu
Walter Struppler
Jiande Sun
Jun Sun
Yonghong Tian
John Vergis
David Virette
Feng Wu
Hongkai Xiong
Rongshan Yu
Oren Yuen
Xinggong Zhang
Daidi Zhong
Huan Zhou

When the IEEE-SA Standards Board approved this standard on 23 August 2013, it had the following membership:

John Kulick, *Chair*
David J. Law, *Vice Chair*
Richard H. Hulett, *Past Chair*
Konstantinos Karachalios, *Secretary*

Masayuki Ariyoshi
Peter Balma
Farooq Bari
Ted Burse
Wael William Diab
Stephen Dukes
Jean-Philippe Faure
Alexander Gelman

Mark Halpin
Gary Hoffman
Paul Houzé
Jim Hughes
Michael Janezic
Joseph L. Koepfinger*
Oleg Logvinov

Ron Petersen
Gary Robinson
Jon Walter Rosdahl
Adrian Stephens
Peter Sutherland
Yatin Trivedi
Phil Winston
Yu Yuan

*Member Emeritus

Also included are the following nonvoting IEEE-SA Standards Board liaisons:

Richard DeBlasio, *DOE Representative*
Michael Janezic, *NIST Representative*

Julie Alessi
IEEE Standards Program Manager, Document Development

Malia Zaman
IEEE Standards Program Manager, Technical Program Development

Introduction

This introduction is not part of IEEE Std 1857.2-2013, IEEE Standard for Advanced Audio Coding.

This standard provides regular high-quality and efficient coding tool sets for compression, decompression, and packaging of the multimedia data to save bandwidth for transmission on the Internet or to save space for storage.

Contents

1. Overview	1
1.1 Scope	1
1.2 Purpose	1
1.3 Audio coding tools.....	1
1.4 Document structure.....	2
2. Definitions, symbols, and abbreviations.....	3
2.1 Definitions	3
2.2 Symbols and abbreviations	6
3. General audio coding.....	10
3.1 Overview	10
3.2 Syntax	11
3.3 General information.....	19
3.4 Context-dependent Bitplane Coding.....	31
3.5 Post-quantization Square Polar Stereo Coding	41
3.6 Quantization	43
3.7 Multi-resolution synthesis	44
3.8 Linear prediction vector quantization decoding tool	46
3.9 Frequency-time domain mapping	49
3.10 Multi-channel extension	50
3.11 Lossless extension	52
4. Lossless audio coding.....	78
4.1 Overview	78
4.2 Semantics and syntax.....	78
4.3 Technical overview.....	82
4.4 Channel decorrelation.....	83
4.5 Integer lifting wavelet transform	86
4.6 Linear predictive coding.....	89
4.7 Residue preprocessing	91
4.8 Entropy coding	94
Annex A (normative) Method for keeping the syncword unique	109
Annex B (normative) AATF and AASF file formats	110
Annex C (normative) Table for general audio coding.....	120
Annex D (informative) Encoder for general audio coding	204
Annex E (normative) Fixed-point reference decoder	216

IEEE Standard for Advanced Audio Coding

IMPORTANT NOTICE: *IEEE Standards documents are not intended to ensure safety, security, health, or environmental protection, or ensure against interference with or from other devices or networks. Implementers of IEEE Standards documents are responsible for determining and complying with all appropriate safety, security, environmental, health, and interference protection practices and all applicable laws and regulations.*

This IEEE document is made available for use subject to important notices and legal disclaimers. These notices and disclaimers appear in all publications containing this document and may be found under the heading “Important Notice” or “Important Notices and Disclaimers Concerning IEEE Documents.” They can also be obtained on request from IEEE or viewed at <http://standards.ieee.org/IPR/disclaimers.html>.

1. Overview

1.1 Scope

This standard specifies the audio compression, decompression, and packaging tools and mechanism to support the transmission and storage of multimedia data over the Internet in a highly efficient way under constraints that include limited complexity and bandwidth.

1.2 Purpose

This standard provides regular high-quality and efficient coding tool sets for compression, decompression, and packaging of multimedia data to save bandwidth for transmission on the Internet or to save space for storage.

1.3 Audio coding tools

In this standard, a set of audio coding tools are defined for transmitting and decoding of recorded music and other audio signals. In this subclause, a short introduction for each coding tool is given. Detailed top level payload definitions, bitstream syntax, semantics, and the decoding process for each coding tool are specified in their corresponding subclauses.

1.3.1 General audio coding

Clause 3 describes the coding method and decoding method of high-quality audio signals. It is applied to high-resolution digital broadcasting, high-density digital storage media, wireless broadband multimedia communication, network broadband, streaming transaction.

This clause provides a mixed framework of lossy and lossless audio coding. It includes a high-quality lossy coding sub-framework and a lossless extension coding sub-framework.

The lossy coding sub-framework supports audio coding of mono, stereo, and multi-channel signals at a sampling frequency of 8 kHz ~ 96 kHz and output bitstream of 16 kb/s ~ 96 kb/s per channel. At the same time, it also supports fine granule scalability; the granule step is 1 kb/s when the coding rate is below 16 kb/s ~ 79 kb/s per channel, and slightly larger than 1 kb/s when the coding rate is 80 kb/s ~ 96 kb/s per channel. In this sub-framework, time-frequency transform, frequency linear prediction and vector quantization, multi-resolution analysis, post-quantization square polar stereo coding, and context-dependent bitplane coding are adopted.

The lossless extension coding sub-framework realizes lossless coding of the residual signal and achieves a lossless framework that is compatible with the lossy coding bitstream. In this sub-framework, channel Decorrelation, integer lifting wavelet, linear prediction, residuals handling, and arithmetic entropy coding are adopted.

1.3.2 Lossless audio coding

Clause 4 describes the lossless audio coding algorithm. It defines an efficient lossless compression technology that can be used for digital media storage, Internet wideband audio, and digital audio broadcasting areas.

Lossless audio coding supports multiple channels up to 32 channels, sampling frequency up to 16 MHz, and multi-bit resolutions: 8 bit, 16 bit, and 24 bit. It also supports random access.

1.4 Document structure

The structure of this document is as follows:

Clause 1: Overview

Clause 2: Definitions, symbols, and abbreviations

Clause 3: General audio coding

Clause 4: Lossless audio coding

Annex A (normative) Method for keeping the sync word unique

Annex B (normative) Audio Video Standard (AVS) Audio Transport Format (AATF) and AVS Audio Storage Format (AASF) file formats

Annex C (normative) Table for general audio coding

Annex D (informative) Encoder for general audio coding

Annex E (normative) Fixed-point reference decoder

2. Definitions, symbols, and abbreviations

2.1 Definitions

For the purposes of this document, the following terms and definitions apply. The *IEEE Standards Dictionary Online* should be consulted for terms not defined in this clause.¹

AASF: Audio Video Standard (AVS) Audio Storage Format.

AATF: Audio Video Standard (AVS) Audio Transport Format.

analysis filterbank: Filterbank in the encoder that transforms a broadband pulse-code modulation (PCM) audio signal into a set of spectral coefficients.

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

audio buffer: A buffer in the system target decoder for storage of compressed audio data.

bitrate: The number of bits that are conveyed or processed to the decoder per unit of time.

bitstream: An ordered series of bits that forms the coded representation of the data.

byte alignment: A bit in a coded bitstream is byte-aligned if its position is a multiple of 8 bits from the first bit in the stream.

byte: Sequence of 8 bits.

CBC band: A basic coding unit of Context-dependent Bitplane Coding (CBC) that contains 32 spectral coefficients.

CBC base layer: The base layer is the coding layer with the lowest quality in the Context-dependent Bitplane Coding (CBC). Each frame has one base layer that contains several sublayers.

CBC bitplane vector: Each bitplane vector contains 4 bits, obtained by Huffman coding of four adjacent spectral coefficients.

CBC enhancement layer: The enhancement layer is the coding layer used to enhance the quality of the base layer in the Context-dependent Bitplane Coding (CBC). Each frame has several sublayers according to the target coding bitrate.

CBC sublayer: Each sublayer contains one or more coding band of Context-dependent Bitplane Coding (CBC).

channel: A sequence of data representing an audio signal intended to be reproduced at one listening position.

channel configuration index: A 4-bit field that indicates the channel configuration of audio output. Each index specifies the number and mapping mode of channels in the current data block.

coded audio bitstream: A coded representation of an audio signal.

¹IEEE Standards Dictionary Online subscription is available at:
http://www.ieee.org/portal/innovate/products/standard/standards_dictionary.html.

coded presentation: A data element as represented in its encoded form.

coding: The process to read the audio signal and generate bitstream.

Context-dependent Bitplane Coding (CBC): An entropy coding method that exploits the context information in the bit layers.

data element: Unit of data that has precise meaning or precise semantics.

decoded stream: The decoded reconstruction of a compressed bitstream.

decoder: An embodiment of a decoding process.

decoding: The process to read the bitstream and generate the audio samples.

encoder: An embodiment of a coding process.

entropy coding: Variable-length lossless coding of the digital representation of a signal that reduces statistical redundancy.

Fast Fourier Transform (FFT): A fast algorithm for performing a discrete Fourier transform (an orthogonal transform).

filterbank: A set of band-pass filters covering the entire audio frequency range.

FLPVQ: Frequency-domain Linear Prediction and Vector Quantization.

frequency-time transform: Transformation of the signal from frequency domain to time domain.

Hann window: A time function applied sample-by-sample to a block of audio samples before Fourier transformation.

IFLPVQ: Inverse Frequency-domain Linear Prediction and Vector Quantization.

IMDCT: Inverse modified discrete cosine transform.

inter-channel decorrelation: Decorrelation between channels to remove the redundancy.

LPC: Linear predictive coding.

LSB: Least significant bit.

MDCT: Modified discrete cosine transform.

modulation window: A data structure of spectral coefficients that is reorganized by the modulation bands after the processing of spectral coefficients. Modulation windows can be sorted as a windows sequence ordered by modulation.

MSB: Most significant bit.

multi-resolution filtering: A multi-resolution time-frequency domain mapping technique based on multirate sampling filter.

Nyquist sampling: Sampling at or above twice the maximum bandwidth of a signal.

Parcor coefficients: Parcor representation of the predictor coefficients.

reserved: Some special syntactic elements that may be used in the future for extensions of this part. All the reserved values should be set to “1” unless specifically stated.

sampling frequency (fs): The number of samples per unit of time taken from a continuous audio signal to make a discrete audio signal.

scalability: An ability of a decoder that decodes ordered sets of bitstream to reconstruct audio streams.

scalefactor band: A set of spectral coefficients that are scaled by one scalefactor.

scalefactor index: A numerical code for a scalefactor.

scalefactor: Factor by which a set of values are scaled before quantization.

side information: Information in the bitstream necessary for controlling the decoding.

spectral coefficient: Discrete frequency domain data output from the analysis filterbank.

spectral envelope: The frequency-amplitude curve of the signal after a time-frequency transform.

square polar stereo code: A stereo coding method that eliminates correlation of the spectral signal of left and right channels by matrix transformation.

square polar stereo mode: An audio coding method using the square polar stereo code algorithm.

steady signal: A type of signal whose energy and other characteristics change slowly over time. It is analyzed by long window sequences.

stuffing: Code words that may be inserted into the coded bitstream and are to be discarded in the decoding process. Their purpose is to increase the bitrate of the stream that would otherwise be lower than the desired bitrate. Beyond that, the additional coding data can use these bytes or bits as well.

syncword: Data of 12 bits embedded in an audio bitstream that indicates the start of a frame.

synthesis filterbank: Filterbank in the decoder that reconstructs a pulse-code modulation (PCM) audio signal from subband samples.

temporal stability two-level decision: A two-level decision method used to describe the temporal stability of the signal time domain and to decide whether frequency-domain linear prediction and vector quantization (FLPQV) is applied.

time-frequency transform: Transformation of the signal from time domain to frequency domain.

transient signal: A type of signal whose energy and other characteristics change rapidly over time. It is analyzed by short window sequences.

variable length code (VLC): A code word assigned by a variable length encoder *See: variable length coding.*

variable length coding: A reversible coding procedure that assigns shorter code words to frequent symbols and longer code words to less frequent symbols.

variable length decoder: A procedure to obtain the symbols encoded with a variable length coding technique.

variable length encoder: A procedure to assign variable length code (VLC) words to symbols.

2.2 Symbols and abbreviations

2.2.1 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, 6/4 and −6/−4 are truncated to 1, and −6/4 and 6/−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 toward $-\infty$.
	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
$\sqrt{}$	Square root
lg	Common (Briggsian) logarithm ($\log_{10}$)
ln	Natural (Napierian) logarithm ($\log_e$)
lb	Binary logarithm ($\log_2$)
ceil()	Ceiling operator. Returns the smallest integer that is greater than or equal to the real-valued argument.

2.2.2 Logical operators

	Logical OR
&&	Logical AND
!	Logical NOT

2.2.3 Relational operators

>	Greater than
$\geq$	Greater than or equal to
<	Less than
$\leq$	Less than or equal to
==	Equal to
$\neq$	Not equal to
max [...]	Maximum value in the argument list.
min [...]	Minimum value in the argument list.

2.2.4 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

2.2.5 Assignment

= Assignment operator.

2.2.6 Mnemonics

The following mnemonics are defined to describe the different data types used in the coded bitstream payload.

bslbf Bit string, left bit first, where “left” is the order in which bit strings are written in ISO/IEC 14496. 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.

vlclbf 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 $\text{block_type} = 2$, $0 \leq \text{window} \leq 2$. (Audio)

The byte order of multi-byte words is most significant byte first.

2.2.7 Constants

π 3.14159265358...

e 2.71828182845...

2.2.8 Method of describing bitstream payload syntax

The bitstream payload retrieved by the decoder is described in the syntax subclause of each clause. Each data item in the bitstream payload is in bold type.

It is described by the following:

- Its name.
- Its length in bits, where “X.Y” indicates that the number of bits is one of the values between X and Y including X and Y. “{X;Y}” means the number of bits is X or Y, depending on the value of other data elements in the bitstream payload.
- A mnemonic for its type and order of transmission.

Data elements forming a logic unit are encapsulated in data functions.

```
data_function ();           Data function call.

data_function () {         Data function entity.
    ...
}
```

The action caused by a decoded data element in a bitstream payload 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 the subclauses following in the syntax subclause of each clause. 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
    data_element           next in the data stream. This repeats until the condition is not
    ...                   true.
}

do {                      The data element always occurs at least once. The data element is
    data_element           repeated until the condition is not true.
    ...
} while ( condition )

if (condition) {           If the condition is true, then the first group of data elements
    data_element           occurs next in the data stream.
    ...
}
else {                    If the condition is not true, then the second group of data
    data_element           elements occurs next in the data stream.
    ...
}

switch (expression) {     If the condition formed by the comparison of expression and
    case const-expr:       const-expr is true, then the data stream continues with the
        data_element;      subsequent data elements. An optionally break statement can be
        break;             used to immediately leave the switch, data elements beyond a
    case const-expr:       break do not occur in the data stream.
        data_element;
    }

for (expr1; expr2; expr3) { Expr1 is an expression specifying the initialization of the loop.
    data_element           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.
```

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

<pre>for (i = 0; i < n; i++) { data_element . . . }</pre>	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.
---	---

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 $n+1^{\text{th}}$ element of an array of data.
data_element [m][n]	data_element [m][n] is the $m+1$, $n+1^{\text{th}}$ element of a two-dimensional array of data.
data_element [l][m][n]	data_element [l][m][n] is the $l+1$, $m+1$, $n+1^{\text{th}}$ 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 this implements a satisfactory decoding procedure. In particular, it defines a correct and error-free input bitstream payload. Actual decoders shall include a means to deal with incorrect bitstream payloads and to find the start of the described elements.

2.2.9 Arithmetic data types

INT32	32-bit signed integer using two's complement
INT64	64-bit signed integer using two's complement

3. General audio coding

3.1 Overview

Clause 3 describes the coding method and decoding method of a high-quality audio signal. It is applied to high-resolution digital broadcasting, high-density digital storage media, wireless broadband multimedia communication, and network broadband streaming transactions.

This clause provides a mixed framework of lossy and lossless audio coding. It includes a high-quality lossy coding sub-framework and a lossless extension coding sub-framework.

The lossy coding sub-framework supports audio coding of mono, stereo, and multi-channel signals at a sampling frequency of 8 kHz ~ 96 kHz and an output bitstream of 16 kb/s ~ 96kb/s per channel. At the same time, it also supports fine granule scalability; the granule step is 1 kb/s when the coding rate is below

16 kb/s ~ 79 kb/s per channel, and slightly larger than 1 kb/s when the coding rate is 80 kb/s ~ 96 kb/s per channel. In this sub-framework, time-frequency transform, frequency linear prediction and vector quantization, multi-resolution analysis, post-quantization square polar stereo coding, and context-dependent bitplane coding are adopted.

The lossless extension coding sub-framework realizes lossless coding of the residual signal and achieves a lossless framework that is compatible with the lossy coding bitstream. In this sub-framework, channel decorrelation, integer lifting wavelet, linear prediction, residuals handling, and arithmetic entropy coding are adopted.

3.2 Syntax

3.2.1 Profile definition

Element	Description / values
coding_profile	2 bits, format of the bitstream: 00 – original lossy bistream 01 – lossy to lossless extension 10 – lossless only bitstream 11 – Reserved
sampling_frequency_index	Support 0x0 – 0xb

In this subclause, the following coding profiles are defined:

- a) Lossy profile (coding_profile = 00)
- b) Lossy to lossless profile (coding_profile = 01)
- c) Lossless only profile (coding_profile = 10)
- d) Reserved (coding_profile = 11)

Note that the sampling_frequency_index from “0xc-0xe” are supported in lossless only profile.

3.2.2 Raw data

A raw data block can be decoded directly. In original data blocks, each frame contains a basic layer and can contain a number of enhanced layers according to objective encoding rate. The syntax of ga_raw_data_block() is shown in Table 3.1.

Table 3.1—Syntax of ga_raw_data_block()

Syntax	No. of bits	Mnemonic
ga_raw_data_block()		
{		
if (2 !=coding_profile) {		
cbc_base_element();		
layer = slayer_size;		
while (data_available () && layer<(top_layer+slayer_size)) {		uimbf
cbc_layer_element(layer);		
layer++;		
}		
byte_alignment();		
if (data_available()) {		
while(data_available())		
extended_cbc_raw_data_block();		
}		
if (fill_enable) {		
if (1==coding_profile) {		
lossless_data_block();		
}		
else {		
for(i=0; i<fill_ele_number+1; i++)		
fill_element(i);		
}		
}		
else {		
lossless_data_length;	16	uimsbf
lossless_data_block();		
}		
}		

Syntax of `cbc_base_element()` is shown in Table 3.2.

Table 3.2—Syntax of `cbc_base_element()`

Syntax	No. of bits	Mnemonic
<code>cbc_base_element()</code>		
{		
<code>frame_length_data();</code>		
<code>mc_present;</code>	1	bslbf
if (<code>mc_present</code>)		
<code>element_length_data_FLFR();</code>		
<code>cbc_scalable_header();</code>		
<code>cbc_general_header(0);</code>		
<code>flpvq_data();</code>		
<code>byte_alignment();</code>		
for (<code>slayer = 0; slayer < slayer_size; slayer++</code>)		
<code>cbc_layer_element(slayer);</code>		
}		

Syntax of `extended_cbc_raw_data_block()` is shown in Table 3.3.

Table 3.3—Syntax of `extended_cbc_raw_data_block()`

Syntax	No. of bits	Mnemonic
<code>extended_cbc_raw_data_block()</code>		
{		
<code>extended_cbc_base_element();</code>		
<code>layer=slayer_size;</code>		
while(<code>data_available()</code> && <code>layer<(top_layer+slayer_size)</code>) {		
<code>cbc_layer_element(layer);</code>		
<code>layer++;</code>		
}		
<code>byte_alignment();</code>		
}		

Syntax of `extended_cbc_base_element()` is shown in Table 3.4.

Table 3.4—Syntax of `extended_cbc_base_element()`

Syntax	No. of bits	Mnemonic
<code>extended_cbc_base_element()</code>		
{		
<code>element_length_data()</code>		uimbf
<code>channel_configuration_index</code>	4	bslbf
<code>scalable_header_reuse</code>	1	bslbf
<code>reserved_bit</code>	1	bslbf
if (! <code>scalable_header_reuse</code>)		
<code>cbc_scalable_header();</code>		
<code>cbc_general_header(1);</code>		
<code>flpvq_data();</code>		
<code>byte_alignment();</code>		
for (<code>slayer = 0; slayer < slayer_size; slayer++</code>)		
<code>cbc_layer_element(slayer);</code>		
}		

Syntax of `element_length_data()` is shown in Table 3.5.

Table 3.5—Syntax of `element_length_data()`

Syntax	No. of bits	Mnemonic
<code>element_length_data()</code>		
{		
base_element_length	7	uimbf
length_end_flag	1	bslbf
while (<code>length_end_flag</code>) {		
chanc_element_length	3	uimbf
length_end_flag	1	bslbf
}		
}		

Syntax of `cbc_scalable_header()` is shown in Table 3.6.

Table 3.6—Syntax of `cbc_scalable_header()`

Syntax	No. of bits	Mnemonic
<code>cbc_scalable_header()</code>		
{		
top_layer;	6	uimbf
base_snf_thr;	2	uimbf
base_band;	5	uimbf
}		

Syntax of `cbc_general_header()` is shown in Table 3.7.

Table 3.7—Syntax of `cbc_general_header()`

Syntax	No. of bits	Mnemonic
<code>cbc_general_header(ext_flag)</code>		
{		
reserved_bit;	1	bslbf
for (<code>ch = 0; ch < nch; ch++</code>)		
scalefactor[ch][0];	8	uimbf
signal_type;	1	
window_shape;	1	uimsbf
if (<code>signal_type == TRANSIENT_TYPE</code>) {		
max_sfb;	4	uimsbf
scale_factor_grouping;	7	uimsbf
coding_band_mode;	1	uimsbf
} else {		
max_sfb;	6	uimsbf
}		
if (<code>nch == 2</code>)		
sc_present;	2	bslbf

Table 3.7—Syntax of cbc_general_header() (continued)

Syntax	No. of bits	Mnemonic
if (!ext_flag) {		
fill_enable	1	uimsbf
if (fill_enable) {		
fill_ele_number	7	uimsbf
for(i=0;i<fill_ele_number+1;i++){		
fill_cnt[i] = fill_length[i] ;	4	uimsbf
if (fill_length[i]==15)		
fill_cnt[i] += fill_esc_length[i] ;	8	uimsbf
}		
}		
}		

Syntax of flpvq_data() is shown in Table 3.8.

Table 3.8—Syntax of flpvq_data()

Syntax	No. of bits	Mnemonic
flpvq_data()		
{		
for (ch = 0; ch < nch; ch++) {		
bs_flp_present[ch] ;	1	uimsbf
if (bs_flp_present[ch]){		
bs_pred_direction[ch] ;	1	uimsbf
bs_vqcode1indx[ch] ;	10	uimsbf
bs_vqcode2indx[ch] ;	10	uimsbf
bs_vqcode3indx[ch] ;	9	uimsbf
bs_vqcode4indx[ch] ;	8	uimsbf
}		
}		
}		

Syntax of cbc_layer_element() is shown in Table 3.9.

Table 3.9—Syntax of cbc_layer_element()

Syntax	No. of bits	Mnemonic
cbc_layer_element(layer)		
{		
layer_cband_si(layer);		
layer_sfb_si(layer);		
cbc_layer_spectra(layer);		
cbc_lower_spectra(layer);		
}		

Syntax of layer_cband_si() is shown in Table 3.10.

Table 3.10—Syntax of layer_cband_si()

Syntax	No. of bits	Mnemonic
layer_cband_si(layer)		
{		
g = layer_group[layer];		
for (ch = 0; ch < nch; ch++) {		
for (cband = layer_start_cband[layer];		
cband < layer_end_cband[layer]; cband++) {		
hcode_cband_si[ch][g][cband];	1..18	bslbf
if (hcode_cband_si[ch][g][cband] == 0x329C8)		
cband_si_escape[ch][g][cband];	4	bslbf
}		
}		
}		

Syntax of layer_sfb_si() is shown in Table 3.11.

Table 3.11—Syntax of layer_sfb_si()

Syntax	No. of bits	Mnemonic
layer_sfb_si(layer)		
{		
g = layer_group[layer];		
for (ch = 0; ch < nch; ch++)		
for (sfb = layer_start_sfb[layer]; sfb < layer_end_sfb[layer]; sfb++) {		
if (stereo_side_info_coded[g][sfb] == 0) {		
if (sc_present != 2) {		
if (sc_present == 1) {		
hcode_sc_used[g][sfb];	1	bslbf
} else if (sc_present == 3) {		
}		
}		
stereo_side_info_coded[g][sfb] = 1;		
}		
hcode_scf_index[ch][g][sfb];	1..19	bslbf
}		
}		

Syntax of cbc_layer_spectra() is shown in Table 3.12.

Table 3.12—Syntax of cbc_layer_spectra()

Syntax	No. of bits	Mnemonic
cbc_layer_spectra(layer)		
{		
g = layer_group[layer];		
start_index[g] = layer_start_index[layer];		
end_index[g] = layer_end_index[layer];		
if (layer < slayer_size)		
thr_snf = base_snf_thr;		
else		

Table 3.12 Syntax of cbc_layer_spectra() (continued)

Syntax	No. of bits	Mnemonic
thr_snf = 0;		
cbc_spectral_data (g, g+1, thr_snf, cur_snf);		
}		

Syntax of cbc_lower_spectra() is shown in Table 3.13.

Table 3.13—Syntax of cbc_lower_spectra()

Syntax	No. of bits	Mnemonic
cbc_lower_spectra(layer)		
{		
for (g = 0; g < num_window_groups; g++) {		
start_index[g] = 0;		
end_index[g] = 0;		
}		
for (play = 0; play < layer; play++) {		
end_index[layer_group[play]] = layer_end_index[play];		
}		
cbc_spectral_data (0, num_window_groups, 0, unc_snf);		
}		

Syntax of cbc_spectral_data() is shown in Table 3.14.

Table 3.14—Syntax of cbc_spectral_data ()

Syntax	No. of bits	Mnemonic
cbc_spectral_data(start_g, end_g, thr_snf, cur_snf)		
{		
if (!layer_data_available()) return;		
for (snf = maxsnf; snf > thr_snf; snf--)		
for (g = start_g; g < end_g; g++)		
for (i = start_index[g]; i < end_index[g]; i++)		
for (ch = 0; ch < nch; ch++) {		
if (cur_snf[ch][g][i] < snf) continue;		
if (!sample[ch][g][i] sign_is_coded[ch][g][i]) {		
model_select();		
upper_vector_mapping();		
hcod_sliced_bit[ch][g][i][snf];	1..13	bslbf
}		
if (sample[ch][g][i] && !sign_is_coded[ch][g][i]) {		
if (layer_data_available()) return;		
hcod_sign[ch][g][i];	1	bslbf
sign_is_coded[ch][g][i] = 1;		
}		
cur_snf[ch][g][i]--;		
if (!layer_data_available()) return;		
}		
}		

Syntax of `fill_element()` is shown in Table 3.15.

Table 3.15—Syntax of `fill_element()`

Syntax	No. of bits	Mnemonic
<code>fill_element(fill_idx)</code>		
{		
fill_type	4	uimsbf
switch(fill_type) {		
case FILL_DFT:		
for(i=0;i<8*(fill_cnt[fill_idx]-1)+4;i++)		
fill_dft_bits[i]	1	uimsbf
}		
}		

Syntax of `frame_length_data()` is shown in Table 3.16.

Table 3.16—Syntax of `frame_length_data()`

Syntax	No. of bits	Mnemonic
<code>frame_length_data()</code>		
{		
base_frame_length;	7	uimbf
length_end_flag;	1	bslbf
enhance_length_cnt = 0;		
while (length_end_flag) {		
enhance_frame_length[enhance_length_cnt++];	3	uimbf
length_end_flag;	1	bslbf
}		
}		

Syntax of `element_length_data_FLFR()` is shown in Table 3.17.

Table 3.17—Syntax of `element_length_data_FLFR()`

Syntax	No. of bits	Mnemonic
<code>element_length_data_FLFR()</code>		
{		
base_element_length_FLFR	7	uimbf
length_end_flag	1	bslbf
while (length_end_flag) {		
ehanc_element_length_FLFR	3	uimbf
length_end_flag	1	bslbf
}		
}		

3.3 General information

3.3.1 Introduction

The AVS audio coder and decoder make use of a number of required and optional modules. Table 3.18 lists the modules and their status such as *required* or *optional*. Required modules are mandatory in any possible profile. Optional modules may not be required in some profiles.

Table 3.18—Modules

Module name	Required / optional
Bitstream Deformatter	Required
Context-dependent Bitplane Decoding	Required
Inverse Quantization	Required
Post-Quantization Domain Square Polar Stereo Decoding	Optional
Multi-resolution Synthesis	Optional
Inverse Frequency-domain Linear Prediction and Vector Quantization	Optional
Inverse Time-Frequency Transform	Required

The basic structure of the AVS audio system is shown in Figure 3-1 and Figure 3-2. As is shown in Table 3.18, there are both required and optional modules in the decoder; refer to Figure 3-2. 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 modules are active in the bitstream in order to arrive at the actual signal spectra as described by the input bitstream, and finally convert the reconstructed spectral coefficients from frequency domain to the time domain.

The input to the Bitstream Deformatter Module is the AVS audio data stream. The module separates each part of the AVS audio data stream into the parts for each module and provides each of the modules with the decoding information related to that module.

The input to the Bitstream Deformatter is: AVS audio coded stream.

The outputs from the Bitstream Deformatter Module are as follows:

- The noiselessly coded spectra
- Context-dependent bitplane decoding lossless decoding control information
- Quantization domain square polar stereo decoding control information
- Signal type information
- Frequency domain linear predictive and vector quantization control information

The Context-dependent Bitplane Decoding Module takes information from the bitstream deformatter, decodes the Huffman coded data, and reconstructs the quantized spectra, mode information of coding band and scalefactor.

The inputs to the Context-dependent Bitplane Decoding Module are as follows:

- The noiselessly coded mode information of coding bands
- The noiselessly coded scalefactors
- The noiselessly coded spectra

The output(s) from the Context-dependent Bitplane Decoding Module is (are) as follows:

- The mode of coding band
- The quantized values for the spectra

The Post-Quantization Square Polar Stereo Decoding Module implements stereo decoding on the square polar of spectra.

The inputs to the Post-Quantization Square Polar Stereo Decoding Module are as follows:

- The quantized values for the spectra
- The square polar stereo and the coupling control information
- The scalefactors after context-dependent bitplane decoding

The output(s) from the Post-Quantization Square Polar Stereo Decoding Module is (are) as follows:

- The quantized spectra after post-quantization square polar stereo decoding
- The scalefactors after post-quantization square polar stereo decoding

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

The inputs to the Inverse Quantization Module are as follows:

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

The output(s) of the Inverse Quantization tool is (are) as follows:

- The scaled, inversely quantized spectra

The Multi-Resolution Synthesis Module implements multi-resolution processing on audio spectral coefficient of transient signal.

The inputs to the Multi-Resolution Synthesis Module are as follows:

- The inversely quantized spectra / the residual spectra

The output(s) from the Multi-Resolution Synthesis Module is (are) as follows:

- The reconstructed spectra / the residual spectra

Frequency-domain Linear Prediction and Vector Quantization (FLPVQ) Module used Frequency-domain Linear Prediction technique and Vector Quantization technique to improve the coding efficiency effectively.

The inputs to the Inverse Frequency-domain Linear Prediction and Vector Quantization (IFLPVQ) Module are as follows:

- The residual spectra

- The vector indices
- The control information

The output(s) from the IFLPVQ Module is (are) as follows:

- Modified discrete cosine transform (MDCT) spectral coefficients

The Inverse Time-Frequency Transform Module applies the inverse of the time-frequency mapping that was carried out in the encoder. An inverse modified discrete cosine transform (IMDCT) is used for this module; the spectral structure of the MDCT coefficients with an input of 1024 samples depends on the window sequence that is in use.

The inputs to the Inverse Time-Frequency Transform Module are as follows:

- The inversely quantized spectra coefficients

The output(s) from the Inverse Time-Frequency Transform Module is (are) as follows:

- The time domain reconstructed audio data

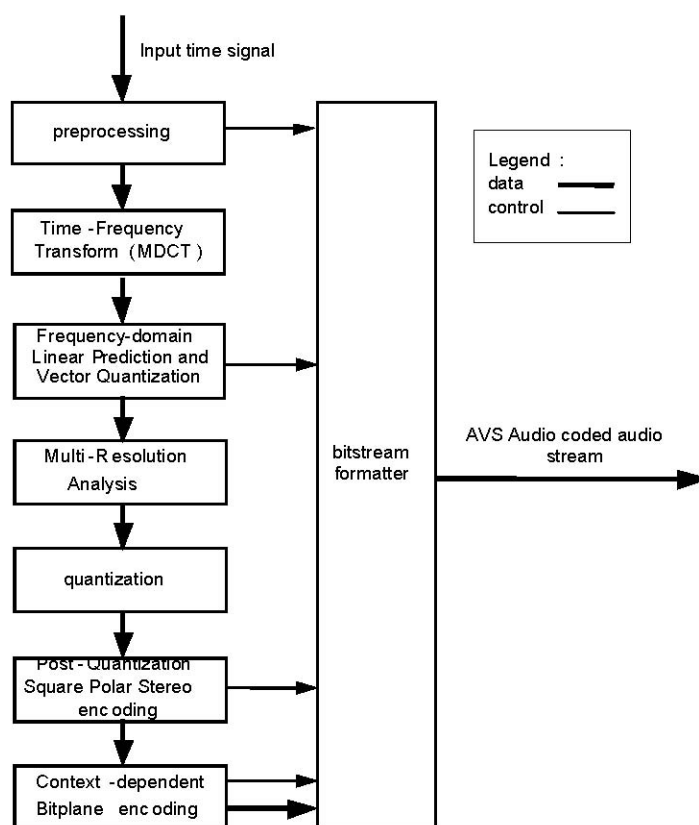


Figure 3-1—AVS audio encoder block diagram

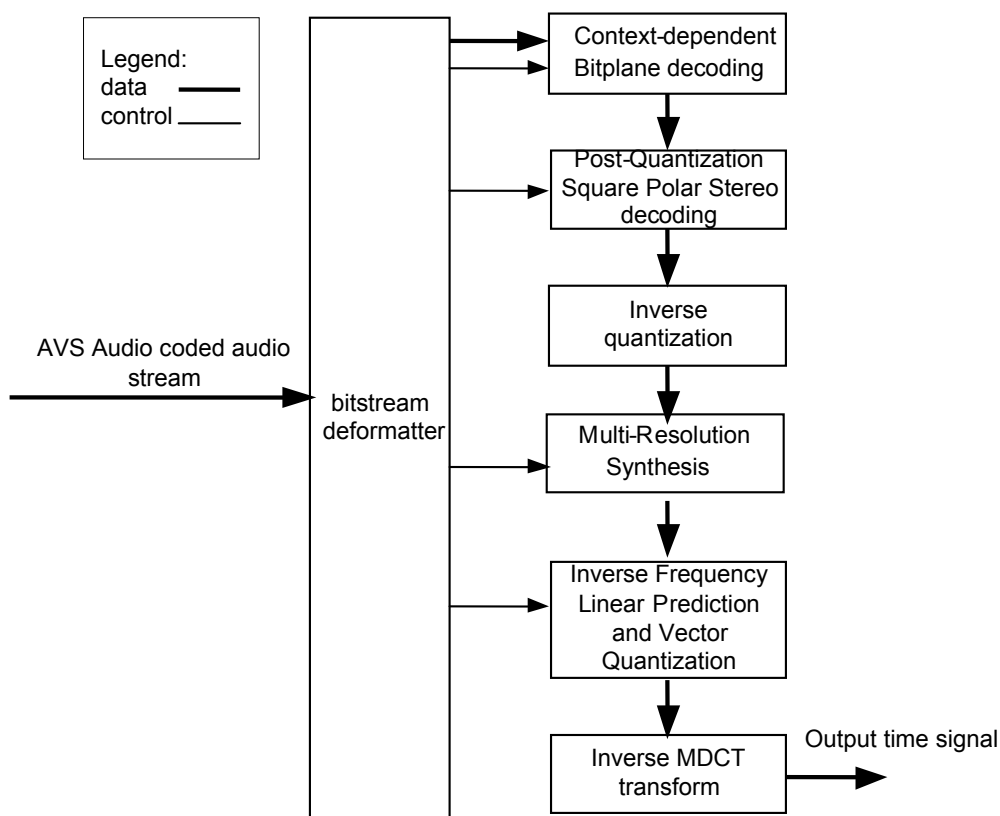


Figure 3-2—AVS audio decoder block diagram

3.3.2 Decoding of raw data

3.3.2.1 Definitions

`ga_raw_data_block ()` Block of raw data that contains 1024 samples of audio data, related information, and other data.

3.3.2.2 Decoding process

Assuming that the start of a `raw_data_block` is known, it can be decoded without any additional transport information to produce 1024 audio samples per output channel. The sampling rate of the audio signal may be specified by an index of sampling frequency (`sampling_frequency_index`).

3.3.2.3 Grouping and interleaving

The spectral coefficients are constructed to form the modulation window sequence after multi-resolution analysis in frequency domain. Each modulation window sequence is divided into different scalefactor bands in the coding process; thus optimal grouping and interleaving can be used to achieve greater coding efficiency.

The coefficients can be grouped such that they share scalefactors among all scalefactor bands within the group. In addition, the coefficients within a group are interleaved by interchanging the order of scalefactor bands and windows, thus providing for greater coding efficiency. To be specific, assume that before interleaving the set of 1024 coefficients c are indexed as:

$$c[g][w][k/4][k\%4]$$

where

g is the index on groups

w is the index on windows within a group

k is the index on coefficients within a window

After interleaving, the coefficients are indexed as $c[g][k/4][w][k\%4]$.

3.3.2.4 Table of scalefactor bands

Table 3.19—Scalefactor bands of STATIONARY_TYPE at 44.1 kHz and 48 kHz

fs [kHz]	44.1, 48
num_swb_station_type	49
swb	swb_offset_station_type
0	0
1	4
2	8
3	12
4	16
5	20
6	24
7	28
8	32
9	36
10	40
11	48
12	56
13	64
14	72
15	80
16	88
17	96
18	108
19	120
20	132
21	144
22	160
23	176
24	196

swb	swb_offset_station_type
25	216
26	240
27	264
28	292
29	320
30	352
31	384
32	416
33	448
34	480
35	512
36	544
37	576
38	608
39	640
40	672
41	704
42	736
43	768
44	800
45	832
46	864
47	896
48	928
	1024

Table 3.20—Scalefactor bands of TRANSIENT_TYPE at 32 kHz, 44.1 kHz, and 48 kHz

fs [kHz]	32, 44.1, 48
num_swb_transient_type	14
swb	swb_offset_transient_type
0	0
1	4
2	8
3	12
4	16
5	20
6	28
7	36

swb	swb_offset_transient_type
8	44
9	56
10	68
11	80
12	96
13	112
	128

Table 3.21—Scalefactor bands of STATIONARY_TYPE at 32 kHz

fs [kHz]	32
num_swb_station_type	51
swb	swb_offset_station_type
0	0
1	4
2	8
3	12
4	16
5	20
6	24
7	28
8	32
9	36
10	40
11	48
12	56
13	64
14	72
15	80
16	88
17	96
18	108
19	120
20	132
21	144
22	160
23	176
24	196
25	216

swb	swb_offset_station_type
26	240
27	264
28	292
29	320
30	352
31	384
32	416
33	448
34	480
35	512
36	544
37	576
38	608
39	640
40	672
41	704
42	736
43	768
44	800
45	832
46	864
47	896
48	928
49	960
50	992
	1024

Table 3.22—Scalefactor bands of STATIONARY_TYPE at 8 kHz

fs [kHz]	8		
num_swb_station_type	40		
swb	swb_offset_station_type	swb	swb_offset_station_type
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 3.23—Scalefactor bands of TRANSIENT_TYPE at 8 kHz

fs [kHz]	8		
num_swb_transient_type	15		
swb	swb_offset_transient_type	swb	swb_offset_transient_type
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 3.24—Scalefactor bands of STATIONARY_TYPE at 11.025 kHz, 12 kHz, and 16 kHz

fs [kHz]	11.025, 12, 16		
num_swb_station_type	43		
swb	swb_offset_station_type	swb	swb_offset_station_type
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 3.25—Scalefactor bands of TRANSIENT_TYPE at 11.025 kHz, 12 kHz, and 16 kHz

fs [kHz]	11.025, 12, 16		
num_swb_transient_type	15		
swb	swb_offset_transient_type	swb	swb_offset_transient_type
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 3.26—Scalefactor bands of STATIONARY_TYPE at 22.05 kHz and 24 kHz

fs [kHz]	22.05 and 24
num_swb_station_type	47
swb	swb_offset_station_type
0	0
1	4
2	8
3	12
4	16
5	20
6	24
7	28
8	32
9	36
10	40
11	44
12	52
13	60
14	68
15	76
16	84
17	92
18	100
19	108
20	116
21	124
22	136
23	148

swb	swb_offset_station_type
24	160
25	172
26	188
27	204
28	220
29	240
30	260
31	284
32	308
33	336
34	364
35	396
36	432
37	468
38	508
39	552
40	600
41	652
42	704
43	768
44	832
45	896
46	960
	1024

Table 3.27—Scalefactor bands of TRANSIENT_TYPE at 22.05 kHz and 24 kHz

fs [kHz]	22.05 and 24
num_swb_transient_type	15
swb	swb_offset_transient_type
0	0
1	4
2	8
3	12
4	16
5	20
6	24
7	28

swb	swb_offset_transient_type
8	36
9	44
10	52
11	64
12	76
13	92
14	108
	128

Table 3.28—Scalefactor bands of STATIONARY_TYPE at 64 kHz

fs [kHz]	64		
num_swb_station_type	47		
swb	swb_offset_station_type	swb	swb_offset_station_type
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 3.29—Scalefactor bands of TRANSIENT_TYPE at 64 kHz

fs [kHz]	64		
num_swb_transient_type	12		
swb	swb_offset_transient_type	swb	swb_offset_transient_type
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 3.30—Scalefactor bands of STATIONARY_TYPE at 88.2 kHz and 96 kHz

fs [kHz]	88.2 and 96		
num_swb_station_type	41		
swb	swb_offset_station_type	swb	swb_offset_station_type
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 3.31 Scalefactor bands of TRANSIENT_TYPE at 88.2 kHz and 96 kHz

fs [kHz]	88.2 and 96		
num_swb_transient_type	12		
swb	swb_offset_transient_type	swb	swb_offset_transient_type
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		

3.3.2.5 Figure for window grouping

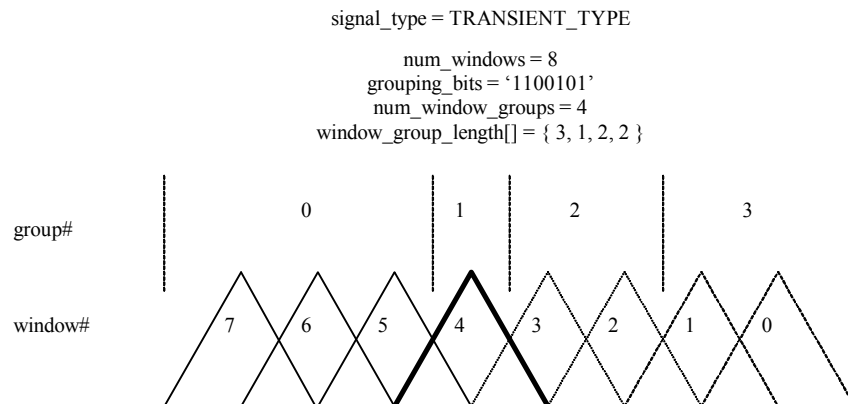


Figure 3-3—Example for window grouping

3.3.3 Decoding of fill data

3.3.3.1 Definitions

fill_element ()	Fill data block, which is located by the byte alignment position of raw audio data. It is used to increase the bitrate or fill the additional audio coding data.
fill_enable	A 1-bit field indicating the enable bit of fill element in cbc_general_header.
fill_ele_number	A 7-bit field indicating the number of fill_element, i.e., fill_ele_number+1.
fill_length	A 4-bit field indicating the initial value for length of fill element in bytes.
fill_esc_length	An 8-bit field indicating the incremental value of length of fill element in bytes when the length is more than 15 bytes.
fill_type	A 4-bit field indicating the type of fill element content.
fill_dft_bits	A 1-bit field indicating the default bit of fill element.

3.3.3.2 Decoding process

If the value of fill_enable is 1, the total number of fill elements can be decoded from cbc_general_header and the length of each fill element can be decoded as well. So the fill_element can be decoded after aligning the decoded bytes of raw audio data.

If the length of fill element is less than 15, then the value of fill_length indicates the length of fill element. Otherwise, the length of fill element is fill_length+fill_esc_length.

The type of fill element is given by fill_type defined as Table 3.32. If Fill_type is 0, then FILL_DFT is used to fill the bitstream. Other values of Fill_type are reserved.

Table 3.32—Definition of fill_type

Symbol	Value of fill_type	Usage
FILL_DFT	'0000'	Fill bitstream
—	Other value	Reserved

3.3.4 Buffer requirements

The size of a minimum decoder buffer for every independent channel with full bandwidth is 6144 bits. The number of bits used to the low-frequency effects (LFE) channel and fill element is given by main buffer.

The bit reservoir is controlled at the encoder. The maximum bit reservoir in the encoder depends on the number of independent channels with full bandwidth 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 buffer size. For example, at 96 kbit/s for a stereo signal at 48 kHz sampling frequency, the maximum bit reservoir size is 12 288 bit – (96 000 bit/s / 48 000 1/s * 1024) = 10 240 bit. For variable bitrate channels the encoder shall operate in a way that the input buffer requirements do not exceed the minimum decoder buffer.

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

The bit_reservoir_state of next frame can be calculated as:

$$\text{bit_reservoir_state}[\text{frame}] = \text{bit_reservoir_state}[\text{frame} - 1] + \text{mean_framelength} - \text{framelength}[\text{frame}] \quad (3.1)$$

where framelength[frame] shall be modified to meet the following condition:

$$0 \leq \text{bit_reservoir_state}[\text{frame}] \leq \text{max_bit_reservoir} \quad (3.2)$$

3.4 Context-dependent Bitplane Coding

3.4.1 Model block description

Context-dependent Bitplane Coding (CBC) selects all of the sub-decoders or part of them according to the output rate and the cut-off condition of the bitstream received. See Figure 3-4. In each sub-decoder, each bit vector is decoded from the highest to the lowest bit, from low- to high-frequency band in turn.

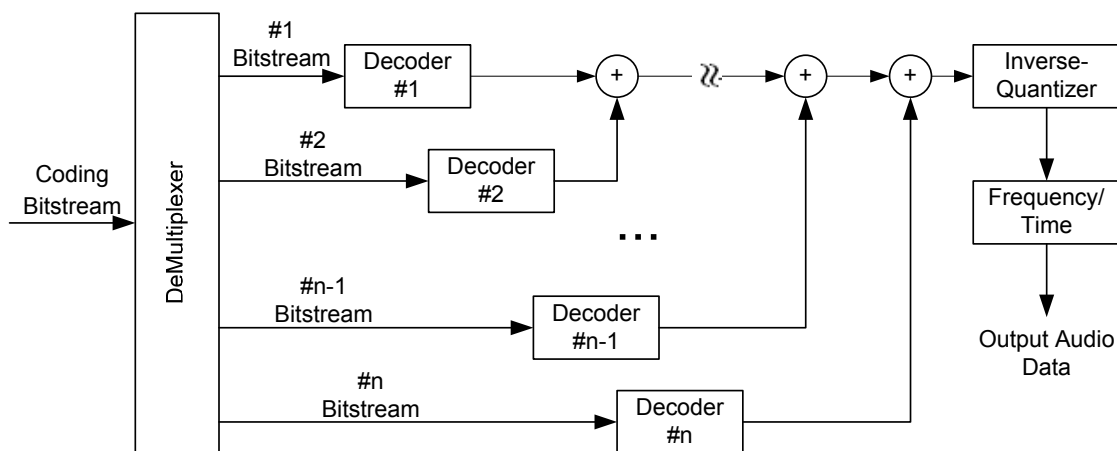


Figure 3-4—CBC decoder principle block diagram

3.4.2 Definitions

raw_data_block	A block that includes the coded audio data. A raw_data_block consists of cbc_base_element and cbc_layer_element basically.
cbc_base_element	Syntax unit of bitstream in the basic layer, including coded spectral data and the relevant side information, etc.
cbc_layer_element	Syntax unit of bitstream in CBC layer, including coded spectral data, relevant side information, etc.
frame_length_data()	Data of frame length, including the length of frame head data in byte unit. Returned value plus 4 bytes is the actual frame length.
mc_Present	Multi-channel bitstream enable bit. “0” represents non-multi-channel data, “1” represents multi-channel data.
top_layer	The number of the highest layer of the scalable bitstream.
base_snf_thr	The significant threshold of spectra data coding in the basic layer.
base_band	Indicates the largest spectral line of the coded data in the basic layer. If the current window is modulation window, it is 4*base_band, else it is 32*base_band.
slayer_size	The number of coding band in the basic layer.
reserved_bit	1 bit, reserved bit.
signal_type	Indicate the type of the current block.
max_sfb	The max number of coding scalefactor band in each group of windows.
scale_factor_grouping	Indicate the division information of the modulation window.

<code>coding_band_mode</code>	The coding band mode of the modulation window.
<code>sc_present</code>	Stereo information.
<code>cbc_scalable_header()</code>	Head information of bitstream scalability supported by CBC.
<code>cbc_general_header()</code>	Head information required by general audio coding.
<code>flpvq_data()</code>	Quantization and coding information of frequency linear predictive spectral coefficient.
<code>scalefactor[ch][0]</code>	Scalefactor value of the first scalefactor band in current frame and current channel.
<code>layer_cband_si()</code>	Side information of coding band.
<code>layer_sfb_si()</code>	Side information of scalefactor band.
<code>cbc_layer_spectra()</code>	Information of new added layer's spectral data, Huffman coding.
<code>cbc_lower_spectra()</code>	Information of pre-layer's uncoded spectral data, Huffman coding.
<code>cbc_spectra_data()</code>	Basic coding unit of spectral data, Huffman coding.

3.4.3 Ancillary decoding process

3.4.3.1 Calculation of layer buffer offset, layer offset

Layer buffer offset is used to control the bit number obtained in each layer during the fine grain scalable (FGS) decoding process. In CBC, according to the largest layer number, the bit number required in the decoding each layer is allocated by averaging. Pseudo-code for the calculation is shown as follows. Here, `frameLength` indicates the bit number of the current frame. `Used_bits` indicates the bit number of the frame head that have been decoded already. `LayerBit` means the total bit number of the enhanced layer. If decoding stream contains `fill_element`, the `frameLength` should be subtracted by `fill_element`, so as to allocate an equal bit number in each layer. The reference pseudo-codes are shown as follows:

```
(3.3)
```

```

/* To calculate the buffer offset of each basic layer */
layerOffset[0] = 16 * (frameLength - used_bits) / (double) (top_layer+16)/(double)
    slayer_size;
for (layer = 1; layer < slayer_size-1; layer++){
    layerOffset[layer] = layerOffset[layer-1] + 16*(frameLength - used_bits) /
        (top_layer+16)/slayer_size;
}
layerOffset[slayer_size-1] = 16 * (frameLength - used_bits) / (top_layer+16);
/* To calculate the buffer offset of each enhanced layer */
layerBit = (frameLength-used_bits) - LayerOffset[slayer_size-1];
for(layer = slayer_size; layer < top_layer+slayer_size-1; layer++) {
    LayerOffset[layer] = LayerOffset[layer-1]+layerBit/top_layer;
}
layerOffset[top_layer+slayer_size-1] = frameLength-used_bits;

```

The largest bit count that is available for each scalable layer (`available_len`) is as follows:

```
(3.4)
```

```

if (layer==0)
    available_len = LayerOffset[0];
else
    available_len += (LayerOffset[layer] - LayerOffset[layer-1]);

```

3.4.3.2 CBC scalefactor band, group and coding band

CBC scalefactor band is defined as follows. In order to increase the coding rate, CBC accepts the concept of Coding Band. A coding band consists of 32 spectral coefficients. For the modulation window sequence, it consists of spectral data that have been interweaved with each other.

The scalefactor band and coding band are the basic elements in the CBC decoding algorithm. Some auxiliary variables are used to describe the decoding process of the coding band and scalefactor. These variables depend on sampling frequency, signal type, scalefactor band, and the number of maximum scalefactor band. The following pseudo-codes show the calculation method of the auxiliary variable `num_swb` (the number of scalefactor band) and `swb_offset` (the offset of scalefactor band). The function `bit_set (bit_field, bit_num)` returns the value of bit number `bit_num` in `bit_field`. The rightmost bit is bit “0”.

```
(3.5)
```

```

fs_index = sampling_frequency_index;
switch (signal_type) {
    case STATIONARY_TYPE:
        num_windows = 1;
        num_window_groups = 1;
        window_group_length[num_window_groups-1] = 1;
        num_swb = num_swb_long_window[fs_index];
        for (sfb = 0; sfb < max_sfb+1; sfb++) {
            swb_offset[0][sfb] = swb_offset_long_window[fs_index][sfb];
        }
        break;
    case TRANSIENT_TYPE:
        num_windows = 8;
        num_window_groups = 1;
        window_group_length[num_window_groups-1] = 1;
        num_swb = num_swb_short_window[fs_index];
        for (i = 0; i < num_windows-1; i++) {
            if (bit_set(scale_factor_grouping, 6-i) == 0) {
                num_window_groups += 1;
                window_group_length[num_window_groups-1] = 1;
            }
        }
        else {
            window_group_length[num_window_groups-1] += 1;
        }
    }

    for (g = 0; g < num_window_groups; g++)
        swb_offset[g][0] = 0;
        for (sfb = 0; sfb < max_sfb+1; sfb++) {
            for (g = 0; g < num_window_groups; g++) {
                swb_offset[g][sfb] = swb_offset_short_window[fs_index][sfb];
                swb_offset[g][sfb] = swb_offset[g][sfb] *
                    window_group_length[g];
            }
        }
        break;
    default:
        break;
}

```

3.4.3.3 Calculation of the layer number in basic layer, `slayer_size`

In CBC, the bandwidth of the basic layer is defined by `base_band` in the bitstream in order to guarantee the quality. The bandwidth is $32 \times \text{base_band}$ in stable-state block coding and $4 \times \text{base_band}$ in the modulation window. To assure that FGS could be realized, the bandwidth limitation expands into enhanced layers in CBC. The bandwidth of every layer depends on the bandwidth of the basic layer, the length of the transform block, and the sampling frequency. The spectrum will be wider with the increase of enhanced layers. The pseudo-codes for calculation the sublayers number included in the basic layer (*slayer_size*) are shown as follows. The “fs” indicates the sampling frequency.

```
(3.6)
```

```

slayer_size = 0;
for (g = 0; g < num_window_groups; g++) {
    if (signal_type == TRANSIENT_TYPE) {

```

```

end_index[g] = base_band * 4 * window_group_length[g];
if (fs == 44100 || fs == 48000) {
    if (end_index[g] % 32 >= 16)
        end_index[g] = (int)(end_index[g] / 32) * 32 + 20;
    else if (end_index[g] % 32 >= 4)
        end_index[g] = (int)(end_index[g] / 32) * 32 + 8;
}
else if (fs == 22050 || fs == 24000 || fs == 32000)
    end_index[g] = (int)(end_index[g] / 16) * 16;
else if (fs == 11025 || fs == 12000 || fs == 16000)
    end_index[g] = (int)(end_index[g] / 32) * 32;
else end_index[g] = (int)(end_index[g] / 64) * 64;
end_cband[g] = (end_index[g] + 31) / 32;
}
else
    end_cband[g] = base_band;
slayer_size += end_cband[g];
}

```

3.4.3.4 Calculation of window group containing the frequency spectrum of scalable layer, layer_group

(3.7)

```

layer = 0;
for (g = 0; g < num_window_groups; g++)
    for (cband = 1; cband <= end_cband[g]; layer++, cband++)
        layer_group[layer] = g;
layer = slayer_size;
for (g = 0; g < num_window_groups; g++)
    for (w = 0; w < window_group_length[g]; w++)
        layer_group[layer++] = g;
for (layer = slayer_size + 8; layer < (top_layer + slayer_size); layer++)
    layer_group[layer] = layer_group[layer - 8];

```

3.4.3.5 Calculation of the number of new added spectral band and coding band in scalable layer, layer_end_index[], layer_end_cband[], layer_start_index[], and layer_start_cband[]

(3.8)

```

layer = 0;
for (g = 0; g < num_window_groups; g++) {
    for (cband = 0; cband < end_cband[g]; cband++) {
        layer_start_cband[layer] = cband;
        end_cband[g] = layer_end_cband[layer] = cband + 1;
        layer_start_index[layer] = cband * 32;
        if ((cband + 1) * 32 > end_index[g])
            layer_end_index[layer++] = end_index[g];
        else
            layer_end_index[layer++] = (cband + 1) * 32;
    }
    if (signal_type == TRANSIENT_TYPE)
        last_index[g] = swb_offset_short_window[fs_index][max_sfb]
            * window_group_length[g];
    else
        last_index[g] = swb_offset_long_window[fs_index][max_sfb];
}
for (layer = slayer_size; layer < (top_layer + slayer_size); layer++) {
    g = layer_group[layer];
    layer_start_index[layer] = end_index[g];
    if (fs == 44100 || fs == 48000) {
        end_index[g] += 8;
        if (end_index[g] % 32 != 0) end_index[g] += 4;
    }
    else if (fs == 22050 || fs == 24000 || fs == 32000)
        end_index[g] += 16;
    else if (fs == 11025 || fs == 12000 || fs == 16000)
        end_index[g] += 32;
    else
        end_index[g] += 64;
    if (end_index[g] > last_index[g])
        end_index[g] = last_index[g];
    layer_end_index[layer] = end_index[g];
    layer_start_cband[layer] = end_cband[g];
    end_cband[g] = layer_end_cband[layer] = (end_index[g] + 31) / 32;
}

```

3.4.3.6 Calculation of newly added scalefactor band in the scalable layer, `layer_end_sfb[]`, and `layer_start_sfb[]`

(3.9)

```
for (g = 0; g < num_window_groups; g++)
    end_sfb[g] = 0;
for (layer = 0; layer < (top_layer+slayer_size); layer++) {
    g = layer_group[layer];
    layer_start_sfb[layer] = end_sfb[g];
    layer_end_sfb[layer] = max_sfb;
    for (sfb = 0; sfb < max_sfb; sfb++) {
        if (layer_end_index[layer] <= swb_offset_short_window[fs_index][sfb] *
            window_group_length[g]) {
            layer_end_sfb[layer] = sfb + 1;
            break;
        }
    }
    end_sfb[g] = layer_end_sfb[layer];
}
```

3.4.4 Decoding process

3.4.4.1 Decode raw_data_block

In the CBC stream, the `raw_data_block` consists of layers. First, `cbc_base_element` is decoded to get data of the basic layer. Then `cbc_layer_element` is decoded to get data of each enhanced layer. The process of decoding `cbc_layer_element` will repeat until the number of layers that have been decoded equals the `top_layer` (the highest one) or the highest decoded layer that has been set.

3.4.4.2 Decode cbc_base_element

`cbc_base_element` consists of `frame_length_data()`, `cbc_scalable_header()`, `cbc_general_header()`, and `cbc_layer_element`.

`frame_length_data()` supports variable length frame. Data in `frame_length_data()` is divided into two parts. One part is the basic frame length field, 8 bits; the other is the enhanced frame length field, 4 bits. Data of variable length frame consist of at least one basic frame length field. It may contain more than one enhanced frame length field. At the end of each field there is an end label, which indicates whether the frame length data ends or not. “0” represents the end of the length-unlimited frame, while “1” means the opposite. The end label of the final length variable frame field shall be “0”. To avoid unpredictable memory consumption, up to 3 enhanced frame lengths are supported.

In order to decode basic layer, the decoding of `cbc_scalable_header()`, `cbc_general_header()`, `layer_cband_si()`, `layer_sfb_si()`, and `cbc_spectra_data()` are included. Decoding of `cbc_base_element` contains decoding of several `cbc_layer_elements`. The basic layer contains several sublayers, the number of which, that is `slayer_size`, is decided by the decoding `base_band`.

3.4.4.3 Decoding cbc_scalable_header

Decoding `cbc_scalable_header` will get the information of CBC scalable. Main processes are as follows:

- Decode `top_layer`, 6 bits
- Decode `base_snf_thr`, 2 bits
- Decode `base_band`, 5 bits

3.4.4.4 Decoding `cbc_general_header`

Decoding `cbc_general_header` will get the basic information of the audio decoding. Main processes are shown as follows:

- Decode `reserved_bit`, 1 bit
- Decode `scalefactor[0]`, 8 bits
- Decode `signal_type`, 1 bit
- Decode `max_sfb`, 4 bits (transient block), 6 bits (stable block)
- Decode `scale_factor_grouping`, 7 bits (transient block)
- Decode `coding_band_mode`, 1 bit (transient block)
- Decode `sc_present`, 2 bits

3.4.4.5 Decoding `cbc_layer_element`

`cbc_layer_element` is the basic unit of the audio data decoding. Decoding data of the basic and the enhanced layer consists of `layer_cband_si()`, `layer_sfb_si()`, `cbc_layer_spectra()`, and `cbc_lower_spectra()`. Main processes are shown as follows:

- Decode `layer_cband_si()`
- Decode `layer_sfb_si()`
- Decode `cbc_layer_spectra()`
- Decode `cbc_lower_spectra()`

3.4.4.6 Decoding `layer_cband_si`

`layer_cband_si()` get the side information of the coding band. The frequency coefficients of CBC are divided into coding bands, and each band consists of 32 coefficients.

The `cband_si` of differential Huffman coding indicates the most significant bit (MSB) value of the coding band, as shown in Table C.1. CBC use `cband_si` to code spectral data by the bit-layer strategy. The pseudo-codes of decoding `cband_si` are shown as follows:

```
(3.10)
```

```

for (ch = 0; ch < nch; ch++) {
    g = layer_group[layer];
    if (coding_band_mode) {
        if (start_cband[g]==0)
            prevSi[ch][g] = 0;
    }
    else {
        if (g==0 && start_cband[g]==0)
            prevSi[ch][0] = 0;
    }
    for (cband = layer_start_cband[layer]; cband < layer_end_cband[layer]; cband++) {
        diff_cband_si = Huffman_decoding();
        if (coding_band_mode)
            cband_si[ch][g][cband] = prevCbandSi[ch][g] - diff_cband_si + 15;
        else
            cband_si[ch][g][cband] = prevCbandSi[ch][0] - diff_cband_si + 15;
        if (cband_si[ch][g][cband] == 30) {
            escape_idx = getbitsfrombuf(4);
            escape = cband_si_esc[escape_idx];
            if (coding_band_mode)
                cband_si[ch][g][cband] = prevCbandSi[ch][g] - escape + 15;
            else

```

```

        cband_si[ch][g][cband] = prevCbandSi[ch][0] - escape + 15;
    }
    if (coding_band_mode)
        prevCbandSi[ch][g] = cband_si[ch][g][cband];
    else
        prevCbandSi[ch][0] = cband_si[ch][g][cband];
}
}

```

coding_band_mode controls whether modulation window, prevCbandSi[ch][g], should be replaced or not. When coding_band_mode equals “1”, prevCbandSi[ch][g] will be replaced by “0” at the beginning of every modulation window, otherwise it will be replaced by “0” only at the beginning of the first modulation window. The Huffman coding of cband_si[ch][g][cband] is shown in Table C.2.

After the cband_si[ch][g][cband] is decoded, the pseudo-code for calculating the highest bit is shown as follows:

```

if (cband_si[ch][g][cband] <= 8)
    msb[ch][g][cband] = (cband_si[ch][g][cband] + 1) / 2;
else
    msb[ch][g][cband] = cband_si[ch][g][cband] - 4;

```

(3.11)

When the side information of differential coding band of Huffman decoding is 30, it means that the current side information is out of the Huffman coding range. In this condition, the following 4 bits decoded stands for the number of the overflow value. From Table 3.33, we can get the value of current side information of the differential coding band, first, and then decode it differentially. The numbers from 9 to 15 of the overflow code are reserved.

Table 3.33—Mapping table of the overflow value of side information differential coding

Escape idx	0	1	2	3	4	5	6	7	8
Escape	−4	−3	−2	−1	30	31	32	33	34

The MSB of each spectral coefficient, cur_snf[][][], can be calculated as follows:

```

/* set current snf */
g = layer_group[layer];
for (ch = 0; ch < nch; ch++) {
    for (i = layer_start_index[layer]; i < layer_end_index[layer]; i++) {
        cband = i / 32;
        cur_snf[ch][g][i] = msb[ch][g][cband];
    }
}

```

(3.12)

3.4.4.7 Decoding cbc_sfb_si

cbc_sfb_si() is used to decode scalefactor. When the scalefactor band does not use stereo coding, the value of each scalefactor is computed from the differential scalefactor decoded by Huffman decoding. The initial value of the differential computation is Scalefactor[ch][0], which is represented with 8 bits in bitstream. The decoding process is shown as follows:

```

prevSf = ScaleFactor[ch][0];
for (ch = 0; ch < nch; ch++) {
    g = layer_group[layer];
    for (sfb = layer_start_sfb[layer]; sfb < layer_end_sfb[layer]; sfb++) {
        diff_scf = Huffman_decoding();
        scf[ch][g][sfb] = diff_scf + prevSf;
        prevSf = scf[ch][g][sfb];
    }
}

```

(3.13)