INTERNATIONAL STANDARD

ISO/IEC 29199-2

Fourth edition
2020-05

Information technology — JPEG XR image coding system —

Part 2: Image coding specification

Technologies de l'information — Système de codage d'image JPEG XR —

Partie 2: Spécification de codage d'image

Reference number
ISO/IEC 29199-2:2020(E)

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Fax: +41 22 749 09 47
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Contents

Page

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.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents) or the IEC list of patent declarations received (see http://patents.iec.ch).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html.

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*, in collaboration with ITU-T. The technically identical text is published as Rec. ITU-T T.832.

This fourth edition cancels and replaces the third edition (ISO/IEC 29199-2:2012), which has been technically revised. It also incorporates the Amendment ISO/IEC 29199-2:2012/Amd.1:2017.

The main changes compared to the previous edition include:

— the specification of additional colour type identifiers;

— the specification of an alternative file storage format based on ISO/IEC 23008-12 for the storage and interchange of JPEG XR coded images and image sequences;

— the specification of media type identifiers for use for use in various internet protocols.

A list of all parts in the ISO/IEC 29199 series can be found on the ISO website.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html.

Introduction

This document specifies requirements and implementation guidelines for the compressed representation of digital images for storage and interchange in a form referred to as JPEG XR. The JPEG XR design provides a practical coding technology for a broad range of applications with excellent compression capability and important additional functionalities. An input image is typically operated on by an encoder to create a JPEG XR coded image. The decoder then operates on the coded image to produce an output image that is either an exact or approximate reconstruction of the input image.

The primary intended application of JPEG XR is the representation of continuous-tone still images such as photographic images. The manner of representation of the compressed image data and the associated decoding process are specified. These processes and representations are generic, that is, they are applicable to a broad range of applications using compressed colour and grayscale images in communications and computer systems and within embedded applications, including mobile devices.

As of 2008, the most widely used digital photography format is a nominal implementation of the first JPEG coding format as specified in ITU-T Recommendation T.81 | ISO/IEC 10918-1. This encoding uses a bit depth of 8 for each of three channels, resulting in 256 representable values per channel (a total of 16 777 216 representable colour values).

More demanding applications may require a bit depth of 16, providing 65 536 representable values for each channel, and resulting in over $2.8 * 10^{14}$ colour values. Additional scenarios may necessitate even greater bit depths and sample representation formats. When memory or processing power is at a premium, as few as five or six bits per channel may be used.

The JPEG XR specification enables greater effective use of compressed imagery with this broadened diversity of application requirements. JPEG XR supports a wide range of colour encoding formats including monochrome, RGB, CMYK and n-component encodings using a variety of unsigned integer, fixed point, and floating point decoded numerical representations with a variety of bit depths. The primary goal is to provide a compressed format specification appropriate for a wide range of applications while keeping the implementation requirements for encoders and decoders simple. A special focus of the design is support for emerging high dynamic range (HDR) imagery applications.

JPEG XR combines the benefits of optimized image quality and compression efficiency together with low-complexity encoding and decoding implementation requirements. It also provides an extensive set of additional functionalities, including:

— high compression capability;

— low computational and memory resource requirements;

— lossless and lossy compression;

— image tile segmentation for random access and large image formats;

— support for low-complexity compressed-domain image manipulations;

— support for embedded thumbnail images and progressive resolution refinement;

— embedded codestream scalability for both image resolution and fidelity;

— alpha plane support;

— bit-exact decoder results for fixed and floating point image formats.

Important detailed design properties include:

— high performance, embedded system friendly compression;

— small memory footprint;

— integer-only operations with no divides;

— a signal processing structure that is highly amenable to parallel processing;

— use of the same signal processing operations for both lossless and lossy compression operation;

— support for a wide range of decoded sample formats (many of which support high dynamic range imagery):

  — monochrome, RGB, CMYK or n-component image representation;

  — 8- or 16-bit unsigned integer;

  — 16- or 32-bit fixed point;

  — 16- or 32-bit floating point;

  — several packed bit formats;

  — 1-bit per sample monochrome;

  — 5- or 10-bit per sample RGB;

  — radiance RGBE.

The algorithm uses a reversible hierarchical lifting-based lapped biorthogonal transform. The transform has lossless image representation capability and requires only a small number of integer processing operations for both encoding and decoding. The processing is based on 16 × 16 macroblocks in the transform domain, which may or may not affect overlapping areas in the spatial domain (with the overlapping property selected under the control of the encoder). The design provides encoding and decoding with a minimal memory footprint suitable for embedded implementations.

The algorithm provides native support for both RGB and CMYK colour types by converting these colour formats to an internal luma-dominant format through the use of a reversible colour transform. In addition, YUV, monochrome and arbitrary n-channel colour formats are supported.

The transforms employed are reversible; both lossless and lossy operations are supported using the same algorithm. Using the same algorithm for both types of operation simplifies implementation, which is especially important for embedded applications.

A wide range of numerical encodings at multiple bit depths are supported: 8-bit and 16-bit formats, as well as additional specialized packed bit formats, are supported for both lossy and lossless compression. (32-bit formats are supported using lossy compression.) Up to 24 bits are retained through the various transforms. While only integer arithmetic is used for internal processing, lossless and lossy coding are supported for floating point and fixed point image data – as well as for integer image formats.

The main body of this document specifies the syntax and semantics of JPEG XR coded images and the associated decoding process that produces an output image from a coded image. Annex A specifies a tag-based file storage format for storage and interchange of such coded images. Annex B specifies profiles and levels, which determine conformance requirements for classes of encoders and decoders. Aspects of colour imagery representations and colour management are discussed in Annex C. The typical expected encoding process is described in Annex D. Annex E contains a media type specification for images encoded according to the tag-based format specified in Annex A for use in various internet protocols. Annex F specifies an alternative file storage format based on

ISO/IEC 23008-12 and associated media type specifications for the storage and interchange of JPEG XR coded images and image sequences. Annexes A, B, E, and F are an integral part of this document and contain normative specifications.

The International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) draw attention to the fact that it is claimed that compliance with this document may involve the use of patents.

ISO and IEC take no position concerning the evidence, validity and scope of these patent rights.

The holders of these patent rights have assured ISO and IEC that they are willing to negotiate licences under reasonable and non-discriminatory terms and conditions with applicants throughout the world. In this respect, the statements of the holders of these patent rights are registered with ISO and IEC. Information may be obtained from the patent database available at www.iso.org/patents.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights other than those in the patent database. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Information technology — JPEG XR image coding system —

Part 2:
Image coding specification

1 Scope

This document specifies a coding format, referred to as JPEG XR, which is designed primarily for continuous-tone photographic content.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

Rec. ITU-T T.833 | ISO/IEC 29199-3, *Information technology — JPEG XR image coding system — Part 3: Motion JPEG XR*

ISO/IEC/IEEE 60559, *Information technology — Microprocessor systems — Floating-Point arithmetic*

ISO/IEC 10646:2017, *Information technology — Universal coded character set (UCS)*

ISO/IEC 23008-12:2017, *Information technology — High efficiency coding and media delivery in heterogeneous environments — Part 12: Image file format*

3 Terms and definitions

For the purposes of this document, the terms, definitions and abbreviated terms specified in ISO/IEC 23008-12 and the following apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at http://www.electropedia.org/

NOTE For the avoidance of doubt, in case of ambiguities, the definitions in this document take precedence over the definitions of ISO/IEC 23008-12 except in regard to the file format specified in Annex F. A sample in the context of ISO/IEC 23008-12 and in Annex F is "all the data associated with a single time". In Annex G, this is meant as all data associated with one coded image, not "element in a two-dimensional image array that comprises an image plane".

3.1
adaptive coefficient normalization
parsing sub-process where *transform coefficients* (3.75) are dynamically partitioned into a *VLC*-coded (3.77) part and a *fixed-length coded* (3.28) part, in a manner designed to control (i.e., "normalize") bits used to represent the VLC-coded part

Note 1 to entry: The fixed-length coded part of DC coefficients and low-pass coefficients is called FLC refinement and the fixed-length coded part of high-pass coefficients is called flexbits.

3.2
adaptive inverse scanning
parsing sub-process where the *zigzag scan order* (3.80) associated with a set of *transform coefficients* (3.75) is dynamically modified, based on the statistics of previously-parsed transform coefficients

3.3
adaptive VLC
parsing sub-process where the code table associated with *VLC* (3.77) parsing of a particular syntax element is switched, among a finite set of fixed tables, based on the statistics of previously-parsed instances of this syntax element

3.4
alpha image plane
optional secondary *image plane* (3.36) associated with an image of the same dimensions as the *luma* (3.45) component of the *primary image plane* (3.57)

Note 1 to entry: The alpha image plane has one component, a luma component.

3.5
block
m×n array of *samples* (3.64), or an m×n array of *transform coefficients* (3.75)

3.6
block index
integer in the range 0 to 15 identifying, by its position in *raster scan order* (3.61), a particular 4×4 *block* (3.5) within a partition of a 16×16 block into 16 4×4 blocks

3.7
byte
sequence of 8 bits

3.8
byte-aligned
bit in a *codestream* (3.13) where its position is an integer multiple of 8 bits from the beginning of the codestream, where the first bit in the codestream is at position 0

3.9
chroma
component (3.14) of the *primary image plane* (3.57) with non-zero index, or the *transform coefficients* (3.75) and sample values associated with this component

3.10
coded block pattern high-pass
syntax element indicating the *coded block status* (3.12), i.e. the presence or absence of non-zero *high-pass coefficients* (3.34), for each of the *blocks* (3.5) in the *macroblock* (3.46)

3.11
coded block pattern low-pass
syntax element indicating the presence or absence of non-zero *low-pass coefficients* (3.44) in the *macroblock* (3.46)

3.12
coded block status
indication of the presence or absence of non-zero *transform coefficients* (3.75) in that *block* (3.5)

3.13
codestream
sequence of bits contained in a sequence of *bytes* (3.7) from which syntax elements are parsed

Note 1 to entry: The most significant bit of the first byte is the first bit of the codestream, the next most significant bit of the first byte is the second bit of the codestream, and so on, to the least significant bit of the first byte (which is the eighth bit of the codestream), followed by the most significant bit of the second byte (which is the ninth bit of the codestream), and so on, up to and including the least significant bit of the last byte of the sequence of bytes (which is the last bit of the codestream).

3.14
component
array of samples associated with an *image plane* (3.36)

3.15
context
possible value of a specific instance of a *context variable* (3.16)

3.16
context variable
variable used in the *parsing process* (3.54) to select which data structure is to be used for the *adaptive VLC* (3.3) parsing of a given syntax element

3.17
DC coefficient
first subset when the *transform coefficients* (3.75), that are contained in a specific *macroblock* (3.46) and a specific *component* (3.14), are partitioned into 3 subsets

3.18
DC-LP array
array of all *DC* (3.17) and *low-pass* (3.44) *transform coefficients* (3.75), for all *macroblocks* (3.46) associated with a specific *component* (3.14)

3.19
decoder
embodiment of a *parsing process* (3.54) and *decoding process* (3.20)

3.20
decoding process
process of computing output sample values from the parsed syntax elements of the *codestream* (3.13)

3.21
dequantization
process of rescaling the quantized *transform coefficients* (3.75) after their value has been parsed from the *codestream* (3.13) and before they are presented to the *inverse transform process* (3.41)

3.22
discriminant
one of DiscrimVal1 or DiscrimVal2, which are the two member variables of an instance of the *adaptive VLC* (3.3) data structure

Note 1 to entry: The adaptive VLC data structure is specified in subclause 5.5.5.

3.23
encoder
embodiment of an *encoding process* (3.24)

3.24
encoding process
process of converting source sample values into a *codestream* (3.13)

3.25
extended image
image (3.35) produced by the *decoding process* (3.20) prior to *windowing* (3.79)

Note 1 to entry: The extended image has a *luma* (3.45) array that is an integer multiple of 16 in width and height.

3.26
file
finite-length sequence of *bytes* (3.7) that is accessible to a *decoder* (3.19) in a manner such that the decoder can obtain access to the data at specified positions within the sequence of bytes

EXAMPLE Access to the data can be achieved by storing the entire sequence of bytes in random access memory or by performing "position seek" operations to specified positions within the sequence of bytes.

3.27
file format
specified structure for the content of a *file* (3.26)

3.28
fixed-length code
FLC
code which assigns a finite set of allowable bit patterns to a specific set of values, where each bit pattern has the same length

3.29
FLC refinement
fixed-length coded (3.28) part of a *DC coefficient* (3.17) or *low-pass coefficient* (3.44) that is parsed using adaptive fixed-length codes

3.30
flexbits
fixed-length coded (3.28) part of the *high-pass coefficient* (3.34) information which is parsed using adaptive fixed-length codes

3.31
frequency band
one of three subsets of the *transform coefficients* (3.75) for an *image* (3.35), which are separately parsed: *DC coefficients* (3.17), *low-pass coefficients* (3.44) and *high-pass coefficients* (3.34)

3.32
frequency mode
codestream (3.13) structure mode where the *DC* (3.17), *low-pass* (3.44), *high-pass* (3.34) and *flexbits* (3.30) *frequency bands* (3.31) for each *tile* (3.72) are grouped separately

3.33
hard tiles
codestream (3.13) structure mode where the overlap operators are not applied across tile boundaries; instead, boundary overlap operators are applied at tile boundaries

3.34
high-pass coefficient
third subset, when the *transform coefficients* (3.75) that are contained in a specific *macroblock* (3.46) and a specific *component* (3.14) are partitioned into 3 subsets

3.35
image
result of the *decoding process* (3.20) consisting of a *primary image plane* (3.57) and an optional *alpha image plane* (3.4)

3.36
image plane
collective term for a grouping of the *components* (3.14) of the *image* (3.35)

3.37
initial level value
one of two values used to compute the *VLC*-coded (3.77) part of a *transform coefficient* (3.75)

3.38
interleaved alpha image plane
alpha image plane (3.4) that is coded in a *codestream* (3.13) in an interleaved manner together with the *primary image plane* (3.57)

3.39
internal colour format
colour format associated with the spatial-domain samples obtained through the *inverse transform process* (3.40) and the *sample reconstruction process* (3.65), and distinguished from the *output colour format* (3.49) associated with the *output formatting process* (3.50)

3.40
inverse core transform
ICT
two steps of the *inverse transform process* (3.40) that involve processing of *transform coefficients* (3.75) associated with each *macroblock* (3.46) independently, with no *overlap filtering* (3.53)

3.41
inverse transform process
part of the *decoding process* (3.20) by which a set of *dequantized* (3.21) *transform coefficients* (3.75) are converted into spatial-domain values

3.42
inverse scanning
process of reordering an ordered set of parsed syntax elements from the *codestream* (3.13) to form an array of *transform coefficients* (3.75) associated with a specific *component* (3.14) and *macroblock* (3.46)

3.43
little-endian form
ordering of the *bytes* (3.7) that represent a numerical value as an integer number of bytes in which the bytes representing the number are in ascending order of significance

Note 1 to entry: With the least significant byte first, followed by the next least significant byte, etc.

3.44
low-pass coefficient
second subset, when the *transform coefficients* (3.75) that are contained in a specific *macroblock* (3.46) and a specific *component* (3.14) are partitioned into 3 subsets

3.45
luma
component (3.14) of an *image plane* (3.36) with index zero, and the *transform coefficients* (3.75) and sample values associated with this component

Note 1 to entry: Although this term is commonly associated with a signal that conveys perceptual brightness information, as used in this document the term is primarily an identifier of a particular array of samples or transform coefficients for an *image* (3.35).

3.46
macroblock
collection of *transform coefficients* (3.75) or samples, across all *components* (3.14), that have the same indices *i* and *j* with respect to a *macroblock partition* (3.47)

3.47
macroblock partition
partitioning of each *component* (3.14), into 16×16, 8×8, or 16×8 *blocks* (3.5), depending on the *internal colour format* (3.39)

3.48
output bit depth
representation, including the number of bits and the interpretation of the bit pattern, used for the sample values of the output *image* (3.35) that are the result of the *decoding process* (3.20)

3.49
output colour format
colour format associated with the output *image* (3.35) that is the result of the *decoding process* (3.20)

3.50
output formatting process
process of converting the arrays of samples (that are the result of the *sample reconstruction process* (3.65)) into the output samples that constitute the output of the *decoding process* (3.20)

Note 1 to entry: This specifies a conversion (if necessary) into the appropriate *output colour format* (3.49) and *output bit depth* (3.48).

3.51
output image height
height of the sub-array of the *luma* (3.45) *component* (3.14) of the *primary image plane* (3.57) that is output by the *decoding process* (3.20)

3.52
output image width
width of the sub-array, of the *luma* (3.45) *component* (3.14) of the *primary image plane* (3.57) that is output by the *decoding process* (3.20)

3.53
overlap filtering
steps of the *inverse transform process* (3.41) that involve processing of *transform coefficients* (3.75) across adjacent *blocks* (3.5) and *macroblocks* (3.46)

Note 1 to entry: When overlap filtering is applied, it is applied across macroblock boundaries as well as block boundaries. When the *codestream* (3.13) uses *soft tiles* (3.67), the overlap filtering is also applied across *tile* (3.72) boundaries. Otherwise, overlap filtering does not occur across tile boundaries.

3.54
parsing process
process of extracting bit sequences from the *codestream* (3.13), converting these bit sequences to syntax element values, and setting the values of global variables for use in the *decoding process* (3.20)

3.55
prediction
process of computing an estimate of the sample value or data element that is currently being decoded

3.56
prediction residual
difference between the result of the *prediction* (3.55) process invoked for a sample or data element, and its intended value

3.57
primary image plane
image plane (3.36) that consists of all *image* (3.35) *components* (3.14) that are not a part of the *alpha image plane* (3.4)

3.58
QP index
integer, which for a particular *frequency band* (3.31) and *macroblock* (3.46) specifies the index into the table of *quantization parameters* (3.60) available for this frequency band and *tile* (3.72)

Note 1 to entry: The *QP index* (3.58) thereby selects, for this macroblock, the quantization parameter used for the *dequantization* (3.21) of the *transform coefficients* (3.75) in the specific *frequency band* (3.31).

3.59
QP set
set of *quantization parameters* (3.60) associated with a particular *frequency band* (3.31), corresponding to the *luma* (3.45) and *chroma* (3.9) *components* (3.14)

3.60
quantization parameter
QP
value used to compute the scaling factor for the *dequantization* (3.21) of a *transform coefficient* (3.75), before the *inverse transform process* (3.41) is applied

3.61
raster scan order
scan order in which a two-dimensional array of values is scanned row-wise from left to right, and the rows are scanned from the top row to the bottom

3.62
refinement
process of modifying a predicted or partially-computed *transform coefficient* (3.75)

3.63
run
number of zero valued coefficient levels that precede a non-zero valued coefficient level in the *zigzag scan order* (3.80) during the *inverse scanning* (3.42) process

3.64
sample
element in a two-dimensional image array that comprises an *image plane* (3.36)

3.65
sample reconstruction process
process of converting *dequantized* (3.21) *transform coefficients* (3.75) into samples of the *image* (3.35)

3.66
separate alpha image plane
alpha image plane (3.4) that is coded in a codestream separately from the *primary image plane* (3.57)

3.67
soft tiles
codestream (3.13) structure mode where the overlap operators are applied across tile boundaries

3.68
spatial co-location
sub-arrays of samples are *spatially co-located* (3.68) across *components* (3.14) when they correspond to the same spatial region of the decoded *image* (3.35)

Note 1 to entry: The *macroblock partition* (3.47) of the image ensures that the *i*-th *macroblock* (3.46) horizontally and *j*-th macroblock vertically across all components are spatially co-located.

3.69
spatial mode
codestream (3.13) structure mode where the *DC* (3.17), *low-pass* (3.44), *high-pass* (3.34) and *flexbits* (3.30) *frequency bands* (3.31) for each specific *macroblock* (3.46) are grouped together

3.70
spatial transformation
element in the *codestream* (3.13) indicating the preferred final displayed orientation of the decoded *image* (3.35)

Note 1 to entry: As specified in subclause 8.3.8.

Note 2 to entry: The *spatial transformation* (3.70) is only a suggestion, and *decoder* (3.19) conformance is checked only for the decoded *image* (3.35) prior to the application of this transformation (i.e. for orientation 0).

3.71
start code
bit pattern that specifies the beginning of a *tile packet* (3.73) or other distinguished, contiguous set of syntax elements in the *codestream* (3.13)

3.72
tile
collection of *macroblocks* (3.46) that have the same indices *i* and *j* with respect to a *tile partition* (3.74)

Note 1 to entry: Each *tile* (3.72) corresponds to the macroblocks for a rectangular region of the *image* (3.35).

3.73
tile packet
contiguous subset of the *codestream* (3.13), which contains the coded syntax elements associated with a specific *tile* (3.72)

3.74
tile partition
partition of the *image* (3.35) into rectangular arrays of *macroblocks* (3.46)

Note 1 to entry: As specified in subclause 6.4.

3.75
transform coefficient
values, associated with each specific *macroblock* (3.46) and specific *component* (3.14), that – after *dequantization* (3.21) – form the input arrays into the *inverse transform process* (3.41)

3.76
universal multiple-octet coded character set transformation format 8
UTF-8
8-bit character set encoding

Note 1 to entry: As specified in ISO/IEC 10646:2017, Annex D.

3.77
variable-length code
VLC
code which assigns a finite set of allowable bit patterns to a specific set of values, where each bit pattern is potentially of a different length

3.78
VLC refinement
one of two values used to compute the *VLC*-coded (3.76) part of a *transform coefficient* (3.75)

Note 1 to entry: The number of bits required to specify the VLC-refinement is dependent on the value of the *initial level value* (3.37). The VLC refinement is added to the initial level value to produce the VLC-coded part of the transform coefficient.

3.79
windowing
selection of *spatially co-located* (3.68) sub-arrays of the *components* (3.14) of all present *image planes* (3.36) associated with an *image* (3.35) that are output by the *decoding process* (3.20)

3.80
zigzag scan order
adaptive ordering for the *inverse scanning* (3.42) process, which assigns array indices to each subsequent *transform coefficient* (3.75) parsed from the *codestream* (3.13)

4 Abbreviated terms

CBPHP	coded block pattern high-pass
CBPLP	coded block pattern low-pass
CIE	International Commission on Illumination
FCC	Federal Communications Commission
FCT	forward core transform
FLC	fixed-length code
HP	high-pass
ICC	International Color Consortium
ICT	inverse core transform
JPEG	Joint Photographic Experts Group
LP	low-pass
LSB	least significant bit
MSB	most significant bit
NTSC	National Television System Committee
QP	quantization parameter
RP	recommended practice
SMPTE	Society of Motion Picture and Television Engineers
UUID	universal unique identifier (as specified by ISO/IEC 11578)
UTF	universal multiple-octet coded character set transformation format (as specified by ISO/IEC 10646)
VLC	variable-length code

5 Conventions

5.1 Conformance language

The keyword "reserved" indicates a provision that is not specified at this time, shall not be used in implementations conforming to this edition of this document and may be specified in the future. The keyword "forbidden" indicates "reserved" and in addition indicates that the provision will never be specified in the future.

5.2 Operators

NOTE Many of the operators used in this document are similar to those used in the C programming language (e.g. as specified in ISO/IEC 9899).

5.2.1 Arithmetic operators

+	addition
−	subtraction (as a binary operator) or negation (as a unary prefix operator)
++	increment by one as a unary postfix operator
− −	decrement by one as a unary postfix operator
*	multiplication
/	integer division, where the result is truncated towards zero
÷	division in mathematical equations where no truncation or rounding is intended
$\frac{x}{y}$	division in mathematical equations where no truncation or rounding is intended
%	x % a is defined as the modulus operator for x >= 0 and a > 0 x % a is defined as −(((−x) % a)) for x < 0 and a > 0

NOTE 1 Although sometimes used as unary prefix operators in the C programming language, the "++" and "− −" arithmetic operators are not used as unary prefix operators in this document.

NOTE 2 The division operators used in this document differ somewhat from those used in the C programming language.

5.2.2 Logical operators

||	logical OR
&&	logical AND
!	logical NOT

TRUE/FALSE convention:

— When a variable or arithmetic expression having a non-zero value is evaluated as a logical condition or as an element of a logical expression, it is evaluated as TRUE, and when a variable or expression having a zero value is evaluated as a logical condition or as an element of a logical expression, it is evaluated as FALSE.

— When the value of a variable or arithmetic expression is compared to the value TRUE (in text or using a relational operator), it is compared to the value 1, and when the value of a variable or arithmetic expression is compared to the value FALSE (in text or using a relational operator), it is compared to the value 0.

— When a variable is set to the value TRUE, it is set to the value 1; and when a variable is set to the value FALSE, it is set to the value 0.

5.2.3 Relational operators

>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
= =	equal to
!=	not equal to

5.2.4 Bit-wise operators

Bit-wise operators operate on bit pattern values that are produced by conversion of an integer value to an equivalent bit pattern value. Bit-wise operators operate on a two's complement representation of the integer value using a number of bits sufficient to represent the integer value (with a bit equal to 0 in the MSB of non-negative integer value representations and otherwise with a bit equal to 1 in the MSB). The result of a bit-wise operator is then interpreted as a two's complement representation of an integer value. The following bit-wise operators are defined:

& AND

When operating on a bit pattern argument that contains fewer bits than the other argument, the shorter argument is extended by adding more significant bits equal to the MSB of the shorter argument such that the number of bits representing the shorter argument is made the same as the number of bits for the longer argument.

| OR

When operating on a bit pattern argument that contains fewer bits than the other argument, the shorter argument is extended by adding more significant bits equal to the MSB of the shorter argument such that the number of bits representing the shorter argument is made the same as the number of bits for the longer argument.

^ XOR

When operating on a bit pattern argument that contains fewer bits than the other argument, the shorter argument is extended by adding more significant bits equal to the MSB of the shorter argument such that the number of bits representing the shorter argument is made the same as the number of bits for the longer argument.

x >> b arithmetic right shift of a two's complement integer representation of x by b binary digits, where b is a non-negative integer

Bits shifted into the MSBs as a result of the right shift have a value equal to the MSB of x prior to the shift operation.

x << b arithmetic left shift of a two's complement integer representation of x by b binary digits, where b is a non-negative integer

Bits shifted into the LSBs as a result of the left shift have a value equal to 0.

5.2.5 Assignment operators

=	assignment operator
+=	x += a is defined as x = x + a
−=	x −= a is defined as x = x − a
^=	x ^= a is defined as x = x ^ a
*=	x *= a is defined as x = x * a
<<=	x <<= a is defined as x = (x << a)
>>=	x >>= a is defined as x = (x >> a)

5.2.6 Precedence order of operators

Operators are listed below in descending order of precedence. If several operators appear in the same line, they have equal precedence. When several operators of equal precedence appear at the same level in an expression, evaluation proceeds according to the associativity of the operator either from right to left or from left to right.

Table 1 — Precedence order of operators

Operators	Type of operation	Associativity
( ), [ ], .	expression	left to right
++, − −	postfix operators	right to left
−, !	unary	
*, /, %, $\frac{x}{y}$	multiplication and division	left to right
+, −	addition and subtraction	left to right
<<, >>	arithmetic shift	left to right
< , >, <=, >=	relational	left to right
= =, !=	equality test	left to right
&, |, ^	bit-wise operator	left to right
&&, | |	logical operators	left to right
?, =, *=, +=, −=, ^=, <<=, >>=	assignment operators	right to left

5.2.7 Pseudocode operations

Pseudocode is organized into "functions" that are specified in tabular form. A sample pseudocode table is presented in Table 2. Pseudocode statements are each expressed as a row of the table. A group of statements enclosed in curly brackets "{...}" is a compound statement and is treated functionally as a single statement. Each function definition begins with a table row specifying the name of the function, the arguments of the function, and containing the opening curly bracket of a compound statement.

Arguments passed to a pseudocode function are listed in parenthesis after the function name, and are comma delimited. Two types of arguments are used in pseudocode function definitions, as follows:

— Non-array variables, which are passed by value (e.g. valueArgument1 in Table 2).

— Arrays, which are passed by reference (e.g. arrayArgument2[ ] in Table 2).

Non-array variables that are passed to a function are addressed within the function using a local variable name, even when a global variable (subclause 5.5) has been used when calling the function. Since non-array variables are passed by value rather than by reference, any changes made to the value of the local variable within the function do not affect the value of the (local or global) variable that was used as a calling argument by the invoking process when the function was called. Since arrays are passed to a function by reference rather than by value, any changes made within the function to the values of entries in such an array (specified using a local array name within the function) do persist after the completion of the pseudocode function. Changes made to the values of global variables (subclause 5.5) that are specified within a function using the name of the global variable also persist after the completion of the pseudocode function.

Pseudocode functions may or may not return a value. When a function returns a value, the value that is returned is specified by a "return" statement that appears as the last statement in the compound statement that specifies the function, and the value that is returned is the value of the expression that is specified after the term "return" in the pseudocode return statement. Functions that do not return a value do not contain a return statement. Table 2 provides an example of a function definition for a function that returns the value of a variable valueReturn.

Table 2 — Example of a pseudocode table

ExamplePseudocode(valueArgument1, arrayArgument2[ ]) {	Reference
statement	
return valueReturn	
}	

The pseudocode convention shown in Table 3 is used to indicate an informative comment.

Table 3 — Example of a pseudocode comment

ExamplePseudocodeComment( ) {	Reference
/* this is a comment start and end */	
}	

The pseudocode convention shown in Table 4 specifies repeated execution of a "condition" checking followed by a "statement" until the "condition" is no longer TRUE.

Table 4 — Example of a pseudocode while statement

ExamplePseudocodeWhileStatement( ) {	Reference
while (condition)	
statement	
}	

The pseudocode convention shown in Table 5 specifies evaluation of an "initial statement" followed by evaluation of a "condition", and when the "condition" is TRUE, it specifies repeated execution of a "primary statement" followed by a "subsequent statement", and repeating the checking of the condition and the execution of the primary statement and subsequent statement until the checked condition no longer evaluates to the value TRUE.

Table 5 — Example of a pseudocode for statement

ExamplePseudocodeForStatement( ) {	Reference
for (initial statement; condition; subsequent statement)	
primary statement	
}	

The pseudocode convention shown in Table 6 specifies that a "statement" is executed if a "condition" is TRUE, and that an "alternate statement" is otherwise performed.

Table 6 — Example of a pseudocode conditional statement

ExamplePseudocodeConditionalStatement( ) {	Reference
if (condition)	
statement	
else	
alternative statement	
}	

— The pseudocode convention shown in Table 7 specifies the initialization of the values of entries in an array. In this example, iArr[0] is set equal to 2, iArr[1] is set equal to 4, iArr[2] is set equal to 6, and iArr[3] is set equal to 8.

Table 7 — Example of the initialization of values in an array in pseudocode

ExamplePseudocodeArrayInitalization( ) {	**Reference**
iArr[ ] = {2, 4, 6, 8}	
}	

5.2.8 Mathematical functions

Ceiling(x) ceiling function

Returns the smallest integer that is greater than or equal to the real-valued argument x.

Floor(x) floor function

Returns the largest integer that is less than or equal to the real-valued argument x.

Ln(x) natural logarithm of x (the base-e logarithm, where e is the natural logarithm base constant 2.718 281 828...)

Max(a, b) maximum of two arguments (as specified in Table 8)

Table 8 — Definition of mathematical function Max( )

Max(a, b) {	**Reference**
if (a >= b)	
valueReturn = a	
else	
valueReturn = b	
return valueReturn	
}	

Min(a, b) minimum of two arguments (as specified in Table 9)

Table 9 — Definition of mathematical function Min( )

Min(a, b) {	**Reference**
if (a <= b)	
valueReturn = a	
else	
valueReturn = b	
return valueReturn	
}	

Abs(x) absolute value of an argument (as specified in Table 10)

Table 10 — Definition of mathematical function Abs( )

Abs(x) {	**Reference**
if (x >= 0)	
valueReturn = x	
else	
valueReturn = −x	
return valueReturn	
}	

Sign(x) sign of an argument (as specified in Table 11)

Table 11 — Definition of mathematical function Sign( )

Sign(x) {	**Reference**
if (x >= 0)	
valueReturn = 1	
else	

valueReturn = −1	
return valueReturn	
}	

Round(x) rounding to integer value (as specified in Table 12)

Table 12 — Definition of mathematical function Round( )

Round(x) {	**Reference**
valueReturn = Sign(x) * Floor(Abs(x) + 0.5)	
return valueReturn	
}	

Clip(x, iLow, iHigh) clip integer value to a range lie between integers iLow and iHigh (as specified in Table 13)

Table 13 — Definition of mathematical function Clip( )

Clip(x, iLow, iHigh) {	**Reference**
valueReturn = Max(x, iLow)	
valueReturn = Min(valueReturn, iHigh)	
return valueReturn	
}	

Sqrt(x) square root of x

Numones(x) returns the number of bits in an argument that are set, for a positive integer argument x that is represented in two's complement arithmetic (as specified in Table 14)

Table 14 — Definition of mathematical function Numones( )

Numones(x) {	**Reference**
valueReturn = 0	
while (x != 0) {	
valueReturn += (x & 1)	
x >>= 1	
}	
return valueReturn	
}	

5.3 Syntax and semantics notation

5.3.1 Method of specifying syntax in tabular form

The syntax tables specify a superset of the syntax of all allowed codestreams. Additional constraints on the syntax may also be specified, either directly or indirectly, in other subclauses.

Table 15 lists an example of pseudocode used to specify the syntax. When the name of a syntax element appears in the first column, it specifies that the syntax element is parsed from the codestream and the codestream pointer is advanced to the next bit position beyond the syntax element in the codestream parsing process.

Subclause 5.3.2 provides an example of how the semantics of a syntax element are specified in this document.

The column with the heading "Descriptor" specifies the parsing process of an associated syntax element as follows:

— i(n): two's complement signed integer using n bits, where the most significant bit is the left-most bit. This indicates a fixed-length syntax element. The value of n is the size of the syntax element in bits. For example, i(3) indicates a 3-bit syntax element, and i(iVar) indicates a syntax element of length iVar, where iVar is a variable computed from the values of other previously parsed syntax elements.

— u(n): unsigned integer using n bits, where the most significant bit is the left-most bit. This indicates a fixed-length syntax element. The value of n is the size of the syntax element in bits. For example, u(3) indicates a 3-

bit syntax element, and u(iVar) indicates a syntax element of length iVar, where iVar is a variable computed from the values of other previously parsed syntax elements.

— le(n): unsigned integer using n bits in little-endian form, where n is an integer multiple of 8. This indicates a fixed-length syntax element. The value of n is the size of the syntax element in bits. For example, le(16) indicates a 16-bit syntax element, and le(iVar) indicates a syntax element of length iVar, where iVar is a variable having a value that is an integer multiple of 8 that is computed from the values of other previously parsed syntax elements.

— e(v): entropy coded syntax element where the most significant bit of the code is the left-most bit. This indicates a variable-length coded syntax element, and a fixed VLC table is used to parse this syntax element.

— ae(v): adaptive entropy coded syntax element where the most significant bit of the code is the left most bit. This indicates a variable-length coded syntax element, where the VLC table used to parse the syntax element is selected adaptively based on the values of other previously parsed syntax elements.

The column with the heading "Reference" provides one or more links to semantics or information about constraints on an associated syntax element.

Table 15 — Syntax table example

HEADER_EXAMPLE( ) {	Descriptor	Reference
/* A statement can be a syntax element or a conditional statement that specifies the presence and type of syntax element */		
/* The conditional statements are expressed in terms of the pseudocode operations defined in subclause 5.2.7 */		
if (condition)		
SYNTAX_ELEMENT_EXAMPLE	u(8)	5.3.2
}		

5.3.2 SYNTAX_ELEMENT_EXAMPLE semantics

SYNTAX_ELEMENT_EXAMPLE is an example 8-bit syntax element having semantics and constraints specified in this subclause, as identified in the "Reference" column of the associated syntax table in subclause 5.3.1.

5.3.3 Syntax functions

The codestream is formatted as an ordered sequence of bytes. These bytes contain sequences of bits. The syntax elements appear within a sequence of bits in the order specified in the syntax tables, and for each syntax element, the most-significant bit of the syntax element representation is the first bit in the sequence of bits that represents the syntax element and the least-significant bit of the syntax element representation is the right-most bit. The bits of the syntax elements shall be extracted from the bytes that represent them by extracting the most-significant bit of the first syntax element from the most-significant bit of the first byte, the next bit of the syntax element from the next less significant bit of the byte, etc., proceeding through to the least-significant bit of the byte and then the most-significant bit of the following byte, if the bit pattern for the syntax element is longer than 8, etc. After the bits of the first syntax element, the same convention shall be followed, starting at the next bit, for the bits of the next syntax element and then for the subsequent syntax elements.

Unless otherwise specified, the bytes of the codestream are ordered in the sequence of bytes that forms the codestream such that the conceptually-first byte is placed first in the sequence of bytes, the second byte is placed second, etc. (i.e., so-called "network" or "big endian" byte ordering is used for the codestream unless otherwise specified).

The syntax functions IS_BYTE_ALIGNED( ) and POS_SEEK(iLoc) are used in the specification of some syntax structures. These functions assume the existence of a codestream pointer referring to the position of the next bit to be read from the codestream by the parsing process. Prior to operation of the parsing process, the codestream pointer refers to the position of the first bit of the first byte of the codestream.

IS_BYTE_ALIGNED( ) is a syntax function specified as follows:

— If the current position in the codestream is on a byte boundary, i.e., the next bit in the codestream is the first bit in a byte, the return value of IS_BYTE_ALIGNED( ) is equal to TRUE.

— Otherwise, the return value of IS_BYTE_ALIGNED( ) is equal to FALSE.

POS_SEEK(iLoc) is a syntax function that sets the codestream pointer to position of the first bit of the byte that is iLoc bytes from the start of the codestream, where iLoc is a non-negative integer argument, and the first byte of the codestream corresponds to iLoc equal to 0.

5.4 Formatting conventions

5.4.1 Variable and array naming conventions

Global variables are in scope throughout more than one subclause. The extent of the scope of each global variable is specified in subclause 5.5. The name of a global variable begins with an upper case letter and includes some lower case letters or numerals, and does not include underscore characters, e.g. as in "ImageWidth". With the exception of image variables (subclause 5.5.1), all other global variables are associated with a specific image plane; therefore, a separate instance of each global variable exists for the primary image plane and for the alpha image plane (in the case where an alpha image plane is present).

In the parsing syntax and pseudocode tables of Clause 8, the global variable IsCurrPlaneAlphaFlag is used to indicate which set of global variables are referenced in the table; if IsCurrPlaneAlphaFlag is equal to FALSE, the global variables referenced are those of the primary image plane; otherwise (IsCurrPlaneAlphaFlag is equal to TRUE), the global variables referenced are those of the alpha image plane.

The scope of local variables is limited to one subclause. They begin with a lower case letter and may include some upper case letters or numerals and do not include underscore characters, e.g. iValue.

Square parentheses are used for the indexing of arrays. Arrays can be either syntax elements or variables. A one-dimensional array is referred to as a list. Array indices count from zero. For example, the first element of arrayExample[ ] is arrayExample[0].

5.4.2 Data structure naming conventions

An instance of a data structure is labelled by bold-faced letters. The member variables of a data structure are formatted like global variables. To reference a member variable of a data structure instance, the data structure instance's name is associated with the member variable with a period ".", e.g. **AbslevelIndexDCLum**.TableIndex indicates that the member variable TableIndex is part of the data structure instance **AbslevelIndexDCLum**.

5.4.3 Syntax element naming conventions

Syntax elements are labelled by a name in upper case letters, in which at least one underscore character is included.

5.4.4 Syntax structure naming conventions

Syntax structure is a term used to refer to a collection of syntax elements. Syntax structures are identified by a name in upper case letters, in which at least one underscore character is included, followed by a pair of parentheses. Within the parentheses, there may be one or more variables. These variables correspond to variables or values that are associated with the pseudocode table for this syntax structure when the syntax structure is invoked within another syntax structure.

5.4.5 Naming conventions for mnemonic constants

Mnemonics are used to refer to constant values taken by syntax elements in the parsing and decoding process. Mnemonics constants are in upper case letters without underscores and may include numbers, e.g. YUV420. The mnemonic constants that are used are defined in Table 16. The mnemonic constant RESERVED is used to specify a value that is reserved for future use.

Table 16 — Defined mnemonic values

Mnemonics	Syntax element and Reference
YUV444, YUV422, YUV420, YONLY, CMYK, CMYKDIRECT, RGB, RGBE, NCOMPONENT	Table 22 for syntax element OUTPUT_CLR_FMT
YUV444, YUV422, YUV420, YONLY, YUVK, NCOMPONENT	Table 28 for syntax element INTERNAL_CLR_FMT
BD1WHITE1, BD8, BD16, BD16S, BD16F, BD32S, BD5, BD10, BD565, BD1BLACK1	Table 23 for syntax element OUTPUT_BITDEPTH
ALL, NOFLEXBITS, NOHIGHPASS, DCONLY	Table 29 for syntax element BANDS_PRESENT

5.4.6 Naming conventions for numerical values

Integer numbers are expressed as bit patterns, hexadecimal values, or decimal numbers. Bit patterns and hexadecimal values have both a numerical value and an associated particular length in bits.

Hexadecimal notation, indicated by prefixing the hexadecimal number by "0x", may be used instead of binary notation to denote a bit pattern having a length that is an integer multiple of 4. For example, 0x41 represents an eight-bit pattern having only its second most significant bit and its least significant bit equal to 1. Numerical values that are specified under a "**Code**" heading in tables that are referred to as "code tables" are bit pattern values (specified as a string of digits equal to 0 or 1 in which the left-most bit is considered the most-significant bit). Other numerical values not prefixed by "0x" are decimal values. When used in expressions, a hexadecimal value is interpreted as having a value equal to the value of the corresponding bit pattern evaluated as a binary representation of an unsigned integer (i.e., as the value of the number formed by prefixing the bit pattern with a sign bit equal to 0 and interpreting the result as a two's complement representation of an integer value). For example, the hexadecimal value 0xF is equivalent to the 4-bit pattern '1111' and is interpreted in expressions as being equal to the decimal number 15.

5.4.7 Array dimensions convention

Arrays of height valHeight and width valWidth are denoted as having dimension valHeight×valWidth. For variable and function names, the character "x" is used as the cross symbol. Otherwise, the cross symbol "×" is used in all other cases.

5.5 Global variables

In subclause 5.5, bold font formatting is used to identify each global variables in the subclause it is described. Changes in value applied to global variables persist beyond single pseudocode functions. The scope of that persistence is specified by the subclause in which the global variable is defined.

5.5.1 Image variables

The following global variables maintain information relevant to the entire image.

ExtendedWidth[i]: This variable specifies the extended image width of component i.

ExtendedHeight[i]: This variable holds the extended image height of component i.

IndexOffsetTile[n]: This variable specifies the offset of the n-th tile packet from the start of the coded image data in bytes.

ImagePrimary[i][x][y]: For each specific triple (i, x, y), where 0 <= i < NumComponents (subclause 8.4.11), 0 <= x < ExtendedWidth[i], 0 <= y < ExtendedHeight[i], the associated variable ImagePrimary[i][x][y] holds the image plane sample values associated with the component i, located at the sample position indicated by the values x and y, for the primary image plane.

ImageAlpha[i][x][y]: For each specific triple (i, x, y), where i = 0, 0 <= x < ExtendedWidth[i], 0 <= y < ExtendedHeight[i], this variable holds the image plane sample value, at the sample position determined by the values x and y, for the alpha image plane.

MBHeight: This variable holds the value associated with the number of vertical macroblock partitions.

MBWidth: This variable holds the value associated with the number of horizontal macroblock partitions.

NumTileCols: This variable holds the value associated with the number of tile partitions in the image horizontally.

NumTileRows: This variable holds the value associated with the number of tile partitions in the image vertically.

TopMBIndexOfTile[i]: This variable holds the value associated with the macroblock index of the top macroblock row of the i-th tile row.

LeftMBIndexOfTile[j]: This variable holds the value associated with the macroblock index of the left macroblock column of the j-th tile column.

NumMBInTile[n]: This variable holds the value associated with the number of macroblocks in the n-th tile.

NumBandsOfPrimary: This variable holds the value associated with the value of NumBands (defined in subclause 5.5.2) for the primary image plane.

SubsequentBytes: This variable holds the value associated with the number of bytes of subsequent data that precede the CODED_TILES( ) (subclause 8.2.2) syntax element and follow the image plane headers and the tiles index table.

5.5.2 Image plane variables

The following global variables maintain information relevant for all tiles of the current image plane.

IsCurrPlaneAlphaFlag: This variable is equal to TRUE if the current plane that is being parsed or decoded is the alpha image plane; otherwise, this variable is equal to FALSE. It is also used to specify which set of image plane variables, tile variables, and macroblock variables are being referenced.

NumComponents: This variable holds the value associated with the number of colour components present in the codestream for the current image plane. For the primary image plane, its value can be obtained by calling DetermineNumComponents( ) (subclause 8.4.11). For the alpha image plane, its value is equal to 1.

NumBands: This variable holds the value associated with the number of frequency bands present in the codestream for the current image plane. Its value can be obtained by calling DetermineNumBands( ) (subclause 8.4.4).

NumLPQPs: This variable holds the value associated with the number of low pass QP sets. This variable may have a constant value over an image plane or it may vary from tile to tile.

NumHPQPs: This variable holds the value associated with the number of high pass QP sets. This variable may have a constant value over an image plane or it may vary from tile to tile.

MBQPIndexLP[MBx][MBy]: (MBx and MBy are defined in subclause 5.5.4) This variable holds the QP index into the table of quantization parameters for LP coefficients, corresponding to the macroblock indexed by MBx and MBy. The same index applies for all colour components.

MBQPIndexHP[MBx][MBy]: (MBx and MBy are defined in subclause 5.5.4) This variable holds the QP index into the table of quantization parameters for HP coefficients, corresponding to the macroblock indexed by MBx and MBy. The same index applies for all colour components.

MbDCLP[MBx][MBy][i][j]: (MBx and MBy are defined in subclause 5.5.4) When j is equal to 0, this variable holds the DC transform coefficient for the macroblock indexed by MBx and MBy, associated with the colour component indexed by i. For non-zero values of the index j, this variable holds the j-th LP transform coefficient for the macroblock indexed by MBx and MBy, and associated with the colour component indexed by i. The index j ranges from 0 to 15 for luma components of all colour formats and chroma components of all colour formats except YUV 4:2:0 and YUV 4:2:2. In the YUV 4:2:0 chroma component case, j ranges from 0 to 3, and in the YUV 4:2:2 chroma component case, j ranges from 0 to 7.

MBBuffer[MBx][MBy][i][j]: (MBx and MBy are defined in subclause 5.5.4) This variable holds the j-th transform coefficient - associated with the colour component i – for the macroblock indexed by MBx and MBy. The index j ranges from 0 to 255.

The ordering of the 256 transform coefficients in the macroblock is as follows: let iBlkIndex represent the block index of a 4×4 block of component i in the macroblock, indexed in raster scan order, with iBlkIndex ranging from 0 to 15. Then the 16 transform coefficients for this block (indexed in raster scan order in the block) are represented by the values of MBBuffer[MBx][MBy][i][j], where j ranges from (16*iBlkIndex + 0) to (16*iBlkIndex + 15), inclusive.

MBCBPHP[MBx][MBy][i]: (MBx and MBy are defined in subclause 5.5.4) This variable holds the coded block status for the macroblock indexed by MBx and MBy, associated with the colour component indexed by i. The association of a bit of this variable to the block in the macroblock is specified in subclause 8.7.17.1, and a bit takes the value 1 if the corresponding block has non-zero HP transform coefficients to be scanned.

PredDCLP[MBx][MBy][i][j]: (MBx and MBy are defined in subclause 5.5.4) This variable holds the predicted DC and LP coefficient values, for the macroblock indexed by MBx and MBy, associated with the colour component indexed by i; the index j ranges from 0 to 6. The predicted DC value corresponds to index 0.

ModelBitsMBHP[MBx][MBy][i]: (MBx and MBy are defined in subclause 5.5.4) This variable holds the value of the member variable MBits[i], associated with the data structure ModelHP as defined in subclause 5.5.6, for the macroblock indexed by MBx and MBy. The index i ranges from 0 to 1. For each macroblock, two values are stored: one value for the luma component, and one value for the chroma components.

NOTE The values ModelBitsMBHP[MBx][MBy][i] are used in the parsing process to communicate the state between the parsing of the syntax structures MB_HP( ) and MB_FLEXBITS( ). See Table 83.

ImagePlane[i][x][y]: This variable holds the sample value associated with the colour component i, located at the position indicated by the values x and y, where 0 <= i < NumComponents, 0 <= x < ExtendedWidth[i], 0 <= y < ExtendedHeight[i], for the current image plane being decoded.

5.5.3 Tile variables

The following global variables maintain information that is relevant for all macroblocks in the current tile:

TileIndexx: This variable holds the column index of the current tile. The value of TileIndexx is in the range 0 <= TileIndexx < NumTileCols.

TileIndexy: This variable holds the row index of the current tile. The value of TileIndexy is in the range 0 <= TileIndexy < NumTileRows.

NumMBInCurrentTile: This variable holds the value associated with the number of macroblocks in the current tile.

DCQuantParam[i]: This variable holds the DC quantization parameter for the colour component i of the current tile.

LPQuantParam[i][j]: This variable holds the LP quantization parameter for the colour component i, and the quantization parameter index j of the current tile.

HPQuantParam[i][j]: This variable holds the HP quantization parameter for the colour component i and the quantization parameter index j of the current tile.

5.5.4 Macroblock variables

The following global variables hold information relevant for a specific macroblock:

MBx: This variable holds the column index of the current macroblock, with respect to the block indices associated with the macroblock partition of the image.

MBy: This variable holds the row index of the current macroblock, with respect to the block indices associated with the macroblock partition of the image.

MBDCMode: This variable holds the DC coefficient prediction mode for the current macroblock.

MBLPMode: This variable holds the LP coefficient prediction mode for the current macroblock.

MBHPMode: This variable holds the HP coefficient prediction mode for the current macroblock.

DCInput[i]: This variable holds the DC transform coefficient value for each colour component i.

LPInput[i][j]: This variable holds the j-th LP transform coefficient value for each colour component i.

HPInputVLC[i][j][k]: This variable holds the most significant bits of the k-th HP transform coefficient value for the j-th block of the macroblock for each colour component i.

HPInputFlex[i][j][k]: This variable holds the least significant bits of the k-th HP transform coefficient value for the j-th block of the macroblock for each colour component i.

IsMBLeftEdgeofTileFlag: This variable indicates whether the current macroblock is along the left edge of the tile. It is set equal to TRUE if MBx is equal to LeftMBIndexOfTile[TileIndexx]. Otherwise, it is set equal to FALSE.

IsMBTopEdgeofTileFlag: This variable indicates whether current macroblock is along the top edge of the tile. It is set equal to TRUE if MBy is equal to TopMBIndexOfTile[TileIndexy]. Otherwise, it is set equal to FALSE.

5.5.5 Data structures for adaptive VLC table selection

5.5.5.1 General

Syntax elements which are parsed using adaptive VLC tables are associated with a set of global state variables. The data structure template **AdaptiveVLC** is used to associate these variables to their respective syntax element.

AdaptiveVLC data structure template member variables are as follows:

— DiscrimVal1: This variable accumulates statistics about the code table choice, used for adaptively switching to other code tables.

— DiscrimVal2: This is a second variable used to accumulate statistics about the code table choice, used for adaptively switching to other code tables. When there are more than two code tables for the parsing of a given syntax element, the associated AdaptiveVLC data structure instance requires two discriminants.

— TableIndex: The index selecting which code table is used for the current macroblock.

— DeltaTableIndex: The index selecting which Delta Table is used for modifying DiscrimVal1, for the current macroblock.

— Delta2TableIndex: The index selecting which Delta Table is used for modifying DiscrimVal2, for the current macroblock.

The collective term discriminant is used, when referring to either DiscrimVal1 or DiscrimVal2; these two member variables are jointly referred to as the discriminants associated with a specific AdaptiveVLC data structure instance.

Subclauses 5.5.5.2 through 5.5.5.4 define the instances of the AdaptiveVLC data structure that are used for the parsing of syntax elements associated with DC, LP, and HP transform coefficients.

NOTE Informative remarks related to this subclause are provided in subclause D.9.

5.5.5.2 DC adaptiveVLC data structure instances

AbslevelIndDCLum: This data structure instance accumulates statistics for the ABS_LEVEL_INDEX syntax element, during parsing of the syntax structure DECODE_ABS_LEVEL( ) (Table 50) from within DECODE_DC( ) (Table 49), referring to luma DC values. The corresponding TableIndex chooses between the code tables for the syntax element ABS_LEVEL_INDEX, which are specified in subclause 8.7.14.5.

AbslevelIndDCChr: This data structure instance accumulates statistics for the ABS_LEVEL_INDEX syntax element, during parsing of the syntax structure DECODE_ABS_LEVEL( ) (Table 50) from within DECODE_DC( ) (Table 49), referring to chroma DC values. The corresponding TableIndex chooses between the code tables for the syntax element ABS_LEVEL_INDEX, which are specified in subclause 8.7.14.5.

5.5.5.3 Low-pass adaptiveVLC data structure instances

DecFirstIndLPLum: This data structure instance accumulates statistics for the FIRST_INDEX syntax element, during parsing of the syntax structure DECODE_FIRST_INDEX( ) (Table 75) from within DECODE_BLOCK( ) (Table 72), referring to luma LP values. The corresponding TableIndex chooses between the code tables for the syntax element FIRST_INDEX, which are specified in subclause 8.7.18.9.7.

DecIndLPLum0: This data structure instance accumulates statistics for the INDEX_A syntax element, during parsing of the syntax structure DECODE_INDEX( ) (Table 74) from within DECODE_BLOCK( ) (Table 72), referring to luma LP values, with the local variable iContext (local to the pseudocode table for DECODE_BLOCK( )) equal to 0. The corresponding TableIndex chooses between the code tables for the syntax element INDEX_A, which are specified in subclause 8.7.18.9.4.

DecIndLPLum1: This data structure instance accumulates statistics for the INDEX_A syntax element, during parsing of the syntax structure DECODE_INDEX( ) (Table 74) from within DECODE_BLOCK( ) (Table 72), referring to luma LP values, with the local variable iContext (local to the pseudocode table for DECODE_BLOCK( )) equal to 1. The corresponding TableIndex chooses between the code tables for the syntax element INDEX_A, which are specified in subclause 8.7.18.9.4.

DecFirstIndLPChr: This data structure instance accumulates statistics for the FIRST_INDEX syntax element, during parsing of the syntax structure DECODE_FIRST_INDEX( ) (Table 75) from within DECODE_BLOCK( ) (Table 72), referring to chroma LP values. The corresponding TableIndex chooses between the code tables for the syntax element FIRST_INDEX, which are specified in subclause 8.7.18.9.7.

DecIndLPChr0: This data structure instance accumulates statistics for the INDEX_A syntax element, during parsing of the syntax structure DECODE_INDEX( ) (Table 74) from within DECODE_BLOCK( ) (Table 72), referring to chroma LP values, with the local variable iContext (local to the pseudocode table for DECODE_BLOCK( )) equal to 0. The corresponding TableIndex chooses between the code tables for the syntax element INDEX_A, which are specified in subclause 8.7.18.9.4.

DecIndLPChr1: This data structure instance accumulates statistics for the INDEX_A syntax element, during parsing of the syntax structure DECODE_INDEX( ) (Table 74) from within DECODE_BLOCK( ) (Table 72), referring to chroma LP values, with the local variable iContext (local to the pseudocode table for DECODE_BLOCK( )) equal to 1. The corresponding TableIndex chooses between the code tables for the syntax element INDEX_A, which are specified in subclause 8.7.18.9.4.

AbsLevelIndLP0: This data structure instance accumulates statistics for the ABS_LEVEL_INDEX syntax element, during parsing of the syntax structure DECODE_ABS_LEVEL( ) (Table 50) from within DECODE_BLOCK( ) (Table 72), referring to LP values with iContext (local to the pseudocode table for DECODE_BLOCK( )) equal to 0. The corresponding TableIndex chooses between the code tables for the syntax element ABS_LEVEL_INDEX, which are specified in subclause 8.7.14.5.

AbsLevelIndLP1: This data structure instance accumulates statistics for the ABS_LEVEL_INDEX syntax element, during parsing of the syntax structure DECODE_ABS_LEVEL( ) (Table 50) from within DECODE_BLOCK( ) (Table 72), referring to LP values with iContext (local to the pseudocode table for DECODE_BLOCK( )) equal to 1. The corresponding TableIndex chooses between the code tables for the syntax element ABS_LEVEL_INDEX, which are specified in subclause 8.7.14.5.

5.5.5.4 High-pass adaptiveVLC data structure instances

DecFirstIndHPLum: This data structure instance accumulates statistics for the FIRST_INDEX syntax element, during parsing of the syntax structure DECODE_FIRST_INDEX( ) (Table 75) from within DECODE_BLOCK( ) (Table 72), referring to luma HP values. The TableIndex chooses between the code tables for the syntax element FIRST_INDEX, which are specified in subclause 8.7.18.9.7.

DecIndHPLum0: This data structure instance accumulates statistics for the INDEX_A syntax element, during parsing of the syntax structure DECODE_INDEX( ) (Table 74) from within DECODE_BLOCK (Table 72), referring to luma HP values, with the local variable iContext (local to the pseudocode table for DECODE_BLOCK( )) equal to 0. The TableIndex chooses between the code tables for the syntax element INDEX_A, which are specified in subclause 8.7.18.9.4.

DecIndHPLum1: This data structure instance accumulates statistics for the INDEX_A syntax element, during parsing of the syntax structure DECODE_INDEX( ) (Table 74) from within DECODE_BLOCK( ) (Table 72), referring to luma HP values, with the local variable iContext (local to the pseudocode table for DECODE_BLOCK( )) equal to 1. The TableIndex chooses between the code tables for the syntax element INDEX_A, which are specified in subclause 8.7.18.9.4.

DecFirstIndHPChr: This data structure instance accumulates statistics for the FIRST_INDEX syntax element, during parsing of the syntax structure DECODE_FIRST_INDEX( ) (Table 75) from within DECODE_BLOCK( ) (Table 72), referring to chroma HP values. The TableIndex chooses between the code tables for the syntax element FIRST_INDEX, which are specified in subclause 8.7.18.9.7.

DecIndHPChr0: This data structure instance accumulates statistics for the INDEX_A syntax element, during parsing of the syntax structure DECODE_INDEX( ) (Table 74) from within DECODE_BLOCK( ) (Table 72), referring to chroma HP values, with the local variable iContext (local to the pseudocode table for DECODE_BLOCK( )) equal to 0. The TableIndex chooses between the code tables for the syntax element INDEX_A, which are specified in subclause 8.7.18.9.4.

DecIndHPChr1: This data structure instance accumulates statistics for the INDEX_A syntax element, during parsing of the syntax structure DECODE_INDEX( ) (Table 74) from within DECODE_BLOCK( ) (Table 72), referring to chroma HP values, with the local variable iContext (local to the pseudocode table for DECODE_BLOCK( )) equal to 1. The TableIndex chooses between the code tables for the syntax element INDEX_A, which are specified in subclause 8.7.18.9.4.

AbsLevelIndHP0: This data structure instance accumulates statistics for the ABS_LEVEL_INDEX syntax element, during parsing of the syntax structure DECODE_ABS_LEVEL( ) (Table 50) from within DECODE_BLOCK( ) (Table 72), referring to HP values with iContext (local to the pseudocode table for DECODE_BLOCK( )) equal to 0. The TableIndex chooses between the code tables for the syntax element ABS_LEVEL_INDEX, which are specified in subclause 8.7.14.5.

AbsLevelIndHP1: This data structure instance accumulates statistics for the ABS_LEVEL_INDEX syntax element, during parsing of the syntax structure DECODE_ABS_LEVEL( ) (Table 50) from within DECODE_BLOCK( ) (Table 72), referring to HP values with iContext (local to the pseudocode table for DECODE_BLOCK( )) equal to 1.

The TableIndex chooses between the code tables for the syntax element ABS_LEVEL_INDEX. This syntax element's code tables are specified in subclause 8.7.14.5.

DecNumCBPHP: This data structure instance accumulates statistics for the NUM_CBPHP syntax element, during parsing of the syntax structure MB_CBPHP( ) (subclause 8.7.17.2). The TableIndex chooses between the code tables for the syntax element NUM_CBPHP, which are specified in subclause 8.7.17.4.1.

DecNumBlkCBPHP: This data structure instance accumulates statistics for the NUM_BLKCBPHP syntax element, during parsing of the syntax structure MB_CBPHP( ) (subclause 8.7.17.2). The TableIndex chooses between the code tables for NUM_BLKCBPHP, which are specified in subclause 8.7.17.4.2.

NOTE The number of code tables for the NUM_BLKCBPHP syntax element is either 5 or 9, depending on the INTERNAL_CLR_FMT of the image.

5.5.6 Adaptive coefficient normalization data structure instances

The parsing of syntax elements associated with transform coefficients involves maintaining state variables which are used for adaptive coefficient normalization. Each frequency band (DC, LP, HP) maintains its own collection of state variables to track these statistics. To denote the association of these variables, a data structure is used. The data structure template **Model** is defined as follows:

Model data structure template member variables:

- MState[i]: This variable maintains the associated state, where MState[0] represents the information for the luma component, and MState[1] represents the information for the chroma components.

- MBits[i]: This variable represents the number of bits, where MBits[0] represents the information for the luma component, and MBits[1] represents the information for the chroma components.

The following instances of the **Model** data structure are used for the DC, LP, and HP bands:

ModelDC: This data structure instance maintains the statistics for the DC band.

ModelLP: This data structure instance maintains the statistics for the LP band.

ModelHP: This data structure instance maintains the statistics for the HP band.

5.5.7 Adaptive CBPHP prediction data structure instance

The CBPHP prediction mechanism is adapted based on the statistics of CBPHP of previous macroblocks. Each colour component maintains its own collection of state variables to track these statistics. To denote the association of these variables, the data structure template **CBPHPModel** is defined as follows:

CBPHPModel has three member variables indexed by i, where i is equal to either 0 or 1. This data structure template holds two independent sets of statistics: one for the luma component, corresponding to i = 0, and one for the chroma components, corresponding to i = 1. The three member variables are as follows:

- CBPHPState[i]: This variable maintains the state.

- CountOnes[i]: This variable represents the count of blocks, computed as specified in subclause 8.10.2, with the value of coded status (derived from CBPHP) equal to 1.

- CountZeroes[i]: This variable represents the count of blocks, computed as specified in subclause 8.10.2, with the value of coded status equal to 0.

The following instance of the CBPHPModel data structure is used for the HP band:

CBPHPModelHP: This data structure instance maintains the CBPHP statistics for the HP band.

5.5.8 Adaptive count CBPLP variables

The following global variables maintain information that is relevant for the parsing of syntax elements associated with CBPLP:

CountZeroCBPLP: This variable holds the sample value associated with a count the number of times the CBPLP is equal to zero.

CountMaxCBPLP: This variable holds the sample value associated with a count the number of times the CBPLP is equal to its maximum (subclause 8.9.3).

5.6 Adaptive VLC deltaDisc tables

The following global variables maintain lists of the appropriate values of deltaDisc when a syntax element is parsed using a VLC having a code table that can be adaptively selected:

AbslevelIndexDelta[i][j]: This variable holds the value associated with the incremental discriminant value for the i-th syntax element and the j-th code table associated with syntax element ABS_LEVEL_INDEX (subclause 8.7.14.5).

FirstIndexDelta[i][j]: This variable holds the value associated with the incremental discriminant value for the i-th syntax element and the j-th code table associated with syntax element FIRST_INDEX (subclause 8.7.18.9.7).

Index1Delta[i][j]: This variable holds the value associated with the incremental discriminant value for the i-th syntax element and the j-th code table associated with syntax element INDEX_A (subclause 8.7.18.9.4).

NumCBPHPDelta[i][j]: This variable holds the value associated with the incremental discriminant value for the i-th syntax element and the j-th code table associated with syntax element NUM_CBPHP (subclause 8.7.17.4.1).

NumBlkCBPHPDelta[i][j]: This variable holds the value associated with the incremental discriminant value for the i-th syntax element and the j-th code table associated with syntax element NUM_BLKCBPHP (subclause 8.7.17.4.2).

5.7 Adaptive inverse scanning tables

The following global variables maintain lists of the various coefficient scanning orders and related statistics:

LowpassScanOrder[i]: This variable holds the value associated with the location where the i-th lowpass transform coefficient is put into a block in the raster scan order.

HighpassHorScanOrder[i]: This variable holds the value associated with the location where the i-th highpass transform coefficient is put into a block in the raster scan order with horizontal scanning.

HighpassVerScanOrder[i]: This variable holds the value associated with the location where the i-th highpass transform coefficient is put into a block in the raster scan order with vertical scanning.

ScanOrder0[i]: This variable holds the value associated with the first of two possible initialization values for the lowpass and highpass coefficient scans.

ScanOrder1[i]: This variable holds the value associated with the second of two possible initialization values for the lowpass and highpass coefficient scans.

LowpassTotals[i]: This variable holds the value associated with the statistics used to determine how the lowpass scan order is updated.

HighpassHorTotals[i]: This variable holds the value associated with the statistics used to determine how the highpass horizontal scan order is updated.

HighpassVerTotals[i]: This variable holds the value associated with the statistics used to determine how the highpass vertical scan order is updated.

ScanTotals[i]: This variable holds the value associated with the initialization of the lists used to determine how the lowpass and highpass scan order is updated.

6 General provisions, provisions specified in annexes, and image and codestream structures

6.1 General

The main body of this document specifies the syntax and semantics which shall be used for JPEG XR coded images and specifies the associated decoding process that produces an output image from a coded image in a manner that shall be matched by the results produced by conforming decoders as specified. Annex A specifies a tag-based file storage format that may be used for storage and interchange of such coded images and to which conformance is required when this storage format is used. Annex B specifies profiles and levels, which determine conformance requirements that shall be followed for classes of encoders and decoders. Aspects of colour imagery representations and colour management are discussed in Annex C. The typical expected encoding process is described in Annex D. Annex E contains a media type specification for images encoded according to the tag-based format specified in Annex A which shall be followed when indicated as specified. Annex F specifies an alternative file storage format based on ISO/IEC 23008-12 which may be used for storage and interchange of coded images, and also defines associated media type specifications for the storage and interchange of JPEG XR coded images and image sequences, which shall be followed when indicated as specified.

The subclauses of Clause 6 specify the image and codestream structures for JPEG XR images. Italic font formatting is used to highlight occurrences of terms defined in Clause 3.

6.2 Image planes and component arrays

An *image* is composed of one or two *image planes*: a *primary image plane*, and, when present, an *alpha image plane*. An *image plane* is an ordered set of *components*. A *component* is an array of samples. The *primary image plane* may have multiple *components*; NumComponents (subclause 8.4.11) specifies the number of *components*, with $1 <= \text{NumComponents} <= 4111$. For this *primary image plane*, each *component* is an ExtendedHeight[i] × ExtendedWidth[i] array of samples, where ExtendedWidth[i] and ExtendedHeight[i] specify (respectively) the width and height of the array for the i-th *component*, for the index i, $0 <= i < \text{NumComponents}$.

For both the *primary image plane* and the *alpha image plane*, the *component* corresponding to the index i = 0 is defined to be the *luma component* of the respective *image plane*; in the case where NumComponents is greater than 1, the *components* of the *primary image plane* corresponding to non-zero indices are defined to be the *chroma components* of this *image plane*.

The *alpha image plane* is an *image plane* that contains exactly one *component*. The dimensions of this *component* are the same as those of the *luma component* of the *primary image plane*.

NOTE The purpose of an *alpha image plane* is to indicate a level of blend of the *primary image plane* with relation to the background on which the *image* is being rendered. A common interpretation of the *alpha image plane* is as a multiplicative processing (normalized to between 0 and 1) applied to the sample values of the *primary image plane*. The normalized value of the *alpha image plane* determines the proportion of the blending. The value 0 indicates full transparency and the maximum representable value indicates full opacity.

ExtendedHeight[0] is referred to as the *extended image* height. It specifies the number of rows in the *luma* array that are processed within the *decoding process*. Its value is set equal to HEIGHT_MINUS1 + 1 + TOP_MARGIN + BOTTOM_MARGIN.

ExtendedWidth[0] is referred to as the *extended image* width. It specifies the number of columns in the *luma* array that are processed within the *decoding process*. Its value is set equal to WIDTH_MINUS1 + 1 + LEFT_MARGIN + RIGHT_MARGIN.

The *chroma component* array sizes are specified such that ExtendedHeight[i] is equal to ExtendedHeight[1] and ExtendedWidth[i] is equal to ExtendedWidth[1] for all i > 1. The values of ExtendedHeight[1] and ExtendedWidth[1] are specified in Table 17.

Table 17 — Pseudocode to calculate chroma component array sizes

CalculateChromaComponentArraySizes( ) {	**Reference**
if (INTERNAL_CLR_FMT = = YUV420) {	
ExtendedHeight[1] = ExtendedHeight[0] / 2	
ExtendedWidth[1] = ExtendedWidth[0] / 2	
} else if (INTERNAL_CLR_FMT = = YUV422) {	
ExtendedHeight[1] = ExtendedHeight[0]	
ExtendedWidth[1] = ExtendedWidth[0] / 2	
} else {	
ExtendedHeight[1] = ExtendedHeight[0]	
ExtendedWidth[1] = ExtendedWidth[0]	
}	
}	

6.3 Image windowing

Image windowing is specified by four syntax elements: TOP_MARGIN, BOTTOM_MARGIN, RIGHT_MARGIN, and LEFT_MARGIN. These syntax elements determine the columns and rows of the *extended image* that are not present in the output *image*. With respect to a *raster scan ordering* in the *luma* array of an *image plane*, the first TOP_MARGIN rows are not output, nor are the last BOTTOM_MARGIN rows; also, the first LEFT_MARGIN columns are not output, nor are the last RIGHT_MARGIN columns of the *luma component*. Similarly, the *spatially co-located* portions of the *chroma components* are not output, in a manner that retains the ratios between the sizes of the arrays for the *chroma components* and that of the *luma component*, as specified in subclause 6.2.

6.4 Image partitioning

The *luma component* is partitioned horizontally and vertically into an integer number of 16×16 *blocks* of samples. Label these *blocks* arrayLumaMB[j][k], where 0 <= j < (ExtendedWidth[0] / 16) and 0 <= k < (ExtendedHeight[0] / 16) are the *block* indices defined by this partitioning. MBHeight is defined to be equal to ExtendedHeight[0] / 16, and MBWidth is defined to be equal to ExtendedWidth[0] / 16.

In a similar fashion, the *chroma components* are partitioned into *blocks* arrayChromaMB[i][j][k] of size 8×8 for 4:2:0 sampling, of size 16×8 for 4:2:2 sampling, or of size 16×16 in the default case.

This partitioning of the *components* into *blocks* is called a *macroblock partition*.

For each specific pair of indices (j, k), the *macroblock* arrayMB[j][k] is defined to be the collection of *blocks* arrayComplonentMB[i][j][k], for 0 <= i < NumComponents. Across all *components* (all values of i), arrayComponent[i][j][k] and arrayChromaMB[i][j][k] are *spatially co-located*.

Let 0 = LeftMBIndexOfTile[0] < LeftMBIndexOfTile[1] < … < LeftMBIndexOfTile[NumTileCols] = MBWidth, and 0 = TopMBIndexOfTile[0] < TopMBIndexOfTile[1] < … < TopMBIndexOfTile[NumTileRows] = MBHeight be two increasing sequences of integers, where the sequences are of length NumTileCols + 1 and NumTileRows + 1, respectively. LeftMBIndexOfTile[ ] is calculated by calling DetermineLeftBoundaryofTile( ) (subclause 8.3.25) and TopMBIndexOfTile[ ] is calculated by calling DetermineTopBoundaryofTile( ) (subclause 8.3.26). Associated with any such pair of sequences, a *tile partition* may be defined: partition the *macroblocks* of each into *tiles* arrayTile[m][n], for 0 <= m < NumTileCols, and 0 <= n < NumTileRows, where arrayTile[m][n] is defined to be the set of all *macroblocks* MB[j][k] LeftMBIndexOfTile[m] <= j < LeftMBIndexOfTile[m+1] and TopMBIndexOfTile[n] <= k < TopMBIndexOfTile[n+1].

The i-th *tile column* corresponds to the set of all *tiles* of the form arrayTile[i][n], for 0 <= n < NumTileRows; similarly, the j-th *tile row* corresponds to the set of all *tiles* arrayTile[m][j], for 0 <= m < NumTileCols. The i-th *tile width* is defined to be LeftMBIndexOfTile[i+1] − LeftMBIndexOfTile[i]; likewise, the i-th *tile height* is defined to be TopMBIndexOfTile[i+1] − TopMBIndexOfTile[i]. Both the *tile* width and the *tile* height correspond to an integer number of *macroblocks*. The *codestream* specifies a *tile partition* for the *image*, which impacts the order of the parsing of sample values associated with the *image*, as specified in the *codestream* parsing and *decoding processes*. The *tile partition* shall satisfy 1 <= NumTileCols <= 4096, and 1 <= NumTileRows <= 4096.

NOTE Figure 1 provides an informative overview of the *image plane* partitions and implicit *windowing* where (a) the *extended image* plane dimension is indicated by the bold rectangle, (b) the output *image plane* edges on the left and bottom are indicated by dashed lines), (c) the 2×4 regular tiling pattern is shown (d) the *macroblock* in the *tile* (1,2) is shown and (e) the *blocks* within the *macroblock* are shown in the expanded subfigure. Colour *components* are not explicitly shown.

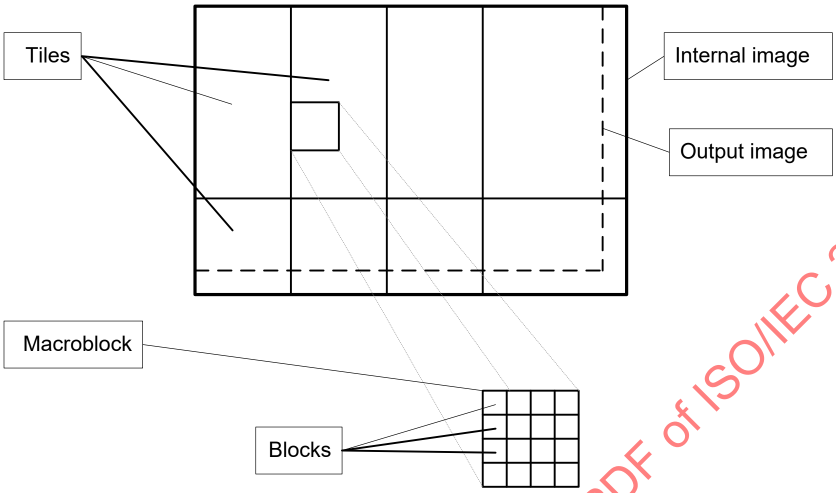


Figure 1 — Informative overview of image partitions and internal windowing

6.5 Transform coefficients and frequency bands

The *decoding process* includes an inverse transform operation. The *transform coefficients* associated with each *component* and each *macroblock* are split into three subsets, or *frequency bands*, which are called the *DC coefficient*, the *low-pass coefficients*, and the *high-pass coefficients*.

For each colour *component* of the *macroblock*, one of the following three conditions holds:

— If the *component* is a *luma component* or a *chroma component* with no down sampling, the following is true. The *component* contains 256 *transform coefficients* for each *macroblock*. These 256 *transform coefficients* are partitioned into three subsets. One set is of size 1, and this coefficient constitutes the *DC coefficient* of this *component*. Another set is of size 15; this set constitutes the *low-pass coefficients*. The third set is of size 240, and constitutes the *high-pass coefficients*.

— If the *component* is a *chroma component* with a sampling rate of ½ for both the horizontal and vertical directions, the following holds true. The *component* contains 64 *transform coefficients* for each *macroblock*. These 64 *transform coefficients* are partitioned into three subsets. One set is of size 1, and this coefficient constitutes the *DC coefficient* of this *component*. Another set is of size 3; this set constitutes the *low-pass coefficients*. The third set is of size 60, and constitutes the *high-pass coefficients*.

— If the *component* is a *chroma component* with a sampling rate of ½ for the horizontal direction and a sampling rate of 1 for the vertical direction, the following holds true. The *component* contains 128 *transform coefficients* for each *macroblock*. These 128 *transform coefficients* are partitioned into three subsets. One set is of size 1, and this coefficient constitutes the *DC coefficient* of this *component*. Another set is of size 7; this set constitutes the *low-pass coefficients*. The third set is of size 120, and constitutes the *high-pass coefficients*.

NOTE 1 The partitioning of *transform coefficients* into three sets, and the use of the terms DC, low-pass, and high-pass is based on the hierarchical nature of the transform.

Transform coefficients are dynamically partitioned into a *VLC*-coded part and a *fixed-length coded* part. The *fixed-length coded* part of the DC and low-pass coefficient is called *FLC refinement*.

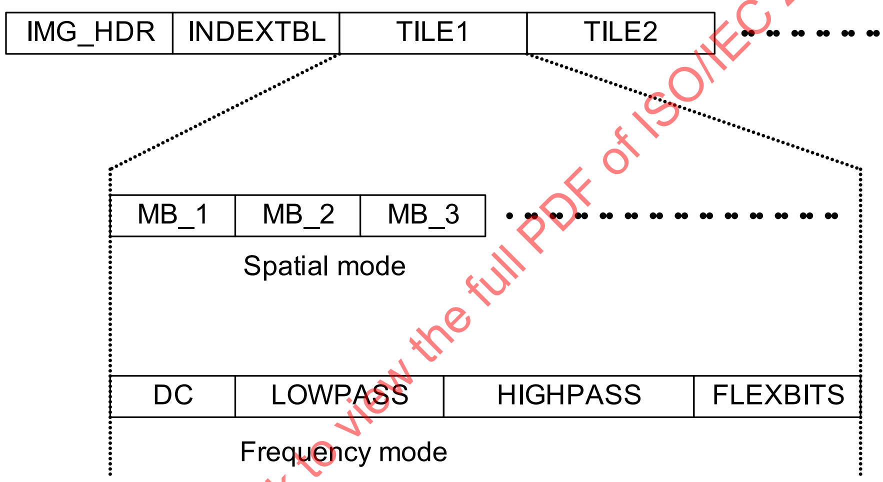
The *fixed-length coded* part of the high-pass coefficient is called *flexbits*. *Flexbits* can be carried in a separate *tile packet* as specified in subclause 6.6.

NOTE 2 This partition of *transform coefficients* is designed to control the number of bits used to represent the *VLC*-coded part.

6.6 Codestream structure

A *codestream* is laid out in one of two orderings called the *spatial mode* and the *frequency mode*. In both modes, the *codestream* is laid out as a header, followed by a sequence of *tile packets*.

NOTE Figure 2 provides an informative overview of the *codestream* structure for the *spatial mode* and the *frequency mode*. The fact that *tiles* can be out of order, and the fact that there can be *codestream* segments of unspecified content between the *tiles* is not shown in Figure 2.



NOTE Image header is followed by a sequence of tiles which are in spatial or frequency mode.

Figure 2 — Informative overview of codestream structure

In the *spatial mode*, a single *tile packet* carries the *codestream* of each *tile* in *macroblock* raster scan order (scanning left to right, top to bottom). The bits associated with each *macroblock* are located together.

In the *frequency mode*, the *codestream* of each *tile* is carried in multiple *tile packets*, where each *tile packet* carries *transform coefficients* of one *frequency band* of that *tile*. The DC *tile packet* carries information of the DC value of each *macroblock*, in *raster scan order*. The LP *tile packet* carries information of the LP coefficients value of each *macroblock*. The HP *tile packet* carries information of the *VLC*-coded part of the HP coefficients of each *macroblock*. Finally, the *flexbits tile packet* carries information regarding the low order bits of the HP coefficients.

6.7 Precision and word length

The SCALED_FLAG syntax element specifies whether scaling is performed in the output formatting stage. If SCALED_FLAG is equal to TRUE, the final output is divided by 8 in the output formatting stage and thus the effective precision of the *decoding processes* such as the inverse transform is higher. If SCALED_FLAG is equal to FALSE, there is no such division operation on the final output and the effective prevision of the *decoding processes* of the inverse transform is lower.

NOTE Encoding with SCALED_FLAG equal to TRUE typically improves rate-distortion performance for lossy coding.

7 Overview of decoder

7.1 General

A block diagram of the decoder, comprising of the parsing process and decoding process, is shown in Figure 3.

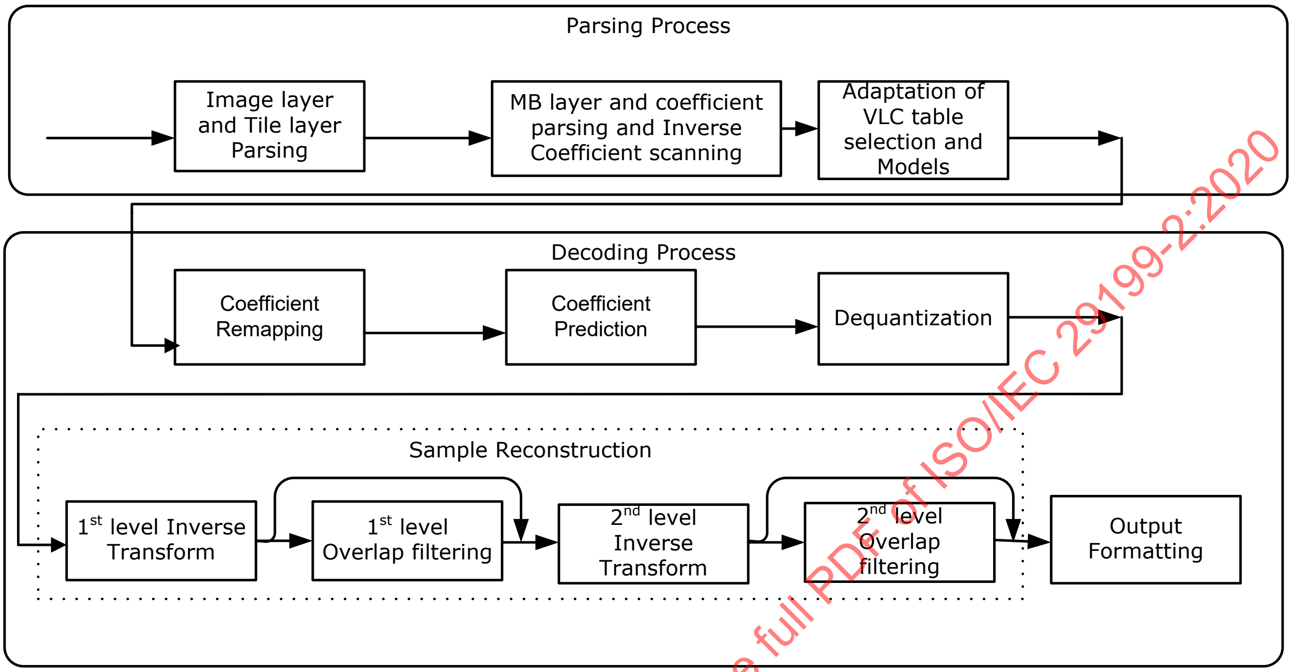


Figure 3 — Informative decoding process block diagram

The parsing process consists of the following stages:

1. image layer and tile layer codestream parsing;
2. macroblock layer codestream parsing which includes parsing the transform coefficients and inverse scanning;
3. adaptation of VLC table selection and context models.

The decoding process consists of the following stages:

4. coefficient remapping;
5. coefficient prediction;
6. dequantization;
7. sample reconstruction, which consists of the following stages:
   a. first-level inverse transform;
   b. when indicated, a first-level overlap filter;
   c. second-level inverse transform;
   d. when indicated, a second-level overlap filter;

8. output formatting.

Clause 8 specifies the stages in the parsing processes. An overview of these steps is provided in subclause 7.2.

Clause 9 specifies the stages in the decoding processes. An overview of these processes is provided in subclause 7.3.

7.2 Overview of parsing process

7.2.1 Overview of image layer codestream parsing

The image level codestream structure is specified in subclause 8.2. It consists of the image header, the header of the primary image plane, and, when present, the header of the alpha image plane. The syntax of the image header is specified by subclause 8.3.

The image plane header defines information that is unique to that plane, and its syntax is specified by subclause 8.4.

The tile index table is used to locate the data that corresponds to a particular tile. The syntax of tile index table is specified by subclause 8.5.

7.2.2 Overview of tile layer codestream parsing

The syntax of the tile layer is specified by subclause 8.7. A tile-packet consists of a tile-packet header, followed by compressed data associated with macroblocks of the tile.

In spatial mode, all the compressed data pertinent to a macroblock is located together in a single tile-packet and the parsing of syntax elements for a spatial-mode tile is specified in subclause 8.7.2.

In frequency mode, each tile packet contains the data associated with a particular transform band; in this mode, a tile-packet is classified as a DC tile-packet, a LP tile-packet, a HP tile-packet, or a flexbits tile-packet.

The syntax elements contained in the DC tile-packet, LP tile-packet, HP tile-packet and flexbits tile-packet are specified in subclauses 8.7.3, 8.7.5, 8.7.7, and 8.7.9, respectively.

If the quantization parameters associated with each band are not specified at the image plane header; they are specified at the tile level.

7.2.3 Overview of macroblock layer codestream parsing

The macroblock layer is parsed to generate the coefficients of the different frequency bands. These coefficients are inverse transformed to reconstruct the macroblock.

Subclause 8.7.11 defines the syntax structure MB_DC( ), which parses the syntax elements related to the DC coefficient, for each component.

Subclause 8.7.16.1 defines the syntax structure MB_LP( ), which parses the syntax elements related to the low-pass coefficients for each component and also performs inverse scanning of the coefficients.

The first step in decoding the HP coefficients involves derivation of CBPHP, which determines which 4×4 blocks of the macroblock have non-zero coefficients. The CBPHP is parsed as specified by the function MB_CBPHP in subclause 8.7.17.2. Subclause 8.7.18.2 defines the syntax structure MB_HP( ) which parses the syntax elements for parsing the VLC part of the HP coefficients, and also performs the inverse scanning of the coefficients. The process of parsing syntax elements related to flexbits and thus refine the HP coefficients is specified in subclause 8.7.19.1.

The VLC table used to parse the syntax elements can be adapted based on the value of previously parsed syntax elements. The adaptation processes for VLC table selection and other context models are also specified in pseudocode in these subclauses.

7.3 Overview of the decoding process

7.3.1 Overview of coefficient mapping

The DC, LP and HP transform coefficients are remapped, and this remapping process is specified in subclause 9.5.

7.3.2 Overview of coefficient prediction

The transform coefficient may be predicted from the coefficients of the neighbouring blocks and macroblocks, and this prediction process is specified in subclause 9.6.

7.3.3 Overview of dequantization

The dequantization process specifies how the transform coefficients are scaled by the quantizer parameter, and this process is specified in subclause 9.8. The derivation of the quantization parameter is specified in subclause 9.7.

7.3.4 Overview of sample reconstruction

Subclause 9.9 defines the sample reconstruction process.

The inverse transform takes a two-level lapped transform. The steps are as follows:

— An inverse core transform (ICT) is applied to each 4×4 block corresponding to reconstructed DC and LP coefficients arranged in an array known as the DC-LP array. The first-level inverse transform process is specified in subclause 9.9.2.

— An overlap filter operation, when indicated, is applied to 4×4 areas evenly straddling blocks in the DC-LP array. For images with soft tiles, this filter is applied to all such blocks. For images with hard tiles, this filter is applied only to the interior of tiles. Further, an overlap filter is applied to boundary 2×4 and 4×2 areas, as well as the four 2×2 corner areas. For images with hard tiles, these filters are additionally applied at tile boundaries. The first-level overlap filtering process is specified in subclause 9.9.3. For INTERNAL_CLR_FMT equal to YUV420 or YUV422, alternate filter operations are applied to the 2×2 interior blocks and 2×1 and 1×2 edge blocks of the chroma components. For these cases, a prediction process is used for the corner samples, denoted 'OverlapPostFilter1' in Figure 4 and detailed in Table 154 and Table 155.

— The resulting array contains coefficients of the 4×4 blocks corresponding to the first-level transform. These coefficients are combined with the reconstructed HP coefficients into a larger array. This coefficient combination process is specified in subclause 9.9.4.

— An ICT is applied to each 4×4 block. The second-level inverse transform process is specified in subclause 9.9.5.

— An overlap filter operation, when indicated, is applied to 4×4 areas evenly straddling blocks in the DC-LP array. For images with soft tiles, this filter is applied to all such blocks. For images with hard tiles, this filter is applied only to the interior of tiles. Further, an overlap filter is applied to boundary 2×4 and 4×2 areas, as well as the four 2×2 corner areas. For images with hard tiles, these filters are additionally applied at tile boundaries. The second-level overlap filtering process is specified in subclause 9.9.6.

The flowchart for the sample reconstruction process is shown in Figure 4.

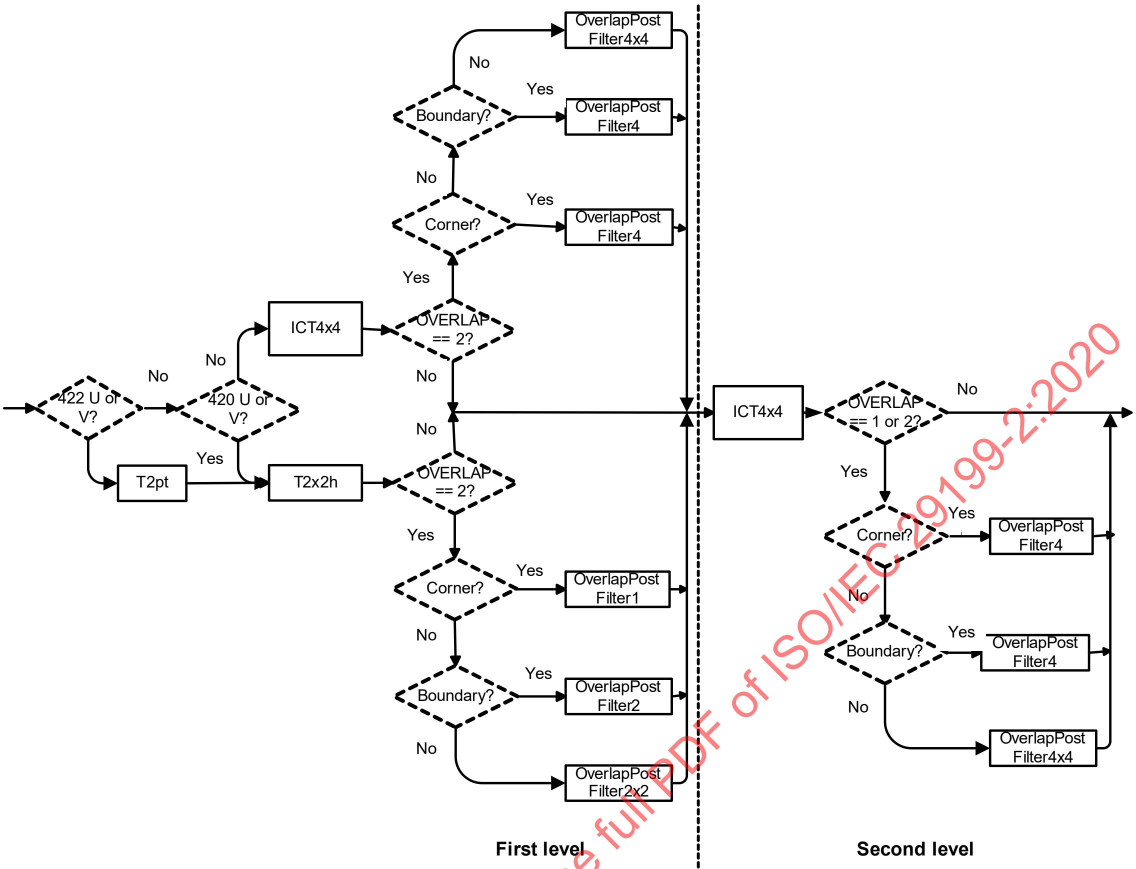


Figure 4 — Informative overview of sample reconstruction process

7.3.5 Overview of output formatting

Subclause 9.10 defines the outputting process which accounts for the various transformations required to handle the different colour formats and bit depths.

8 Syntax, semantics, and parsing process

8.1 General

This clause specifies the codestream layout, and the processes related to parsing syntax elements from the codestream. The parsing of syntax elements requires information about the order of syntax elements as they occur in the codestream, along with the manner of correctly interpreting these syntax elements. At a given point in the parsing of the codestream, the order and presence of syntax elements is conditional upon the state of the decoder itself at that time (based on the previously parsed and interpreted syntax elements as specified by the pseudocode of this subclause).

This clause also specifies the adaptation processes that are associated with variable-length decoding, and with adaptive coefficient normalization. These adaptation processes require specific state variables to be maintained by the decoder in order to properly parse the syntax elements of the codestream. Therefore, the processes of initializing and updating these state variables are also specified in this clause.

The codestream is comprised of the following layers: image, tile, macroblock and block. Furthermore, the macroblock and block layers are laid out differently for the spatial and frequency modes of the codestream. The parsing processes of this subclause are organized by this hierarchy. Below the macroblock level, the parsing processes are further grouped by frequency band: separate syntax structures specify the decoding of the DC, LP and HP frequency bands.

8.2 CODED_IMAGE( )

8.2.1 Syntax structure

The CODED_IMAGE( ) syntax structure is specified by Table 18.

NOTE Throughout the parsing of syntax elements, it is assumed that, if ALPHA_IMAGE_PLANE_FLAG is equal to TRUE, there are two sets of parsed syntax elements: one set corresponding to the primary image plane and used if IsCurrPlaneAlphaFlag is equal to FALSE, and one set corresponding to the alpha image plane and used if IsCurrPlaneAlphaFlag is equal to TRUE.

Table 18 — CODED_IMAGE( ) syntax structure

CODED_IMAGE( ) {	**Descriptor**	**Reference**
IMAGE_HEADER( )		8.3
IsCurrPlaneAlphaFlag = FALSE		
IMAGE_PLANE_HEADER( )		8.4
if (ALPHA_IMAGE_PLANE_FLAG) {		
IsCurrPlaneAlphaFlag = TRUE		
IMAGE_PLANE_HEADER( )		8.4
}		
if (INDEX_TABLE_PRESENT_FLAG)		
INDEX_TABLE_TILES( )		8.5
SubsequentBytes = VLW_ESC( )		8.2.4
if (SubsequentBytes > 0) {		
iBytes = PROFILE_LEVEL_INFO( )		8.6
valueAdditionalBytes = SubsequentBytes − iBytes		
for (iBytes = 0; iBytes < valueAdditionalBytes; iBytes++)		
RESERVED_A_BYTE	u(8)	8.2.3
}		
CODED_TILES( )		8.7
}		

8.2.2 SubsequentBytes

SubsequentBytes specifies the number of bytes of subsequent data that precede the CODED_TILES( ). The value of this variable is determined by a VLW_ESC( ) syntax structure as specified in subclause 8.2.4. When SubsequentBytes is not equal to 0, it is a requirement of codestream conformance that SubsequentBytes shall not be less than 4.

The value of the variable valueAdditionalBytes is derived from the value of SubsequentBytes as shown in the syntax structure table of subclause 8.2.1.

The value of valueAdditionalBytes shall be equal to 0 in codestreams encoded according to this edition of this document. The use of other values of valueAdditionalBytes is reserved for future specification by ITU-T | ISO/IEC. Decoders shall allow this variable to have any value and shall use this value to determine the quantity of RESERVED_A_BYTE syntax elements that follow.

NOTE The purpose of the specification for decoders to allow this variable to have any value is to enable the future definition of a backward-compatible usage of different values of this variable.

8.2.3 RESERVED_A_BYTE

RESERVED_A_BYTE is an 8-bit syntax element. The use of this syntax element is reserved for future specification by ITU-T | ISO/IEC. When present, the value of this syntax element shall be ignored by the decoder.

NOTE The purpose of the specification for decoders to ignore the value of RESERVED_A_BYTE is to enable the future definition of a backward-compatible usage of this syntax element.

8.2.4 VLW_ESC( )

8.2.4.1 Syntax structure

The VLW_ESC( ) syntax structure is specified by Table 19.

Table 19 — VLW_ESC( ) syntax structure

VLW_ESC( ) {	Descriptor	Reference
FIRST_BYTE	u(8)	8.2.4.2
if (FIRST_BYTE < 0xFB) {		
SECOND_BYTE	u(8)	8.2.4.3
iValue = FIRST_BYTE * 256 + SECOND_BYTE		
} else if (FIRST_BYTE = = 0xFB) {		
FOUR_BYTES	u(32)	8.2.4.4
iValue = FOUR_BYTES		
} else if (FIRST_BYTE = = 0xFC) {		
EIGHT_BYTES	u(64)	8.2.4.5
iValue = EIGHT_BYTES		
} else /* FIRST_BYTE is 0xFD, or 0xFE, or 0xFF */		
iValue = 0 /* Escape Mode */		
return iValue		
}		

8.2.4.2 FIRST_BYTE

FIRST_BYTE is an 8-bit syntax element that affects the computation of iValue as specified in subclause 8.2.4.1.

8.2.4.3 SECOND_BYTE

SECOND_BYTE is an 8-bit syntax element that affects the computation of iValue as specified in subclause 8.2.4.1.

8.2.4.4 FOUR_BYTES

FOUR_BYTES is a 32-bit syntax element that affects the computation of iValue as specified in subclause 8.2.4.1.

8.2.4.5 EIGHT_BYTES

EIGHT_BYTES is a 64-bit syntax element that affects the computation of iValue as specified in subclause 8.2.4.1.

8.3 IMAGE_HEADER( )

8.3.1 Syntax structure

The IMAGE_HEADER( ) syntax structure is specified by Table 20.

Table 20 — IMAGE_HEADER( ) syntax structure

IMAGE_HEADER( ) {	Descriptor	Reference
GDI_SIGNATURE	u(64)	8.3.2
RESERVED_B	u(4)	8.3.3
HARD_TILING_FLAG	u(1)	8.3.4
RESERVED_C	u(3)	8.3.5
TILING_FLAG	u(1)	8.3.6
FREQUENCY_MODE_CODESTREAM_FLAG	u(1)	8.3.7
SPATIAL_XFRM_SUBORDINATE	u(3)	8.3.8
INDEX_TABLE_PRESENT_FLAG	u(1)	8.3.9
OVERLAP_MODE	u(2)	8.3.10
SHORT_HEADER_FLAG	u(1)	8.3.11
LONG_WORD_FLAG	u(1)	8.3.12
WINDOWING_FLAG	u(1)	8.3.13
TRIM_FLEXBITS_FLAG	u(1)	8.3.14
RESERVED_D	u(1)	8.3.15
RED_BLUE_NOT_SWAPPED_FLAG	u(1)	8.3.16
PREMULTIPLIED_ALPHA_FLAG	u(1)	8.3.17
ALPHA_IMAGE_PLANE_FLAG	u(1)	8.3.18
OUTPUT_CLR_FMT	u(4)	8.3.19
OUTPUT_BITDEPTH	u(4)	8.3.20
if (SHORT_HEADER_FLAG) {		
WIDTH_MINUS1	u(16)	8.3.21
HEIGHT_MINUS1	u(16)	8.3.22
} else {		
WIDTH_MINUS1	u(32)	8.3.21
HEIGHT_MINUS1	u(32)	8.3.22
}		
if (TILING_FLAG) {		
NUM_VER_TILES_MINUS1	u(12)	8.3.23
NUM_HOR_TILES_MINUS1	u(12)	8.3.24
}		
for (n = 0; n < NUM_VER_TILES_MINUS1; n++)		
if (SHORT_HEADER_FLAG)		
TILE_WIDTH_IN_MB[n]	u(8)	8.3.25
else		
TILE_WIDTH_IN_MB[n]	u(16)	8.3.25
for (n = 0; n < NUM_HOR_TILES_MINUS1; n++)		
if (SHORT_HEADER_FLAG)		
TILE_HEIGHT_IN_MB[n]	u(8)	8.3.26
else		
TILE_HEIGHT_IN_MB[n]	u(16)	8.3.26
if (WINDOWING_FLAG) {		
TOP_MARGIN	u(6)	8.3.27
LEFT_MARGIN	u(6)	8.3.28
BOTTOM_MARGIN	u(6)	8.3.29
RIGHT_MARGIN	u(6)	8.3.30
}		
}		

8.3.2 GDI_SIGNATURE

GDI_SIGNATURE is a 64-bit syntax element that identifies the codestream. It shall have the value 0x574D50484F544F00.

NOTE This signature corresponds to "WMPHOTO" using the UTF-8 character set encoding specified in ISO/IEC 10646:2017, Annex D, followed by a byte equal to 0.

8.3.3 RESERVED_B

RESERVED_B is a 4-bit syntax element that shall be equal to 1 in all codestreams conforming to this edition of this document. All other values are reserved.

NOTE Alternative values for RESERVED_B could be specified in the future as an indication of a codestream that is not compatible with prior decoder versions.

8.3.4 HARD_TILING_FLAG

HARD_TILING_FLAG is a 1-bit syntax element. If HARD_TILING_FLAG is equal to TRUE, overlap filtering is not performed across tile boundaries (hard tiles). Otherwise (HARD_TILING_FLAG is equal to FALSE), overlap filtering is performed across tile boundaries (soft tiles).

8.3.5 RESERVED_C

RESERVED_C is a 3-bit syntax element that shall be equal to 1 in all codestreams conforming to this edition of this document. All other values are reserved. Decoders conforming to this edition of this document shall ignore the value of RESERVED_C.

NOTE The purpose of the specification for decoders to ignore the value of RESERVED_C is to enable the future definition of a backward-compatible usage of different values of this syntax element.

8.3.6 TILING_FLAG

TILING_FLAG is a 1-bit syntax element. If TILING_FLAG is equal to TRUE, both the syntax elements NUM_VER_TILES_MINUS1 and NUM_HOR_TILES_MINUS1 are present in the codestream. Otherwise, these syntax elements are not present, and the number of tiles is equal to 1.

8.3.7 FREQUENCY_MODE_CODESTREAM_FLAG

FREQUENCY_MODE_CODESTREAM_FLAG is a 1-bit syntax element.

If FREQUENCY_MODE_CODESTREAM_FLAG is equal to FALSE, the codestream is laid out in the spatial mode. If FREQUENCY_MODE_CODESTREAM_FLAG is equal to TRUE, the codestream is laid out in the frequency mode.

8.3.8 SPATIAL_XFRM_SUBORDINATE

SPATIAL_XFRM_SUBORDINATE is a 3-bit syntax element that, in the absence of any over-riding indication as determined by the application or by a file format usage context, indicates a preferred spatial transformation that should be applied to the decoded image, as specified by Table 21 as follows.

- The "RCW" table column, when equal to 1, indicates a 90 degree clockwise rotation request of the output image.
- The "FlipH" table column, when equal to 1, indicates a horizontal flip request of the output image.
- The "FlipV" table column, when equal to 1, indicates a vertical flip request of the output image.
- The "Example" table column visually illustrates the application of the requested transformation to an image of the character "P".
- The "Fill" table column indicates the location of the [0][0] image sample coordinate position after application of the requested transformation, as follows:
  - "TL" indicates that row 0 represents the top edge of the image and column 0 represents the left edge of the image.

— "BL" indicates that row 0 represents the bottom edge of the image and column 0 represents the left edge of the image.

— "TR" indicates that row 0 represents the top edge of the image and column 0 represents the right edge of the image.

— "BR" indicates that row 0 represents the bottom edge of the image and column 0 represents the right edge of the image.

— "RT" indicates that row 0 represents the right edge of the image and column 0 represents the top edge of the image.

— "RB" indicates that row 0 represents the right edge of the image and column 0 represents the bottom edge of the image.

— "LT" indicates that row 0 represents the left edge of the image and the column 0 represents the top edge of the image.

— "LB" indicates that row 0 represents the left edge of the image and column 0 represents the bottom edge of the image.

NOTE The TIFF 6.0 specification includes an "Orientation" tag with a similar purpose. The TIFF Orientation tag values that correspond to the SPATIAL_XFRM_SUBORDINATE values 0, 1, 2, 3, 4, 5, 6, and 7 are 1, 4, 2, 3, 6, 7, 5, and 8, respectively.

Table 21 — Interpretation of SPATIAL_XFRM_SUBORDINATE

SPATIAL_XFRM_SUBORDINATE	RCW	FlipH	FlipV	Example	Fill
0	0	0	0	**p**	TL
1	0	0	1	**b**	BL
2	0	1	0	**q**	TR
3	0	1	1	**d**	BR
4	1	0	0		RT
5	1	0	1		RB
6	1	1	0		LT
7	1	1	1		LB

8.3.9 INDEX_TABLE_PRESENT_FLAG

INDEX_TABLE_PRESENT_FLAG is a 1-bit syntax element that specifies whether the index table is present in the codestream. If FREQUENCY_MODE_CODESTREAM_FLAG is equal to TRUE, or NUM_VER_TILES_MINUS1 is greater than 0, or NUM_HOR_TILES_MINUS1 is greater than 0, it is a requirement of codestream conformance that INDEX_TABLE_PRESENT_FLAG shall be equal to TRUE. If INDEX_TABLE_PRESENT_FLAG is equal to TRUE, the index table is present in the codestream. Otherwise, the index table is not present in the codestream. See subclause 8.5.

8.3.10 OVERLAP_MODE

OVERLAP_MODE is a 2-bit syntax element that specifies the overlap processing mode.

When OVERLAP_MODE is equal to 0, no overlap filtering is performed. Otherwise, if OVERLAP_MODE is equal to 1, only the second level overlap filtering is performed. Otherwise, if OVERLAP_MODE is equal to 2, both first level and second level overlap filtering are performed. The value 3 is reserved.

NOTE The trade-offs between complexity and compression efficiency related to the different overlap modes are discussed in subclause D.4.

8.3.11 SHORT_HEADER_FLAG

SHORT_HEADER_FLAG is a 1-bit syntax element that specifies the number of bits required to represent the syntax elements for the width and the height of the image and the tiles. If SHORT_HEADER_FLAG is equal to TRUE, WIDTH_MINUS1 and HEIGHT_MINUS1 are 16-bit syntax elements, and TILE_WIDTH_IN_MB[n], when present, and

TILE_HEIGHT_IN_MB[n], when present, are 8-bit syntax elements. Otherwise, WIDTH_MINUS1 and HEIGHT_MINUS1 are 32-bit syntax elements, and TILE_WIDTH_IN_MB[n], when present, and TILE_HEIGHT_IN_MB[n], when present, are 16-bit syntax elements.

8.3.12 LONG_WORD_FLAG

LONG_WORD_FLAG is a 1-bit syntax element that specifies the range of values of variables associated with the decoding process (Clause 9). The constraints imposed by LONG_WORD_FLAG equal to FALSE have the following scope:

— All values of the global array variable MbDCLP[ ][ ][ ][ ]

— All values of the global array variable MBBuffer[ ][ ][ ][ ]

— All values of the global array variable PredDCLP[ ][ ][ ][ ]

— Values of global array variable ImagePlane[ ][ ][ ] in the sample reconstruction process specified in subclause 9.9.

NOTE 1 Values of the global array variable ImagePlane[ ][ ][ ] in the output formatting process specified in subclause 9.10 are not included in the scope of the constraints imposed by LONG_WORD_FLAG equal to FALSE.

— All values of local variables used in the sample reconstruction process specified in subclause 9.9 except the index variables used to address the elements of arrays.

NOTE 2 Local variables used in the output formatting process specified in subclause 9.10 are not included in the scope of the constraints imposed by LONG_WORD_FLAG equal to FALSE.

The variables associated with the decoding process specified in Clause 9 shall be constrained as follows:

— If LONG_WORD_FLAG is equal to TRUE, it is a requirement for codestream conformance to this document that the range of values of all variables associated with the decoding process (Clause 9) shall not exceed the signed 32-bit range from -2^{31} range to $2^{31}-1$, inclusive, although the range of values of these variables may exceed the signed 16-bit range from -2^{15} range to $2^{15}-1$, inclusive. Thus all variables of the decoding process can be stored by decoders as 32 bit two's complement integers while producing output conforming to this document (regardless of the value of LONG_WORD_FLAG).

— Otherwise (LONG_WORD_FLAG is equal to FALSE), it is a requirement for codestream conformance to this document that the range of values of the specific affected variables listed above shall not exceed the signed 16-bit range from -2^{15} range to $2^{15}-1$, inclusive, and that the range of values of all intermediate expressions and other variables associated with the decoding process for the codestream shall not exceed the signed 32-bit range from -2^{31} range to $2^{31}-1$, inclusive. In this case, it is sufficient for decoders to store the affected variables of the decoding process as 16 bit two's complement integers in order to produce output images conforming to this document.

NOTE 3 Decoder implementations need not use two's complement arithmetic using fixed-word-length storage and processing. However, in the case where such a representation is used, 32-bit variable storage and 32-bit arithmetic processing elements are sufficient to decode an image, regardless of the value of LONG_WORD_FLAG. The LONG_WORD_FLAG element can be used by the decoder to optimize its resource usage for the sample reconstruction process.

8.3.13 WINDOWING_FLAG

WINDOWING_FLAG is a 1-bit syntax element that specifies whether syntax elements specifying windowing dimensions (TOP_MARGIN, LEFT_MARGIN, BOTTOM_MARGIN, and RIGHT_MARGIN as specified in subclauses 8.3.27 through 8.3.30) are present in the codestream. If WINDOWING_FLAG is equal to TRUE, these syntax elements are present in the codestream. If WINDOWING_FLAG is equal to FALSE, these syntax elements are not present in the codestream.

8.3.14 TRIM_FLEXBITS_FLAG

TRIM_FLEXBITS_FLAG is a 1-bit syntax element that specifies whether TRIM_FLEXBITS syntax element is present in the TILE_SPATIAL( ) syntax structure and TILE_FLEXBITS( ) syntax structure. If TRIM_FLEXBITS_FLAG is equal to TRUE, TRIM_FLEXBITS is present. Otherwise, TRIM_FLEXBITS is not present.

8.3.15 RESERVED_D

RESERVED_D is a 1-bit syntax element. The value of RESERVED_D shall be equal to 0. The value 1 is reserved. Decoders shall ignore the value of this syntax element.

NOTE The purpose of the specification for decoders to ignore the value of RESERVED_D is to enable the future definition of a backward-compatible usage of different values of this syntax element.

8.3.16 RED_BLUE_NOT_SWAPPED_FLAG

RED_BLUE_NOT_SWAPPED_FLAG is a 1-bit syntax element. Its interpretation is specified as follows:

— If OUTPUT_CLR_FMT is equal to RGB and OUTPUT_BITDEPTH is equal to BD5, BD565, or BD10, the value of RED_BLUE_NOT_SWAPPED_FLAG affects the operation of the output formatting process.

— Otherwise, the value of RED_BLUE_NOT_SWAPPED_FLAG shall be equal to 0 in all codestreams conforming to this edition of this document. The value 1 is reserved. Decoders conforming to this edition of this document shall ignore the value of RED_BLUE_NOT_SWAPPED_FLAG in this case.

NOTE The specification of semantics for RED_BLUE_NOT_SWAPPED_FLAG was not included in the original edition of this document. The specification of RED_BLUE_NOT_SWAPPED_FLAG was added later to address a problem with respect to the observed behaviour of existing implementations. In principle, when OUTPUT_CLR_FMT is equal to RGB and OUTPUT_BITDEPTH is equal to BD5, BD565, or BD10, the value 1 for RED_BLUE_NOT_SWAPPED_FLAG should ordinarily provide better compression capability than the value 0. However, early product implementations of this document have operated in a manner corresponding to the value 0.

8.3.17 PREMULTIPLIED_ALPHA_FLAG

PREMULTIPLIED_ALPHA_FLAG is a 1-bit syntax element that can be used, when an alpha image plane is present, to indicate that the coded image channels other than the alpha channel are considered to be in pre-multiplied form in relation to the alpha channel.

NOTE 1 The designation of an alpha channel as pre-multiplied indicates that the decoded sample values do not require multiplication by the alpha channel values when performing compositing (as any necessary such multiplication process was performed as a pre-processing step prior to encoding).

When PREMULTIPLIED_ALPHA_FLAG is equal to TRUE in the IMAGE_HEADER( ) of the coded image that contains the alpha image plane, the channels other than the alpha channel are indicated to be in pre-multiplied form in relation to the alpha channel.

When no alpha image plane is present, PREMULTIPLIED_ALPHA_FLAG shall be equal to FALSE, and decoders shall ignore the value of this syntax element.

When an alpha image plane is present as a separate alpha image plane, PREMULTIPLIED_ALPHA_FLAG shall be equal to FALSE in the IMAGE_HEADER( ) of the coded image that does not contain the alpha image plane, and decoders shall ignore the value of this syntax element in the IMAGE_HEADER( ) of the coded image that does not contain the alpha image plane.

When an alpha image plane is present and PREMULTIPLIED_ALPHA_FLAG is equal to FALSE in the IMAGE_HEADER( ) of the coded image that contains the alpha image plane, other indicators provided by other means not specified in the main body of this document should be used to determine whether the channels other than the alpha channel (when present) are considered to be in pre-multiplied form in relation to the alpha channel. When an alpha image plane is present and PREMULTIPLIED_ALPHA_FLAG is equal to FALSE in the

IMAGE_HEADER( ) of the coded image that contains the alpha image plane and such other indicators are not available, it is suggested that the default interpretation should be that the channels other than the alpha channel are considered not to be in pre-multiplied form in relation to the alpha channel.

NOTE 2 The specification of semantics for PREMULTIPLIED_ALPHA_FLAG was not included in the original edition of this document. The specification of PREMULTIPLIED_ALPHA_FLAG was added later to correct for the ambiguity of interpretation resulting from absence of such an indicator (when no indication is provided by other means outside the coded image syntax). In the original edition of this document, the bit corresponding to PREMULTIPLIED_ALPHA_FLAG was required to be equal to 0 and decoders were required to ignore the value of this bit.

NOTE 3 When the file format specified in Annex A is used, the PIXEL_FORMAT value indicates whether the channels other than the alpha channel (when present) are considered to be in pre-multiplied form in relation to the alpha channel, and the value of PREMULTIPLIED_ALPHA_FLAG is required to be consistent with the PIXEL_FORMAT value. When the codestream is conveyed by some means other than the file format specified in Annex A, some indicator may be available to indicate whether the channels other than the alpha channel (when present) are considered to be in pre-multiplied form in relation to the alpha channel, and the value of PREMULTIPLIED_ALPHA_FLAG should be set to be consistent with any such indicator.

8.3.18 ALPHA_IMAGE_PLANE_FLAG

ALPHA_IMAGE_PLANE_FLAG is a 1-bit syntax element that specifies whether an alpha image plane is present in the codestream. If ALPHA_IMAGE_PLANE_FLAG is equal to TRUE, an alpha image plane is present. If ALPHA_IMAGE_PLANE_FLAG is equal to FALSE, no alpha image plane is present in the codestream.

NOTE If ALPHA_IMAGE_PLANE_FLAG is equal to FALSE, an alpha image plane can be carried as a separate image within a system or file.

8.3.19 OUTPUT_CLR_FMT

OUTPUT_CLR_FMT is a 4-bit syntax element that specifies the colour format of the output image as specified in Table 22.

Table 22 — OUTPUT_CLR_FMT

OUTPUT_CLR_FMT	Mnemonic
0	YONLY
1	YUV420
2	YUV422
3	YUV444
4	CMYK
5	CMYKDIRECT
6	NCOMPONENT
7	RGB
8	RGBE
9-15	RESERVED

If IsCurrPlaneAlphaFlag is equal to TRUE, the value of OUTPUT_CLR_FMT shall be equal to 0.

For the cases where OUTPUT_CLR_FMT is equal to YUV420, YUV422, or YUV444, there are three output colour components: the component corresponding to index 0 is the Y component, while the U and V correspond to colour components 1 and 2, respectively. For CMYK and CMYKDIRECT, components 0, 1, 2, and 3 correspond respectively to the C, M, Y, and K components, respectively. For RGB, components 0, 1 and 2 correspond respectively to R, G, and B, and for RGBE, components 0, 1, 2, and 3 correspond respectively to the R, G, B, and E components.

8.3.20 OUTPUT_BITDEPTH

OUTPUT_BITDEPTH is a 4-bit syntax element that specifies the bit depth and corresponding representation of the output image, as specified in Table 23. BD1WHITE1, BD1BLACK1, BD8, BD16, BD5 and BD10 are unsigned integer formats, corresponding to 1, 1, 8, 16, 5 and 10 bits per component, respectively, having a representation specified in subclause 9.10.8. In BD1WHITE1, a value of 0 indicates the minimum level or black for the specific channel and the value 1 indicates the maximum value for that channel. In BD1BLACK1, a value of 1 indicates the minimum level or black for the specific channel and the value 0 indicates the maximum value for that channel. BD16S and BD32S are signed (two's complement) integer formats corresponding to 16 and 32 bits per component, respectively. BD16F is 16-bit Half float (1-bit sign, 5-bit exponent and 10-bit mantissa where the most significant bit is the sign bit) having a representation as specified in subclause 9.10.7.3. BD32F is 32-bit float (1-bit sign, 8-bit exponent, and 23 bit mantissa where the most significant bit is the sign bit) having a representation as specified in subclause 9.10.7.3. BD565 corresponds to unsigned integer formats where the R, G and B components have 5, 6 and 5 bits, respectively, having a representation as specified in subclause 9.10.8.

The values of OUTPUT_BITDEPTH and OUTPUT_CLR_FMT shall be constrained as specified in subclause 9.10.

NOTE Subclause 9.10.7 provides more details on the representation of these formats.

Table 23 — OUTPUT_BITDEPTH

OUTPUT_BITDEPTH	Mnemonic
0	BD1WHITE1
1	BD8
2	BD16
3	BD16S
4	BD16F
5	RESERVED
6	BD32S
7	BD32F
8	BD5
9	BD10
10	BD565
11-14	RESERVED
15	BD1BLACK1

8.3.21 WIDTH_MINUS1

WIDTH_MINUS1 plus 1 is the output image width. If SHORT_HEADER_FLAG is equal to TRUE, WIDTH_MINUS1 is a 16-bit syntax element. Otherwise, WIDTH_MINUS1 is a 32-bit syntax element. When OUTPUT_CLR_FMT is equal to YUV420 or YUV422, it is a requirement of codestream conformance to this document that the value of WIDTH_MINUS1 + 1 shall be an integer multiple of 2.

It is a requirement of codestream conformance to this document that the value of WIDTH_MINUS1 + 1 + LEFT_MARGIN + RIGHT_MARGIN shall be an integer multiple of 16. When INTERNAL_CLR_FMT is equal to YUV420 or YUV422 and OVERLAP_MODE is equal to 2, it is a requirement of codestream conformance to this document that the value of WIDTH_MINUS1 + 1 + LEFT_MARGIN + RIGHT_MARGIN shall be greater than or equal to 32.

NOTE Images with INTERNAL_CLR_FMT equal to YUV420 or YUV422 and OVERLAP_MODE equal to 2 must have a width of at least 2 macroblocks due to the adjacent coefficient residual process described in subclause 9.9.3.1.

8.3.22 HEIGHT_MINUS1

HEIGHT_MINUS1 plus 1 is the output image height. If SHORT_HEADER_FLAG is equal to TRUE, HEIGHT_MINUS1 is a 16-bit syntax element. Otherwise, HEIGHT_MINUS1 is a 32-bit syntax element. When OUTPUT_CLR_FMT is equal to YUV420, it is a requirement of codestream conformance to this document that the value of HEIGHT_MINUS1 + 1 shall be an integer multiple of 2.

It is a requirement of codestream conformance to this document that the value of HEIGHT_MINUS1 + 1 + TOP_MARGIN + BOTTOM_MARGIN shall be an integer multiple of 16.

8.3.23 NUM_VER_TILES_MINUS1

NUM_VER_TILES_MINUS1 is a 12-bit syntax element that is present when TILING_FLAG is equal to TRUE, and specifies the number of tiles in a row minus 1. When NUM_VER_TILES_MINUS1 is not present, its value shall be inferred to be equal to 0.

NOTE "Vertical" indicates that the partitioning of the image corresponding to these tiles runs in the vertical direction.

8.3.24 NUM_HOR_TILES_MINUS1

NUM_HOR_TILES_MINUS1 is a 12-bit syntax element that is present when TILING_FLAG is equal to TRUE, and specifies the number of tiles in a column minus 1. When NUM_HOR_TILES_MINUS1 is not present, its value shall be inferred to be equal to 0.

NOTE "Horizontal" indicates that the partitioning of the image corresponding to these tiles runs in the horizontal direction.

8.3.25 TILE_WIDTH_IN_MB[n]

TILE_WIDTH_IN_MB[n] is a syntax element that specifies the width (in macroblock units) of the n-th tile column, where the 0-th tile column is the left-most tile column in the image, and subsequent tile columns are numbered consecutively, left to right. If SHORT_HEADER_FLAG is equal to TRUE, TILE_WIDTH_IN_MB[n] is an 8-bit syntax element. Otherwise, it is a 16-bit syntax element.

The width of the right-most tile in macroblock units is derived by subtracting the cumulative width of the previous tiles from the width of the coded area in macroblock units ExtendedWidth[0] divided by 16.

The position of the left boundary of the tile, LeftMBIndexOfTile[ ], is calculated by calling DetermineLeftBoundaryofTile( ) in the pseudocode in Table 24.

Table 24 — Pseudocode to determine the position of the left boundaries of the tiles

DetermineLeftBoundaryofTile( ) {	**Reference**
LeftMBIndexOfTile[0] = 0	
for (n = 0; n < NUM_VER_TILES_MINUS1; n++)	
LeftMBIndexOfTile[n+1] = LeftMBIndexOfTile[n] + TILE_WIDTH_IN_MB[n]	
LeftMBIndexOfTile[NUM_VER_TILES_MINUS1 + 1] = MBWidth	
}	

When INTERNAL_CLR_FMT is equal to YUV420 or YUV422, OVERLAP_MODE is equal to 2, and HARD_TILING_FLAG is equal to TRUE, TILE_WIDTH_IN_MB[n] shall be greater than or equal to 2 for all tiles.

When INTERNAL_CLR_FMT is equal to YUV420 or YUV422, OVERLAP_MODE is equal to 2, and HARD_TILING_FLAG is equal to TRUE, MBWidth − LeftMBIndexOfTile[NUM_VER_TILES_MINUS1] shall be greater than or equal to 2.

8.3.26 TILE_HEIGHT_IN_MB[n]

TILE_HEIGHT_IN_MB[n] is a syntax element that specifies the height (in macroblock units) of the n-th tile row, where the 0-th tile row is the top tile row in the image, and subsequent tile rows are numbered consecutively, top to bottom. If SHORT_HEADER_FLAG is equal to TRUE, TILE_HEIGHT_IN_MB[n] is an 8-bit syntax element. Otherwise, it is a 16-bit syntax element.

The height of the bottom tile in macroblock units is derived by subtracting the transmitted heights (plus 1) from the height of the coded area in macroblock units ExtendedHeight[0] divided by 16.

The position of the top boundary of the tile, TopMBIndexOfTile[ ], is calculated by calling DetermineTopBoundaryofTile( ) as specified in the pseudocode in Table 25.

Table 25 — Pseudocode to determine the position of the top boundaries of the tiles

DetermineTopBoundaryofTile( ) {	Reference
TopMBIndexOfTile[0] = 0	
for (n = 0; n < NUM_HOR_TILES_MINUS1; n++)	
TopMBIndexOfTile[n+1] = TopMBIndexOfTile[n] + TILE_HEIGHT_IN_MB[n]	
TopMBIndexOfTile[NUM_HOR_TILES_MINUS1 + 1] = MBHeight	
}	

The number of macroblocks in a tile, NumMBInTile[ ], is calculated by calling DetermineNumMBInTile( ) as specified in the pseudocode in Table 26.

Table 26 — Pseudocode to determine the number of macroblocks in each tile

DetermineNumMBInTile( ) {	Reference
n = 0	
for (i = 0; i < NUM_HOR_TILES_MINUS1 + 1; i++)	
for (j = 0; j < NUM_VER_TILES_MINUS1 + 1; j++) {	
NumMBInTile[n] = TILE_HEIGHT_IN_MB[i] * TILE_WIDTH_IN_MB[j]	
n++	
}	
}	

8.3.27 TOP_MARGIN

TOP_MARGIN is a 6-bit syntax element that is present when WINDOWING_FLAG is equal to TRUE, and specifies the vertical offset of the top boundary of the output image relative to the top edge of the extended image. When TOP_MARGIN is not present, its value shall be inferred to be equal to 0. When OUTPUT_CLR_FMT is equal to YUV420, it is a requirement of codestream conformance to this document that the value of TOP_MARGIN shall be an integer multiple of 2.

8.3.28 LEFT_MARGIN

LEFT_MARGIN is a 6-bit syntax element that is present when WINDOWING_FLAG is equal to TRUE, and specifies the horizontal offset of the left boundary of the output image relative to the left edge of the extended image. When LEFT_MARGIN is not present, its value shall be inferred to be equal to 0. When OUTPUT_CLR_FMT is equal to YUV420 or YUV422, it is a requirement of codestream conformance to this document that the value of LEFT_MARGIN shall be an integer multiple of 2.

8.3.29 BOTTOM_MARGIN

BOTTOM_MARGIN is a 6-bit syntax element that is present when WINDOWING_FLAG is equal to TRUE, and specifies the vertical offset of the bottom of the output image relative to the bottom edge of the extended image. When BOTTOM_MARGIN is not present, its value shall be inferred as follows:

— If HEIGHT_MINUS1 + 1 is an integer multiple of 16, BOTTOM_MARGIN shall be inferred to be equal to 0.

— Otherwise, BOTTOM_MARGIN shall be inferred to be equal to 16 – ((HEIGHT_MINUS1 + 1) % 16).

When OUTPUT_CLR_FMT is equal to YUV420, it is a requirement of codestream conformance to this document that the value of BOTTOM_MARGIN shall be an integer multiple of 2.

8.3.30 RIGHT_MARGIN

RIGHT_MARGIN is a 6-bit syntax element that is present when WINDOWING_FLAG is equal to TRUE, and specifies the horizontal offset of the right boundary of the output image relative to the right edge of the extended image. When RIGHT_MARGIN is not present, its value shall be inferred as follows:

— If WIDTH_MINUS1 + 1 is an integer multiple of 16, RIGHT_MARGIN shall be inferred to be equal to 0.

— Otherwise, RIGHT_MARGIN shall be inferred to be equal to 16 − ((WIDTH_MINUS1 + 1) % 16).

When OUTPUT_CLR_FMT is equal to YUV420 or YUV422, it is a requirement of codestream conformance to this document that the value of RIGHT_MARGIN shall be an integer multiple of 2.

8.4 IMAGE_PLANE_HEADER( )

8.4.1 Syntax structure

The IMAGE_PLANE_HEADER( ) syntax structure is specified by Table 27.

Table 27 — IMAGE_PLANE_HEADER( ) syntax structure

<table>
<tr><th>IMAGE_PLANE_HEADER( ) {</th><th>Descriptor</th><th>Reference</th></tr>
<tr><td>INTERNAL_CLR_FMT</td><td>u(3)</td><td>8.4.2</td></tr>
<tr><td>SCALED_FLAG</td><td>u(1)</td><td>8.4.3</td></tr>
<tr><td>BANDS_PRESENT</td><td>u(4)</td><td>8.4.4</td></tr>
<tr><td>if (INTERNAL_CLR_FMT = = YUV444 | | INTERNAL_CLR_FMT = = YUV420 | | INTERNAL_CLR_FMT = = YUV422) {</td><td></td><td></td></tr>
<tr><td>if (INTERNAL_CLR_FMT = = YUV420 | | INTERNAL_CLR_FMT = = YUV422) {</td><td></td><td></td></tr>
<tr><td>RESERVED_E_BIT</td><td>u(1)</td><td>8.4.5</td></tr>
<tr><td>CHROMA_CENTERING_X</td><td>u(3)</td><td>8.4.6</td></tr>
<tr><td>} else /* INTERNAL_CLR_FMT = = YUV444 */</td><td></td><td></td></tr>
<tr><td>RESERVED_F</td><td>u(4)</td><td>8.4.7</td></tr>
<tr><td>if (INTERNAL_CLR_FMT = = YUV420) {</td><td></td><td></td></tr>
<tr><td>RESERVED_G_BIT</td><td>u(1)</td><td>8.4.8</td></tr>
<tr><td>CHROMA_CENTERING_Y</td><td>u(3)</td><td>8.4.9</td></tr>
<tr><td>} else</td><td></td><td></td></tr>
<tr><td>RESERVED_H</td><td>u(4)</td><td>8.4.10</td></tr>
<tr><td>} else if (INTERNAL_CLR_FMT = = NCOMPONENT) {</td><td></td><td></td></tr>
<tr><td>NUM_COMPONENTS_MINUS1</td><td>u(4)</td><td>8.4.11</td></tr>
<tr><td>if (NUM_COMPONENTS_MINUS1 = = 0xF)</td><td></td><td></td></tr>
<tr><td>NUM_COMPONENTS_EXTENDED_MINUS16</td><td>u(12)</td><td>8.4.12</td></tr>
<tr><td>else</td><td></td><td></td></tr>
<tr><td>RESERVED_H</td><td>u(4)</td><td>8.4.10</td></tr>
<tr><td>}</td><td></td><td></td></tr>
<tr><td>if (OUTPUT_BITDEPTH = = BD16 | | OUTPUT_BITDEPTH = = BD16S | | OUTPUT_BITDEPTH = = BD32S)</td><td></td><td></td></tr>
<tr><td>SHIFT_BITS</td><td>u(8)</td><td>8.4.13</td></tr>
<tr><td>if (OUTPUT_BITDEPTH = = BD32F) {</td><td></td><td></td></tr>
<tr><td>LEN_MANTISSA</td><td>u(8)</td><td>8.4.14</td></tr>
<tr><td>EXP_BIAS</td><td>i(8)</td><td>8.4.15</td></tr>
<tr><td>}</td><td></td><td></td></tr>
<tr><td>DC_IMAGE_PLANE_UNIFORM_FLAG</td><td>u(1)</td><td>8.4.16</td></tr>
<tr><td>if (DC_IMAGE_PLANE_UNIFORM_FLAG)</td><td></td><td></td></tr>
<tr><td>DC_QP( )</td><td></td><td>8.4.22</td></tr>
<tr><td>if (BANDS_PRESENT != DCONLY) {</td><td></td><td></td></tr>
<tr><td>RESERVED_I_BIT</td><td>u(1)</td><td>8.4.17</td></tr>
<tr><td>LP_IMAGE_PLANE_UNIFORM_FLAG</td><td>u(1)</td><td>8.4.18</td></tr>
<tr><td>if (LP_IMAGE_PLANE_UNIFORM_FLAG) {</td><td></td><td></td></tr>
<tr><td>NumLPQPs = 1</td><td></td><td></td></tr>
<tr><td>LP_QP( )</td><td></td><td>8.4.23</td></tr>
<tr><td>}</td><td></td><td></td></tr>
<tr><td>if (BANDS_PRESENT != NOHIGHPASS) {</td><td></td><td></td></tr>
<tr><td>RESERVED_J_BIT</td><td>u(1)</td><td>8.4.19</td></tr>
<tr><td>HP_IMAGE_PLANE_UNIFORM_FLAG</td><td>u(1)</td><td>8.4.20</td></tr>
<tr><td>if (HP_IMAGE_PLANE_UNIFORM_FLAG) {</td><td></td><td></td></tr>
<tr><td>NumHPQPs = 1</td><td></td><td></td></tr>
</table>

HP_QP( )		8.4.24
}		
}		
}		
while (!IS_BYTE_ALIGNED( ))		
BYTE_ALIGNMENT_BIT	u(1)	8.4.21
}		

8.4.2 INTERNAL_CLR_FMT

INTERNAL_CLR_FMT is a 3-bit syntax element that specifies the internal colour format of the coded image as specified in Table 28. For OUTPUT_BITDEPTH of BD16F and BD32F and OUTPUT_CLR_FMT of RGBE, only YUV444 shall be used. The values of INTERNAL_CLR_FMT and OUTPUT_CLR_FMT shall be constrained as specified in subclause 9.10.

Table 28 — INTERNAL_CLR_FMT

INTERNAL_CLR_FMT	Mnemonic
0	YONLY
1	YUV420
2	YUV422
3	YUV444
4	YUVK
5	RESERVED
6	NCOMPONENT
7	RESERVED

When IsCurrPlaneAlphaFlag is equal to TRUE, the value of INTERNAL_CLR_FMT shall be equal to 0.

For the cases where INTERNAL_CLR_FMT is equal to YUV420, YUV422, or YUV444, there are three internal colour components: the component corresponding to index 0 is the Y component, and the U and V correspond to colour components 1 and 2, respectively. For YUVK, components 0, 1, 2, and 3 correspond to the Y, U, V, and K components, respectively.

8.4.3 SCALED_FLAG

SCALED_FLAG is a 1-bit syntax element that specifies whether scaling is performed in the output formatting stage. If SCALED_FLAG is equal to FALSE, scaling is not performed. If SCALED_FLAG is equal to TRUE, scaling is performed as specified in subclause 9.10.6.

8.4.4 BANDS_PRESENT

BANDS_PRESENT is a 4-bit syntax element that indicates whether the various frequency bands are present in the codestream, as specified in Table 29.

Table 29 — BANDS_PRESENT

BANDS_PRESENT	Mnemonic	Interpretation
0	ALL	All subbands are present
1	NOFLEXBITS	Flexbits is not present
2	NOHIGHPASS	Flexbits and HP are not present
3	DCONLY	Only DC is present
4-15	RESERVED	

The number of bands present in the codestream, NumBands, is specified by calling DetermineNumBands( ), which is specified by the pseudocode in Table 30. When IsCurrPlaneAlphaFlag is equal to TRUE, it is a requirement of codestream conformance to this document that the value of NumBands shall be less than or equal to the value of NumBandsOfPrimary.

Table 30 — Pseudocode to determine the number of bands present in the codestream, NumBands

DetermineNumBands( ) {	**Reference**
if (BANDS_PRESENT = = ALL)	
NumBands = 4	
else if (BANDS_PRESENT = = NOFLEXBITS)	
NumBands = 3	
else if (BANDS_PRESENT = = NOHIGHPASS)	
NumBands = 2	
else /* (BANDS_PRESENT = = DCONLY) */	
NumBands = 1	
if (IsCurrPlaneAlphaFlag = = FALSE)	
NumBandsOfPrimary = NumBands	
}	

8.4.5 RESERVED_E_BIT

RESERVED_E_BIT is a 1-bit syntax element and is present when INTERNAL_CLR_FMT is equal to YUV422 or INTERNAL_CLR_FMT is equal to YUV420.

When RESERVED_E_BIT is present, its value shall be equal to 0. The value 1 for RESERVED_E_BIT is reserved.

The value of RESERVED_E_BIT shall be ignored by decoders.

NOTE The purpose of the specification for decoders to ignore the value of RESERVED_E_BIT is to enable the future definition of a backward-compatible usage of different values of this syntax element.

8.4.6 CHROMA_CENTERING_X

CHROMA_CENTERING_X is a 3-bit syntax element. It is present when INTERNAL_CLR_FMT is equal to YUV422 or YUV420. When CHROMA_CENTERING_X is not present, its value should be inferred to be equal to 0.

CHROMA_CENTERING_X indicates the positioning alignment of the chroma sampling grid with respect to the luma sampling grid. When present and in the range of 0 to 4, inclusive, CHROMA_CENTERING_X indicates that the left-most sample of each row of each chroma array of the image is considered to be horizontally positioned at the position CHROMA_CENTERING_X with respect to the left-most sample of each row of the luma array, in units of quarter luma sample positions. When present and equal to 7, CHROMA_CENTERING_X indicates that the positioning alignment is unknown or unspecified.

For example, when CHROMA_CENTERING_X is equal to 0 and INTERNAL_CLR_FMT is equal to YUV422 or YUV420, each chroma sample is considered to be horizontally located at the same position as the left-most sample of a pair of luma samples.

The value of CHROMA_CENTERING_X shall be equal to 0, 1, 2, 3, 4, or 7. The values 5 and 6 are reserved. Decoders conforming to this edition of this document should treat the values 5 and 6 as equivalent to the value 7.

NOTE CHROMA_CENTERING_X is useful to aid in performing appropriate upsampling conversion from 4:2:0 or 4:2:2 to 4:4:4. However, the use of CHROMA_CENTERING_X is not required for decoder conformance to this document, as the manner of performing such an upsampling process is outside the scope of this document.

8.4.7 RESERVED_F

RESERVED_F is a 4-bit syntax element that is present when INTERNAL_CLR_FMT is equal to YUV444.

When RESERVED_F is present, its value shall be equal to 0. Decoders shall ignore the value of this syntax element.

NOTE The specification for decoders to ignore the value of RESERVED_F is to enable the future definition of a backward-compatible usage of different values of this syntax element.

8.4.8 RESERVED_G_BIT

RESERVED_G_BIT is a 1-bit syntax element that is present when INTERNAL_CLR_FMT is equal to YUV420.

When RESERVED_G_BIT is present, its value shall be equal to 0. The value 1 for RESERVED_G_BIT is reserved.

Decoders shall ignore the value of RESERVED_G_BIT.

NOTE The specification for decoders to ignore the value of RESERVED_G_BIT is to enable the future definition of a backward-compatible usage of different values of this syntax element.

8.4.9 CHROMA_CENTERING_Y

CHROMA_CENTERING_Y is a 3-bit syntax element. It is present when INTERNAL_CLR_FMT is equal to YUV420. When CHROMA_CENTERING_Y is not present, its value should be inferred to be equal to 0.

CHROMA_CENTERING_Y indicates the positioning alignment of the chroma sampling grid with respect to the luma sampling grid. When present and in the range of 0 to 4, inclusive, CHROMA_CENTERING_Y indicates that the top-most sample of each column of each chroma array of the image is considered to be vertically positioned at the position CHROMA_CENTERING_Y with respect to the top-most sample of each column of the luma array, in units of quarter luma sample positions. When present and equal to 7, CHROMA_CENTERING_Y indicates that the positioning alignment is unknown or unspecified.

For example, when CHROMA_CENTERING_Y is equal to 0 and INTERNAL_CLR_FMT is equal to YUV420, each chroma sample is considered to be vertically located at the same position as the upper sample of a pair of luma samples.

The value of CHROMA_CENTERING_Y shall be equal to 0, 1, 2, 3, 4, or 7. The values 5 and 6 are reserved. Decoders conforming to this edition of this document should treat the values 5 and 6 as equivalent to the value 7.

NOTE CHROMA_CENTERING_Y is useful to aid in performing appropriate upsampling conversion from 4:2:0 to 4:2:2 or 4:4:4. However, the use of CHROMA_CENTERING_Y is not required for decoder conformance to this document, as the manner of performing such an upsampling process is outside the scope of this document.

8.4.10 RESERVED_H

RESERVED_H is a 4-bit syntax element that is present when INTERNAL_CLR_FMT is equal to one of YUV444, YUV422, or NCOMPONENT.

When RESERVED_H is present, its value shall be equal to 0. Decoders shall ignore the value of this syntax element.

NOTE The specification for decoders to ignore the value of RESERVED_H is to enable the future definition of a backward-compatible usage of different values of this syntax element.

8.4.11 NUM_COMPONENTS_MINUS1

NUM_COMPONENTS_MINUS1 is a 4-bit syntax element that is present when INTERNAL_CLR_FMT is equal to NCOMPONENT.

The number of components, NumComponents, is specified in pseudocode in Table 31.

8.4.12 NUM_COMPONENTS_EXTENDED_MINUS16

NUM_COMPONENTS_EXTENDED_MINUS16 is a 12-bit syntax element that is present when NUM_COMPONENTS_MINUS1 is equal to 0xF.

The number of components, NumComponents, is specified in pseudocode in Table 31.

Table 31 — Pseudocode to determine the number of components present in the codestream

DetermineNumComponents( ) {	**Reference**
if (INTERNAL_CLR_FMT = = NCOMPONENT)	
if (NUM_COMPONENTS_MINUS1 = = 0xF)	
NumComponents = NUM_COMPONENTS_EXTENDED_MINUS16 + 16	
else	
NumComponents = NUM_COMPONENTS_MINUS1 + 1	
else if (INTERNAL_CLR_FMT= = YONLY)	
NumComponents = 1	
else if (INTERNAL_CLR_FMT = = YUV420 | | INTERNAL_CLR_FMT = = YUV422 | | INTERNAL_CLR_FMT = = YUV444)	
NumComponents = 3	
else if (INTERNAL_CLR_FMT = = YUVK)	
NumComponents = 4	
}	

8.4.13 SHIFT_BITS

SHIFT_BITS is an 8-bit syntax element that is present when OUTPUT_BITDEPTH is equal to BD16, BD16S, or BD32S. SHIFT_BITS is used to left-shift the sample values in the output formatting stage as specified in subclause 9.10.7.

8.4.14 LEN_MANTISSA

LEN_MANTISSA is an 8-bit syntax element that is present when OUTPUT_BITDEPTH is equal to BD32F. It specifies the number of mantissa bits that are specified by the integer representation of floating point data as specified in subclause 9.10.7 prior to output conversion processing.

8.4.15 EXP_BIAS

EXP_BIAS is an 8-bit syntax element that is present when OUTPUT_BITDEPTH is equal to BD32F. This element specifies the bias of the exponent in the representation of floating point data as specified in subclause 9.10.7.

8.4.16 DC_IMAGE_PLANE_UNIFORM_FLAG

DC_IMAGE_PLANE_UNIFORM_FLAG is a 1-bit syntax element that specifies whether a single QP set is used for the DC band for all the macroblocks in the corresponding image plane. If DC_IMAGE_PLANE_UNIFORM_FLAG is equal to TRUE, a single QP set is used for the DC band of all tiles in the image (and hence for all macroblocks of all tiles), and this QP set is present in the image plane header. In this case, the DC QP set used for all tiles shall be set equal to the value specified in the image plane header. If DC_IMAGE_PLANE_UNIFORM_FLAG is equal to FALSE, the DC band of each tile may use a different QP set, and these QP sets are specified in the tile headers.

8.4.17 RESERVED_I_BIT

RESERVED_I_BIT is a 1-bit syntax element. It is a requirement of codestream conformance to this document that the value of RESERVED_I_BIT shall be equal to FALSE. Decoders shall ignore (remove from the codestream and discard) the value of RESERVED_I_BIT.

NOTE The purpose of the specification for decoders to ignore the value of RESERVED_I_BIT is to enable the future definition of a backward-compatible usage of different values of this syntax element.

8.4.18 LP_IMAGE_PLANE_UNIFORM_FLAG

LP_IMAGE_PLANE_UNIFORM_FLAG is a 1-bit syntax element that specifies whether a single QP set is used for the LP band. If LP_IMAGE_PLANE_UNIFORM_FLAG is equal to TRUE, a single QP set is used for the LP band of all tiles in the image (and hence for all macroblocks of all tiles), and this QP set is specified in the image plane header. In this case, the LP QP set for all tiles shall be set equal to the corresponding values specified in the image plane header. If LP_IMAGE_PLANE_UNIFORM_FLAG is equal to FALSE, the LP band of each tile may use a different QP set, and these QP sets are specified in the tile header.

8.4.19 RESERVED_J_BIT

RESERVED_J_BIT is a 1-bit syntax element. It is a requirement of codestream conformance to this document that the value of RESERVED_J_BIT shall be equal to FALSE. Decoders shall ignore (remove from the codestream and discard) the value of RESERVED_J_BIT.

NOTE The purpose of the specification for decoders to ignore the value of RESERVED_J_BIT is to enable the future definition of a backward-compatible usage of different values of this syntax element.

8.4.20 HP_IMAGE_PLANE_UNIFORM_FLAG

HP_IMAGE_PLANE_UNIFORM_FLAG is a 1-bit syntax element that specifies whether a single QP set is used for the HP band. If HP_IMAGE_PLANE_UNIFORM_FLAG is equal to TRUE, a single QP set shall be used for the HP band of all tiles in the image (and hence for all macroblocks of all tiles), and this QP set is specified in the image plane header. In this case, the HP QP set for all tiles shall be set equal to the values specified in the image plane header. If HP_IMAGE_PLANE_UNIFORM_FLAG is equal to FALSE, multiple QP sets may be used for the HP bands of each colour component in each tile, and these QP sets are specified in the tile headers.

8.4.21 BYTE_ALIGNMENT_BIT

BYTE_ALIGNMENT_BIT is a 1-bit syntax element. When it is present, its value shall be equal to 0. The value 1 is reserved.

8.4.22 DC_QP( )

8.4.22.1 Syntax structure

The DC_QP( ) syntax structure is specified by Table 32.

Table 32 — DC_QP( ) syntax structure

DC_QP( ) {	Descriptor	Reference
if (NumComponents != 1)		
COMPONENT_MODE	u(2)	8.4.22.2
if (COMPONENT_MODE = = UNIFORM)		
DC_QUANT	u(8)	8.4.22.3
else if (COMPONENT_MODE = = SEPARATE) {		
DC_QUANT_LUMA	u(8)	8.4.22.4
DC_QUANT_CHROMA	u(8)	8.4.22.5
} else if (COMPONENT_MODE = = INDEPENDENT)		
for (i = 0; i < NumComponents; i++)		
DC_QUANT_CH[i]	u(8)	8.4.22.6
}		

NOTE This function, DC_QP( ), is called from two locations: IMAGE_PLANE_HEADER or TILE_HEADER_DC. Care should be taken to use the correct value of DC_QUANT_CH[ ] when quantization parameters vary on a per tile basis.

8.4.22.2 COMPONENT_MODE

COMPONENT_MODE is a 2-bit syntax element that is present if NumComponents > 1, and specifies whether the colour components use or do not use the same QP set across components as specified in subclauses 8.4.22, 8.4.23 and 8.4.24. If NumComponents = = 1, the value of COMPONENT_MODE is inferred to be UNIFORM.

Table 33 — COMPONENT_MODE

Value	COMPONENT_MODE
0	UNIFORM
1	SEPARATE
2	INDEPENDENT
3	RESERVED

8.4.22.3 DC_QUANT

DC_QUANT is an 8-bit syntax element that is present if COMPONENT_MODE is equal to UNIFORM. In this case, the value of the DC QP for all the colour components shall be set to DC_QUANT.

8.4.22.4 DC_QUANT_LUMA

DC_QUANT_LUMA is an 8-bit syntax element that is present if COMPONENT_MODE is equal to SEPARATE. In this case, the value of the DC QP for the luma component shall be set to DC_QUANT_LUMA.

8.4.22.5 DC_QUANT_CHROMA

DC_QUANT_CHROMA is an 8-bit syntax element that is present if COMPONENT_MODE is equal to SEPARATE. In this case, the value of the DC QP for the chroma components shall be set to DC_QUANT_CHROMA.

8.4.22.6 DC_QUANT_CH[i]

DC_QUANT_CH[i] is an 8-bit syntax element that is present if COMPONENT_MODE is equal to INDEPENDENT. In this case, the value of the DC QP for the i-th colour component shall be set to DC_QUANT_CH[i].

8.4.23 LP_QP( )

8.4.23.1 Syntax structure

The LP_QP( ) syntax structure is specified by Table 34.

Table 34 – LP_QP( ) syntax structure

LP_QP( ) {	Descriptor	Reference
for (q = 0; q < NumLPQPs; q++) {		
if (NumComponents != 1)		
COMPONENT_MODE	u(2)	8.4.22.2
if (COMPONENT_MODE = = UNIFORM)		
LP_QUANT[q]	u(8)	8.4.23.2
else if (COMPONENT_MODE = = SEPARATE) {		
LP_QUANT_LUMA[q]	u(8)	8.4.23.3
LP_QUANT_CHROMA[q]	u(8)	8.4.23.4
} else if (COMPONENT_MODE = = INDEPENDENT)		
for (i = 0; i < NumComponents; i++)		
LP_QUANT_CH[i][q]	u(8)	8.4.23.5
}		
}		

NOTE This function, LP_QP( ), is called from two locations: IMAGE_PLANE_HEADER or TILE_HEADER_LOWPASS. Care should be taken to use the correct value of LP_QUANT_CH[ ][ ] when quantization parameters vary on a per tile basis.

8.4.23.2 LP_QUANT[q]

LP_QUANT[q] is an 8-bit syntax element that is present if COMPONENT_MODE is equal to UNIFORM. In this case, the value of the q-th LP QP for all the colour components shall be set to LP_QUANT[q].

8.4.23.3 LP_QUANT_LUMA[q]

LP_QUANT_LUMA[q] is an 8-bit syntax element that is present if COMPONENT_MODE is equal to SEPARATE. In this case, the value of the q-th LP QP for the luma component shall be set to LP_QUANT_LUMA[q].

8.4.23.4 LP_QUANT_CHROMA[q]

LP_QUANT_CHROMA[q] is an 8-bit syntax element that is present if COMPONENT_MODE is equal to SEPARATE. In this case, the value of the q-th LP QP for the chroma components shall be set to LP_QUANT_CHROMA[q].

8.4.23.5 LP_QUANT_CH[i][q]

LP_QUANT_CH[i][q] is an 8-bit syntax element that is present if COMPONENT_MODE is equal to INDEPENDENT. In this case, the value of the q-th LP QP for the i-th colour component shall be set to LP_QUANT_CH[i][q].

8.4.24 HP_QP( )

8.4.24.1 Syntax structure

The HP_QP( ) syntax structure is specified by Table 35.

Table 35 — HP_QP( ) syntax structure

HP_QP( ) {	Descriptor	Reference
for (q = 0; q < NumHPQPs; q++) {		
if (NumComponents != 1)		
COMPONENT_MODE	u(2)	8.4.22.2
if (COMPONENT_MODE = = UNIFORM)		
HP_QUANT[q]	u(8)	8.4.24.2
else if (COMPONENT_MODE = = SEPARATE) {		
HP_QUANT_LUMA[q]	u(8)	8.4.24.3
HP_QUANT_CHROMA[q]	u(8)	8.4.24.4
} else if (COMPONENT_MODE = = INDEPENDENT)		
for (i = 0; i < NumComponents; i++)		
HP_QUANT_CH[i][q]	u(8)	8.4.24.5
}		
}		

NOTE This function, HP_QP( ), is called from two locations: IMAGE_PLANE_HEADER or TILE_HEADER_HIGHPASS. Care should be taken to use the correct value of HP_QUANT_CH[ ][ ] when quantization parameters vary on a per tile basis.

8.4.24.2 HP_QUANT[q]

HP_QUANT[q] is an 8-bit syntax element that is present if COMPONENT_MODE is equal to UNIFORM. In this case, the value of the q-th HP QP for all the colour components shall be set to HP_QUANT[q].

8.4.24.3 HP_QUANT_LUMA[q]

HP_QUANT_LUMA[q] is an 8-bit syntax element that is present if COMPONENT_MODE is equal to SEPARATE. In this case, the value of the q-th HP QP for the luma component shall be set to HP_QUANT_LUMA[q].

8.4.24.4 HP_QUANT_CHROMA[q]

HP_QUANT_CHROMA[q] is an 8-bit syntax element that is present if COMPONENT_MODE is equal to SEPARATE. In this case, the value of the q-th HP QP for the chroma components shall be set to HP_QUANT_CHROMA[q].

8.4.24.5 HP_QUANT_CH[i][q]

HP_QUANT_CH[i][q] is an 8-bit syntax element that is present if COMPONENT_MODE is equal to INDEPENDENT. In this case, the value of the q-th HP QP for the i-th colour component shall be set to HP_QUANT_CH[i][q].

8.5 INDEX_TABLE_TILES( )

8.5.1 Syntax structure

The INDEX_TABLE_TILES( ) syntax structure is specified by Table 36.

Table 36 — INDEX_TABLE_TILES( ) syntax structure

INDEX_TABLE_TILES( ) {	**Descriptor**	**Reference**
if (FREQUENCY_MODE_CODESTREAM_FLAG = = FALSE)		
valueNumIndexTableEntries = (NUM_HOR_TILES_MINUS1 + 1) * (NUM_VER_TILES_MINUS1 + 1)		
else /* FREQUENCY_MODE_CODESTREAM_FLAG = = TRUE */		
valueNumIndexTableEntries = (NUM_HOR_TILES_MINUS1 + 1) * (NUM_VER_TILES_MINUS1 + 1) * NumBandsOfPrimary		
INDEX_TABLE_STARTCODE	u(16)	8.5.2
for (n = 0; n < valueNumIndexTableEntries; n++)		
IndexOffsetTile[n] = VLW_ESC( )		8.5.3
}		

8.5.2 INDEX_TABLE_STARTCODE

INDEX_TABLE_STARTCODE is a 16-bit syntax element which indicates the start of the INDEX_TABLE_TILES( ). This element shall have the value 0x0001. Other values of INDEX_TABLE_STARTCODE are reserved.

8.5.3 IndexOffsetTile[n]

IndexOffsetTile[n] is a variable that specifies the offset of the n-th tile packet from the start of the coded image data. The value of this variable is determined by a VLW_ESC( ) syntax structure.

The ordering of this information is as follows: Index offset elements corresponding to each tile shall be consecutively ordered in low-to-high order of the frequency, i.e. DC followed by low-pass, high-pass, and flexbits. IndexOffsetTile entries are ordered in the raster scan order of the respective tiles, i.e. left-to-right for the top row of the tile, followed by left-to-right for the next row, and so on through the bottom row of the tile.

For spatial mode codestreams, only one IndexOffsetTile is sent per tile. For images with missing sub-bands (such as when BANDS_PRESENT is not equal to ALL), Index Offset elements are sent only for the sub-bands that are present. An example of this syntax element for an image with four spatial tiles and two frequency bands (DC and LP, i.e. BANDS_PRESENT is equal to NOHIGHPASS) is specified below. Here, pDCTile[n] and pLPTile[n] are the index offset elements of the DC and LP bands of tile n:

pDCTile[0] pLPTile[0] pDCTile[1] pLPTile[1] pDCTile[2] pLPTile[2] pDCTile[3] pLPTile[3]

When the number of tile packets is 1, the index offset of the only packet is 0.

8.6 PROFILE_LEVEL_INFO( )

8.6.1 Syntax structure

The PROFILE_LEVEL_INFO( ) syntax structure is specified by Table 37.

Table 37 — PROFILE_LEVEL_INFO( ) syntax structure

PROFILE_LEVEL_INFO( ) {	Descriptor	Reference
numBytes = 0		
for (iLast = 0; iLast = = 0; iLast = LAST_FLAG) {		
PROFILE_IDC	u(8)	8.6.2
LEVEL_IDC	u(8)	8.6.3
RESERVED_L	u(15)	8.6.4
LAST_FLAG	u(1)	8.6.5
numBytes += 4		
}		
return numBytes		
}		

8.6.2 PROFILE_IDC

PROFILE_IDC (when present) is an 8-bit syntax element. When present, the values of PROFILE_IDC and LEVEL_IDC indicate a set of profile and level constraints as specified in Annex B.

8.6.3 LEVEL_IDC

LEVEL_IDC (when present) is an 8-bit syntax element. When present, the values of PROFILE_IDC and LEVEL_IDC indicate a set of profile and level constraints as specified in Annex B.

8.6.4 RESERVED_L

RESERVED_L is a 15-bit syntax element. When present, the value of RESERVED_L shall be equal to 0. Other values are reserved. Decoders shall ignore the value of this syntax element.

NOTE The purpose of the specification for decoders to ignore the value of RESERVED_L is to enable the future definition of a backward-compatible usage of different values of this syntax element.

No particular combination of PROFILE_IDC, LEVEL_IDC, and RESERVED_L shall appear more than once in the PROFILE_LEVEL_INFO( ) syntax structure.

8.6.5 LAST_FLAG

LAST_FLAG (when present) is a 1-bit syntax element. It indicates whether the preceding PROFILE_IDC, LEVEL_IDC, and RESERVED_L syntax elements are the last such syntax elements in the PROFILE_LEVEL_INFO( ) syntax structure.

8.7 CODED_TILES( )

8.7.1 Syntax structure

The CODED_TILES( ) syntax structure is specified by Table 38.

Table 38 — CODED_TILES( ) syntax structure

CODED_TILES( ) {	**Descriptor**	**Reference**
if (FREQUENCY_MODE_CODESTREAM_FLAG = = FALSE)		
for (n = 0; n < (NUM_HOR_TILES_MINUS1 + 1) * (NUM_VER_TILES_MINUS1 + 1); n++) {		
NumMBInCurrentTile = NumMBInTile[n]		
POS_SEEK(IndexOffsetTile[n])		
TILE_SPATIAL( )		8.7.2
}		
else { /* FREQUENCY_MODE_CODESTREAM_FLAG = = TRUE */		
for (n = 0; n < (NUM_HOR_TILES_MINUS1 + 1) * (NUM_VER_TILES_MINUS1 + 1); n++) {		
NumMBInCurrentTile = NumMBInTile[n]		
POS_SEEK(IndexOffsetTile[n * NumBandsOfPrimary])		
TILE_DC( )		8.7.3
}		
if (NumBandsOfPrimary > 1)		
for (n = 0; n < (NUM_HOR_TILES_MINUS1 + 1) * (NUM_VER_TILES_MINUS1 + 1); n++) {		
NumMBInCurrentTile = NumMBInTile[n]		
POS_SEEK(IndexOffsetTile[n * NumBandsOfPrimary + 1])		
TILE_LOWPASS( )		8.7.5
}		
if (NumBandsOfPrimary > 2)		
for (n = 0; n < (NUM_HOR_TILES_MINUS1 + 1) * (NUM_VER_TILES_MINUS1 + 1); n++) {		
NumMBInCurrentTile = NumMBInTile[n]		
POS_SEEK(IndexOffsetTile[n * NumBandsOfPrimary + 2])		
TILE_HIGHPASS( )		8.7.7
}		
if (NumBandsOfPrimary > 3)		
for (n = 0; n < (NUM_HOR_TILES_MINUS1 + 1) * (NUM_VER_TILES_MINUS1 + 1); n++) {		
NumMBInCurrentTile = NumMBInTile[n]		
POS_SEEK(IndexOffsetTile[n * NumBandsOfPrimary + 3])		
TILE_FLEXBITS( )		8.7.9
}		
}		
}		

8.7.2 TILE_SPATIAL( )

The TILE_SPATIAL( ) syntax structure is specified by Table 39.

Table 39 — TILE_SPATIAL( ) syntax structure

TILE_SPATIAL( ) {	Descriptor	Reference
TILE_STARTCODE	u(24)	8.7.10.1
ARBITRARY_BYTE	u(8)	8.7.10.2
if (TRIM_FLEXBITS_FLAG)		
TRIM_FLEXBITS	u(4)	8.7.10.3
IsCurrPlaneAlphaFlag = FALSE		
TILE_HEADER_DC( )		8.7.4
if (BANDS_PRESENT != DCONLY) { /* BANDS_PRESENT of Primary Plane */		
TILE_HEADER_LOWPASS( )		8.7.6
if (BANDS_PRESENT != NOHIGHPASS)		
TILE_HEADER_HIGHPASS( )		8.7.8
}		
if (ALPHA_IMAGE_PLANE_FLAG) {		
IsCurrPlaneAlphaFlag = TRUE		
TILE_HEADER_DC( )		8.7.4
if (BANDS_PRESENT != DCONLY) { /* BANDS_PRESENT of Alpha plane */		
TILE_HEADER_LOWPASS( )		8.7.6
if (BANDS_PRESENT != NOHIGHPASS)		
TILE_HEADER_HIGHPASS( )		8.7.8
}		
}		
for (n = 0; n < NumMBInCurrentTile; n++) {		
IsCurrPlaneAlphaFlag = FALSE		
if (BANDS_PRESENT != DCONLY) {		
if (NumLPQPs > 1 && USE_DC_QP_FLAG = = FALSE)		
LP_QP_INDEX[n] = DECODE_QP_INDEX(NumLPQPs)		8.7.10.10
if (BANDS_PRESENT != NOHIGHPASS && NumHPQPs > 1 && USE_LP_QP_FLAG = = FALSE)		
HP_QP_INDEX[n] = DECODE_QP_INDEX(NumHPQPs)		8.7.10.10
}		
MB_DC( )		8.7.11
if (BANDS_PRESENT != DCONLY) {		
MB_LP( )		8.7.16.1
if (BANDS_PRESENT != NOHIGHPASS) {		
MB_CBPHP( )		8.7.17.2
MB_HP_FLEX( )		8.7.18.3
}		
}		
if (ALPHA_IMAGE_PLANE_FLAG) {		
IsCurrPlaneAlphaFlag = TRUE		
if (BANDS_PRESENT != DCONLY) { /* BANDS_PRESENT of Alpha plane*/		
if (NumLPQPs > 1 && USE_DC_QP_FLAG = = FALSE)		
LP_QP_INDEX[n] = DECODE_QP_INDEX(NumLPQPs)		8.7.10.10
if (BANDS_PRESENT != NOHIGHPASS && NumHPQPs > 1 && USE_LP_QP_FLAG = = FALSE)		
HP_QP_INDEX[n] = DECODE_QP_INDEX(NumHPQPs)		8.7.10.10
}		
MB_DC( )		8.7.11
if (BANDS_PRESENT != DCONLY) {		
MB_LP( )		8.7.16.1
if (BANDS_PRESENT != NOHIGHPASS) {		
MB_CBPHP( )		8.7.17.2

TILE_SPATIAL( ) {	Descriptor	Reference
MB_HP_FLEX( )		8.7.18.3
}		
}		
} /* for if (ALPHA_IMAGE_PLANE_FLAG) */		
} /* for (n = 0; n < NumMBInCurrentTile; n++) */		
while (!IS_BYTE_ALIGNED( ))		
BYTE_ALIGNMENT_BIT	u(1)	8.4.21
}		

8.7.3 TILE_DC( )

The TILE_DC( ) syntax structure is specified by Table 40.

Table 40 — TILE_DC( ) syntax structure

TILE_DC( ) {	Descriptor	Reference
TILE_STARTCODE	u(24)	8.7.10.1
ARBITRARY_BYTE	u(8)	8.7.10.2
IsCurrPlaneAlphaFlag = FALSE		
TILE_HEADER_DC( )		8.7.4
if (ALPHA_IMAGE_PLANE_FLAG) {		
IsCurrPlaneAlphaFlag = TRUE		
TILE_HEADER_DC( )		8.7.4
}		
for (n = 0; n < NumMBInCurrentTile; n++) {		
IsCurrPlaneAlphaFlag = FALSE		
MB_DC( )		8.7.11
if (ALPHA_IMAGE_PLANE_FLAG) {		
IsCurrPlaneAlphaFlag = TRUE		
MB_DC( )		8.7.11
}		
}		
while (!IS_BYTE_ALIGNED( ))		
BYTE_ALIGNMENT_BIT	u(1)	8.4.21
}		

8.7.4 TILE_HEADER_DC( )

The TILE_HEADER_DC( ) syntax structure is specified by Table 41.

Table 41 — TILE_HEADER_DC( ) syntax structure

TILE_HEADER_DC( ) {	Descriptor	Reference
if (DC_IMAGE_PLANE_UNIFORM_FLAG = = FALSE)		
DC_QP( )		8.4.22
}		

8.7.5 TILE_LOWPASS( )

The TILE_LOWPASS( ) syntax structure is specified by Table 42.

Table 42 — TILE_LOWPASS( ) syntax structure

TILE_LOWPASS( ) {	Descriptor	Reference
TILE_STARTCODE	u(24)	8.7.10.1
ARBITRARY_BYTE	u(8)	8.7.10.2
IsCurrPlaneAlphaFlag = FALSE		
if (BANDS_PRESENT != DCONLY) /* BANDS_PRESENT of primary image plane */		
TILE_HEADER_LOWPASS( )		8.7.6
if (ALPHA_IMAGE_PLANE_FLAG) {		
IsCurrPlaneAlphaFlag = TRUE		
if (BANDS_PRESENT != DCONLY) /* BANDS_PRESENT of alpha image plane */		
TILE_HEADER_LOWPASS( )		8.7.6
}		
for (n = 0; n < NumMBInCurrentTile; n++) {		
IsCurrPlaneAlphaFlag = FALSE		
if (BANDS_PRESENT != DCONLY) { /* BANDS_PRESENT of primary image plane */		
if (NumLPQPs > 1 && USE_DC_QP_FLAG = = FALSE)		
LP_QP_INDEX[n] = DECODE_QP_INDEX(NumLPQPs) /* primary image plane */		8.7.10.10
MB_LP( )		8.7.16.1
}		
if (ALPHA_IMAGE_PLANE_FLAG) {		
IsCurrPlaneAlphaFlag = TRUE		
if (BANDS_PRESENT != DCONLY) { /* BANDS_PRESENT of alpha image plane */		
if (NumLPQPs > 1 && USE_DC_QP_FLAG = = FALSE)		
LP_QP_INDEX[n] = DECODE_QP_INDEX(NumLPQPs) /* alpha image plane */		8.7.10.10
MB_LP( )		8.7.16.1
}		
}		
}		
while (!IS_BYTE_ALIGNED( ))		
BYTE_ALIGNMENT_BIT	u(1)	8.4.21
}		

8.7.6 TILE_HEADER_LOWPASS( )

The TILE_HEADER_LOWPASS( ) syntax structure is specified by Table 43.

Table 43 — TILE_HEADER_LOWPASS( ) syntax structure

TILE_HEADER_LOWPASS( ) {	Descriptor	Reference
if (LP_IMAGE_PLANE_UNIFORM_FLAG = = FALSE) {		
USE_DC_QP_FLAG	u(1)	8.7.10.4
if (USE_DC_QP_FLAG)		
NumLPQPs = 1		
else {		
NUM_LP_QPS_MINUS1	u(4)	8.7.10.5
NumLPQPs = NUM_LP_QPS_MINUS1 + 1		
LP_QP( )		8.4.23
}		
}		
}		

8.7.7 TILE_HIGHPASS( )

The TILE_HIGHPASS( ) syntax structure is specified by Table 44.

Table 44 — TILE_HIGHPASS( ) syntax structure

TILE_HIGHPASS( ) {	**Descriptor**	**Reference**
TILE_STARTCODE	u(24)	8.7.10.1
ARBITRARY_BYTE	u(8)	8.7.10.2
IsCurrPlaneAlphaFlag = FALSE		
if (BANDS_PRESENT != DCONLY && BANDS_PRESENT != NOHIGHPASS) /* BANDS_PRESENT of primary image plane */		
TILE_HEADER_HIGHPASS( )		8.7.8
if (ALPHA_IMAGE_PLANE_FLAG) {		
IsCurrPlaneAlphaFlag = TRUE		
if (BANDS_PRESENT != DCONLY && BANDS_PRESENT != NOHIGHPASS) /* BANDS_PRESENT of alpha image plane */		
TILE_HEADER_HIGHPASS( )		8.7.8
}		
for (n = 0; n < NumMBInCurrentTile; n++) {		
IsCurrPlaneAlphaFlag = FALSE		
if (BANDS_PRESENT != DCONLY && BANDS_PRESENT != NOHIGHPASS) { /* BANDS_PRESENT of primary image plane */		
if (NumHPQPs > 1 && USE_LP_QP_FLAG = = FALSE)		
HP_QP_INDEX[n] = DECODE_QP_INDEX(NumHPQPs)		8.7.10.10
MB_CBPHP( )		8.7.17.2
MB_HP( )		8.7.18.2
}		
if (ALPHA_IMAGE_PLANE_FLAG) {		
IsCurrPlaneAlphaFlag = TRUE		
if (BANDS_PRESENT != DCONLY && BANDS_PRESENT != NOHIGHPASS) { /* BANDS_PRESENT of alpha image plane */		
if (NumHPQPs > 1 && USE_LP_QP_FLAG = = FALSE)		
HP_QP_INDEX[n] = DECODE_QP_INDEX(NumHPQPs)		8.7.10.10
MB_CBPHP( )		8.7.17.2
MB_HP( )		8.7.18.2
}		
}		
}		
while (!IS_BYTE_ALIGNED( ))		
BYTE_ALIGNMENT_BIT	u(1)	8.4.21
}		

8.7.8 TILE_HEADER_HIGHPASS( )

The TILE_HEADER_HIGHPASS( ) syntax structure is specified by Table 45.

Table 45 — TILE_HEADER_HIGHPASS( ) syntax structure

TILE_HEADER_HIGHPASS( ) {	Descriptor	Reference
if (HP_IMAGE_PLANE_UNIFORM_FLAG = = FALSE) {		
USE_LP_QP_FLAG	u(1)	8.7.10.6
if (USE_LP_QP_FLAG)		
NumHPQPs = NumLPQPs		
else {		
NUM_HP_QPS_MINUS1	u(4)	8.7.10.7
NumHPQPs = NUM_HP_QPS_MINUS1 + 1		
HP_QP( )		8.4.24
}		
}		
}		

8.7.9 TILE_FLEXBITS( )

The TILE_FLEXBITS( ) syntax structure is specified by Table 46.

Table 46 — TILE_FLEXBITS( ) syntax structure

TILE_FLEXBITS( ) {	Descriptor	Reference
TILE_STARTCODE	u(24)	8.7.10.1
ARBITRARY_BYTE	u(8)	8.7.10.2
if (TRIM_FLEXBITS_FLAG)		
TRIM_FLEXBITS	u(4)	8.7.10.3
for (n = 0; n < NumMBInCurrentTile; n++) {		
IsCurrPlaneAlphaFlag = FALSE		
if (BANDS_PRESENT = = ALL) /* BANDS_PRESENT of primary image plane */		
MB_FLEXBITS( )		8.7.19.1
if (ALPHA_IMAGE_PLANE_FLAG) {		
IsCurrPlaneAlphaFlag = TRUE		
if (BANDS_PRESENT = = ALL) /* BANDS_PRESENT of alpha image plane */		
MB_FLEXBITS( )		8.7.19.1
}		
}		
while (!IS_BYTE_ALIGNED( ))		
BYTE_ALIGNMENT_BIT	u(1)	8.4.21
}		

8.7.10 Tile-level semantics

8.7.10.1 TILE_STARTCODE

TILE_STARTCODE is a 24-bit syntax element that is present at the beginning of tile-level syntax structures. The value of TILE_STARTCODE shall be equal to 0x000001.

NOTE 1 Decoders can check the value of TILE_STARTCODE to ensure that it has the correct value. If some value other than 0x000001 is detected, decoders can infer the presence of an error condition. It is suggested that the subsequent data for any tiles that begin with an incorrect value of TILE_STARTCODE is discarded. When such an error condition is detected and the tile is not a flexbits tile-packet, it is suggested for the decoder to infer zero values for the transform coefficients in such a packet. When such an error condition is detected and the tile is a flexbits tile-packet, it is suggested for the decoder to infer zero values for all flexbits of such a tile-packet. Alternative approaches to handling such conditions can be preferable in some uses.

NOTE 2 There is no guarantee that a byte-aligned 24-bit pattern evaluating to 0x000001 will not occur at any other location in the codestream. Therefore, TILE_STARTCODE can only be used to reconfirm the start of a tile in conjunction with the index table entries and not as a guaranteed indicator of the start of a tile.

8.7.10.2 ARBITRARY_BYTE

ARBITRARY_BYTE is an 8-bit syntax element. This syntax element may have any value. The value of this syntax element shall be ignored by the decoder.

8.7.10.3 TRIM_FLEXBITS

TRIM_FLEXBITS is a 4-bit syntax element that is present if TRIM_FLEXBITS_FLAG is equal to TRUE. Otherwise, TRIM_FLEXBITS shall be inferred to be equal to 0.

NOTE The number of bits per transform coefficient that are present in the flexbits tile-packet is specified by the value of (ModelBitsMBHP[MBx][MBy][i] − TRIM_FLEXBITS) as specified in subclause 8.7.18.3 and subclause 8.7.19.1.

8.7.10.4 USE_DC_QP_FLAG

USE_DC_QP_FLAG is a 1-bit syntax element which specifies whether the LP band uses the same QP set as the DC band. If USE_DC_QP_FLAG is equal to TRUE, the values of the LP QP set are set to those of the DC band QP set; otherwise, the values of the LP QP set are explicitly specified in the codestream. When USE_DC_QP_FLAG is not present, its value shall be inferred to be equal to FALSE.

8.7.10.5 NUM_LP_QPS_MINUS1

NUM_LP_QPS_MINUS1 is a 4-bit syntax element that is present if LP_IMAGE_PLANE_UNIFORM_FLAG is equal to FALSE. This syntax element specifies the number of LP band QPs, per colour component in each tile, minus 1.

8.7.10.6 USE_LP_QP_FLAG

USE_LP_QP_FLAG is a 1-bit syntax element that specifies whether the HP band uses the same QP sets as the LP band. If USE_LP_QP_FLAG is equal to TRUE, the values of the HP QP sets are set to those of the LP band QP sets; otherwise, the values of the HP QP sets are explicitly specified in the codestream. When USE_LP_QP_FLAG is not present, its value shall be inferred to be equal to FALSE.

8.7.10.7 NUM_HP_QPS_MINUS1

NUM_HP_QPS_MINUS1 is a 4-bit syntax element that is present if HP_IMAGE_PLANE_UNIFORM_FLAG is equal to FALSE. This syntax element specifies the number of HP band QPs, per colour component in each tile, minus 1.

8.7.10.8 LP_QP_INDEX[n]

LP_QP_INDEX[n] is a variable-length syntax element that is present when BANDS_PRESENT is not equal to DCONLY, NumLPQPs is greater than 1, and USE_DC_QP_FLAG is equal to FALSE. It specifies the QP index used for the LP band of the n-th macroblock, in raster scan order, of the tile. The LP band QP for each colour component shall be derived from the q-th QP set when LP_QP_INDEX[n] takes the value q. The LP QP index is parsed using the syntax structure DECODE_QP_INDEX( ). When LP_QP_INDEX[n] is not present, its value shall be inferred to be equal to 0.

8.7.10.9 HP_QP_INDEX[n]

HP_QP_INDEX[n] is a variable-length syntax element that is present when BANDS_PRESENT is not equal to DCONLY or NOHIGHPASS, NumHPQPs is greater than 1, and USE_LP_QP_FLAG is equal to FALSE. It specifies the QP index for the HP band of the n-th macroblock, in raster scan order, of the tile. The HP band QP for each colour component shall be derived from the q-th QP set when HP_QP_INDEX[n] takes the value q. The HP QP index is parsed using the syntax structure DECODE_QP_INDEX( ). When HP_QP_INDEX[n] is not present, its value shall be inferred as follows:

— If USE_LP_QP_FLAG is equal to TRUE, HP_QP_INDEX[n] shall be inferred to be equal to LP_QP_INDEX[n].

— Otherwise, HP_QP_INDEX[n] shall be inferred to be equal to 0.

8.7.10.10 DECODE_QP_INDEX( )

DECODE_QP_INDEX( ) is called when there is a table of quantization parameters associated with either the LP or HP band. When called, DECODE_QP_INDEX( ) returns the index into this table, that represents the quantization parameter to be used. This syntax structure takes the parameter iNumQP, which specifies the size of the relevant quantization parameter table.

The syntax structure DECODE_QP_INDEX( ) is specified by Table 47.

Table 47 — DECODE_QP_INDEX( ) syntax structure

DECODE_QP_INDEX(iNumQP) {	Descriptor	Reference
iBitsQPIndex[ ] = {0, 0, 1, 1, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4}		
iBits = iBitsQPIndex[iNumQP]		
IS_QPINDEX_NONZERO_FLAG	u(1)	8.7.10.11
if (IS_QPINDEX_NONZERO_FLAG = = FALSE)		
iQPIndex = 0		
else { /* iBits > 0 as iNumQP > 1 */		
QPINDEX_REF	u(iBits)	8.7.10.12
iQPIndex = QPINDEX_REF + 1		
}		
return iQPIndex		
}		

8.7.10.11 IS_QPINDEX_NONZERO_FLAG

IS_QPINDEX_NONZERO_FLAG is a 1-bit syntax element. If IS_QPINDEX_NONZERO_FLAG is equal to TRUE, the QP index is derived from the syntax element QPINDEX_REF. Otherwise, the QP index is set to 0.

8.7.10.12 QPINDEX_REF

QPINDEX_REF is a syntax element that specifies the QP index when IS_QPINDEX_NONZERO_FLAG is equal to TRUE. The value of QPINDEX_REF shall be in the range of 0 to iNumQP−2. All other values are reserved.

8.7.11 MB_DC( )

The MB_DC( ) syntax structure is specified by Table 48.

Table 48 — MB_DC( ) syntax structure

MB_DC( ) {	Descriptor	Reference
/* IsCurrPlaneAlphaFlag is equal to TRUE for parsing alpha image plane, and IsCurrPlaneAlphaFlag is equal to FALSE for parsing primary image plane */		
iBand = 0 /* 0 = DC band, 1 = LP band, 2 = HP band */		
bInitializeContext = (IsMBLeftEdgeOfTileFlag && IsMBTopEdgeOfTileFlag)		
if (bInitializeContext) {		
InitializeDCVLC( )		8.8.3.1
InitializeModelMB(ModelDC, iBand)		8.12.1
}		
iLapMean[ ] = {0, 0}		
if (INTERNAL_CLR_FMT = = YONLY | | INTERNAL_CLR_FMT = = YUVK | | INTERNAL_CLR_FMT = = NCOMPONENT)		
for (n=0; n < NumComponents; n++) {		
IS_DC_CH_FLAG	u(1)	8.7.14.1

MB_DC( ) {	Descriptor	Reference
bAbsLevel = IS_DC_CH_FLAG		
m = 0		
if (n != 0)		
m = 1		
if (bAbsLevel)		
iLapMean[m] = iLapMean[m] + 1		
bChroma = FALSE /* Luma */		
DCInput[n] = DECODE_DC(ModelDC.MBits[m], iBand, bChroma, bAbsLevel)		8.7.12
}		
else { /* INTERNAL_CLR_FMT is not YONLY, YUVK, or NCOMPONENT */		
VAL_DC_YUV /* Parse with VAL_DC_YUV Code table */	e(v)	8.7.14.2
/* Luma (Y) DC Parsing */		
bAbsLevel = ((VAL_DC_YUV & 4) != 0)		
if (bAbsLevel)		
iLapMean[0] = iLapMean[0] + 1		
bChroma = FALSE /* i.e., Luma */		
DCInput[0] = DECODE_DC(ModelDC.MBits[0], iBand, bChroma, bAbsLevel)		8.7.12
/* First chroma (U) DC Parsing */		
bAbsLevel = ((VAL_DC_YUV & 2) != 0)		
if (bAbsLevel)		
iLapMean[1] = iLapMean[1] + 1		
bChroma = TRUE /* i.e., Chroma */		
DCInput[1] = DECODE_DC(ModelDC.MBits[1], iBand, bChroma, bAbsLevel)		8.7.12
/* Second chroma (V) DC Parsing */		
bAbsLevel = ((VAL_DC_YUV & 1) != 0)		
if (bAbsLevel)		
iLapMean[1] = iLapMean[1] + 1 /* Same index for U and V */		
bChroma = TRUE /* i.e., Chroma */		
DCInput[2] = DECODE_DC(ModelDC.MBits[1], iBand, bChroma, bAbsLevel)		8.7.12
}		
UpdateModelMB(iLapMean[ ], ModelDC, iBand)		8.12.2
bResetContext = (MBx = = (LeftMBIndexOfTile[TileIndexx + 1] − 1) | | (MBx − LeftMBIndexOfTile[TileIndexx]) % 16 = = 0)		
if (bResetContext)		
AdaptDC( )		8.8.4.1
}		

8.7.12 DECODE_DC( )

Table 49 — DECODE_DC( ) syntax structure

DECODE_DC(iModelBits, iBand, bChroma, bAbsLevel) {	Descriptor	Reference
iDC = 0		
iContext = 0		
if (bAbsLevel)		
iDC = DECODE_ABS_LEVEL(iBand, bChroma, iContext) − 1		8.7.13
if (iModelBits) {		
DC_REF	u(iModelBits)	8.7.14.3
iDCRef = DC_REF		
iDC = (iDC<< iModelBits) | iDCRef		
}		
if (iDC != 0) {		
SIGN_FLAG	u(1)	8.7.14.4
if (SIGN_FLAG)		
iDC = −iDC		

DECODE_DC(iModelBits, iBand, bChroma, bAbsLevel) {	Descriptor	Reference
}		
return iDC		
}		

8.7.13 DECODE_ABS_LEVEL( )

Table 50 — DECODE_ABS_LEVEL( ) syntax structure

DECODE_ABS_LEVEL(iBand, bChroma, iContext) {	Descriptor	Reference
/* **sAdaptVLC** is local instance of AdaptiveVLC data structure */		
if (iBand = = 0) /* DC */		
if (bChroma)		
sAdaptVLC = AbsLevelIndDCChr		
else		
sAdaptVLC = AbsLevelIndDCLum		
else if (iBand = = 1) /* LP */		
if (iContext)		
sAdaptVLC = AbsLevelIndLP1		
else		
sAdaptVLC = AbsLevelIndLP0		
else if (iBand = = 2) /* HP */		
if (iContext)		
sAdaptVLC = AbsLevelIndHP1		
else		
sAdaptVLC = AbsLevelIndHP0		
iRemap[ ] = {2, 3, 4, 6, 10, 14}		
iFixedLen[ ] = {0, 0, 1, 2, 2, 2}		
ABS_LEVEL_INDEX /* Parse with table indexed by **sAdaptVLC**.TableIndex */	ae(v)	8.7.14.5
sAdaptVLC.DiscrimVal1 += AbslevelIndexDelta[0][ABS_LEVEL_INDEX]		Table 86
if (ABS_LEVEL_INDEX < 6) {		
iFixed = iFixedLen[ABS_LEVEL_INDEX]		
iLevel = iRemap[ABS_LEVEL_INDEX]		
if (iFixed > 0) {		
LEVEL_REF	u(iFixed)	8.7.14.6
iLevel += LEVEL_REF		
}		
} else { /* Escape mode */		
FIXED_NUM	u(4)	8.7.14.7
iFixed = FIXED_NUM + 4		
if (iFixed = = 19) {		
FIXED_NUM_EXT	u(2)	8.7.14.8
iFixed += FIXED_NUM_EXT		
if (iFixed = = 22) {		
FIXED_NUM_EXT2	u(3)	8.7.14.9
iFixed += FIXED_NUM_EXT2		
}		
}		
LEVEL_REF	u(iFixed)	8.7.14.6
iLevel = 2 + (1 << iFixed) + LEVEL_REF		
}		
return iLevel		
}		

8.7.14 Macroblock DC( ) semantics

8.7.14.1 IS_DC_CH_FLAG

IS_DC_CH_FLAG is a 1-bit syntax element that is present if INTERNAL_CLR_FMT is one of YONLY, YUVK, or NCOMPONENT. If IS_DC_CH_FLAG is equal to TRUE, the variable-length coded part of the DC coefficient of the corresponding colour component is specified in the codestream. If IS_DC_CH_FLAG is equal to FALSE, the variable-length coded part of the DC coefficient of the corresponding colour component is equal to 0.

8.7.14.2 VAL_DC_YUV

VAL_DC_YUV is a variable-length syntax element that is present if INTERNAL_CLR_FMT is not one of YONLY, YUVK, or NCOMPONENT. The value of VAL_DC_YUV is a 3-bit number, which jointly specifies the zero/non-zero status of the DC coefficients of the Y, U and V, respectively (i.e., (VAL_DC_YUV & 4) specifies the Y, (VAL_DC_YUV & 2) specifies the U, and (VAL_DC_YUV & 1) specifies the V). The code table used in parsing VAL_DC_YUV is specified in Table 51.

Table 51 — Code table for VAL_DC_YUV

Code	Value
10	0
001	1
0000 1	2
0001	3
11	4
010	5
0000 0	6
011	7

8.7.14.3 DC_REF

DC_REF is a syntax element which specifies the FLC refinement in the DC value. The number of bits, iModelBits, needed to specify DC_REF is computed as specified in subclause 8.7.12.

8.7.14.4 SIGN_FLAG

SIGN_FLAG is a 1-bit syntax element which specifies the sign of a coefficient. If SIGN_FLAG is equal to TRUE, the coefficient is negative. Otherwise, the coefficient is positive.

8.7.14.5 ABS_LEVEL_INDEX

ABS_LEVEL_INDEX is a variable-length syntax element that has a value in the range of 0 to 6, inclusive. This syntax element is used in the computation of the VLC-coded part of the transform coefficient. The VLC-coded part of a transform coefficient is parsed in two stages: the initial level value and the VLC refinement. If ABS_LEVEL_INDEX is less than 6, the initial level and the number of bits required to specify the VLC refinement are specified by this syntax element. If ABS_LEVEL_INDEX is equal to 6, further syntax elements are parsed to determine the initial level value and the number of bits required to specify the VLC refinement, as specified in subclause 8.7.13.

The coding of this syntax element uses one of two tables, adaptively determined as specified by the parsing process (see subclause 8.8). The two code tables are specified in Table 52.

Table 52 — Code table for ABS_LEVEL_INDEX

Code 0	Code 1	Value
01	1	0
10	01	1
11	001	2
001	0001	3
0001	00001	4
00000	000000	5
00001	000001	6

8.7.14.6 LEVEL_REF

LEVEL_REF is a syntax element which specifies the VLC refinement. The number of bits, iFixed, needed to specify this syntax element is computed as specified in subclause 8.7.13 from ABS_LEVEL_INDEX if ABS_LEVEL_INDEX is less than 6, or from FIXED_NUM, FIXED_NUM_EXT, and FIXED_NUM_EXT2 if ABS_LEVEL_INDEX is greater than or equal to 6.

8.7.14.7 FIXED_NUM

FIXED_NUM is a 4-bit syntax element that is present if ABS_LEVEL_INDEX is equal to 6. It specifies the number of bits needed to specify the initial level value.

8.7.14.8 FIXED_NUM_EXT

FIXED_NUM_EXT is a 2-bit syntax element that is present if FIXED_NUM is equal to 15. It specifies the number of extension bits needed to specify the initial level value.

8.7.14.9 FIXED_NUM_EXT2

FIXED_NUM_EXT2 is a 3-bit syntax element that is present if FIXED_NUM is equal to 15 and FIXED_NUM_EXT is equal to 3. It specifies the number of additional extension bits needed to specify the initial level value.

8.7.15 Macroblock low-pass

8.7.16 General

This subclause specifies the derivation of the LP coefficients of the blocks in a macroblock. The presence of non-zero entropy coded LP coefficients, i.e. coded block pattern low-pass, in a macroblock is represented by the variable iCBPLP as computed, as specified in Table 53, from the syntax elements CBPLP_YUV1 or CBPLP_YUV2 or CBPLP_CH_BIT.

If INTERNAL_CLR_FMT is not equal to YUV420 or YUV422, the coded block status of the n-th colour component is specified by ((iCBPLP >> n) & 1). If INTERNAL_CLR_FMT is equal to YUV420 or YUV422, the coded block status of the luma component is specified by (iCBPLP & 1). If the coded block status bit of a component is non-zero, there can be up to 15 non-zero LP coefficients associated with that component. These coefficients are parsed by invoking the process DECODE_BLOCK( ) specified by subclause 8.7.18.5, and the inverse scanning order is determined by invoking the process AdaptiveLPScan( ).

If INTERNAL_CLR_FMT is equal to YUV420 or YUV422, the coded block status of the U and V component is jointly specified by ((iCBPLP >> 1) & 1). If the coded block status bit is non-zero, the LP coefficients of U and V are parsed jointly by invoking the process DECODE_BLOCK( ) specified in subclause 8.7.18.5. The U and V coefficients are interleaved, and a fixed inverse scanning order (specified by iRemapArr and iRemapOffset) is used. If INTERNAL_CLR_FMT is equal to YUV420, there can be up to 3 U and 3 V coefficients and the inverse scanning order is U[1], V[1], U[2], V[2], U[3], V[3]. If INTERNAL_CLR_FMT is equal to YUV422, there can be up to 7 U and 7 V coefficients and the inverse scanning order is U[4], V[4], U[1], V[1], U[2], V[2], U[3], V[3], U[5], V[5], U[6], V[6].

The value of the LP coefficients is refined by invoking the process REFINE_LP( ), and this process is invoked irrespective of the value of iCBPLP.

8.7.16.1 MB_LP( )

The MB_LP( ) syntax structure is specified by Table 53.

Table 53 — MB_LP( ) syntax structure

MB_LP( ) {	Descriptor	Reference
/* IsCurrPlaneAlphaFlag is equal to TRUE for parsing alpha image plane, and IsCurrPlaneAlphaFlag is equal to FALSE for parsing primary image plane */		
iBand = 1 /* 0 = DC 1 = LP, 2 = HP */		
iTranspose444[ ] = {0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15}		
iTranspose422[ ] = {0, 2, 1, 3, 4, 6, 5, 7}		
iTranspose420[ ] = {0, 2, 1, 3}		
bInitializeContext = (IsMBLeftEdgeOfTileFlag && IsMBTopEdgeOfTileFlag)		
if (bInitializeContext) {		
InitializeCountCBPLP( )		8.9.2
InitializeLPVLC( )		8.8.3.2
InitializeAdaptiveScanLP( )		8.11.2
InitializeModelMB(ModelLP, iBand)		8.12.1
}		
bResetTotals = ((MBx − LeftMBIndexOfTile[TileIndexx]) % 16) = = 0)		
if (bResetTotals)		
ResetTotalsAdaptiveScanLP( )		8.11.4
iLapMean[ ] = {0, 0}		
if (INTERNAL_CLR_FMT = = YUV422 | | INTERNAL_CLR_FMT = = YUV420)		
iFullPlanes = 2		
else		
iFullPlanes = NumComponents		
if (INTERNAL_CLR_FMT = = YUV420 | | INTERNAL_CLR_FMT = = YUV422 | | INTERNAL_CLR_FMT = = YUV444) {		
iMax = iFullPlanes * 4 − 5 /* Max value of CBPLP */		
if (CountZeroCBPLP <= 0 | | CountMaxCBPLP < 0) {		
CBPLP_YUV1	e(v)	8.7.16.3.1
if (CountMaxCBPLP < CountZeroCBPLP)		
iCBPLP = iMax − CBPLP_YUV1		
else		
iCBPLP = CBPLP_YUV1		
} else {		
CBPLP_YUV2	u(iFullPlanes)	8.7.16.3.2
iCBPLP = CBPLP_YUV2		
}		
UpdateCountCBPLP(iCBPLP, iMax)		8.9.3
} else {		
iCBPLP = 0		
for (n=0; n < NumComponents; n++) {		
CBPLP_CH_BIT	u(1)	8.7.16.3.3
iCBPLP |= (CBPLP_CH_BIT << n)		
}		
}		
for (n = 0; n < NumComponents; n++) {		
if (INTERNAL_CLR_FMT = = YUV420)		
jMax = 3		
else if (INTERNAL_CLR_FMT = = YUV422)		
jMax = 7		
else		
jMax = 15		
for (j = 0; j <= jMax; j++)		
LPInput[k][j] = 0		
}		
for (n = 0; n < iFullPlanes; n++) {		

MB_LP( ) {	Descriptor	Reference
if (n = = 0)		
iIndex = 0		
else		
iIndex = 1		
iNumNonZero = 0		
if ((iCBPLP >> n) & 1) {		
for (i = 0; i < 32; i++)		
iRLCoeffs[i] = 0		
iLocation = 1		
if ((INTERNAL_CLR_FMT = = YUV420) && n)		
iLocation = 10		
if ((INTERNAL_CLR_FMT = = YUV422) && n)		
iLocation = 2		
iNumNonZero = DECODE_BLOCK(iIndex, iRLCoeffs[ ], iBand, iLocation)		8.7.18.5
if ((INTERNAL_CLR_FMT = = YUV420 | | INTERNAL_CLR_FMT = = YUV422) && n) {		
iTemp[14] = 0 /* Initializing the array iTemp to zero. */		
iRemapArr[ ] = {4, 1, 2, 3, 5, 6, 7}		
iRemapOffset = 0		
if (INTERNAL_CLR_FMT = = YUV420)		
iRemapOffset = 1		
if (INTERNAL_CLR_FMT = = YUV422)		
iCountChr = 14		
else		
iCountChr = 6		
i = 0		
for (k = 0; k< iNumNonZero; k++) {		
i += iRLCoeffs[k * 2]		
iTemp[i] = iRLCoeffs[k * 2 + 1]		
i++		
}		
for (k = 0; k < iCountChr; k++) {		
iRemap = iRemapArr[(k >> 1) + iRemapOffset]		
if (INTERNAL_CLR_FMT = = YUV420)		
LPInput[(k & 1) + 1][iTranspose420[iRemap]] = iTemp[k]		
else		
LPInput[(k & 1) + 1][iTranspose422[iRemap]] = iTemp[k]		
}		
} else {		
i = 1		
for (k = 0; k< iNumNonZero; k++) {		
i += iRLCoeffs[k*2]		
AdaptiveLPScan(n, i, iRLCoeffs[k * 2 + 1]) /* Updates LPInput */		8.11.6
i++		
}		
}		
} /* if ((iCBPLP>>n) & 1) */		
iModelBits = ModelLP.MBits[iIndex]		
iLapMean[iIndex] += iNumNonZero		
if (iModelBits)		
if ((INTERNAL_CLR_FMT= =YUV420) && n)		
for (k = 1; k < 4; k++) {		
LPInput[1][iTranspose420[k]]= REFINE_LP(LPInput[1][iTranspose420[k]], iModelBits)		8.7.16.2
LPInput[2][iTranspose420[k]] = REFINE_LP(LPInput[2][iTranspose420[k]], iModelBits)		8.7.16.2
}		
else if ((INTERNAL_CLR_FMT= =YUV422) && n)		
for (k = 1; k < 8; k++) {		
LPInput[1][iTranspose422[k]]=		8.7.16.2

MB_LP( ) {	Descriptor	Reference
REFINE_LP(LPInput[1][iTranspose422[k]], iModelBits)		
LPInput[2][iTranspose422[k]] = REFINE_LP(LPInput[2][iTranspose422[k]], iModelBits)		8.7.16.2
}		
else		
for (k = 1; k < 16; k++)		
LPInput[n][iTranspose444[k]] = REFINE_LP(LPInput[n][iTranspose444[k]], iModelBits)		8.7.16.2
} /* for (n=0 ... */		
UpdateModelMB(iLapMean[ ], ModelLP, iBand)		8.12.2
bResetContext = (MBx = = (LeftMBIndexOfTile[TileIndexx + 1] − 1) | | (MBx − LeftMBIndexOfTile[TileIndexx]) % 16 = = 0)		
if (bResetContext)		
AdaptLP( )		8.8.4.2
}		

8.7.16.2 REFINE_LP( )

Table 54 — REFINE_LP( ) syntax structure

REFINE_LP(iCoeff, iModelBits) {	Descriptor	Reference
COEFF_REF	u(iModelBits)	8.7.16.3.4
if (iCoeff > 0) {		
iCoeff <<= iModelBits		
iCoeff += COEFF_REF		
} else if (iCoeff < 0) {		
iCoeff <<= iModelBits		
iCoeff −= COEFF_REF		
} else {		
iCoeff = COEFF_REF		
if (iCoeff) {		
SIGN_FLAG	u(1)	8.7.14.4
if (SIGN_FLAG)		
iCoeff = −iCoeff		
}		
}		
return iCoeff		
}		

8.7.16.3 MB_LP( ) and REFINE_LP( ) semantics

8.7.16.3.1 CBPLP_YUV1

CBPLP_YUV1 is a syntax element that is present if INTERNAL_CLR_FMT is one of YUV420, YUV422, or YUV444, and also CountZeroCBPLP is less than or equal to 0 or CountMaxCBPLP is less than 0 it jointly specifies the coded block pattern low-pass of the Y, U and V colour components as follows:

— If INTERNAL_CLR_FMT is YUV444, the parsing of CBPLP_YUV1 is specified by Table 55.

— If INTERNAL_CLR_FMT is YUV420 or YUV422, the parsing of CBPLP_YUV1 is specified by Table 56.

NOTE If (CountZeroCBPLP > 0 && CountMaxCBPLP >= 0), the coded block pattern low-pass band is computed as specified in subclause 8.7.16.1.

Table 55 — Code table for CBPLP_YUV1 when INTERNAL_CLR_FMT is equal toYUV444

Code	Value
0	0
100	1
1010	2
1011	3
1100	4
1101	5
1110	6
1111	7

Table 56 — Code table for CBPLP_YUV1 when INTERNAL_CLR_FMT is equal to YUV420 or YUV422

Code	Value
0	0
10	1
110	2
111	3

8.7.16.3.2 CBPLP_YUV2

CBPLP_YUV2 is a syntax element that is present when INTERNAL_CLR_FMT is equal to YUV420, YUV422, or YUV444, and CountZeroCBPLP is greater than 0 and CountMaxCBPLP is greater than or equal to 0. The number of bits required to specify this syntax element is specified by iFullPlanes.

8.7.16.3.3 CBPLP_CH_BIT

CBPLP_CH_BIT is a 1-bit syntax element that is present for each colour component in an image when INTERNAL_CLR_FMT is not one of YUV422, YUV420, or YUV444. It specifies the coded block pattern low-pass of the corresponding colour component. If CBPLP_CH_BIT is equal to 0, all the coefficients in the LP band for this macroblock of the corresponding colour component are set to the value 0. If CBPLP_CH_BIT is equal to 1, the LP band for this macroblock of the corresponding component is non-zero.

8.7.16.3.4 COEFF_REF

COEFF_REF is a syntax element that refines the value of LP coefficients. The number of bits used to parse this syntax element is specified by iModelBits.

8.7.17 Coded block pattern high-pass (CBPHP)

8.7.17.1 General

The CBPHP derivation process is hierarchical, and proceeds as follows:

First step: the syntax element NUM_CBPHP and REFINE_CBPHP( ) process specify the residual CBPHP status of block-groups where each block-group consists of multiple blocks as specified below.

If INTERNAL_CLR_FMT is equal to YUV444, YUV422, or YUV420, there is a NUM_CBPHP syntax element for each macroblock and each block-group consists of 2×2 group of luma blocks, and the co-located chroma blocks. For each 2×2 group of luma block, the co-located chroma blocks consists of a) 2×2 group of U blocks and 2×2 group of V blocks for YUV444, b) 2×1 group of U blocks and 2×1 group of V blocks for YUV422, and c) 1 U block and 1 V block for YUV420. Thus, there are 4 block-groups in each macroblock. NUM_CBPHP takes a value between 0 and 4 and specifies the number of block-groups where CBPHP residual values are non-zero, i.e. block-groups that have CBPHP status that differ from their predicted values. The REFINE_CBPHP( ) process is invoked to determine which of the block groups has non-zero residual CBPHP values.

If INTERNAL_CLR_FMT is equal to YONLY, YUVK, or NCOMPONENT, there is a NUM_CBPHP syntax for each colour component in that macroblock, and each block-group consists of 2×2 groups of blocks in that colour component.

Thus, there are 4 block-groups for each colour component. NUM_CBPHP takes a value between 0 and 4 and specifies the number of block groups (in that colour component) where CBPHP residual values are non-zero, and the REFINE_CBPHP( ) process is invoked to determine which of the block groups has non-zero residual CBPHP values.

Second step: If the residual CBPHP status of a block-group is equal to 0, the residual CBPHP of all the blocks in the group is inferred to be equal to 0. If the residual CBPHP status of a given block-group is non-zero, the NUM_BLKCBPHP and subsequent syntax elements are used to indicate the residual CBPHP of specific blocks in that block-group as summarized below:

If INTERNAL_CLR_FMT is equal to YUV444, YUV422, or YUV420, there is a NUM_BLKCBPHP syntax element for each block-group with non-zero residual CBPHP status. NUM_BLKCBPHP takes a value between 0 and 8. If NUM_BLKCBPHP plus 1 is less than 6, it indicates that the residual CBPHP of all the chroma blocks in the block group are equal to 0. The luma blocks that have non-zero residual CBPHP are indicated by the value of NUM_BLKCBPHP and CODE_INC. If NUM_BLKCBPHP plus 1 is greater than or equal to 6, the residual CBPHP of at least some chroma blocks in this block-group are non-zero, and the syntax elements CHR_CBPHP is parsed to specify if a) the U, or b) V, or c) both U and V, colour components have blocks with non-zero residual CBPHP. In this case, the luma blocks that have non-zero residual CBPHP are indicated by the value of NUM_BLKCBPHP, VAL_INC and CODE_INC.

If INTERNAL_CLR_FMT is equal to YUV444 or YUV422, the chroma component with non-zero residual CBPHP has multiple blocks. If INTERNAL_CLR_FMT is equal to YUV444, the syntax element NUM_CH_BLK and the process REFINE_CBPHP( ) specify the chroma blocks that have non-zero residual CBPHP. If INTERNAL_CLR_FMT is equal to YUV422, the syntax element CBPHP_CH_BLK specifies the chroma blocks that have non-zero residual CBPHP.

If INTERNAL_CLR_FMT is equal to YONLY, YUVK, or NCOMPONENT, there is a NUM_BLKCBPHP syntax element for each block-group (in each colour component) where residual CBPHP status is non-zero. NUM_BLKCBPHP takes a value between 0 and 4. The blocks in this block-group that have non-zero residual CBPHP are indicated by the value of NUM_BLKCBPHP and CODE_INC.

Third step: On the completion of the second step for all block groups in a macroblock, the residual CBPHP values for all the blocks in colour component i are stored in the corresponding iDiffCBPHP[i] variable. These values are stored in a hierarchical raster scan order, where each consecutive nibble of 4 bits corresponds to one 2×2 block group. Within each nibble, the blocks of a block-group are in raster scan order, and 2×2 block-groups in a macroblock are also in raster scan order. If INTERNAL_CLR_FMT is equal to YUV422 or YUV420), the hierarchical scan order for the chroma components is identical to the normal scan order.

The PredCBPHP( ) process is invoked to compute the actual CBPHP values from the residual CBPHP values. The value of ((MBCBPHP[MBx][MBy][i] >> j) & 1) specifies the coded block status of the j-th block (in the same hierarchical raster scan order as iDiffCBPHP) associated with the i-th colour component in the macroblock indexed by MBx and MBy.

8.7.17.2 MB_CBPHP( )

The MB_CBPHP( ) syntax structure is specified by Table 57.

Table 57 — MB_CBPHP( ) syntax structure

MB_CBPHP( ) {	Descriptor	Reference
/* IsCurrPlaneAlphaFlag is equal to TRUE for parsing alpha image plane, and IsCurrPlaneAlphaFlag is equal to FALSE for parsing primary image plane */		
/* **sAdaptVLC** is local instance of AdaptiveVLC data structure */		
iFLC[ ] = {0, 2, 1, 2, 2, 0}		
iOff[ ] = {0, 4, 2, 8, 12, 1}		
iOut[ ] ={0, 15, 3, 12, 1, 2, 4, 8, 5, 6, 9, 10, 7, 11, 13, 14}		
iDiffCBPHP[NumComponents] =0 /* Initializing the array to zero */		
bInitializeContext = (IsMBLeftEdgeOfTileFlag && IsMBTopEdgeOfTileFlag)		
if (bInitializeContext)		
InitializeCBPHPVLC( )		8.8.3.4
if (INTERNAL_CLR_FMT = = YUVK | | INTERNAL_CLR_FMT = = NCOMPONENT)		
iComponent = NumComponents		
else		
iComponent = 1		
for (i = 0; i < iComponent; i++) {		
sAdaptVLC = **DecNumCBPHP** /* reference AdaptiveVLC struct for NUM_CBPHP */		
NUM_CBPHP /* using **sAdaptVLC** */	ae(v)	8.7.17.4.1
sAdaptVLC.DiscrimVal1 += NumCBPHPDelta[**sAdaptVLC**.DeltaTableIndex][NUM_CBPHP] /* **sAdaptVLC**.DeltaTableIndex is equal to 0 */		Table 89
iCBPHP = REFINE_CBPHP(NUM_CBPHP)		8.7.17.3
for (iBlock = 0; iBlock < 4; iBlock++)		
if (iCBPHP & (1 << iBlock)) {		
sAdaptVLC = **DecNumBlkCBPHP** /* reference AdaptiveVLC struct for NUM_BLKCBPHP */		
NUM_BLKCBPHP /* using **sAdaptVLC** */	ae(v)	8.7.17.4.2
sAdaptVLC.DiscrimVal1 += NumBlkCBPHPDelta[**sAdaptVLC**.DeltaTableIndex][NUM_BLKCBPHP]		Table 90, Table 91
iVal = NUM_BLKCBPHP+1		
iBlkCBPHP = 0		
if (iVal >= 6) { /* Is chroma */		
CHR_CBPHP	e(v)	8.7.17.4.3
iBlkCBPHP = 0x10 * (CHR_CBPHP + 1)		
if (iVal >= 9) {		
VAL_INC	e(v)	8.7.17.4.4
iVal += VAL_INC		
}		
iVal −= 6		
}		
iCode = iOff[iVal]		
if (iFLC[iVal]) {		
CODE_INC	u(iFLC[iVal])	8.7.17.4.5
iCode += CODE_INC		
}		
iBlkCBPHP += iOut[iCode]		
if (INTERNAL_CLR_FMT = = YUV444) {		
iDiffCBPHP[0] |= ((iBlkCBPHP & 0x0F) << (iBlock * 4))		
for (k = 0; k < 2; k++)		
if ((iBlkCBPHP >> (k + 4)) & 0x01) {		
NUM_CH_BLK	e(v)	8.7.17.4.6
iCBPHPChr = REFINE_CBPHP(NUM_CH_BLK + 1)		8.7.17.3

MB_CBPHP( ) {	Descriptor	Reference
iDiffCBPHP[k + 1] |= (iCBPHPChr << (iBlock * 4))		
}		
/* INTERNAL_CLR_FMT = = YUV444 */		
} else if (INTERNAL_CLR_FMT = = YUV422) {		
iDiffCBPHP[0] |= ((iBlkCBPHP & 0x0F) << (iBlock * 4))		
for (k = 0; k < 2; k++)		
if ((iBlkCBPHP >> (k + 4)) & 0x01) {		
iShift[4] = {0, 1, 4, 5}		
CBPHP_CH_BLK	e(v)	8.7.17.4.7
iCBPHPChr = iShift[CBPHP_CH_BLK + 1]		
iDiffCBPHP[k + 1] |= (iCBPHPChr << iShift[iBlock])		
}		
} else if (INTERNAL_CLR_FMT = = YUV420) {		
iDiffCBPHP[0] |= ((iBlkCBPHP & 0x0F) << (iBlock * 4))		
iDiffCBPHP[1] |= (((iBlkCBPHP >> 4) & 0x01) << iBlock)		
iDiffCBPHP[2] |= (((iBlkCBPHP >> 5) & 0x01) << iBlock)		
} else /* Default */		
iDiffCBPHP[i] |= ((iBlkCBPHP) << (iBlock * 4))		
} /* if (iCBPHP...) */		
} /* i */		
PredCBPHP(iDiffCBPHP)		8.7.17.5.1
}		

8.7.17.3 REFINE_CBPHP( )

Table 58 — REFINE_CBPHP( ) syntax structure

REFINE_CBPHP(iNum) {	Descriptor	Reference
if (iNum = = 2) {		
REF_CBPHP1	e(v)	8.7.17.4.8
iRef = REF_CBPHP1		
} else if (iNum = = 1) {		
REF_CBPHP	u(2)	8.7.17.4.9
iRef = (1<<REF_CBPHP)		
} else if (iNum = = 3) {		
REF_CBPHP	u(2)	8.7.17.4.9
iRef = (0x0F ^ (1<<REF_CBPHP))		
} else if (iNum = = 4)		
iRef = 0x0F		
else		
iRef = 0		
return iRef		
}		

8.7.17.4 MB_CBPHP( ) and REFINE_CBPHP( ) semantics

8.7.17.4.1 NUM_CBPHP

NUM_CBPHP is a variable syntax element that specifies the number of block-groups where CBPHP residual values CBPHP status differs from their predicted values as specified in subclause 8.7.17.1. NUM_CBPHP is parsed using one of two VLC tables specified in Table 59. The adaptive VLC structure used to parse NUM_CBPHP is initialized to the VLC table corresponding to Code 0 as specified in subclause 8.8.3.4, and the structure is updated as specified in subclause 8.8.4.

Table 59 — Code table for NUM_CBPHP( )

Code 0	Code 1	Value
1	1	0
01	000	1
001	001	2
0000	010	3
0001	011	4

8.7.17.4.2 NUM_BLKCBPHP

NUM_BLKCBPHP is a variable-length syntax element that specifies the CBPHP status of each block-group where residual CBPHP status is non-zero as specified in subclause 8.7.17.1. If INTERNAL_CLR_FMT is equal to YUVK, NCOMPONENT, or YONLY, NUM_BLKCBPHP is coded using one of the two VLC tables specified in Table 60. Otherwise, NUM_BLKCBPHP is parsed using one of the two VLC tables specified in Table 61. The adaptive VLC structure used to parse NUM_BLKCBPHP is initialized to the VLC table corresponding to Code 0 as specified in subclause 8.8.3.4, and the structure is updated as specified in subclause 8.8.4.

Table 60 — Code table for NUM_BLKCBPHP if INTERNAL_CLR_FMT is one of {YUVK, NCOMPONENT, YONLY}

Code 0	Code 1	Value
1	1	0
01	000	1
001	001	2
0000	010	3
0001	011	4

Table 61 — Code table for NUM_BLKCBPHP if INTERNAL_CLR_FMT is not one of {YUVK, NCOMPONENT, YONLY}

Code 0	Code 1	Value
010	1	0
00000	001	1
0010	010	2
00001	0001	3
00010	000001	4
1	011	5
011	00001	6
00011	0000000	7
0011	0000001	8

8.7.17.4.3 CHR_CBPHP

CHR_CBPHP is a syntax element that specifies the chroma components have non-zero CBPHP in a block-group as specified in subclause 8.7.17.1. The VLC used to parse CHR_CBPHP is specified in Table 62.

Table 62 — Code table for CHR_CBPHP, VAL_INC, and CBPHP_CH_BLK

Code	Value
1	0
01	1
00	2

NOTE Non-zero residual CBPHP in a) U component blocks are indicated by CHR_CBPHP = 0, b) V component blocks are indicated by CHR_CBPHP = 1, and c) both U and V component blocks are indicated by CHR_CBPHP = 2.

8.7.17.4.4 VAL_INC

VAL_INC is a syntax element that refines the CBPHP of a block-group as specified in subclause 8.7.17.1 when NUM_BLKCBPHP plus 1 is greater than or equal to 9. The VLC that specifies the parsing of VAL_INC is specified in Table 62.

8.7.17.4.5 CODE_INC

CODE_INC is a syntax element that specifies the location of coded blocks in a block-group as specified in subclause 8.7.17.1. The size of this syntax element is specified by iFLC[iVal], where iFLC[ ] and iVal are specified in subclause 8.7.17.2.

8.7.17.4.6 NUM_CH_BLK

When INTERNAL_CLR_FMT is equal to YUV444, NUM_CH_BLK is a syntax element that specifies the number of coded chroma blocks in a 2×2 block-group as specified in subclause 8.7.17.1. The VLC that specifies the parsing of NUM_CH_BLK is specified in Table 63.

Table 63 — Code table for NUM_CH_BLK

Code	Value
1	0
01	1
000	2
001	3

8.7.17.4.7 CBPHP_CH_BLK

When INTERNAL_CLR_FMT is equal to YUV422, CBPHP_CH_BLK is a syntax element that refines the chroma CBPHP for a block-group as specified in subclause 8.7.17.1. The VLC that specifies the parsing of CBPHP_CH_BLK is specified in Table 62.

8.7.17.4.8 REF_CBPHP1

REF_CBPHP1 is a variable size syntax element that refines the CBPHP of a block-group as specified in subclause 8.7.17.1. The VLC that specifies the parsing of REF_CBPHP1 is specified in Table 64.

Table 64 — Code table for REF_CBPHP1

Code	Value
00	3
01	5
100	6
101	9
110	10
111	12

8.7.17.4.9 REF_CBPHP

REF_CBPHP is a 2-bit syntax element that refines the CBPHP of a block-group as specified in subclause 8.7.17.1.

8.7.17.5 CBPHP prediction

The CBPHP of neighbouring blocks is used to predict the CBPHP of current block as specified by subclause 8.7.17.5.1. The prediction of CBPHP in each component is performed independently. The prediction of CBPHP for the U and V components in the YUV422 case is specified by subclause 8.7.17.5.3.The prediction of CBPHP for the U and V components in the YUV420 is specified by subclause 8.7.17.5.4. In all other cases, the prediction of CBPHP is specified by subclause 8.7.17.5.2. After the CBPHP of the current block is reconstructed, the CBPHP prediction model is updated as specified by subclause 8.10.2.

8.7.17.5.1 PredCBPHP( )

Table 65 — Pseudocode for function PredCBPHP( )

PredCBPHP(iDiffCBPHP[ ]) {	**Reference**
bInitializeContext = (IsMBLeftEdgeOfTileFlag && IsMBTopEdgeOfTileFlag)	
if (bInitializeContext)	
InitializeCBPHPModel( )	8.10.1
if (INTERNAL_CLR_FMT = = YUV420 | | INTERNAL_CLR_FMT = = YUV422)	
iComponent = 1	
else	
iComponent = NumComponents	
for (i = 0; i < iComponent; i++)	
MBCBPHP[MBx][MBy][i] = PredCBPHP444(i, iDiffCBPHP)	8.7.17.5.2
if (INTERNAL_CLR_FMT = = YUV422) {	
MBCBPHP[MBx][MBy][1] = PredCBPHP422(1, iDiffCBPHP)	8.7.17.5.3
MBCBPHP[MBx][MBy][2] = PredCBPHP422(2, iDiffCBPHP)	8.7.17.5.3
} else if (INTERNAL_CLR_FMT = = YUV420) {	
MBCBPHP[MBx][MBy][1] = PredCBPHP420(1, iDiffCBPHP)	8.7.17.5.4
MBCBPHP[MBx][MBy][2] = PredCBPHP420(2, iDiffCBPHP)	8.7.17.5.4
}	
}	

8.7.17.5.2 PredCBPHP444( )

Table 66 — Pseudocode for function PredCBPHP444( )

PredCBPHP444(i, iDiffCBPHP) {	**Reference**
c1= 0	
if (i > 0)	
c1 = 1	
iCBPHP = iDiffCBPHP[i]	
if (**CBPHPModelHP**.CBPHPState[c1] = = 0) {	
if (IsMBLeftEdgeofTileFlag)	
if (IsMBTopEdgeofTileFlag)	
iCBPHP ^= 1	
else	
iCBPHP ^= ((MBCBPHP[MBx][MBy−1][i] >> 10) & 1)	
else	
iCBPHP ^= ((MBCBPHP[MBx−1][MBy][i] >> 5) & 1)	
iCBPHP ^= (0x02 & (iCBPHP << 1))	
iCBPHP ^= (0x10 & (iCBPHP << 3))	
iCBPHP ^= (0x20 & (iCBPHP << 1))	
iCBPHP ^= ((iCBPHP & 0x33) << 2)	
iCBPHP ^= ((iCBPHP & 0x00CC) << 6)	
iCBPHP ^= ((iCBPHP & 0x3300) << 2)	
} else if (**CBPHPModelHP**.CBPHPState[c1] = = 2)	
iCBPHP ^= 0x0000FFFF	
iNOrig = Numones(iCBPHP)	
UpdateCBPHPModel(c1, iNOrig)	8.10.2
return iCBPHP	
}	

8.7.17.5.3 PredCBPHP422( )

Table 67 — Pseudocode for function PredCBPHP422( )

PredCBPHP422(i, iDiffCBPHP[ ]) {	**Reference**
iCBPHP = iDiffCBPHP[i]	
if (**CBPHPModelHP**.CBPHPState[1] = = 0) {	
if (IsMBLeftEdgeofTileFlag)	
if (IsMBTopEdgeofTileFlag)	
iCBPHP ^= 1	
else	
iCBPHP ^= ((MBCBPHP[MBx][MBy−1][i] >> 6) & 1)	
else	
iCBPHP ^= ((MBCBPHP[MBx−1][MBy][i] >> 1) & 1)	
iCBPHP ^= ((iCBPHP & 0x01) << 1)	
iCBPHP ^= ((iCBPHP & 0x03) << 2)	
iCBPHP ^= ((iCBPHP & 0x0C) << 2)	
iCBPHP ^= ((iCBPHP & 0x30) << 2)	
} else if (**CBPHPModelHP**.CBPHPState[1] = = 2)	
iCBPHP ^= 0x00FF	
iNOrig = Numones(iCBPHP) * 2	
UpdateCBPHPModel(1, iNOrig)	8.10.2
return iCBPHP	
}	

8.7.17.5.4 PredCBPHP420( )

Table 68 — Pseudocode for function PredCBPHP420( )

PredCBPHP420(i, iDiffCBPHP[ ]) {	**Reference**
iCBPHP = iDiffCBPHP[i]	
if (**CBPHPModelHP**.CBPHPState[1] = = 0) {	
if (IsMBLeftEdgeofTileFlag)	
if (IsMBTopEdgeofTileFlag)	
iCBPHP ^= 1	
else	
iCBPHP ^= ((MBCBPHP[MBx][MBy−1][i] >> 2) & 1)	
else	
iCBPHP ^= ((MBCBPHP[MBx−1][MBy][i] >> 1) & 1)	
iCBPHP ^= (0x02 & (iCBPHP << 1))	
iCBPHP ^= ((iCBPHP & 0x3) << 2)	
} else if (**CBPHPModelHP**.CBPHPState[1] = = 2)	
iCBPHP ^= 0x0F	
iNOrig = Numones(iCBPHP) * 4	
UpdateCBPHPModel(1, iNOrig)	8.10.2
return iCBPHP	
}	

8.7.18 Macroblock high-pass

8.7.18.1 General

The presence of non-zero HP coefficients in j-th block in colour component is specified by ((MBCBPHP[MBx][MBy][i] >> j) & 1), where the blocks are scanned in the hierarchical raster scan order specified in subclause 8.7.17.1. If there are non-zero coefficients in a block, these coefficients are parsed by invoking the process DECODE_BLOCK_ADAPTIVE( ), specified in subclause 8.7.18.4, which, in turn, invokes the process DECODE_BLOCK( ), specified in subclause 8.7.18.5, for parsing the coefficients, and invokes the process AdaptiveHPScan( ) to determine the inverse scanning order of the coefficients.

8.7.18.2 MB_HP( )

Table 69 — MB_HP( ) syntax structure

MB_HP( ) {	Descriptor	Reference
/* IsCurrPlaneAlphaFlag is equal to TRUE for parsing alpha image plane, and IsCurrPlaneAlphaFlag is equal to FALSE for parsing primary image plane */		
iBand = 2		
iHierScanOrder[ ] = {0, 1, 4, 5, 2, 3, 6, 7, 8, 9, 12, 13, 10, 11, 14, 15}		
bInitializeContext = (IsMBLeftEdgeOfTileFlag && IsMBTopEdgeOfTileFlag)		
if (bInitializeContext) {		
InitializeHPVLC( )		8.8.3.3
InitializeAdaptiveScanHP( )		8.11.3
InitializeModelMB(ModelHP, iBand)		8.12.1
}		
bResetTotals = (((MBx – LeftMBIndexOfTile[TileIndexx]) % 16) = = 0)		
if (bResetTotals)		
ResetTotalsAdaptiveScanHP( )		8.11.5
iLapMean[ ] = {0, 0}		
for (i = 0; i < NumComponents; i++) {		
bChroma = (i > 0)		
iNBlocks = 4		
if (bChroma && INTERNAL_CLR_FMT = = YUV420)		
iNBlocks = 1		
else if (bChroma && INTERNAL_CLR_FMT = = YUV422)		
iNBlocks = 2		
iCBPHP = MBCBPHP[MBx][MBy][i]		
for (iBlock = 0; iBlock < iNBlocks * 4; iBlock++) {		
iBlockMap = iBlock		
if (iNBlocks = = 4)		
iBlockMap = iHierScanOrder[iBlock]		
for (k = 0; k < 16; k++)		
HPInputVLC[i][iBlock][k] = 0		
iNumNonZero = DECODE_BLOCK_ADAPTIVE(iCBPHP & 1, bChroma, i, iBlockMap)		8.7.18.4
iLapMean[bChroma] += iNumNonZero		
iCBPHP >>= 1		
}		
}		
ModelBitsMBHP[MBx][MBy][0] = ModelHP.MBits[0]		
ModelBitsMBHP[MBx][MBy][1] = ModelHP.MBits[1]		
UpdateModelMB(iLapMean[ ], ModelHP, iBand)		8.12.2
bResetContext = (MBx = = (LeftMBIndexOfTile[TileIndexx + 1] – 1) || (MBx – LeftMBIndexOfTile[TileIndexx]) % 16 = = 0)		
if (bResetContext)		
AdaptHP( )		8.8.4.3
}		

8.7.18.3 MB_HP_FLEX( )

Table 70 — MB_HP_FLEX( ) syntax structure

MB_HP_FLEX( ) {	**Descriptor**	**Reference**
/* IsCurrPlaneAlphaFlag is equal to TRUE for parsing alpha image plane, and IsCurrPlaneAlphaFlag is equal to FALSE for parsing primary image plane */		
iBand = 2		
iHierScanOrder[ ] = {0, 1, 4, 5, 2, 3, 6, 7, 8, 9, 12, 13, 10, 11, 14, 15}		
bInitializeContext = (IsMBLeftEdgeOfTileFlag && IsMBTopEdgeOfTileFlag)		
if (bInitializeContext) {		
InitializeHPVLC( )		8.8.3.3
InitializeAdaptiveScanHP( )		8.11.3
InitializeModelMB(ModelHP, iBand)		8.12.1
}		
bResetTotals = (((MBx – LeftMBIndexOfTile[TileIndexx]) % 16) = = 0)		
if (bResetTotals)		
ResetTotalsAdaptiveScanHP( )		8.11.5
iLapMean[ ] = {0, 0}		
for (i = 0; i < NumComponents; i++) {		
iIndex = 0		
bChroma = i > 0		
if (i> 0)		
iIndex = 1		
iModelBits = ModelHP.MBits[iIndex]		
iNBlocks = 4		
if (bChroma && INTERNAL_CLR_FMT = = YUV420)		
iNBlocks = 1		
else if (bChroma && INTERNAL_CLR_FMT = = YUV422)		
iNBlocks = 2		
iCBPHP = MBCBPHP[MBx][MBy][i]		
for (iBlock = 0; iBlock < iNBlocks*4; iBlock++) {		
iBlockMap = iBlock		
if (iNBlocks = = 4)		
iBlockMap = iHierScanOrder[iBlock]		
for (k = 0; k < 16; k++)		
HPInputVLC[i][iBlock][k] = 0		
iNumNonZero = DECODE_BLOCK_ADAPTIVE(iCBPHP & 1, bChroma, i, iBlockMap)		8.7.18.4
if (BANDS_PRESENT != NOFLEXBITS)		
BLOCK_FLEXBITS(i, iBlockMap, iModelBits, TRIM_FLEXBITS)		8.7.19.2
iLapMean[bChroma] += iNumNonZero		
iCBPHP >>= 1		
}		
}		
ModelBitsMBHP[MBx][MBy][0] = ModelHP.MBits[0]		
ModelBitsMBHP[MBx][MBy][1] = ModelHP.MBits[1]		
UpdateModelMB(iLapMean[ ], ModelHP, iBand)		8.12.2
bResetContext = (MBx = = (LeftMBIndexOfTile[TileIndexx + 1] – 1) | | (MBx – LeftMBIndexOfTile[TileIndexx]) % 16 = = 0)		
if (bResetContext)		
AdaptHP( )		8.8.4.3
}		

8.7.18.4 DECODE_BLOCK_ADAPTIVE( )

Table 71 — DECODE_BLOCK_ADAPTIVE( ) syntax structure

DECODE_BLOCK_ADAPTIVE(bNoSkip, bChroma, iComponent, iBlock) {	Descriptor	Reference
iBand = 2 /* 0 = DC 1 = LP, 2 = HP */		
for (i = 0; i < 32; i++)		
iLocalCoeff[i] = 0		
iLocation = 1		
iNumNonZero = 0		
if (bNoSkip) {		
iNumNonZero = DECODE_BLOCK(bChroma, iLocalCoeff[ ], iBand, iLocation)		8.7.18.5
k = iLocation		
for (kk = 0; kk < iNumNonZero; kk++) {		
k += iLocalCoeff[kk * 2]		
AdaptiveHPScan(iComponent, iBlock, k, iLocalCoeff[kk * 2 + 1])		8.11.7
k++		
}		
}		
return iNumNonZero		
}		

8.7.18.5 DECODE_BLOCK( )

Table 72 — DECODE_BLOCK( ) syntax structure

DECODE_BLOCK(bChroma, iCoeff[ ], iBand, iLocation) {	Descriptor	Reference
iNumNZ = 1		
iFirstIndex = DECODE_FIRST_INDEX(bChroma, iBand)		8.7.18.8
SIGN_FLAG	u(1)	8.7.14.4
iSR = (iFirstIndex & 1)		
iSRn = (iFirstIndex >> 2)		
iContext = (iSR & iSRn)		
if (iFirstIndex & 2)		
iCoeff[1] = DECODE_ABS_LEVEL(iBand, bChroma, iContext)		8.7.13
else		
iCoeff[1] = 1		
if (SIGN_FLAG)		
iCoeff[1] = −iCoeff[1]		
iCoeff[0] = 0		
if (iSR = = 0)		
iCoeff[0] = DECODE_RUN(15 − iLocation)		8.7.18.6
iLocation += iCoeff[0] + 1		
while (iSRn != 0) {		
iSR = (iSRn & 1)		
iCoeff[iNumNZ * 2] = 0		
if (iSR = = 0)		
iCoeff[iNumNZ * 2] = DECODE_RUN(15 − iLocation)		8.7.18.6
iLocation += (iCoeff[iNumNZ * 2] + 1)		
iIndex = DECODE_INDEX(iLocation, bChroma, iBand, iContext)		8.7.18.7
iSRn = (iIndex >> 1)		
iContext &= iSRn		
SIGN_FLAG	u(1)	8.7.14.4
if (iIndex & 1)		
iCoeff[(iNumNZ * 2) + 1] = DECODE_ABS_LEVEL(iBand, bChroma, iContext)		8.7.13
else		
iCoeff[(iNumNZ * 2) + 1] = 1		
if (SIGN_FLAG)		
iCoeff[(iNumNZ * 2) + 1] = −iCoeff[(iNumNZ * 2) + 1]		

DECODE_BLOCK(bChroma, iCoeff[ ], iBand, iLocation) {	Descriptor	Reference
iNumNZ++		
}		
return iNumNZ		
}		

8.7.18.6 DECODE_RUN( )

Table 73 — DECODE_RUN( ) syntax structure

DECODE_RUN(iMaxRun) {	Descriptor	Reference
iRemap[ ] = {1, 2, 3, 5, 7, 1, 2, 3, 5, 7, 1, 2, 3, 4, 5}		
iRunBin[ ] = {−1, −1, −1, −1, 2, 2, 2, 1, 1, 1, 1, 0, 0, 0, 0}		
iRunFixedLength[ ] = {0, 0, 1, 1, 3, 0, 0, 1, 1, 2, 0, 0, 0, 0, 1}		
if (iMaxRun < 5) {		
iRun = 1;		
if (iMaxRun != 1) {		
RUN_VALUE	e(v)	8.7.18.9.1
iRun = RUN_VALUE		
}		
} else {		
RUN_INDEX	e(v)	8.7.18.9.2
iIndex = RUN_INDEX + 5* iRunBin[iMaxRun]		
iFixed = iRunFixedLength[iIndex]		
iRun = iRemap[iIndex]		
if (iFixed) {		
RUN_REF	u(iFixed)	8.7.18.9.3
iRun += RUN_REF		
}		
}		
return iRun		
}		

8.7.18.7 DECODE_INDEX( )

Table 74 — DECODE_INDEX( ) syntax structure

DECODE_INDEX(iLocation, bChroma, iBand, iContext) {	**Descriptor**	**Reference**
/* **sAdaptVLC** is local instance of AdaptiveVLC data structure */		
if (iBand = = 1) /* LP */		
if (bChroma)		
if (iContext)		
sAdaptVLC = DecIndLPChr1		
else		
sAdaptVLC = DecIndLPChr0		
else /* Luma */		
if (iContext)		
sAdaptVLC = DecIndLPLum1		
else		
sAdaptVLC = DecIndLPLum0		
else if (iBand = = 2) /* HP */		
if (bChroma)		
if (iContext)		
sAdaptVLC = DecIndHPChr1		
else		
sAdaptVLC = DecIndHPChr0		
else /* Luma */		
if (iContext)		
sAdaptVLC = DecIndHPLum1		
else		
sAdaptVLC = DecIndHPLum0		
if (iLocation < 15) {		
INDEX_A /* variable-length parse with **sAdaptVLC** */	ae(v)	8.7.18.9.4
/* update the discriminants for **sAdaptVLC** */		
sAdaptVLC.DiscrimVal1 += Index1Delta[**sAdaptVLC**.DeltaTableIndex][INDEX_A]		Table 88
sAdaptVLC.DiscrimVal2 += Index1Delta[**sAdaptVLC**.Delta2TableIndex][INDEX_A]		Table 88
iIndex = INDEX_A		
} else if (iLocation = = 15) {		
INDEX_B	e(v)	8.7.18.9.5
iIndex = INDEX_B		
} else {		
INDEX_C_FLAG	u(1)	8.7.18.9.6
iIndex = INDEX_C_FLAG		
}		
return iIndex		
}		

8.7.18.8 DECODE_FIRST_INDEX( )

Table 75 — DECODE_FIRST_INDEX( ) syntax structure

DECODE_FIRST_INDEX(bChroma, iBand) {	Descriptor	Reference
/* **sAdaptVLC** is local instance of AdaptiveVLC data structure */		
if (iBand = = 1) /* LP */		
if (bChroma)		
sAdaptVLC = **DecFirstIndLPChr**		
else /* Luma */		
sAdaptVLC = **DecFirstIndLPLum**		
else if (iBand = = 2) /* HP */		
if (bChroma)		
sAdaptVLC = **DecFirstIndHPChr**		
else /* Luma */		
sAdaptVLC = **DecFirstIndHPLum**		
FIRST_INDEX /* Decode with **sAdaptVLC** */	ae(v)	8.7.18.9.7
/* update Discriminants for **sAdaptVLC** */		
sAdaptVLC.DiscrimVal1 += FirstIndexDelta[**sAdaptVLC**.DeltaTableIndex][FIRST_INDEX]		Table 87
sAdaptVLC.DiscrimVal2 += FirstIndexDelta[**sAdaptVLC**.Delta2TableIndex][FIRST_INDEX]		Table 87
return FIRST_INDEX		
}		

8.7.18.9 Block-level semantics

8.7.18.9.1 RUN_VALUE

RUN_VALUE is a variable-length syntax element that is present when iMaxRun is greater than 1 and iMaxRun is less than 5. It specifies the value of run. If iMaxRun is equal to 2, the parsing of RUN_VALUE is specified by Table 76. Otherwise, if iMaxRun is equal to 3, the parsing of RUN_VALUE is specified by Table 77. Otherwise, (if iMaxRun is equal to 4), the parsing of RUN_VALUE is specified by Table 78.

Table 76 — Code table of RUN_VALUE if iMaxRun = = 2

Code	Value
1	1
0	2

Table 77 — Code table of RUN_VALUE if iMaxRun = = 3

Code	Value
1	1
01	2
00	3

Table 78 — Code table of RUN_VALUE if iMaxRun = = 4

Code	Value
1	1
01	2
001	3
000	4

8.7.18.9.2 RUN_INDEX

RUN_INDEX is a variable-length syntax element that is present when iMaxRun is greater than or equal to 5. It specifies the value of iRun in subclause 8.7.18.6. The parsing of RUN_INDEX is specified by Table 79.

Table 79 — Code table of RUN_INDEX

Code	Value
1	0
01	1
001	2
0000	3
0001	4

8.7.18.9.3 RUN_REF

RUN_REF is a fixed-length syntax element that specifies the value of iRun in subclause 8.7.18.6. The presence and size of the RUN_REF syntax element is indicated by iFixed, as specified in subclause 8.7.18.6.

8.7.18.9.4 INDEX_A

INDEX_A is a variable-length syntax element that is present when iLocation is less than 15. It has a value in the range of 0 to 5, inclusive. The coding of this symbol uses one of four tables. The choice of table is adaptively determined as specified in subclause 8.8.4. The VLC tables are specified in Table 80.

Table 80 — Code table for INDEX_A

Code 0	Code 1	Code 2	Code 3	Value
1	01	0000	0 0000	0
0 0000	0000	0001	0 0001	1
001	10	01	01	2
0 0001	0001	10	1	3
01	11	11	0001	4
0001	001	001	001	5

INDEX_A jointly codes the following two events:

— The binary event of whether the magnitude of the next non-zero coefficient is equal to 1 or greater than 1 as follows:

  — If (INDEX_A & 1) is equal to 0, this magnitude is equal to 1.

  — Otherwise, this magnitude is greater than 1.

— The ternary of event whether this coefficient is the last coefficient in the block, and if there are more non-zero coefficients, whether the run before the next non-zero coefficient is zero or non-zero, as follows:

  — If (INDEX_A >> 1) is equal to 0, this coefficient is the last coefficient in the block.

  — Otherwise, if (INDEX_A >> 1) is equal to 1, the run before the next non-zero coefficient is zero.

  — Otherwise (i.e. when (INDEX_A >> 1) is equal to 2), the run before the next non-zero coefficient is non-zero.

NOTE Thus INDEX_A has an alphabet size of 2*3= 6.

8.7.18.9.5 INDEX_B

INDEX_B is a variable-length syntax element that is present when iLocation is equal to 15. It has a value in the range of 0 to 3, inclusive. The VLC table is specified in Table 81.

Table 81 — Code table for INDEX_B

Code	Value
0	0
10	2
110	1
111	3

INDEX_B jointly codes the following two events:

— The binary event of whether the magnitude of the next non-zero coefficient is equal to 1 or greater than 1 as follows:

  — If (INDEX_B & 1) is equal to 0, this magnitude is equal to 1.

  — Otherwise, this magnitude is greater than 1.

— The binary event of whether this coefficient is the last coefficient in the block or if there are more non-zero coefficients, as follows:

  — If (INDEX_B >> 1) is equal to 0, this coefficient is the last coefficient in the block.

  — Otherwise, the run before the next non-zero coefficient is zero.

NOTE Thus INDEX_B has an alphabet size of 2*2= 4.

8.7.18.9.6 INDEX_C_FLAG

INDEX_C_FLAG is a 1-bit syntax element that is present when iLocation is equal to 16. It specifies the presence of subsequent run/level symbols.

8.7.18.9.7 FIRST_INDEX

FIRST_INDEX is a variable-length syntax element that has a value in the range of 0 to 11, inclusive. The coding of this syntax element uses one of five tables. The choice of table is adaptively determined as specified in subclause 8.8.4. The VLC tables are specified in Table 82.

Table 82 — Code table for FIRST_INDEX

Code 0	Code 1	Code 2	Code 3	Code 4	Value
0000 1	0010	11	001	010	0
0000 01	0001 0	001	11	1	1
0000 000	0000 00	0000 000	0000 000	0000 001	2
0000 001	0000 01	0000 001	0000 1	0001	3
0010 0	0011	0000 1	0001 0	0000 010	4
010	010	010	010	011	5
0010 1	0001 1	0000 010	0000 001	0000 0000	6
1	11	011	011	0010	7
0011 0	011	100	0001 1	0000 011	8
0001	100	101	100	0011	9
0011 1	0000 1	0000 011	0000 01	0000 0001	10
011	101	0001	101	0000 1	11

FIRST_INDEX jointly codes the following three events:

— The binary event of whether the run before the first non-zero coefficient is non-zero or zero as follows:

  — If (FIRST_INDEX & 1) is equal to 0, this run is non-zero.
  — Otherwise this run is zero.

— The binary event of whether the magnitude of the first non-zero coefficient is equal to 1 or greater than 1 as follows:

  — If (FIRST_INDEX & 2) is equal to 0, this magnitude is equal to 1.
  — Otherwise, this magnitude is greater than 1.

— The ternary event of whether the first coefficient is the last coefficient in the block, and if there are more non-zero coefficients whether the run before the next non-zero coefficient is zero or non-zero, as follows.

  — If (FIRST_INDEX >> 2) is equal to 0, the first coefficient is the last coefficient in the block.
  — Otherwise, if (FIRST_INDEX >> 2) is equal to 1, the run before the next non-zero coefficient is zero.
  — Otherwise ((FIRST_INDEX >> 2) is equal to 2), the run before the next non-zero coefficient is non-zero.

NOTE Thus FIRST_INDEX has an alphabet size of 2*3*3 = 12.

8.7.19 Macroblock FLEXBITS

8.7.19.1 MB_FLEXBITS( )

Table 83 — MB_FLEXBITS( ) syntax structure

MB_FLEXBITS( ) {	Descriptor	Reference
/* IsCurrPlaneAlphaFlag is equal to TRUE for parsing alpha image plane, and IsCurrPlaneAlphaFlag is equal to FALSE for parsing primary image plane */		
iHierScanOrder[ ] = {0, 1, 4, 5, 2, 3, 6, 7, 8, 9, 12, 13, 10, 11, 14, 15}		
for (i = 0; i < NumComponents; i++) {		
iIndex = 0		
if (i>0)		
iIndex = 1		
iModelBits = ModelBitsMBHP[MBx][MBy][iIndex]		
iNBlocks = 4		
if ((iIndex = = 1) && (INTERNAL_CLR_FMT = = YUV420))		
iNBlocks = 1		
else if (iIndex && (INTERNAL_CLR_FMT = = YUV422))		
iNBlocks = 2		
for (iBlock = 0; iBlock < iNBlocks * 4; iBlock++) {		
iBlockMap = iBlock		
if (iNBlocks = = 4)		
iBlockMap = iHierScanOrder[iBlock]		
BLOCK_FLEXBITS(i, iBlockMap, iModelBits, TRIM_FLEXBITS)		8.7.19.2
}		
}		
}		

NOTE Informative remarks related to this subclause are provided in subclause D.9.

8.7.19.2 BLOCK_FLEXBITS( )

Table 84 — BLOCK_FLEXBITS( ) syntax structure

BLOCK_FLEXBITS(iComponent, iBlock, iModelBits, iTrimFlexBits) {	Descriptor	Reference
iTranspose444[ ] = {0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15}		
iFlexBitsLeft = iModelBits − iTrimFlexBits		
if (iFlexBitsLeft < 0)		
iFlexBitsLeft = 0		
if (iFlexBitsLeft)		
for (n =1; n < 16; n++)		
HPInputFlex[iComponent][iBlock][iTranspose444[n]]= DECODE_FLEX(HPInputVLC[iComponent][iBlock][iTranspose444[n]], iFlexBitsLeft) << iTrimFlexBits		8.7.19.3
}		

8.7.19.3 DECODE_FLEX( )

Table 85 — DECODE_FLEX( ) syntax structure

DECODE_FLEX(iVLCCoeff, iFlexBitsLeft) {	Descriptor	Reference
FLEX_REF	u(iFlexBitsLeft)	8.7.19.4.1
if (iVLCCoeff > 0)		
iFlexCoeff = FLEX_REF		
else if (iVLCCoeff < 0)		
iFlexCoeff = −FLEX_REF		
else {		
iFlexCoeff = FLEX_REF		
if (iFlexCoeff) {		
SIGN_FLAG	u(1)	8.7.14.4
if (SIGN_FLAG)		
iFlexCoeff = −iFlexCoeff		
}		
}		
return iFlexCoeff		
}		

8.7.19.4 FLEXBITS semantics

8.7.19.4.1 FLEX_REF

FLEX_REF is a syntax element that specifies the flexbits part of the HP coefficient. The size of this syntax element is specified by iFlexBits.

8.8 Adaptive VLC code table selection

8.8.1 General

Adaptive VLC Table Selection is a process by which the entropy coding method adapts to image statistics and thus provides better compression. First, a small number of representative VLC tables are predefined. These tables are designed so as to be suitable for a wide range of statistics. During the entropy coding process, the most appropriate code table is selected based on the history of recently-coded symbols. The VLC tables and the information required for adaptation are precomputed as follows

Let the set of VLC tables be: **vT1**, **vT2**, ..., **vTn** where the ordering of the tables is predefined based on their relative similarity. That is **vT1** and **vT2** exhibit greater similarity to each other than **vT1** and **vT3** do to each other. The measure of similarity between tables can be qualified by using relative entropy. Let table **vTi** be a VLC table used for entropy coding of symbol iX, and let **vTj** and **vTk** be the two nearest tables (in terms of similarity). For each value of symbol iX, a metric deltaDisc estimating the relative advantage of coding this value using table **vTj** or table **vTk** instead of table **vTi** is precomputed and stored in tables **vTi.**DeltaTableIndex and **vTi.**Delta2TableIndex.

This deltaDisc is positive if this symbol is more efficiently coded by the new table instead of the current table. This deltaDisc is negative if this symbol is less efficiently coded by the new table instead of the current table. For example, if a given value of the symbol requires 3 bits in the new table, while it requires 5 bits in the current table, this value is more efficiently coded in the new table and the corresponding deltaDisc is (5 − 3 = 2).

The Table selection proceeds as follows: After entropy coding a symbol, the adaptation process computes the relative advantage of the two nearest tables for coding this symbol. The weights obtained from the tables **vTi**.DeltaTableIndex and **vTi**.Delta2TableIndex are added to two discriminants, **vTi**.DiscrimVal1 and **vTi**.DiscrimVal2, that are used for accumulating the statistics regarding code table transition. The AdaptVLCTable1( ) (subclause 8.8.4.4) and the AdaptVLCTable2( ) (subclause 8.8.4.5) pseudocode functions specify how these discriminants are compared to predefined threshold, and decide whether to continue to use the current table or to transition to one of the nearest table. If there is a transition, the discriminants are reset to zero.

As the discriminants are computed based on previously coded symbols both at the encoder and at the decoder, there is no need for additional side information to signal the selected VLC. For coding some symbols, there are only two code tables, and there is only one possible transition. The adaptation complexity is further reduced in these cases as there is only the DeltaTableIndex and only one discriminant is required.

NOTE Informative remarks related to this subclause are provided in subclause D.9.

8.8.2 Adaptive VLC deltaDisc tables

When a syntax element is parsed using a VLC having a code table that can be adaptively selected, the associated AdaptiveVLC data structure's member variables DiscrimVal1 and DiscrimVal2 are modified by adding an amount deltaDisc. The specific value of this deltaDisc is dependent on the syntax element being parsed, the VLC Code Table currently being used to parse this syntax element, the current value of this syntax element, and (when there are two discriminants) which discriminant is being modified.

Based on these factors, the appropriate values of deltaDisc are specified by collecting them in tables. For each syntax element, a distinct collection of tables of deltaDisc values are defined; in the sequel these tables will be called deltaDisc tables. A syntax element which is parsed using adaptive VLC tables will have N code tables for each syntax element, for some positive integer N > 1. If there are N code tables for a syntax element, there are N − 1 deltaDisc tables (one deltaDisc table for switching between code tables i and i + 1, for i ranging between 0 and N − 2 inclusive). The AdaptiveVLC data structure associates to DiscrimVal1 the associated variable DeltaTableIndex (Delta2TableIndex for DiscrimVal2); DeltaTableIndex defines which of the N − 1 deltaDisc tables is in use for the current macroblock.

The syntax element itself takes values between 0 and M − 1, where M represents the number of entries in this syntax element's code table. For each of these values, there is defined an associated deltaDisc value. In this way, when the syntax element takes the value iVal, the modification of an AdaptiveVLC data structures discriminants is specified as follows (here, **sAdaptVLC** is a local variable used in place of particular AdaptiveVLC data structure instance):

— **sAdaptVLC**.DiscrimVal1 is incremented by deltaDisc[**sAdaptVLC**.DeltaTableIndex][iVal].

— **sAdaptVLC**.DiscrimVal2 is incremented by deltaDisc[**sAdaptVLC**.Delta2TableIndex][iVal].

Table 86, Table 87, Table 88, Table 89, Table 90, and Table 91 specify the set of deltaDisc code tables for adaptive VLC syntax elements.

For the syntax element ABS_LEVEL_INDEX (subclause 8.7.14.5), there is one deltaDisc table for switching between Code Tables 0 and 1, as specified in Table 86.

Table 86 — Constant table AbslevelIndexDelta[m][n]

Index value n	Value for m = 0
0	1
1	0
2	−1
3	−1
4	−1
5	−1
6	−1

For the syntax element FIRST_INDEX (subclause 8.7.18.9.7), there are four Delta tables, with the table associating a different deltaDisc value for each DeltaTableIndex and each value of FIRST_INDEX; the deltaDisc values are specified in Table 87.

Table 87 — Constant table FirstIndexDelta[m][n]

Index value n	Value for m = 0	Value for m = 1	Value for m = 2	Value for m = 3
0	1	2	−1	0
1	1	2	1	1
2	1	−1	0	0
3	1	−1	2	1
4	1	−1	0	−2
5	0	0	0	0
6	0	−2	0	−1
7	−1	−1	0	−1
8	2	0	−2	−2
9	1	0	0	−1
10	0	−2	1	−2
11	0	−1	1	−2

For the syntax element INDEX_A (subclause 8.7.18.9.4), there are three Delta tables, as specified by Table 88.

Table 88 — Constant table Index1Delta[m][n]

Index value n	Value for m = 0	Value for m = 1	Value for m = 2
0	−1	−2	−1
1	1	0	−1
2	1	0	0
3	1	2	1
4	0	0	−2
5	1	0	0

For the syntax element NUM_CBPHP (subclause 8.7.17.4.1), there is one deltaDisc table for switching between Code Tables 0 and 1 as specified in Table 89.

Table 89 — Constant table NumCBPHPDelta[m][n]

Index value n	Value for m = 0
0	0
1	−1
2	0
3	1
4	1

For the syntax element NUM_BLKCBPHP (subclause 8.7.17.4.2), there is one deltaDisc table for switching between Code Tables 0 and 1. In the case where the INTERNAL_CLR_FMT is YONLY, NCOMPONENT, or YUVK, the code tables have five symbols and the deltaDisc table is specified by Table 90.

Table 90 — Constant table NumBlkCBPHPDelta[m][n] when INTERNAL_CLR_FMT is YONLY, NCOMPONENT, or YUVK

Index value n	Value for m = 0
0	0
1	−1
2	0
3	1
4	1

For all other values of INTERNAL_CLR_FMT, the code table for NUM_BLKCBPHP has nine symbols, and the deltaDisc table is specified by Table 91.

Table 91 — Constant table NumBlkCBPHPDelta for INTERNAL_CLR_FMT other than YONLY, NCOMPONENT, and YUVK

Index value n	Value for m = 0
0	2
1	2
2	1
3	1
4	−1
5	−2
6	−2
7	−2
8	−3

8.8.3 Initialization

The relevant adaptive VLC data structures that are associated with each of the three bands DC, LP and HP are initialized by the functions InitializeDCVLC( ), InitializeLPVLC( ), and InitializeHPVLC( ). These functions are specified in subclauses 8.8.3.1, 8.8.3.2, and 8.8.3.3, respectively. The adaptive VLC structure associated with CBPHP is initialized by the function InitializeCBPHPVLC( ) that is specified in subclause 8.8.3.4.

These functions in turn make use of the functions InitializeVLCTable1( ) and InitializeVLCTable2( ) which are specified in subclauses 8.8.3.5 and 8.8.3.6, respectively.

NOTE InitializeVLCTable1( ) is used for initializing VLC code tables when there are exactly two code tables. If there are more than two code table tables, InitializeVLCTable2( ) is used.

8.8.3.1 InitializeDCVLC( )

Table 92 — Pseudocode for function InitializeDCVLC( )

InitializeDCVLC( ) {	**Reference**
AbsLevelIndDCLum = InitializeVLCTable1(**AbsLevelIndDCLum**)	8.8.3.5
AbsLevelIndDCChr = InitializeVLCTable1(**AbsLevelIndDCChr**)	8.8.3.5
}	

8.8.3.2 InitializeLPVLC( )

Table 93 — Pseudocode for function InitializeLPVLC( )

InitializeLPVLC( ) {	Reference
DecFirstIndLPLum = InitializeVLCTable2(**DecFirstIndLPLum**)	8.8.3.6
DecIndLPLum0 = InitializeVLCTable2(**DecIndLPLum0**)	8.8.3.6
DecIndLPLum1 = InitializeVLCTable2(**DecIndLPLum1**)	8.8.3.6
DecFirstIndLPChr = InitializeVLCTable2(**DecFirstIndLPChr**)	8.8.3.6
DecIndLPChr0 = InitializeVLCTable2(**DecIndLPChr0**)	8.8.3.6
DecIndLPChr1 = InitializeVLCTable2(**DecIndLPChr1**)	8.8.3.6
AbsLevelIndLP0 = InitializeVLCTable1(**AbsLevelIndLP0**)	8.8.3.5
AbsLevelIndLP1 = InitializeVLCTable1(**AbsLevelIndLP1**)	8.8.3.5
}	

8.8.3.3 InitializeHPVLC( )

Table 94 — Pseudocode for function InitializeHPVLC( )

InitializeHPVLC( ) {	Reference
DecFirstIndHPLum = InitializeVLCTable2(**DecFirstIndHPLum**)	8.8.3.6
DecIndHPLum0 = InitializeVLCTable2(**DecIndHPLum0**)	8.8.3.6
DecIndHPLum1 = InitializeVLCTable2(**DecIndHPLum1**)	8.8.3.6
DecFirstIndHPChr = InitializeVLCTable2(**DecFirstIndHPChr**)	8.8.3.6
DecIndHPChr0 = InitializeVLCTable2(**DecIndHPChr0**)	8.8.3.6
DecIndHPChr1 = InitializeVLCTable2(**DecIndHPChr1**)	8.8.3.6
AbsLevelIndHP0 = InitializeVLCTable1(**AbsLevelIndHP0**)	8.8.3.5
AbsLevelIndHP1 = InitializeVLCTable1(**AbsLevelIndHP1**)	8.8.3.5
}	

8.8.3.4 InitializeCBPHPVLC( )

Table 95 — Pseudocode for function InitializeCBPHPVLC( )

InitializeCBPHPVLC( ) {	Reference
DecNumCBPHP = InitializeVLCTable1(**DecNumCBPHP**)	8.8.3.5
DecNumBlkCBPHP = InitializeVLCTable1(**DecNumBlkCBPHP**)	8.8.3.5
}	

8.8.3.5 InitializeVLCTable1( )

Table 96 — Pseudocode for function InitializeVLCTable1( )

InitializeVLCTable1(sAdaptVLC) {	Reference
/* **sAdaptVLC** is an instance of the AdaptiveVLC data structure */	
sAdaptVLC.TableIndex = 0	
sAdaptVLC.DeltaTableIndex = 0	
sAdaptVLC.DiscrimVal1 = 0	
return **sAdaptVLC**	
}	

8.8.3.6 InitializeVLCTable2( )

Table 97 — Pseudocode for function InitializeVLCTable2( )

InitializeVLCTable2(sAdaptVLC) {	**Reference**
/* **sAdaptVLC** is an instance of the AdaptiveVLC data structure */	
sAdaptVLC.TableIndex = 1	
sAdaptVLC.DeltaTableIndex = 0	
sAdaptVLC.Delta2TableIndex = 1	
sAdaptVLC.DiscrimVal1 = 0	
sAdaptVLC.DiscrimVal2 = 0	
return **sAdaptVLC**	
}	

8.8.4 Update of adaptive VLC code table selection

The relevant adaptive VLC data structures that are associated with each of the three bands DC, LP and HP are updated by the functions AdaptDC( ), AdaptLP( ), and AdaptHP( ), respectively. The pseudocode for the functions AdaptDC( ), AdaptLP( ), and AdaptHP( ) are specified in subclause 8.8.4.1, subclause 8.8.4.2, and subclause 8.8.4.3, respectively:

The functions AdaptLP( ) and AdaptHP( ) perform the updates by using the functions AdaptVLCTable1( ) and AdaptVLCTable2( ) which are specified in subclauses 8.8.4.4 and 8.8.4.5.

8.8.4.1 AdaptDC( )

Table 98 — Pseudocode for function AdaptDC( )

AdaptDC( ) {	**Reference**
AbsLevelIndDCLum = AdaptVLCTable1(**AbsLevelIndDCLum**)	8.8.4.4
AbsLevelIndDCChr = AdaptVLCTable1(**AbsLevelIndDCChr**)	8.8.4.4
}	

8.8.4.2 AdaptLP( )

Table 99 — Pseudocode for function AdaptLP( )

AdaptLP( ) {	**Reference**
DecFirstIndLPLum = AdaptVLCTable2(**DecFirstIndLPLum**, 4)	8.8.4.5
DecIndLPLum0 = AdaptVLCTable2(**DecIndLPLum0**, 3)	8.8.4.5
DecIndLPLum1 = AdaptVLCTable2(**DecIndLPLum1**, 3)	8.8.4.5
DecFirstIndLPChr = AdaptVLCTable2(**DecFirstIndLPChr**, 4)	8.8.4.5
DecIndLPChr0 = AdaptVLCTable2(**DecIndLPChr0**, 3)	8.8.4.5
DecIndLPChr1 = AdaptVLCTable2(**DecIndLPChr1**, 3)	8.8.4.5
AbsLevelIndLP0 = AdaptVLCTable1(**AbsLevelIndLP0**)	8.8.4.4
AbsLevelIndLP1 = AdaptVLCTable1(**AbsLevelIndLP1**)	8.8.4.4
}	

8.8.4.3 AdaptHP( )

Table 100 — Pseudocode for function AdaptHP( )

AdaptHP( ) {	**Reference**
DecFirstIndHPLum = AdaptVLCTable2(**DecFirstIndHPLum**, 4)	8.8.4.5
DecIndHPLum0 = AdaptVLCTable2(**DecIndHPLum0**, 3)	8.8.4.5
DecIndHPLum1 = AdaptVLCTable2(**DecIndHPLum1**, 3)	8.8.4.5
DecFirstIndHPChr = AdaptVLCTable2(**DecFirstIndHPChr**, 4)	8.8.4.5
DecIndHPChr0 = AdaptVLCTable2(**DecIndHPChr0**, 3)	8.8.4.5
DecIndHPChr1 = AdaptVLCTable2(**DecIndHPChr1**, 3)	8.8.4.5
AbsLevelIndHP0 = AdaptVLCTable1(**AbsLevelIndHP0**)	8.8.4.4
AbsLevelIndHP1 = AdaptVLCTable1(**AbsLevelIndHP**1)	8.8.4.4
DecNumCBPHP = AdaptVLCTable1(**DecNumCBPHP**)	8.8.4.4
DecNumBlkCBPHP = AdaptVLCTable1(**DecNumBlkCBPHP**)	8.8.4.4
}	

8.8.4.4 AdaptVLCTable1( )

AdaptVLCTable1( )is used for choosing VLC code tables when there are exactly two code tables. In this case, the index TableIndex takes only the values 0 or 1, and there is only one parameter (DiscrimVal1) which determines the selection of VLC code tables. DeltaTableIndex takes only the value 0, and there is only the one deltaDisc table.

Table 101 — Pseudocode for function AdaptVLCTable1( )

AdaptVLCTable1(sAdaptVLC) {	**Reference**
/* **sAdaptVLC** is an instance of the AdaptiveVLC data structure */	
iMaxTableIndex = 1 /* Only two code tables */	
cLowerBound = −8	
cUpperBound = 8	
/* **sAdaptVLC**.DeltaTableIndex = 0, since only 2 code tables */	
if (**sAdaptVLC**.DiscrimVal1 < cLowerBound && **sAdaptVLC**.TableIndex != 0) {	
sAdaptVLC.TableIndex– –	
sAdaptVLC.DiscrimVal1 = 0	
} else if (**sAdaptVLC**.DiscrimVal1 > cUpperBound && **sAdaptVLC**.TableIndex != iMaxTableIndex) {	
sAdaptVLC.TableIndex++	
sAdaptVLC.DiscrimVal1 = 0	
} else {	
/* no change to table, but clip the discriminant */	
if (**sAdaptVLC**.DiscrimVal1 < −64)	
sAdaptVLC.DiscrimVal1 = −64	
if (**sAdaptVLC**.DiscrimVal1 > 64)	
sAdaptVLC.DiscrimVal1 = 64	
}	
return **sAdaptVLC**	
}	

8.8.4.5 AdaptVLCTable2( )

AdaptVLCTable2( ) is used for choosing VLC code tables when there are more than two possible code tables. In this case, the index TableIndex can take values between 0 and the maximum table index for that set of VLC Code tables. This maximum table index is contained in the parameter iMaxTableIndex. For AdaptVLCTable2( ), there are two parameters (DiscrimVal1 and DiscrimVal2) which determines the selection of VLC code tables. DiscrimVal1 determines whether the code table index should be decreased, while DiscrimVal2 determines whether the code table index should be increased.

Table 102 — Pseudocode for function AdaptVLCTable2( )

AdaptVLCTable2(sAdaptVLC, iMaxTableIndex) {	**Reference**
/* **sAdaptVLC** is an instance of the AdaptiveVLC struct */	
/* iMaxTableIndex – max table index possible for this struct instance */	
bChange = FALSE	
iDiscrimLow = **sAdaptVLC**.DiscrimVal1	
iDiscrimHigh = **sAdaptVLC**.DiscrimVal2	
cLowerBound = −8	
cUpperBound = 8	
if (iDiscrimLow < cLowerBound && **sAdaptVLC**.TableIndex != 0) {	
sAdaptVLC.TableIndex– –	
bChange = TRUE	
} else if (iDiscrimHigh > cUpperBound && **sAdaptVLC**.TableIndex != iMaxTableIndex) {	
sAdaptVLC.TableIndex++	
bChange = TRUE	
}	
if (bChange) {	
sAdaptVLC.DiscrimVal1 = 0	
sAdaptVLC.DiscrimVal2 = 0	
if (**sAdaptVLC**.TableIndex = = iMaxTableIndex) {	
sAdaptVLC.DeltaTableIndex = **sAdaptVLC**.TableIndex – 1	
sAdaptVLC.Delta2TableIndex = **sAdaptVLC**.TableIndex – 1	
} else if (**sAdaptVLC**.TableIndex = = 0) {	
sAdaptVLC.DeltaTableIndex = **sAdaptVLC**.TableIndex	
sAdaptVLC.Delta2TableIndex = **sAdaptVLC**.TableIndex	
} else {	
sAdaptVLC.DeltaTableIndex = **sAdaptVLC**.TableIndex – 1	
sAdaptVLC.Delta2TableIndex = **sAdaptVLC**.TableIndex	
}	
} else { /* no change to table, but clip the discriminant */	
if (**sAdaptVLC**.DiscrimVal1 < −64)	
sAdaptVLC.DiscrimVal1 = −64	
if (**sAdaptVLC**.DiscrimVal1 > 64)	
sAdaptVLC.DiscrimVal1 = 64	
if (**sAdaptVLC**.DiscrimVal2 < −64)	
sAdaptVLC.DiscrimVal2 = −64	
if (**sAdaptVLC**.DiscrimVal2 > 64)	
sAdaptVLC.DiscrimVal2 = 64	
}	
return **sAdaptVLC**	
}	

8.9 Adaptation of CBPLP state variables

8.9.1 General

The parsing of CBPLP depends on the value of the variables CountZeroCBPLP and CountMaxCBPLP. The functions specified in this subclause specify the initialization and updating of these constants.

8.9.2 InitializeCountCBPLP( )

The variable CountZeroCBPLP and CountMaxCBPLP for LP coefficients are initialized by the function InitializeCountCBPLP( ) specified in Table 103.

Table 103 — Pseudocode for function InitializeCountCBPLP( )

InitializeCountCBPLP( ) {	**Reference**
CountZeroCBPLP = 1	
CountMaxCBPLP = 1	
}	

8.9.3 UpdateCountCBPLP( )

The function UpdateCountCBPLP( ) updates the variables CountZeroCBPLP and CountMaxCBPLP. The pseudocode for this function is specified in Table 104.

Table 104 — Pseudocode for function UpdateCountCBPLP( )

UpdateCountCBPLP(iCBPLP, iMax) {	**Reference**
CountZeroCBPLP += 1 – (4 * (iCBPLP = = 0))	
CountZeroCBPLP = Max(−8, Min(7,CountZeroCBPLP))	
CountMaxCBPLP += 1 – (4 * (iCBPLP = = iMax))	
CountMaxCBPLP = Max(−8, Min(7,CountMaxCBPLP))	
}	

8.10 Adaptive CBPHP prediction

CBPHP prediction depends on the value of member variables of the data structure instance **CBPHPModelHP**. The functions specified in this subclause specify the initialization and updating of this data structure.

8.10.1 InitializeCBPHPModel( )

The data structure instance **CBPHPModelHP** is initialized in the function InitializeCBPHPModel( ) specified in Table 105.

Table 105 — Pseudocode for function InitializeCBPHPModel( )

InitializeCBPHPModel( ) {	**Reference**
CBPHPModelHP.CBPHPState[0] = **CBPHPModelHP**.CBPHPState[1] = 0	
CBPHPModelHP.CountOnes[0] = **CBPHPModelHP**.CountOnes[1] = −4	
CBPHPModelHP.CountZeroes[0] = **CBPHPModelHP**.CountZeroes[1] = 4	
}	

8.10.2 UpdateCBPHPModel( )

The variables associated with the data structure instance **CBPHPModelHP** are updated by the function UpdateCBPHPModel( ) as specified in Table 106.

Table 106 — Pseudocode for function UpdateCBPHPModel( )

UpdateCBPHPModel(i, iNOrig) {	**Reference**
iNDiff = 3	
CBPHPModelHP.CountOnes[i] += iNOrig − iNDiff	
CBPHPModelHP.CountOnes[i] = Clip(**CBPHPModelHP**.CountOnes[i], −16,15)	
CBPHPModelHP.CountZeroes[i] += 16 − iNOrig − iNDiff	
CBPHPModelHP.CountZeroes[i] = Clip(**CBPHPModelHP**.CountZeroes[i], −16, 15)	
if (**CBPHPModelHP**.CountOnes[i] < 0)	
if (**CBPHPModelHP**.CountOnes[i] < CBPHPModelHP.CountZeroes[i])	
CBPHPModelHP.CBPHPState[i] = 1	
else	
CBPHPModelHP.CBPHPState[i] = 2	
else if (**CBPHPModelHP**.CountZeroes[i] < 0)	
CBPHPModelHP.CBPHPState[i] = 2	
else	
CBPHPModelHP.CBPHPState[i] = 0	
}	

8.11 Adaptive inverse scanning

The parsing of syntax elements corresponding to LP and HP coefficients depends on the state of the inverse scanning tables LowpassScanOrder[i], HighpassHorScanOrder[i], and HighpassVerScanOrder[i]. The functions specified in this subclause define the initialization and updating of these tables.

8.11.1 Adaptive inverse scanning tables

The inverse scanning order of transform coefficients is a permutation of the integers 1 to 15. Let the integer i represent the order in which a given transform coefficient is parsed from the codestream, and let the local example list listScanOrder[ ] specify an inverse scanning order as follows: the i-th transform coefficient is put into the block in the j-th position in raster scan order, where j is equal to listScanOrder[i].

The three lists LowpassScanOrder[ ], HighpassHorScanOrder[ ], and HighpassVerScanOrder[ ], are used to specify the inverse scanning order of LP coefficients, HP coefficients in the case of prediction from the left (subclause 9.6), and HP coefficients in the case of prediction from the top, respectively. These lists are initialized to scan orders as specified below. However, the lists are adaptive, and thus may change over the course of parsing, based on the statistics of non-zero transform coefficients in the codestream.

The three lists LowpassScanOrder[ ], HighpassHorScanOrder[ ], and HighpassVerScanOrder[ ] are initialized as specified in subclauses 8.11.2 and 8.11.3. The initial orders are specified by the two lists ScanOrder0[ ] and ScanOrder1[ ], which are specified by Table 107.

Table 107 — Definitions of ScanOrder0 and ScanOrder1

i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
ScanOrder0[i]	4	1	5	8	2	9	6	12	3	10	13	7	14	11	15
ScanOrder1[i]	1	2	5	4	3	6	9	8	7	12	15	13	10	11	14

Each of the lists LowpassScanOrder[ ], HighpassHorScanOrder[ ] and HighpassVerScanOrder[ ] also has an associated list that determines how the scan order is updated. These corresponding lists are LowpassTotals[ ], HighpassHorTotals[ ], and HighpassVerTotals[ ], respectively. These associated lists are initialized to be equal to the list ScanTotals[ ], which is specified by Table 108.

Table 108 — Definition of ScanTotals

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ScanTotals[i]	32	30	28	26	24	22	20	18	16	14	12	10	8	6	4

The three lists LowpassScanOrder[ ], HighpassHorScanOrder[ ], and HighpassVerScanOrder[ ] are updated as specified in subclause 8.11.

8.11.2 InitializeAdaptiveScanLP( )

The adaptive inverse scanning tables for LP coefficients are initialized in the function InitializeAdaptiveScanLP( ) specified in Table 109.

Table 109 — Pseudocode for function InitializeAdaptiveScanLP( )

InitializeAdaptiveScanLP( ) {	Reference
for (i = 1; i <= 15; i++) {	
LowpassScanOrder[i] = ScanOrder0[i]	ScanOrder0[i] specified in subclause 8.11.1
LowpassTotals[i] = ScanTotals[i]	ScanTotals[i] specified in subclause 8.11.1
}	
}	

8.11.3 InitializeAdaptiveScanHP( )

The adaptive inverse scanning tables for HP coefficients are initialized in the function InitializeAdaptiveScanHP( ) specified in Table 110.

Table 110 — Pseudocode for function InitializeAdaptiveScanHP( )

InitializeAdaptiveScanHP( ) {	Reference
for (i = 1; i <= 15; i++) {	
HighpassHorScanOrder[i] = ScanOrder0[i]	ScanOrder0[i] specified in subclause 8.11.1
HighpassVerScanOrder[i] = ScanOrder1[i]	ScanOrder1[i] specified in subclause 8.11.1
HighpassHorTotals[i] = ScanTotals[i]	ScanTotals[i] specified in subclause 8.11.1
HighpassVerTotals[i] = ScanTotals[i]	ScanTotals[i] specified in subclause 8.11.1
}	
}	

8.11.4 ResetTotalsAdaptiveScanLP( )

The list LowpassTotals of the adaptive inverse scanning tables for LP coefficients is reset in the function ResetTotalsAdaptiveScanLP( ) specified in Table 111.

Table 111 — Pseudocode for function ResetTotalsAdaptiveScanLP( )

ResetTotalsAdaptiveScanLP( ) {	Reference
for (i = 1; i <= 15; i++)	
LowpassTotals[i] = ScanTotals[i]	ScanTotals[i] specified in subclause 8.11.1
}	

8.11.5 ResetTotalsAdaptiveScanHP( )

The lists HighpassHorTotals and HighpassVerTotals of the adaptive inverse scanning tables for HP coefficients are reset in the function ResetTotalsAdaptiveScanHP( ) specified in Table 112.

Table 112 — Pseudocode for function ResetTotalsAdaptiveScanHP( )

ResetTotalsAdaptiveScanHP( ) {	**Reference**
for (i = 1; i <= 15; i++) {	
HighpassHorTotals[i] = ScanTotals[i]	ScanTotals[i] specified in subclause 8.11.1
HighpassVerTotals[i] = ScanTotals[i]	ScanTotals[i] specified in subclause 8.11.1
}	
}	

8.11.6 AdaptiveLPScan( )

The function AdaptiveLPScan( ) updates the list LPInput[k], and also updates the variables associated with tracking and modifying the LP scan order LowpassScanOrder[i] as specified in Table 113.

Table 113 — Pseudocode for function AdaptiveLPScan( )

AdaptiveLPScan(n, i, iValue) {	**Reference**
k = LowpassScanOrder[i]	
LPInput[n][k] = iValue	
LowpassTotals[i]++	
if ((i > 1) && (LowpassTotals[i] > LowpassTotals[i−1])) {	
valueTemp = LowpassTotals[i]	
LowpassTotals[i] = LowpassTotals[i−1]	
LowpassTotals[i−1] = valueTemp	
valueTemp = LowpassScanOrder[i]	
LowpassScanOrder[i] = LowpassScanOrder[i−1]	
LowpassScanOrder[i−1] = valueTemp	
}	
}	

8.11.7 AdaptiveHPScan( )

The function AdaptiveHPScan( ) updates the list HPInputVLC[iComponent][iBlock][k], and also updates the variables associated with tracking and modifying the HP scan orders HighpassHorScanOrder[i] and HighpassVerScanOrder[i] as specified in Table 114.

Table 114 — Pseudocode for function AdaptiveHPScan( )

AdaptiveHPScan(iComponent, iBlock, i, iValue) {	**Reference**
if (MBHPMode = = 1) { /* vertical scan order */	
k = HighpassVerScanOrder[i]	
HighpassVerTotals[i]++	
HPInputVLC[iComponent][iBlock][k] = iValue	
if ((i > 1) && (HighpassVerTotals[i] > HighpassVerTotals[i−1])) {	
valueTemp = HighpassVerTotals[i]	
HighpassVerTotals[i] = HighpassVerTotals[i−1]	
HighpassVerTotals[i−1] = valueTemp	
valueTemp = HighpassVerScanOrder[i]	
HighpassVerScanOrder[i] = HighpassVerScanOrder[i−1]	
HighpassVerScanOrder[i−1] = valueTemp	
}	
} else { /* horizontal scan order */	
k = HighpassHorScanOrder[i]	
HighpassHorTotals[i]++	
HPInputVLC[iComponent][iBlock][k] = iValue	
if ((i > 1) && (HighpassHorTotals[i] > HighpassHorTotals[i−1])) {	
valueTemp = HighpassHorTotals[i]	
HighpassHorTotals[i] = HighpassHorTotals[i−1]	
HighpassHorTotals[i−1] = valueTemp	
valueTemp = HighpassHorScanOrder[i]	
HighpassHorScanOrder[i] = HighpassHorScanOrder[i−1]	
HighpassHorScanOrder[i−1] = valueTemp	
}	
}	
}	

The variable MBHPMode is computed during the HP prediction direction computation process specified in subclause 9.6.3.2. The scan order is selected based on the value of the variable MBHPMode. AdaptiveHPScan( ) shall only be invoked on a macroblock after the HP prediction direction computation process specified in subclause 9.6.3.2 has been invoked and completed for this macroblock. The HP prediction direction process shall be invoked only after the completion of the LP transform coefficient parsing process.

NOTE See subclause 9.6 for more information.

8.12 Adaptive coefficient normalization

8.12.1 InitializeModelMB( )

The initialization of the **Model** data structure is specified by the function InitializeModelMB( ) as specified in Table 115.

Table 115 — Pseudocode for function InitializeModelMB( )

InitializeModelMB(Model, iBand) {	**Reference**
/* iBand is the frequency band (DC = 0, LP = 1, HP = 2) */	
Model.MState[0] = **Model**.MState[1] = 0	
Model.MBits[0] = **Model**.MBits[1] = (2 − iBand) * 4	
}	

8.12.2 UpdateModelMB( )

The adaptation of the **Model** data structure is specified by the function UpdateModelMB( ) as specified in Table 116.

Table 116 — Pseudocode for function UpdateModelMB( )

UpdateModelMB(iLapMean[ ], Model, iBand) {	**Reference**
/* INTERNAL_CLR_FMT is the colour format of the image */	
/* iBand is the frequency band (DC = 0, LP = 1, HP = 2) */	
iModelWeight = 70	
iWeight0[3] = {240 /*DC*/, 12 /*LP*/, 1}	
iWeight1[3][MAX_COMPONENTS] = {	
{0, 240, 120, 80, 60, 48, 40,34, 30, 27, 24, 22, 20, 18, 17, 16},	
{0, 12, 6, 4, 3, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1},	
{0, 16, 8, 5, 4, 3, 3, 2, 2, 2, 2, 1, 1, 1, 1, 1}	
}	
iWeight2[6] = {120, 37, 2 /*YUV420*/, 120, 18, 1 /*YUV422*/}	
iLapMean[0] *= iWeight0[iBand]	
if (INTERNAL_CLR_FMT = = YUV420)	
iLapMean[1] *= iWeight2[iBand]	
else if (INTERNAL_CLR_FMT = = YUV422)	
iLapMean[1] *= iWeight2[3 + iBand]	
else {	
iLapMean[1] *= iWeight1[iBand][NumComponents − 1]	
if (iBand = = 2)	
iLapMean[1] >>= 4	
}	
iNumModels = 2	
if (INTERNAL_CLR_FMT = = YONLY)	
iNumModels = 1	
for (j = 0; j < iNumModels; j++) {	
iMS = **Model**.MState[j]	
iDelta = ((iLapMean[j] − iModelWeight) >> 2)	
if (iDelta <= −8) {	
iDelta += 4	
if (iDelta < −16)	
iDelta = −16	
iMS += iDelta	
if (iMS < −8)	
if (**Model**.MBits[j] = = 0)	
iMS = −8	
else {	
iMS = 0	
Model.MBits[j]− −	
}	
} else if (iDelta >= 8) {	
iDelta −= 4	
if (iDelta > 15)	
iDelta = 15	
iMS += iDelta	
if (iMS > 8)	
if (**Model**.MBits[j] >= 15) {	
Model.MBits[j] = 15	
iMS = 8	
} else {	
iMS = 0	
Model.MBits[j]++	
}	
}	
Model.MState[j] = iMS	
}	
}	

9 Decoding process

9.1 General

This clause specifies the decoding process. The decoding process is interdependent with the initialization of variables and parsing of syntax elements as specified in Clause 8.

The decoding process specified in this clause is distinguished from the codestream parsing process in the following manner: the codestream parsing process manages all control flow regarding the correct parsing of codestream syntax elements. This includes maintaining state variables for adaptive VLC selection, adaptive coefficient normalization, and other related information. The processes in this clause therefore are written with the assumption that, when they are invoked, the input variables required for this process have been correctly parsed from the codestream.

The decoding process is specified so that the decoded samples from any two JPEG XR decoders will be numerically identical. Any decoder which produces results that match the process specified here conforms to the requirements of this document.

The image decoding process proceeds as specified in subclause 9.2.

9.2 Image decoding

The outputs of this process are the output samples of the image.

The image decoding process proceeds as in Table 117.

Table 117 — Pseudocode for function ImageDecoding( )

<table>
<tr><th>ImageDecoding( ) {</th><th>Reference</th></tr>
<tr><td>ImagePlaneDecoding( )
/* resulting sample values are stored in the variables ImagePlane[i][x][y] */</td><td>9.3</td></tr>
<tr><td>if ((OUTPUT_CLR_FMT = = RGB) &&
((OUTPUT_BITDEPTH = = BD5) | | (OUTPUT_BITDEPTH = = BD565) | |
(OUTPUT_BITDEPTH = = BD10))) /* Packed RGB */</td><td></td></tr>
<tr><td>outputArrays = 1</td><td></td></tr>
<tr><td>else if ((OUTPUT_CLR_FMT = = RGB) | | (OUTPUT_CLR_FMT = = YUV444) | |
(OUTPUT_CLR_FMT = = YUV422) | | (OUTPUT_CLR_FMT = = YUV420))</td><td></td></tr>
<tr><td>outputArrays = 3</td><td></td></tr>
<tr><td>else if (OUTPUT_CLR_FMT = = RGBE)</td><td></td></tr>
<tr><td>outputArrays = 4</td><td></td></tr>
<tr><td>else</td><td></td></tr>
<tr><td>outputArrays = NumComponents</td><td></td></tr>
<tr><td>for (i = 0; i < outputArrays; i++) {</td><td></td></tr>
<tr><td>if ((i > 0) && (OUTPUT_CLR_FMT = = YUV420))</td><td></td></tr>
<tr><td>outputHeight = (HEIGHT_MINUS1 + 1) / 2</td><td></td></tr>
<tr><td>else</td><td></td></tr>
<tr><td>outputHeight = HEIGHT_MINUS1 + 1</td><td></td></tr>
<tr><td>if ((OUTPUT_BITDEPTH = = BD1WHITE1) | |
(OUTPUT_BITDEPTH = = BD1BLACK1)) /* Horizonally packed flags */</td><td></td></tr>
<tr><td>outputWidth = WIDTH_MINUS1 / 8 + 1</td><td></td></tr>
<tr><td>else if ((i > 0) &&
((OUTPUT_CLR_FMT = = YUV422) | | (OUTPUT_CLR_FMT = = YUV420)))</td><td></td></tr>
<tr><td>outputWidth = (WIDTH_MINUS1 + 1) / 2</td><td></td></tr>
<tr><td>else</td><td></td></tr>
<tr><td>outputWidth = WIDTH_MINUS1 + 1</td><td></td></tr>
<tr><td>for (y = 0; y < outputHeight; y++)</td><td></td></tr>
<tr><td>for (x = 0; x < outputWidth; x++)</td><td></td></tr>
<tr><td>ImagePrimary[i][x][y] = ImagePlane[i][x][y]</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>if (ALPHA_IMAGE_PLANE_FLAG = = TRUE)</td><td></td></tr>
</table>

ImageDecoding( ) {	Reference
ImagePlaneDecoding( ) /* resulting sample values, corresponding to the alpha image plane, are stored in the variables ImagePlane[0][x][y] */	9.3
for (y = 0; y <= HEIGHT_MINUS1; y++)	
for (x = 0; x <= WIDTH_MINUS1; x++)	
ImageAlpha[0][x][y] = ImagePlane[0][x][y]	
}	

NOTE Throughout the parsing of syntax elements in Clause 8, it is assumed that if ALPHA_IMAGE_PLANE_FLAG is equal to TRUE, there are two sets of parsed syntax elements: one set corresponding to the primary image plane, and one set corresponding to the alpha image plane. In the same manner, this subclause assumes that there are two sets of global variables being used in the decoding process, corresponding to the primary and alpha image planes, respectively.

9.3 Image plane decoding

This process is invoked for each image plane.

The outputs of this process are the decoded samples for this image plane, ImagePlane[i][x][y].

The image plane decoding process proceeds as specified in Table 118.

Table 118 — Pseudocode for function ImagePlaneDecoding( )

ImagePlaneDecoding( ) {	Reference
ImagePlaneDCQP( )	9.7.1.1
ImagePlaneLPQP( )	9.7.2.1
ImagePlaneHPQP( )	9.7.3.1
for (TileIndexy = 0; TileIndexy < NumTileRows; TileIndexy++)	
for (TileIndexx = 0; TileIndexx < NumTileCols; TileIndexx++)	
TileTransformCoefficientProcessing( ) /* At this point, transform coefficients for the entire image plane have been obtained */	9.4.1
SampleReconstruction( ) /* This process performs both levels of the inverse transform and overlap operations on the entire image */	9.9.1
OutputFormatting( )	9.10.2
}	

9.4 Tile transform coefficient processing

9.4.1 Overview

This process is invoked for each tile. The inputs to this process are the horizontal and vertical indices of the current tile.

The outputs of this process are transform coefficients for each macroblock in the tile.

The transform coefficient processing proceeds as specified in Table 119.

Table 119 — Pseudocode for function TileTransformCoefficientProcessing( )

TileTransformCoefficientProcessing( ) {	Reference
TileLevelDCQP( )	9.7.1.2
TileLevelLPQP( )	9.7.2.2
TileLevelHPQP( )	9.7.3.2
n = 0	
for (MBy = TopMBIndexOfTile[TileIndexy]; MBy < TopMBIndexOfTile[TileIndexy + 1]; MBy++)	
for (MBx = LeftMBIndexOfTile[TileIndexx]; MBx < LeftMBIndexOfTile[TileIndexx + 1]; MBx++) {	
MBQPIndexLP[MBx][MBy] = LP_QP_INDEX[n]	
MBQPIndexHP[MBx][MBy] = HP_QP_INDEX[n]	
DCTransformCoefficientDecoding( )	9.4.2
LPTransformCoefficientDecoding( )	9.4.3
HPTransformCoefficientDecoding( )	9.4.4
n += 1	
}	
}	

NOTE The computation of the global variable MBHPMode is dependent upon the completion of the LP transform coefficient decoding process. MBHPMode is computed at the beginning of the HP coefficient decoding process.

9.4.2 DC transform coefficient decoding

The outputs of this process are DC transform coefficients for each colour component of the current macroblock.

The DC transform coefficient processing proceeds as specified in Table 120.

Table 120 — Pseudocode for function DCTransformCoefficientDecoding( )

DCTransformCoefficientDecoding( ) {	Reference
DCMBCoefficientRemap( )	9.5.1
DCPredictionGeneral( )	9.6.1.1
DequantizeDCCoefficients( )	9.8.1
}	

9.4.3 Low-pass transform coefficient decoding

The outputs of this process are LP transform coefficients for each colour component of the current macroblock.

The LP transform coefficient processing proceeds as specified in Table 121.

Table 121 — Pseudocode for function LPTransformCoefficientDecoding( )

LPTransformCoefficientDecoding( ) {	Reference
LPMBCoefficientRemap( )	9.5.2
LPPredictionGeneral( )	9.6.2.1
DequantizeLPCoefficients( )	9.8.2
}	

9.4.4 High-pass transform coefficient decoding

Input to this process is the value of MBHPMode for the current macroblock.

The outputs of this process are HP transform coefficients for each colour component of the current macroblock.

The HP transform coefficient processing proceeds as specified in Table 122.

Table 122 — Pseudocode for function HPTransformCoefficientDecoding( )

HPTransformCoefficientDecoding( ) {	**Reference**
CalcHPPredMode( )	9.6.3.2
HPMBCoefficientRemap( )	9.5.3
DequantizeHPCoefficients( )	9.8.3
HPCoefficientPrediction( )	9.6.3.3
}	

9.5 Coefficient remapping

This subclause specifies the coefficient remapping processes for DC, LP and HP coefficients.

9.5.1 DC coefficient remapping

Input to this process is a list DCInput[i] of DC level values for each colour component i, and the variables MBx and MBy, which identify the current macroblock.

Output of this process is the list of values MbDCLP[MBx][MBy][i][0], of DC transform coefficients, for each colour component i of the current macroblock.

NOTE The values DCInput[i] are the outputs of the MB_DC( ) syntax structure of subclause 8.7.11.

The values in the array MbDCLP[MBx][MBy][i][0] are set by Table 123.

Table 123 — Pseudocode for function DCMBCoefficientRemap( )

DCMBCoefficientRemap( ) {	**Reference**
for (i = 0; i < NumComponents; i++)	
MbDCLP[MBx][MBy][i][0] = DCInput[i]	
}	

9.5.2 Low-pass coefficient remapping

Inputs to this process are a list of variables LPInput[i][j] which hold the j-th LP transform coefficient value for each colour component i.

Output of this process is the array of variables MbDCLP[MBx][MBy][i][j] which hold the j-th LP transform coefficients, indexed in raster scan order, of colour component i.

The LP coefficient remapping process proceeds as specified by Table 124.

Table 124 — Pseudocode for function LPMBCoefficientRemap( )

LPMBCoefficientRemap( ) {	**Reference**
for (i = 0; i < NumComponents; i++) {	
if (i != 0) && ((INTERNAL_CLR_FMT = = YUV422) | | (INTERNAL_CLR_FMT = = YUV420))	
if (INTERNAL_CLR_FMT = = YUV422)	
for (j = 1; j <= 7; j++)	
if (BANDS_PRESENT != DCONLY)	
MbDCLP[MBx][MBy][i][j] = LPInput[i][j]	
else	
MbDCLP[MBx][MBy][i][j] = 0	
else /* INTERNAL_CLR_FMT = = YUV420 */	
for (j = 1; j <= 3; j++)	
if (BANDS_PRESENT != DCONLY)	
MbDCLP[MBx][MBy][i][j] = LPInput[i][j]	
else	
MbDCLP[MBx][MBy][i][j] = 0	
else	
for (j = 1; j <= 15; j++)	
if (BANDS_PRESENT != DCONLY)	
MbDCLP[MBx][MBy][i][j] = LPInput[i][j]	
else	
MbDCLP[MBx][MBy][i][j] = 0	
}	
}	

9.5.3 High-pass macroblock coefficient remapping

The HP coefficient remapping process proceeds as in Table 125.

Table 125 — Pseudocode for function HPMBCoefficientRemap( )

HPMBCoefficientRemap( ) {	**Reference**
for (i = 0; i < NumComponents; i++) {	
if (i != 0 && INTERNAL_CLR_FMT = = YUV420)	
jMax = 3	
else if (i != 0 && INTERNAL_CLR_FMT = = YUV422)	
jMax = 7	
else	
jMax = 15	
for (j = 0; j <= jMax; j++)	
HPBlockCoefficientRemap(i, j)	9.5.4
}	
}	

9.5.4 High-pass block coefficient remapping

Inputs to this process are: the array HPInputVLC[currentComponent][blkIndex][j] for the current colour component currentComponent, and the current block index blkIndex, with j ranging from 1 to 15, the array HPInputFlex[currentComponent][blkIndex][j] for the current colour component currentComponent, and the current block index blkIndex, with j ranging from 1 to 15, the variable ModelBits[MBx][MBy], representing the number of flexbits for the current macroblock, and the variables MBx and MBy, which identify the current macroblock.

Output of this process are the values MBBuffer[MBx][MBy][currentComponent][k] of HP transform coefficients, with k ranging from (16 * blkIndex + 1) to (16 * blkIndex + 15), corresponding to the current block. Pseudocode for this process is in Table 126.

Table 126 — Pseudocode for function HPBlockCoefficientRemap( )

HPBlockCoefficientRemap(currentComponent, blkIndex) {	Reference
if (currentComponent = = 0)	
iIndex = 0	
else	
iIndex = 1	
for (j = 1; j <= 15; j++) {	
k = 16 * blkIndex + j	
if (BANDS_PRESENT = = ALL | | BANDS_PRESENT = = NOFLEXBITS)	
MBBuffer[MBx][MBy][currentComponent][k] = HPInputVLC[currentComponent][blkIndex][j] << ModelBitsMBHP[MBx][MBy][iIndex]	
else	
MBBuffer[MBx][MBy][currentComponent][k] = 0	
if (BANDS_PRESENT = = ALL)	
MBBuffer[MBx][MBy][currentComponent][k] += HPInputFlex[currentComponent][blkIndex][j]	
}	
}	

9.6 Transform coefficient prediction

9.6.1 DC coefficient prediction

9.6.1.1 Overview of DC prediction

Four modes are defined for the prediction of the DC coefficient of a macroblock. These modes are:

- Predict from left
- Predict from top
- Predict from left and top
- No prediction

The prediction mode is determined from the position of the macroblock, as well as the DC values to the left, top and top-left of the macroblock. Further, if the image has chroma components, the corresponding DC values of the chroma components are also used.

9.6.1.2 DCPredictionGeneral( )

Inputs to this process are the Boolean variables IsMBLeftEdgeofTileFlag and IsMBTopEdgeofTileFlag, as well as the values MbDCLP[MBx][MBy][i][0], for each colour component i of the current macroblock. The variable IsMBLeftEdgeofTileFlag is equal to TRUE when the current macroblock is at the left edge of the tile; IsMBTopEdgeofTileFlag is equal to TRUE when the current macroblock is at the top edge of the current tile.

NOTE The values MbDCLP[MBx][MBy][i][0] come from the DC coefficient remapping process of subclause 9.5.1.

Outputs of this process are the updated values MbDCLP[MBx][MBy][i][0], for each colour component i of the current macroblock.

The DC prediction process proceeds as in Table 127.

Table 127 — Pseudocode for function DCPredictionGeneral( )

DCPredictionGeneral( ) {	Reference
CalcDCPredMode( )	9.6.1.3
DCCoefficientPrediction( )	9.6.1.4
UpdateDCPredictionVariables( )	9.6.1.5
}	

9.6.1.3 DC prediction direction computation

Inputs to this process are the variables IsMBLeftEdgeofTileFlag and IsMBTopEdgeofTileFlag, and the values PredDCLP[MBx−1][MBy][i][0], PredDCLP[MBx][MBy−1][i][0], and PredDCLP[MBx−1][MBy−1][i][0], for each colour component i.

Output of this process is the value of MBDCMode. The possible values of MBDCMode are as follows: 0 specifies prediction from the left macroblock, 1 specifies prediction from the top macroblock, 2 specifies prediction from both the top and left macroblocks, and 3 specifies no prediction.

The DC prediction direction process proceeds as shown by Table 128.

Table 128 — Pseudocode for function CalcDCPredMode( )

CalcDCPredMode( ) {	Reference
if (IsMBLeftEdgeofTileFlag = = TRUE && IsMBTopEdgeofTileFlag = = TRUE)	
MBDCMode = 3 /* no prediction */	
else if (IsMBLeftEdgeofTileFlag = = TRUE && IsMBTopEdgeofTileFlag = = FALSE)	
MBDCMode = 1 /* prediction from top only */	
else if (IsMBLeftEdgeofTileFlag = = FALSE && IsMBTopEdgeofTileFlag = = TRUE)	
MBDCMode = 0 /* prediction from left only */	
else /* if (IsMBLeftEdgeofTileFlag = = FALSE && IsMBTopEdgeofTileFlag = = FALSE) */ {	
iLeft = PredDCLP[MBx−1][MBy][0][0]	
iTop = PredDCLP[MBx][MBy−1][0][0]	
iTopLeft = PredDCLP[MBx−1][MBy−1][0][0]	
if (INTERNAL_CLR_FMT = = Y_ONLY || INTERNAL_CLR_FMT = =NCOMPONENT) {	
iStrHor = Abs(iTopLeft − iLeft)	
iStrVer = Abs(iTopLeft − iTop)	
} else {	
iLeftU = PredDCLP[MBx−1][MBy][1][0]	
iTopU = PredDCLP[MBx][MBy−1][1][0]	
iTopLeftU = PredDCLP[MBx−1][MBy−1][1][0]	
iLeftV = PredDCLP[MBx−1][MBy][2][0]	
iTopV = PredDCLP[MBx][MBy−1][2][0]	
iTopLeftV = PredDCLP[MBx−1][MBy−1][2][0]	
iScale = 2	
if (INTERNAL_CLR_FMT = = YUV420)	
iScale = 8	
if (INTERNAL_CLR_FMT = = YUV422)	
iScale = 4	
iStrHor = Abs(iTopLeft − iLeft) * iScale + Abs(iTopLeftU − iLeftU) + Abs(iTopLeftV − iLeftV)	
iStrVer = Abs(iTopLeft − iTop) * iScale + Abs(iTopLeftU − iTopU) + Abs(iTopLeftV − iTopV)	
}	
iOrWt = 4	
if ((iStrHor * iOrWt) < iStrVer)	
MBDCMode = 1	

CalcDCPredMode( ) {	Reference
else if ((iStrVer * iOrWt) < iStrHor)	
MBDCMode = 0	
else	
MBDCMode = 2	
}	
}	

The value MBDCMode is used in subsequent stages of the DC prediction process.

9.6.1.4 DC coefficient prediction

This process occurs when MBDCMode is not equal to 3.

Inputs to this process are the variable MBDCMode representing the DC prediction direction, and the array variable MbDCLP[MBx][MBy][i][0] for each colour component i.

Outputs to this process are the updated values MbDCLP[MBx][MBy][i][0], for each colour component i of the current macroblock.

The DC coefficient prediction process proceeds according to Table 129.

Table 129 — Pseudocode for function DCCoefficientPrediction( )

<table>
<tr><th>DCCoefficientPrediction( ) {</th><th>Reference</th></tr>
<tr><td>for (i = 0; i < NumComponents; i++) {</td><td></td></tr>
<tr><td>iLeft = PredDCLP[MBx−1][MBy][i][0]</td><td></td></tr>
<tr><td>iTop = PredDCLP[MBx][MBy−1][i][0]</td><td></td></tr>
<tr><td>if (MBDCMode = = 0)</td><td></td></tr>
<tr><td>MbDCLP[MBx][MBy][i][0] += iLeft</td><td></td></tr>
<tr><td>else if (MBDCMode = = 1)</td><td></td></tr>
<tr><td>MbDCLP[MBx][MBy][i][0] += iTop</td><td></td></tr>
<tr><td>else if (MBDCMode = = 2) {</td><td></td></tr>
<tr><td>if (i = = 0) | | ((INTERNAL_CLR_FMT != YUV420) &&
(INTERNAL_CLR_FMT != YUV422))</td><td></td></tr>
<tr><td>MbDCLP[MBx][MBy][i][0] += (iTop + iLeft) >> 1</td><td></td></tr>
<tr><td>else /* (INTERNAL_CLR_FMT = = YUV420 | |
INTERNAL_CLR_FMT = = YUV422)*/</td><td></td></tr>
<tr><td>MbDCLP[MBx][MBy][i][0] +=
(iTop + iLeft + 1) >> 1</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>}</td><td></td></tr>
</table>

9.6.1.5 Update of DC prediction variables

Outputs of this process are the updated variables PredDCLP[MBx][MBy][i], for each colour component i, where MBx and MBy are indexing the current macroblock.

The update of DC prediction variables process proceeds as shown by Table 130.

Table 130 — Pseudocode for function UpdateDCPredictionVariables( )

UpdateDCPredictionVariables( ) {	Reference
for (i = 0; i < NumComponents; i++)	
PredDCLP[MBx][MBy][i][0] = MbDCLP[MBx][MBy][i][0]	
}	

9.6.2 Low-pass prediction

9.6.2.1 Overview of low-pass prediction

Three modes are defined for the prediction of the LP coefficient of the inner transform of a macroblock. These modes are:

— Prediction from left when MBLPMode is equal to 0;

— Prediction from top when MBLPMode is equal to 1;

— No prediction when MBLPMode is equal to 2.

The LP coefficient prediction mode (MBLPMode) is determined by the DC coefficient prediction mode, together with the quantization parameters of both the current block and the block from which the DC values were predicted.

This rule ensures that prediction of LP coefficients does not take place across macroblocks with different quantization parameters.

Not all of the LP coefficients associated with a macroblock are predicted. The definition and indices of DC and LP coefficients that are predicted is shown in Figure 5. The DC coefficient of the blocks shown in dark gray is at position 0, and the LP coefficients that can be predicted are shown in light gray. For the colour format YUV422, the LP coefficient associated with position 5 is predicted from position 1 in Figure 5. if MBDCMode is equal to 1, irrespective of the value of MBLPMode, and the LP coefficient associated with position 4 (indicated by crosshatch) can be predicted from both top and left.

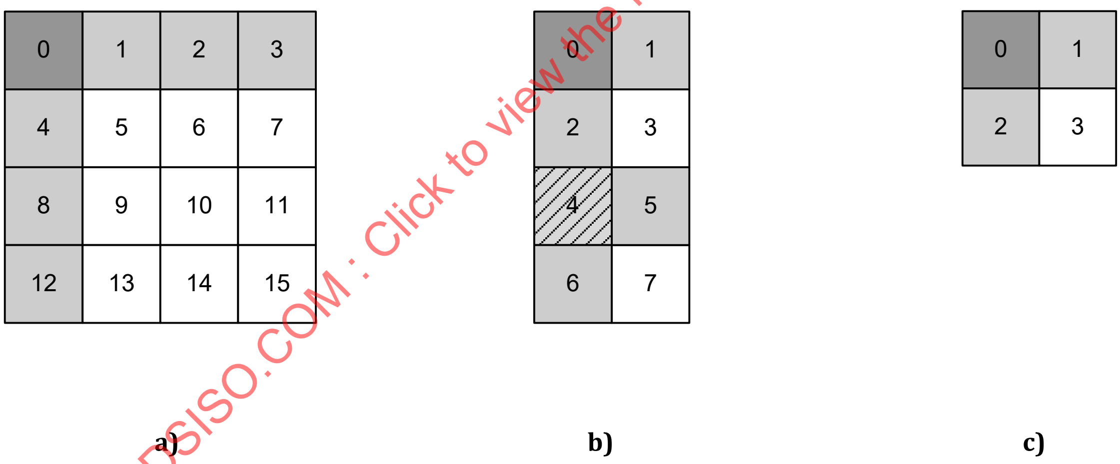


Figure 5 — DC and LP coefficients in (a) 4×4, (b) 422 chroma, and (c) 420 chroma block

9.6.2.2 LPPredictionGeneral( )

Inputs to this process are the variable MBDCMode for the current macroblock, as well as the values MbDCLP[MBx][MBy][i][j], for each colour component i of the current macroblock, and index j referencing the LP transform coefficients, indexed in raster scan order.

NOTE The values MbDCLP[MBx][MBy][i][j] come from the LP coefficient remapping process of subclause 9.5.2.

Outputs to this process are the values MbDCLP[MBx][MBy][i][j], for each colour component i of the current macroblock.

The LP prediction process proceeds as in Table 131.

Table 131 — Pseudocode for function LPPredictionGeneral( )

LPPredictionGeneral( ) {	**Reference**
CalcLPPredMode( )	9.6.2.3
LPCoefficientPrediction( )	9.6.2.4
UpdateLPPredictionVariables( )	9.6.2.5
}	

9.6.2.3 Low-pass prediction direction computation

Inputs to this process are the variables MBDCMode and MBQPIndexLP[MBx][MBy].

Output to this process is the value of MBLPMode. The possible values of MBLPMode are as follows: the value 0 represents prediction from the left macroblock, the value 1 represents prediction from the top macroblock, and the value 2 represents no prediction.

The LP prediction direction process proceeds as in Table 132.

Table 132 — Pseudocode for function CalcLPPredMode( )

CalcLPPredMode( ) {	**Reference**
if (MBDCMode = = 0 && MBQPIndexLP[MBx][MBy] = = MBQPIndexLP[MBx−1][MBy])	
MBLPMode = 0	
else if (MBDCMode = = 1 && MBQPIndexLP[MBx][MBy] = = MBQPIndexLP[MBx][MBy−1])	
MBLPMode = 1	
else	
MBLPMode = 2	
}	

9.6.2.4 Low-pass coefficient prediction

Inputs to this process are: the variable MBLPMode representing the LP prediction direction; the variables MBx and MBy, which index the current macroblock in the image; and the variables PredDCLP[MBx][MBy][i][j].

Outputs to this process are the updated values MbDCLP[MBx][MBy][i][j], for each colour component i of the current macroblock, and j an index referencing the LP transform coefficients, indexed in raster scan order.

The LP coefficient prediction process proceeds as in Table 133.

Table 133 — Pseudocode for function LPCoefficientPrediction( )

LPCoefficientPrediction( ) {	**Reference**
for (i = 0; i < NumComponents; i++) {	
if (i = = 0 | | ((INTERNAL_CLR_FMT != YUV420) && (INTERNAL_CLR_FMT != YUV422))) {	
if (MBLPMode = = 0) {	
MbDCLP[MBx][MBy][i][4] += PredDCLP[MBx−1][MBy][i][4]	
MbDCLP[MBx][MBy][i][8] += PredDCLP[MBx−1][MBy][i][5]	
MbDCLP[MBx][MBy][i][12] += PredDCLP[MBx−1][MBy][i][6]	
} else if (MBLPMode = = 1) {	
MbDCLP[MBx][MBy][i][1] += PredDCLP[MBx][MBy−1][i][1]	
MbDCLP[MBx][MBy][i][2] += PredDCLP[MBx][MBy−1][i][2]	
MbDCLP[MBx][MBy][i][3] += PredDCLP[MBx][MBy−1][i][3]	
}	
} else if (INTERNAL_CLR_FMT = = YUV420) { /* i is equal to 1 or 2 here */	
if (MBLPMode = = 0) /* Prediction from left */	
MbDCLP[MBx][MBy][i][2] += PredDCLP[MBx−1][MBy][i][2]	

LPCoefficientPrediction( ) {	Reference
else if (MBLPMode = = 1) /* Prediction from top */	
MbDCLP[MBx][MBy][i][1] += PredDCLP[MBx][MBy−1][i][1]	
} else if (INTERNAL_CLR_FMT = = YUV422) /* i is equal to 1 or 2 here */	
if (MBLPMode = = 0) { /* Prediction from left */	
MbDCLP[MBx][MBy][i][4] += PredDCLP[MBx−1][MBy][i][4]	
MbDCLP[MBx][MBy][i][2] += PredDCLP[MBx−1][MBy][i][2]	
MbDCLP[MBx][MBy][i][6] += PredDCLP[MBx−1][MBy][i][6]	
} else if (MBLPMode = = 1) { /* Prediction from top */	
MbDCLP[MBx][MBy][i][4] += PredDCLP[MBx][MBy−1][i][4]	
MbDCLP[MBx][MBy][i][1] += PredDCLP[MBx][MBy−1][i][5]	
MbDCLP[MBx][MBy][i][5] += MbDCLP[MBx][MBy][i][1] /* In this line, prediction occurs using the *current* macroblock's data */	
} else if (MBDCMode = = 1)	
MbDCLP[MBx][MBy][i][5] += MbDCLP[MBx][MBy][i][1] /* When the colour format is YUV422, MBLPMode is equal to 2 (no prediction), and MBDCMode is equal to 1 (prediction from the top), the LP coefficient associated with j=5 is predicted from that for j=1 */	
}	
}	

9.6.2.5 Update of low-pass prediction variables

Inputs to this process are the variables MBx and MBy, which index the current macroblock in the image.

Outputs of this process are the variables PredDCLP[MBx][MBy][i][j], for each colour component i, and selected LP indexes j.

The update of LP prediction variables process proceeds as in Table 134.

Table 134 — Pseudocode for function UpdateLPPredictionVariables( )

UpdateLPPredictionVariables( ) {	Reference
for (i = 0; i < NumComponents; i++) {	
if (i = = 0 | | ((INTERNAL_CLR_FMT != YUV420) && (INTERNAL_CLR_FMT != YUV422))) {	
PredDCLP[MBx][MBy][i][1] = MbDCLP[MBx][MBy][i][1]	
PredDCLP[MBx][MBy][i][2] = MbDCLP[MBx][MBy][i][2]	
PredDCLP[MBx][MBy][i][3] = MbDCLP[MBx][MBy][i][3]	
PredDCLP[MBx][MBy][i][4] = MbDCLP[MBx][MBy][i][4]	
PredDCLP[MBx][MBy][i][5] = MbDCLP[MBx][MBy][i][8]	
PredDCLP[MBx][MBy][i][6] = MbDCLP[MBx][MBy][i][12]	
} else if (INTERNAL_CLR_FMT = = YUV420) {	
PredDCLP[MBx][MBy][i][1] = MbDCLP[MBx][MBy][i][1]	
PredDCLP[MBx][MBy][i][2] = MbDCLP[MBx][MBy][i][2]	
} else if (INTERNAL_CLR_FMT = = YUV422) {	
PredDCLP[MBx][MBy][i][1] = MbDCLP[MBx][MBy][i][1]	
PredDCLP[MBx][MBy][i][2] = MbDCLP[MBx][MBy][i][2]	
PredDCLP[MBx][MBy][i][4] = MbDCLP[MBx][MBy][i][4]	
PredDCLP[MBx][MBy][i][5] = MbDCLP[MBx][MBy][i][5]	
PredDCLP[MBx][MBy][i][6] = MbDCLP[MBx][MBy][i][6]	
}	
}	
}	

9.6.3 High-pass prediction

9.6.3.1 Overview of high-pass prediction

There are two prediction processes involving HP coefficients: the HP prediction direction process, and the HP prediction process. The process that computes HP prediction direction also sets the HP direction variable MBHPMode which determines the initial HP inverse scanning order. Therefore, the process that computes HP prediction direction shall be executed before the HP coefficient inverse scanning process. The HP prediction process is executed after the HP coefficient remapping process.

Information in the LP transform coefficients is used to compute a simple metric to determine the *orientation* of prediction of HP coefficients associated with each macroblock. Three modes are defined for the prediction of the HP coefficients of the outer transform. The same mode is used for all blocks within a macroblock for which in-macroblock prediction is possible. For blocks that have no valid reference within the macroblock, null prediction is used. The three modes are:

— Prediction from left when MBHPMode is equal to 0

— Prediction from top when MBHPMode is equal to 1

— No prediction when MBHPMode is equal to 2

Prediction from left is shown in Figure 6. Prediction from top is similar, with the pattern of arrows transposed to point downwards.

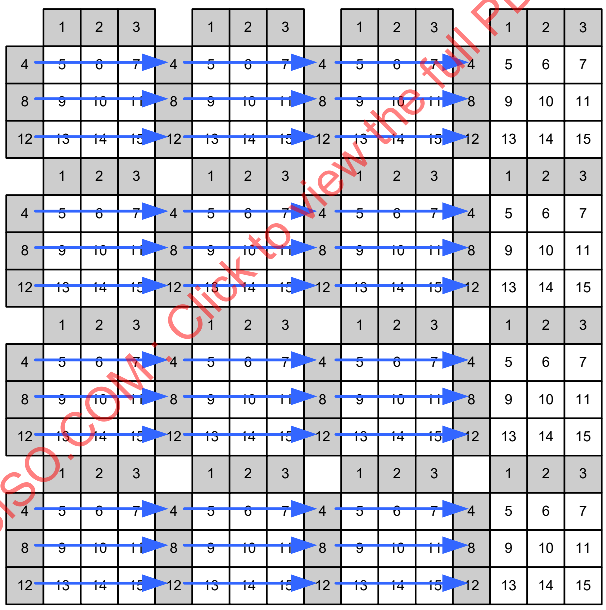


Figure 6 — HP prediction from left

NOTE In the implementation of a decoder, the only information that needs to be available for future use is 1 DC + 6 LP = 7 coefficients per macroblock component (fewer for YUV420 / YUV422 chrominance). Therefore, at most for YUV444, 21 coefficients need to be cached per macroblock. Further, the coefficients used for prediction from left can be discarded after the next macroblock is predicted. For YUV444, therefore, it is necessary to only cache 12 coefficients per macroblock for use in the next row of macroblocks. More state is required on the encoder side: the HP coefficients must be maintained throughout this process, as the encoding of HP coefficients is dependent on the encoding of LP coefficients.

9.6.3.2 High-pass prediction direction computation

Inputs to this process are the variables MBx and MBy, indexing the location of the current macroblock in the image.

Output of this process is the variable MBHPMode for the current macroblock.

The HP prediction process proceeds as in Table 135.

Table 135 — Pseudocode for function CalcHPPredMode( )

CalcHPPredMode( ) {	**Reference**
iStrHor = Abs(MbDCLP[MBx][MBy][0][1]) + Abs(MbDCLP[MBx][MBy][0][2]) + Abs(MbDCLP[MBx][MBy][0][3])	
iStrVer = Abs(MbDCLP[MBx][MBy][0][4]) + Abs(MbDCLP[MBx][MBy][0][8]) + Abs(MbDCLP[MBx][MBy][0][12])	
if ((INTERNAL_CLR_FMT != YONLY) && (INTERNAL_CLR_FMT != NCOMPONENT)) {	
for (i = 1; i <= 2; i++) {	
iStrHor += Abs(MbDCLP[MBx][MBy][i][1])	
if (INTERNAL_CLR_FMT = = YUV420)	
iStrVer += Abs(MbDCLP[MBx][MBy][i][2])	
else if (INTERNAL_CLR_FMT = = YUV422) {	
iStrVer += Abs(MbDCLP[MBx][MBy][i][2]) + Abs(MbDCLP[MBx][MBy][i][6])	
iStrHor += Abs(MbDCLP[MBx][MBy][i][5])	
} else	
iStrVer += Abs(MbDCLP[MBx][MBy][i][4])	
}	
}	
iOrWt = 4	
if (iStrHor * iOrWt < iStrVer)	
MBHPMode = 0 /* predict from left */	
else if (iStrVer * iOrWt < iStrHor)	
MBHPMode = 1 /* predict from top */	
else	
MBHPMode = 2 /* no prediction */	
}	

9.6.3.3 High-pass prediction

Inputs to this process are: the variable MBHPMode, which indicates the HP prediction direction; the values MBx and MBy, which index the current macroblock in the image; and the values MBBuffer[MBx][MBy][i][j], which hold the HP transform coefficients obtained from the HP coefficient remapping process of subclause 9.5.3. The outputs of this process are the updated values MBBuffer[MBx][MBy][i][k] of HP transform coefficients.

The HP prediction process proceeds as in Table 136.

Table 136 — Pseudocode for function HPCoefficientPrediction( )

HPCoefficientPrediction( ) {	Reference
if (INTERNAL_CLR_FMT = = YUV420) | | (INTERNAL_CLR_FMT = = YUV422)	
iComponents = 1	
else	
iComponents = NumComponents	
for (i = 0; i < iComponents; i++) {	
if (MBHPMode = = 0) {	
blkId[ ] = {1,2,3,5,6,7,9,10,11,13,14,15}	
for (j = 0; j < 12; j++) {	
MBBuffer[MBx][MBy][i][16*blkId[j] + 4] += MBBuffer[MBx][MBy][i][16*(blkId[j] − 1) + 4]	
MBBuffer[MBx][MBy][i][16*blkId[j] + 8] += MBBuffer[MBx][MBy][i][16*(blkId[j] − 1) + 8]	
MBBuffer[MBx][MBy][i][16*blkId[j] + 12] += MBBuffer[MBx][MBy][i][16*(blkId[j] − 1) + 12]	
}	
} else if (MBHPMode = = 1) {	
blkId[ ] = {4,5,6,7,8,9,10,11,12,13,14,15}	
for (j = 0; j < 12; j++) {	
MBBuffer[MBx][MBy][i][16*blkId[j] + 1] += MBBuffer[MBx][MBy][i][16*(blkId[j] − 4) + 1]	
MBBuffer[MBx][MBy][i][16*blkId[j] + 2] += MBBuffer[MBx][MBy][i][16*(blkId[j] − 4) + 2]	
MBBuffer[MBx][MBy][i][16*blkId[j] + 3] += MBBuffer[MBx][MBy][i][16*(blkId[j] − 4) + 3]	
}	
}	
}	
if (INTERNAL_CLR_FMT = = YUV420) {	
for (i = 1; i <= 2; i++) {	
if (MBHPMode = = 0) {	
blkId[ ] = {1,3}	
for (j = 0; j < 2; j++) {	
MBBuffer[MBx][MBy][i][16*blkId[j] + 4] += MBBuffer[MBx][MBy][i][16*(blkId[j] − 1) + 4]	
MBBuffer[MBx][MBy][i][16*blkId[j] + 8] += MBBuffer[MBx][MBy][i][16*(blkId[j] − 1) + 8]	
MBBuffer[MBx][MBy][i][16*blkId[j] + 12] += MBBuffer[MBx][MBy][i][16*(blkId[j] − 1) + 12]	
}	
} else if (MBHPMode = = 1) {	
blkId[ ] = {2,3}	
for (j = 0; j < 2; j++) {	
MBBuffer[MBx][MBy][i][16*blkId[j] + 1] += MBBuffer[MBx][MBy][i][16*(blkId[j] − 2) + 1]	
MBBuffer[MBx][MBy][i][16*blkId[j] + 2] += MBBuffer[MBx][MBy][i][16*(blkId[j] − 2) + 2]	
MBBuffer[MBx][MBy][i][16*blkId[j] + 3] += MBBuffer[MBx][MBy][i][16*(blkId[j] − 2) + 3]	
}	
}	
}	
} else if (INTERNAL_CLR_FMT = = YUV422) {	
for (i = 1; i <= 2; i++) {	
if (MBHPMode = = 0) {	
blkId[ ] = {1,3,5,7}	
for (j = 0; j < 4; j++) {	

HPCoefficientPrediction( ) {	Reference
MBBuffer[MBx][MBy][i][16*blkId[j] + 4] += MBBuffer[MBx][MBy][i][16*(blkId[j] − 1) + 4]	
MBBuffer[MBx][MBy][i][16*blkId[j] + 8] += MBBuffer[MBx][MBy][i][16*(blkId[j] − 1) + 8]	
MBBuffer[MBx][MBy][i][16*blkId[j] + 12] += MBBuffer[MBx][MBy][i][16*(blkId[j] − 1) + 12]	
}	
} else if (MBHPMode = = 1) {	
blkId[ ] = {2,4,6,3,5,7}	
for (j = 0; j < 6; j++) {	
MBBuffer[MBx][MBy][i][16*blkId[j] + 1] += MBBuffer[MBx][MBy][i][16*(blkId[j] − 2) + 1]	
MBBuffer[MBx][MBy][i][16*blkId[j] + 2] += MBBuffer[MBx][MBy][i][16*(blkId[j] − 2) + 2]	
MBBuffer[MBx][MBy][i][16*blkId[j] + 3] += MBBuffer[MBx][MBy][i][16*(blkId[j] − 2) + 3]	
}	
}	
}	
}	
}	

9.7 Derivation of quantization parameters

9.7.1 Derivation of DC quantization parameters

9.7.1.1 Image plane level derivation of DC quantization parameters

This process derives the array DCQuantParam[i] of image plane level DC quantization parameters for each colour component i, if these parameters are specified at the image plane level; otherwise, it does nothing, and it is expected that the tile-level derivation process will derive the array DCQuantParam[i], for each tile in the image plane.

The image plane level derivation process of DC quantization parameters proceeds as in Table 137.

Table 137 — Pseudocode for function ImagePlaneDCQP( )

ImagePlaneDCQP( ) {	Reference
if (DC_IMAGE_PLANE_UNIFORM_FLAG = = TRUE)	
AssignDCQuantizationParameters( )	9.7.1.3
}	

NOTE If DC_IMAGE_PLANE_UNIFORM_FLAG is equal to FALSE, the quantization parameters are specified at tile level.

9.7.1.2 Tile level derivation of DC quantization parameters

If these parameters are specified at the tile level, this process derives the array DCQuantParam[i] of tile-level DC quantization parameters for each colour component i. Otherwise, it does nothing and it is assumed that the image plane level derivation process already derived the array DCQuantParam[i].

The tile-level derivation process of DC quantization parameters proceeds as in Table 138.

Table 138 — Pseudocode for function TileLevelDCQP( )

TileLevelDCQP( ) {	Reference
if (DC_IMAGE_PLANE_UNIFORM_FLAG = = FALSE)	
AssignDCQuantizationParameters( )	9.7.1.3
}	

NOTE If DC_IMAGE_PLANE_UNIFORM_FLAG is equal to TRUE, quantization parameters were set at the image plane level.

9.7.1.3 Assignment of DC quantization parameters

The assignment process of DC quantization parameters proceeds as in Table 139.

Table 139 — Pseudocode for function AssignDCQuantizationParameters( )

AssignDCQuantizationParameters( ) {	**Reference**
if (COMPONENT_MODE = = UNIFORM)	
for (i = 0; i <=NumComponents−1; i++)	
DCQuantParam[i] = DC_QUANT	
else if (COMPONENT_MODE = = SEPARATE) {	
DCQuantParam[0] = DC_QUANT_LUMA	
for (i = 1; i <=NumComponents−1; i++)	
DCQuantParam[i] = DC_QUANT_CHROMA	
} else if (COMPONENT_MODE = = INDEPENDENT)	
for (i = 0; i <=NumComponents−1; i++)	
DCQuantParam[i] = DC_QUANT_CH[i]	
}	

9.7.2 Derivation of low-pass quantization parameters

9.7.2.1 Image plane level derivation of low-pass quantization parameters

This process derives the values LPQuantParam[i][j] of image plane level LP quantization parameters, for each colour component i, and each allowable index j (for image plane level LP quantization parameters, j can only take the value 0). These values are derived if LP_IMAGE_PLANE_UNIFORM_FLAG is equal to TRUE, which indicates that these quantization parameters are specified at the image plane level; otherwise, it does nothing, and it is expected that the tile-level derivation process will derive the array LPQuantParam[i][j], for each tile in the image plane.

The image plane level derivation process of LP quantization parameters proceeds as in Table 140.

Table 140 — Pseudocode for function ImagePlaneLPQP( )

ImagePlaneLPQP( ) {	**Reference**
if (LP_IMAGE_PLANE_UNIFORM_FLAG)	
AssignLPQuantizationParameters( )	9.7.2.3
}	

NOTE 1 When LP_IMAGE_PLANE_UNIFORM_FLAG is equal to TRUE, NumLPQPs is always equal to 1.

NOTE 2 If LP_IMAGE_PLANE_UNIFORM_FLAG is equal to FALSE, quantization parameters are set at the tile level.

9.7.2.2 Tile level derivation of low-pass quantization parameters

This process derives the values LPQuantParam[i][j] of tile-level LP quantization parameters, for each colour component i, and each allowable index j (ranging from 0 to NumLPQPs−1). These values are derived if these quantization parameters are specified at the tile level; otherwise, it does nothing, and the image plane level derivation process derived the array LPQuantParam[i][j], used for each tile in the current image plane.

The tile-level derivation process of LP quantization parameters proceeds as in Table 141.

Table 141 — Pseudocode for function TileLevelLPQP( )

TileLevelLPQP( ) {	Reference
if ((LP_IMAGE_PLANE_UNIFORM_FLAG != TRUE) && (USE_DC_QP_FLAG = = TRUE)) {	
NumLPQPs = 1	
for (i = 0; i <=NumComponents−1; i++)	
LPQuantParam[i][0] = DCQuantParam[i]	
} else if (LP_IMAGE_PLANE_UNIFORM_FLAG != TRUE)	
AssignLPQuantizationParameters( )	9.7.2.3
}	

NOTE When LP_IMAGE_PLANE_UNIFORM_FLAG is equal to TRUE, nothing is done in this function, because quantization parameters were set at image plane level.

9.7.2.3 Assignment of low-pass quantization parameters

The assignment process of LP quantization parameters proceeds as in Table 142.

Table 142 — Pseudocode for function AssignLPQuantizationParameters( )

AssignLPQuantizationParameters( ) {	Reference
for (j = 0; j < NumLPQPs; j++) {	
if (COMPONENT_MODE = = UNIFORM)	
for (i = 0; i <=NumComponents−1; i++)	
LPQuantParam[i][j] = LP_QUANT[j]	
else if (COMPONENT_MODE = = SEPARATE) {	
LPQuantParam[0][j] = LP_QUANT_LUMA[j]	
for (i = 1; i <=NumComponents−1; i++)	
LPQuantParam[i][j] = LP_QUANT_CHROMA[j]	
} else if (COMPONENT_MODE = = INDEPENDENT)	
for (i = 0; i <=NumComponents−1; i++)	
LPQuantParam[i][j] = LP_QUANT_CH[i][j]	
}	
}	

9.7.3 Derivation of high-pass quantization parameters

9.7.3.1 Image plane level derivation of high-pass quantization parameters

This process derives the values HPQuantParam[i][j] of image plane level HP quantization parameters, for each colour component i, and each allowable index j. These values are derived if these quantization parameters are specified at the image plane level; otherwise, it does nothing, and it is expected that the tile-level derivation process will derive the array HPQuantParam[i][j], for each tile in the image plane.

The image plane level derivation process of HP quantization parameters proceeds as in Table 143.

Table 143 — Pseudocode for function ImagePlaneHPQP( )

ImagePlaneHPQP( ) {	Reference
if (HP_IMAGE_PLANE_UNIFORM_FLAG)	
AssignHPQuantizationParameters( )	9.7.3.3
}	

NOTE 1 When HP_IMAGE_PLANE_UNIFORM_FLAG is equal to TRUE, NumHPQPs is always equal to 1.

NOTE 2 When HP_IMAGE_PLANE_UNIFORM_FLAG is equal to FALSE, quantization parameters are set at the tile level.

9.7.3.2 Tile level derivation of high-pass quantization parameters

This process derives the values HPQuantParam[i][j] of tile-level HP quantization parameters, for each colour component i, and each allowable index j (ranging from 0 to NumHPQPs−1). These values are derived if these quantization parameters are specified at the tile level; otherwise, this process has no effect, and it is expected that the image plane level derivation process derived the array HPQuantParam[i][j], used for each tile in the current image plane.

The tile-level derivation process of HP quantization parameters proceeds as in Table 144.

Table 144 — Pseudocode for function TileLevelHPQP( )

TileLevelHPQP( ) {	**Reference**
if ((HP_IMAGE_PLANE_UNIFORM_FLAG != TRUE) && (USE_LP_QP_FLAG = = TRUE)) {	
NumHPQPs = NumLPQPs	
for (i = 0; i <=NumComponents−1; i++)	
for (j = 0; j <=NumLPQPs; j++)	
HPQuantParam[i][j] = LPQuantParam[i][j]	
} else if (HP_IMAGE_PLANE_UNIFORM_FLAG != TRUE)	
AssignHPQuantizationParameters( )	9.7.3.3
}	

NOTE When HP_IMAGE_PLANE_UNIFORM_FLAG is equal to TRUE, nothing is done in this function, because quantization parameters were set at image plane level.

9.7.3.3 Assignment of high-pass quantization parameters

The assignment process of HP quantization parameters proceeds as in Table 145.

Table 145 — Pseudocode for function AssignHPQuantizationParameters( )

AssignHPQuantizationParameters( ) {	**Reference**
for (j = 0; j < NumHPQPs; j++) {	
if (COMPONENT_MODE = = UNIFORM)	
for (i = 0; i <=NumComponents−1; i++)	
HPQuantParam[i][j] = HP_QUANT[j]	
else if (COMPONENT_MODE = = SEPARATE) {	
HPQuantParam[0][j] = HP_QUANT_LUMA[j]	
for (i = 1; i <=NumComponents−1; i++)	
HPQuantParam[i][j] = HP_QUANT_CHROMA[j]	
} else if (COMPONENT_MODE = = INDEPENDENT)	
for (i = 0; i <=NumComponents−1; i++)	
HPQuantParam[i][j] = HP_QUANT_CH[i][j]	
}	
}	

9.8 Dequantization

9.8.1 Dequantization of DC coefficients

This process is applied for the DC coefficients of a macroblock, for all colour components.

Input to this process is the array DCQuantParam[i] of DC quantization parameters for each colour component i; the array MbDCLP[MBx][MBy][i][0] of DC transform coefficients for each colour component i of the current macroblock; and the variables MBx and MBy which identify the current macroblock in the image.

This process uses the local variable array iQuantScalingFactor[i], holding the scaling factor, for each colour component i.

Output of this process is an array of scaled DC transform coefficients MbDCLP[MBx][MBy][i][0], for each colour component i.

The dequantization process for DC coefficients proceeds as in Table 146.

Table 146 — Pseudocode for function DequantizeDCCoefficients( )

DequantizeDCCoefficients( ) {	**Reference**
for (i = 0; i < NumComponents; i++) {	
if (i = = 0)	
iQuantScalingFactor[i] = QuantMap(DCQuantParam[i], 1)	9.8.4
else	
iQuantScalingFactor[i] = QuantMap(DCQuantParam[i], 0)	9.8.4
MbDCLP[MBx][MBy][i][0] = MbDCLP[MBx][MBy][i][0] * iQuantScalingFactor[i]	
}	
}	

9.8.2 Dequantization of low-pass coefficients

This process is applied for the LP coefficients of an entire macroblock, for all colour components.

Inputs to this process are the values LPQuantParam[i][j] of LP quantization parameters, for each colour component i and index j; the quantization parameter index MBQPIndexLP[MBx][MBy]; the array MbDCLP[MBx][MBy][i][j] of LP transform coefficients, with i representing the colour component, and j referencing the LP transform coefficients, indexed in raster scan order.

Output of this process is an array of scaled LP transform coefficients MbDCLP[MBx][MBy][i][j], for each colour component i and index j ranging from 1 to 15, referencing the respective LP transform coefficient.

The dequantization process for LP coefficients proceeds as in Table 147.

Table 147 — Pseudocode for function DequantizeLPCoefficients( )

DequantizeLPCoefficients( ) {	**Reference**
for (i = 0; i < NumComponents; i++) {	
k = MBQPIndexLP[MBx][MBy]	
valueQP[i] = LPQuantParam[i][k]	
if (i = = 0)	
iQuantScalingFactor[i] = QuantMap(valueQP[i], 1)	9.8.4
else	
iQuantScalingFactor[i] = QuantMap(valueQP[i], 0)	9.8.4
if (i = = 0) /* Luma Component */	
for (j = 1; j <= 15; j++)	
MbDCLP[MBx][MBy][i][j] = MbDCLP[MBx][MBy][i][j] * iQuantScalingFactor[i]	
else if ((INTERNAL_CLR_FMT != YUV422) && (INTERNAL_CLR_FMT != YUV420))	
for (j = 1; j <= 15; j++)	
MbDCLP[MBx][MBy][i][j] = MbDCLP[MBx][MBy][i][j] * iQuantScalingFactor[i]	
else if (INTERNAL_CLR_FMT = = YUV422)	
for (j = 1; j <= 7; j++)	
MbDCLP[MBx][MBy][i][j] = MbDCLP[MBx][MBy][i][j] * iQuantScalingFactor[i]	
else /* if (INTERNAL_CLR_FMT = = YUV420) */	
for (j = 1; j <= 3; j++)	
MbDCLP[MBx][MBy][i][j] = MbDCLP[MBx][MBy][i][j] * iQuantScalingFactor[i]	
}	
}	

9.8.3 Dequantization of high-pass coefficients

This process is applied for the HP coefficients of an entire macroblock, for all colour components.

Inputs to this process are the values HPQuantParam[i][j] of HP quantization parameters, for each colour component i and index j; the quantization parameter index MBQPIndexHP[MBx][MBy]; the array MBBuffer[MBx][MBy][i][j] of HP transform coefficients, where i and j are indices, with i representing the colour component, and j ranging from 1 to 255.

Output of this process is an array of scaled HP transform coefficients MBBuffer[MBx][MBy][i][j], for each colour component i and index j ranging from 1 to 255, referencing the respective HP transform coefficient.

The dequantization process for HP coefficients proceeds as in Table 148.

Table 148 — Pseudocode for function DequantizeHPCoefficients( )

DequantizeHPCoefficients( ) {	**Reference**
for (i = 0; i < NumComponents; i++) {	
k = MBQPIndexHP[MBx][MBy]	
valueQP[i] = HPQuantParam[i][k]	
iQuantScalingFactor[i] = QuantMap(valueQP[i], 1)	9.8.4
if (i = = 0) /* Luma Component */	
for (blkIndex = 0; blkIndex <= 15; blkIndex++)	
for (j = 1; j <= 15; j++)	
MBBuffer[MBx][MBy][i][16*blkIndex + j] = MBBuffer[MBx][MBy][i][16*blkIndex + j] * iQuantScalingFactor[i]	
else if ((INTERNAL_CLR_FMT != YUV422) && (INTERNAL_CLR_FMT != YUV420))	
for (blkIndex = 0; blkIndex <= 15; blkIndex++)	
for (j = 1; j <= 15; j++)	
MBBuffer[MBx][MBy][i][16*blkIndex + j] = MBBuffer[MBx][MBy][i][16*blkIndex + j] * iQuantScalingFactor[i]	
else if (INTERNAL_CLR_FMT = = YUV422)	
for (blkIndex = 0; blkIndex <= 7; blkIndex++)	
for (j = 1; j <= 15; j++)	
MBBuffer[MBx][MBy][i][16*blkIndex + j] = MBBuffer[MBx][MBy][i][16*blkIndex + j] * iQuantScalingFactor[i]	
else /* if (INTERNAL_CLR_FMT = = YUV420) */	
for (blkIndex = 0; blkIndex <= 3; blkIndex++)	
for (j = 1; j <= 15; j++)	
MBBuffer[MBx][MBy][i][16*blkIndex + j] = MBBuffer[MBx][MBy][i][16*blkIndex + j] * iQuantScalingFactor[i]	
}	
}	

9.8.4 QuantMap( )

The function QuantMap is used above to compute the scaling parameters based on the parsed syntax elements QP. The pseudocode for this function is specified in Table 149.

Table 149 — Pseudocode for function QuantMap( )

QuantMap(iQP, iScaledShift) {	**Reference**
if (0 = = iQP)	
iQuantScalingFactorResult = 1	
else if (!SCALED_FLAG) {	
iNotScaledShift = −2	
if (iQP < 32) {	
iMan = (iQP + 3) >> 2	
iExp = 0	
} else if (iQP < 48) {	
iMan = (16 + (iQP % 16) + 1) >> 1	
iExp = (iQP>>4) + iNotScaledShift	
} else {	
iMan = 16 + (iQP % 16)	
iExp = (iQP>>4) −1 + iNotScaledShift	
}	
iQuantScalingFactorResult = iMan << iExp	
} else { /* SCALED_FLAG is TRUE, but not (0 = = iQP) */	
if (iQP < 16) {	
iMan = iQP	
iExp = iScaledShift	
} else {	
iMan = 16 + (iQP % 16)	
iExp = ((iQP >> 4) − 1) + iScaledShift	
}	
iQuantScalingFactorResult = iMan << iExp	
}	
return iQuantScalingFactorResult	
}	

NOTE The input parameter iScaledShift takes the value of either 0 or 1, dependent on the component and band. When SCALED_FLAG is equal to TRUE, the quantization scaling factor value can be modified by a power of 2. See the NOTE in subclause 9.9.2.

9.9 Sample reconstruction

9.9.1 Overview

Inputs to this process are the values MbDCLP[MBx][MBy][i][j] and MBBuffer[MBx][MBy][i][j] for the entire image plane, and the syntax element OVERLAP_MODE.

Outputs of this process are the decoded samples for the image plane.

The sample reconstruction process proceeds as in Table 150.

Table 150 — Pseudocode for function SampleReconstruction( )

SampleReconstruction( ) {	**Reference**
FirstLevelInverseTransform( )	9.9.2
if (OVERLAP_MODE = = 2)	
FirstLevelOverlapFiltering( )	9.9.3
SecondLevelCoefficientCombination( )	9.9.4
SecondLevelInverseTransform( )	9.9.5
if (OVERLAP_MODE != 0)	
SecondLevelOverlapFiltering( )	9.9.6
}	

NOTE Because the first-level overlap filtering process in general involves interaction with adjacent macroblocks, the first-level transform process must be complete for these adjacent macroblocks, prior to the overlap filtering process being invoked. This precedence relationship also holds between the second level processes: the second level's transform process must be complete for the adjacent macroblocks prior to the second level overlap filtering process.

9.9.2 First level inverse transform

Inputs to this process are the values MbDCLP[MBx][MBy][i][j] for the entire colour component.

Outputs to this process are the modified values MbDCLP[MBx][MBy][i][j] for the current macroblock.

The first-level inverse transform process is specified as in Table 151.

Table 151 — Pseudocode for function FirstLevelInverseTransform( )

FirstLevelInverseTransform( ) {	**Reference**
for (i = 0; i < NumComponents; i++)	
for (MBy = 0; MBy < MBHeight; MBy++)	
for (MBx = 0; MBx < MBWidth; MBx++)	
if (i = = 0) | | ((INTERNAL_CLR_FMT != YUV420) && (INTERNAL_CLR_FMT != YUV422)) {	
ICT4x4(MbDCLP[MBx][MBy][i][ ])	9.9.7.1
if ((i > 0) && SCALED_FLAG)	
for (j = 0; j <= 15; j++)	
MbDCLP[MBx][MBy][i][j] = 2 * MbDCLP[MBx][MBy][i][j]	
} else if (INTERNAL_CLR_FMT = = YUV420) {	
T2x2h(MbDCLP[MBx][MBy][i][ ], 0)	9.9.7.2
arrayLocal[ ] = {MbDCLP[MBx][MBy][i][1], MbDCLP[MBx][MBy][i][2]}	
InvPermute2pt(arrayLocal[ ])	9.9.7.6
MbDCLP[MBx][MBy][i][1] = arrayLocal[0]	
MbDCLP[MBx][MBy][i][2] = arrayLocal[1]	
if ((i > 0) && SCALED_FLAG)	
for (j = 0; j <= 3; j++)	
MbDCLP[MBx][MBy][i][j] = 2 * MbDCLP[MBx][MBy][i][j]	
} else if (INTERNAL_CLR_FMT = = YUV422) {	
arrayLocal[ ] = {MbDCLP[MBx][MBy][i][0], MbDCLP[MBx][MBy][i][4]}	
T2pt(arrayLocal[ ])	9.9.7.7
MbDCLP[MBx][MBy][i][0] = arrayLocal[0]	
MbDCLP[MBx][MBy][i][4] = arrayLocal[1]	
T2x2h(MbDCLP[MBx][MBy][i][ ], 0)	9.9.7.2
arrayLocal[ ] = {MbDCLP[MBx][MBy][i][1], MbDCLP[MBx][MBy][i][2]}	
InvPermute2pt(arrayLocal[ ])	9.9.7.6
MbDCLP[MBx][MBy][i][1] = arrayLocal[0]	
MbDCLP[MBx][MBy][i][2] = arrayLocal[1]	
arrayLocal[ ] = {MbDCLP[MBx][MBy][i][4], MbDCLP[MBx][MBy][i][6], MbDCLP[MBx][MBy][i][5], MbDCLP[MBx][MBy][i][7]}	

FirstLevelInverseTransform( ) {	Reference
T2x2h(arrayLocal[ ], 0)	9.9.7.2
MbDCLP[MBx][MBy][i][4] = arrayLocal[0]	
MbDCLP[MBx][MBy][i][6] = arrayLocal[1]	
MbDCLP[MBx][MBy][i][5] = arrayLocal[2]	
MbDCLP[MBx][MBy][i][7] = arrayLocal[3]	
arrayLocal[ ] = {MbDCLP[MBx][MBy][i][5], MbDCLP[MBx][MBy][i][6]}	
InvPermute2pt(arrayLocal[ ])	9.9.7.6
MbDCLP[MBx][MBy][i][5] = arrayLocal[0]	
MbDCLP[MBx][MBy][i][6] = arrayLocal[1]	
if ((i > 0) && SCALED_FLAG)	
for (j = 0; j <= 7; j++)	
MbDCLP[MBx][MBy][i][j] = 2 * MbDCLP[MBx][MBy][i][j]	
}	
}	

NOTE The purpose of the multiplication by 2 for Chroma components, in circumstances where scaling is involved, is to re-normalize the chroma with respect to the Y component. Due to possible conversion from RGB to YUV during encoding, the U and V components may have a numerical range that has increased by one bit. If SCALED_FLAG is equal to TRUE, the dynamic range of the (DC and LP) U and V component values could potentially grow beyond 16 bits, due to the numerical range expansion associated with the two levels of transform on the encode side (for the DC and LP coefficients). Therefore, the quantization parameter for these chroma components is set to half the value used for luma components. The coefficients are scaled by this factor of two at the end of the first-level transform process.

9.9.3 First level overlap filtering

9.9.3.1 Overview

NOTE The process specification below formalizes the geometric nature of the overlap filtering process. The various cases are described below:

— interior: At every point where four 4×4 blocks meet in a corner, the 4×4 overlap filter process is applied to the 4×4 block straddling these four blocks evenly (i.e., overlapping with a 2×2 corner of each block). When HARD_TILING_FLAG is equal to FALSE, the 4×4 overlap filter process is applied across tile boundaries as well.

— top and bottom two rows: Along both the top two sample rows and the bottom two sample rows, a 4-point overlap filter process is applied evenly across adjacent block boundaries (overlapping with a 1×2 strip of each block). When HARD_TILING_FLAG is equal to TRUE, the 4-point overlap filter process is applied across the top 2 rows and bottom 2 rows of tiles as well.

— right-most and left-most two columns: Along both the left-most two sample columns and the right-most two sample columns, a 4-point overlap filter process is applied evenly across adjacent block boundaries (overlapping with a 2×1 strip of each block). When HARD_TILING_FLAG is equal to TRUE, the 4-point overlap filter process is applied across the top 2 columns and bottom 2 columns of tiles as well.

— four corners: Over the corner 2×2 blocks in the top-left, top-right, bottom-left and bottom-right, a 4-point overlap filter process is applied in a raster scan order (top-left, top-right, bottom-left, then bottom-right). When HARD_TILING_FLAG is equal to TRUE, the 4-point overlap filter process is applied to the four corners of each tile as well.

Additionally, when INTERNAL_CLR_FMT is equal to either YUV422 or YUV420, alternate processes are considered for the chroma components for each of the above cases. These case are described below:

— chroma interior: The 2×2 overlap filter process is applied to the 2×2 block straddling interior block boundaries. When HARD_TILING_FLAG is equal to FALSE, the 2×2 overlap filter process is applied across tile boundaries as well.

— chroma top and bottom rows: Along both the top sample row and the bottom sample row, a 2-point overlap filter process is applied evenly across adjacent block boundaries. When HARD_TILING_FLAG is equal to TRUE, the 2-point overlap filter process is applied across the top row and bottom row of tiles as well.

— chroma right-most and left-most columns: Along both the left-most sample column and the right-most sample column, a 2-point overlap filter process is applied evenly across adjacent block boundaries. When HARD_TILING_FLAG is equal to TRUE, the 2-point overlap filter process is applied across the top column and bottom column of tiles as well.

— chroma four corners: Over the corner 1×1 blocks in the top-left, top-right, bottom-left and bottom-right, an adjacent coefficient residual process (specified in subclause 9.9.3.3 and subclause 9.9.3.4) is applied. When HARD_TILING_FLAG is equal to TRUE, the adjacent coefficient residual process is applied to the four corners of each tile as well.

Inputs to this process are the values MbDCLP[MBx][MBy][i][j] for the entire image plane, and the values MBWidth and MBHeight.

Outputs to this process are the modified values MbDCLP[MBx][MBy][i][j] for the current macroblock.

The first-level overlap filtering process is specified in Table 152.

Table 152 — Pseudocode for function FirstLevelOverlapFiltering( )

FirstLevelOverlapFiltering( ) {	**Reference**
for (i = 0; i < NumComponents; i++)	
if ((i = =0) | | ((INTERNAL_CLR_FMT != YUV420) &&(INTERNAL_CLR_FMT != YUV422)))	
FirstLevelOverlapFilteringPrimary(i)	9.9.3.2
else if (INTERNAL_CLR_FMT = = YUV422)	
FirstLevelOverlapFiltering422(i)	9.9.3.3
else if (INTERNAL_CLR_FMT = = YUV420)	
FirstLevelOverlapFiltering420(i)	9.9.3.4
}	

9.9.3.2 FirstLevelOverlapFilteringPrimary( )

Pseudocode for the function FirstLevelOverlapFilteringPrimary( ) is specified in Table 153.

Table 153 — Pseudocode for function FirstLevelOverlapFilteringPrimary( )

FirstLevelOverlapFilteringPrimary(i) {	**Reference**
for (Ty = 0; Ty <= (NumTileRows − 1); Ty++) {	
for (Tx = 0; Tx <= (NumTileCols − 1); Tx++) {	
for (y = TopMBIndexOfTile[Ty]; y <= (TopMBIndexOfTile[Ty + 1] − 2); y++)	
for (x = LeftMBIndexOfTile[Tx]; x <= (LeftMBIndexOfTile[Tx + 1] − 2); x++)	
FirstLevelCallOverlapPostFilter4x4(i, x, y)	9.9.3.5
if ((Tx = = 0) | | (HARD_TILING_FLAG = = TRUE)) { /* Left edge */	
x = LeftMBIndexOfTile[Tx]	
for (y = TopMBIndexOfTile[Ty]; y <= TopMBIndexOfTile[Ty + 1] − 2; y++) {	
arrayLocal[ ] = {MbDCLP[x][y][i][8], MbDCLP[x][y][i][12], MbDCLP[x][y+1][i][0], MbDCLP[x][y+1][i][4]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
MbDCLP[x][y][i][8] = arrayLocal[0]	
MbDCLP[x][y][i][12] = arrayLocal[1]	
MbDCLP[x][y+1][i][0] = arrayLocal[2]	
MbDCLP[x][y+1][i][4] = arrayLocal[3]	
arrayLocal[ ] = {MbDCLP[x][y][i][9], MbDCLP[x][y][i][13], MbDCLP[x][y+1][i][1], MbDCLP[x][y+1][i][5]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
MbDCLP[x][y][i][9] = arrayLocal[0]	
MbDCLP[x][y][i][13] = arrayLocal[1]	
MbDCLP[x][y+1][i][1] = arrayLocal[2]	
MbDCLP[x][y+1][i][5] = arrayLocal[3]	
}	
}	
if ((Ty = = 0) | | (HARD_TILING_FLAG = = TRUE)) { /* Top edge */	
y = TopMBIndexOfTile[Ty]	
for (x = LeftMBIndexOfTile[Tx]; x <= LeftMBIndexOfTile[Tx + 1] − 2; x++) {	
arrayLocal[ ] = {MbDCLP[x][y][i][2], MbDCLP[x][y][i][3], MbDCLP[x+1][y][i][0], MbDCLP[x+1][y][i][1]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
MbDCLP[x][y][i][2] = arrayLocal[0]	

FirstLevelOverlapFilteringPrimary(i) {	**Reference**
MbDCLP[x][y][i][3] = arrayLocal[1]	
MbDCLP[x+1][y][i][0] = arrayLocal[2]	
MbDCLP[x+1][y][i][1] = arrayLocal[3]	
arrayLocal[ ] = {MbDCLP[x][y][i][6], MbDCLP[x][y][i][7], MbDCLP[x+1][y][i][4], MbDCLP[x+1][y][i][5]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
MbDCLP[x][y][i][6] = arrayLocal[0]	
MbDCLP[x][y][i][7] = arrayLocal[1]	
MbDCLP[x+1][y][i][4] = arrayLocal[2]	
MbDCLP[x+1][y][i][5] = arrayLocal[3]	
}	
}	
if ((Tx = = NumTileCols − 1) | | (HARD_TILING_FLAG = = TRUE)) { /* Right edge */	
x = LeftMBIndexOfTile[Tx + 1] − 1	
for (y = TopMBIndexOfTile[Ty]; y <= TopMBIndexOfTile[Ty + 1] − 2; y++) {	
arrayLocal[ ] = {MbDCLP[x][y][i][10], MbDCLP[x][y][i][14], MbDCLP[x][y+1][i][2], MbDCLP[x][y+1][i][6]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
MbDCLP[x][y][i][10] = arrayLocal[0]	
MbDCLP[x][y][i][14] = arrayLocal[1]	
MbDCLP[x][y+1][i][2] = arrayLocal[2]	
MbDCLP[x][y+1][i][6] = arrayLocal[3]	
arrayLocal[ ] = {MbDCLP[x][y][i][11], MbDCLP[x][y][i][15], MbDCLP[x][y+1][i][3], MbDCLP[x][y+1][i][7]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
MbDCLP[x][y][i][11] = arrayLocal[0]	
MbDCLP[x][y][i][15] = arrayLocal[1]	
MbDCLP[x][y+1][i][3] = arrayLocal[2]	
MbDCLP[x][y+1][i][7] = arrayLocal[3]	
}	
}	
if ((Ty = = NumTileRows − 1) | | (HARD_TILING_FLAG = = TRUE)) { /* Bottom edge */	
y = TopMBIndexOfTile[Ty + 1] − 1	
for (x = LeftMBIndexOfTile[Tx]; x <= LeftMBIndexOfTile[Tx + 1] − 2; x++) {	
arrayLocal[ ] = {MbDCLP[x][y][i][10], MbDCLP[x][y][i][11], MbDCLP[x+1][y][i][8], MbDCLP[x+1][y][i][9]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
MbDCLP[x][y][i][10] = arrayLocal[0]	
MbDCLP[x][y][i][11] = arrayLocal[1]	
MbDCLP[x+1][y][i][8] = arrayLocal[2]	
MbDCLP[x+1][y][i][9] = arrayLocal[3]	
arrayLocal[ ] = {MbDCLP[x][y][i][14], MbDCLP[x][y][i][15], MbDCLP[x+1][y][i][12], MbDCLP[x+1][y][i][13]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
MbDCLP[x][y][i][14] = arrayLocal[0]	
MbDCLP[x][y][i][15] = arrayLocal[1]	
MbDCLP[x+1][y][i][12] = arrayLocal[2]	
MbDCLP[x+1][y][i][13] = arrayLocal[3]	
}	
}	
if (((Tx = = 0) && (Ty = = 0)) | | (HARD_TILING_FLAG = = TRUE)) { /* Top left corner */	
x = LeftMBIndexOfTile[Tx]	
y = TopMBIndexOfTile[Ty]	
arrayLocal[ ] = {MbDCLP[x][y][i][0], MbDCLP[x][y][i][1], MbDCLP[x][y][i][4], MbDCLP[x][y][i][5]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
MbDCLP[x][y][i][0] = arrayLocal[0]	
MbDCLP[x][y][i][1] = arrayLocal[1]	
MbDCLP[x][y][i][4] = arrayLocal[2]	
MbDCLP[x][y][i][5] = arrayLocal[3]	

FirstLevelOverlapFilteringPrimary(i) {	Reference
}	
if (((Tx = = NumTileCols − 1) && (Ty = = 0)) | | (HARD_TILING_FLAG = = TRUE)) { /* Top right corner */	
x = LeftMBIndexOfTile[Tx + 1] − 1	
y = TopMBIndexOfTile[Ty]	
arrayLocal[ ] = {MbDCLP[x][y][i][2], MbDCLP[x][y][i][3], MbDCLP[x][y][i][6], MbDCLP[x][y][i][7]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
MbDCLP[x][y][i][2] = arrayLocal[0]	
MbDCLP[x][y][i][3] = arrayLocal[1]	
MbDCLP[x][y][i][6] = arrayLocal[2]	
MbDCLP[x][y][i][7] = arrayLocal[3]	
}	
if (((Tx = = 0) && (Ty = = NumTileRows − 1)) | | (HARD_TILING_FLAG = = TRUE)) { /* Bottom left corner */	
x = LeftMBIndexOfTile[Tx]	
y = TopMBIndexOfTile[Ty + 1] − 1	
arrayLocal[ ] = {MbDCLP[x][y][i][8], MbDCLP[x][y][i][9], MbDCLP[x][y][i][12], MbDCLP[x][y][i][13]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
MbDCLP[x][y][i][8] = arrayLocal[0]	
MbDCLP[x][y][i][9] = arrayLocal[1]	
MbDCLP[x][y][i][12] = arrayLocal[2]	
MbDCLP[x][y][i][13] = arrayLocal[3]	
}	
if (((Tx = = NumTileCols − 1) && (Ty = = NumTileRows − 1)) | | (HARD_TILING_FLAG = = TRUE)) { /* Bottom right corner */	
x = LeftMBIndexOfTile[Tx + 1] − 1	
y = TopMBIndexOfTile[Ty + 1] − 1	
arrayLocal[ ] = {MbDCLP[x][y][i][10], MbDCLP[x][y][i][11], MbDCLP[x][y][i][14], MbDCLP[x][y][i][15]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
MbDCLP[x][y][i][10] = arrayLocal[0]	
MbDCLP[x][y][i][11] = arrayLocal[1]	
MbDCLP[x][y][i][14] = arrayLocal[2]	
MbDCLP[x][y][i][15] = arrayLocal[3]	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Tx != NumTileCols − 1)) { /* Right across for soft tiles */	
x = LeftMBIndexOfTile[Tx + 1] − 1	
for (y = TopMBIndexOfTile[Ty]; y <= (TopMBIndexOfTile[Ty + 1] − 2); y++)	
FirstLevelCallOverlapPostFilter4x4(i, x, y)	9.9.3.5
}	
if ((HARD_TILING_FLAG = = FALSE) && (Ty != NumTileRows − 1)) { /* Bottom across for soft tiles */	
y = TopMBIndexOfTile[Ty + 1] − 1	
for (x = LeftMBIndexOfTile[Tx]; x <= (LeftMBIndexOfTile[Tx + 1] − 2); x++)	
FirstLevelCallOverlapPostFilter4x4(i, x, y)	9.9.3.5
}	
if ((HARD_TILING_FLAG = = FALSE) && (Tx != NumTileCols − 1) && (Ty != NumTileRows − 1)) { /* Bottom across for soft tiles */	
x = LeftMBIndexOfTile[Tx + 1] − 1	
y = TopMBIndexOfTile[Ty + 1] − 1	
FirstLevelCallOverlapPostFilter4x4(i, x, y)	9.9.3.5
}	
if ((HARD_TILING_FLAG = = FALSE) && (Tx = = 0) && (Ty != NumTileRows − 1)) { /* Left edge for soft tiles */	
x = LeftMBIndexOfTile[Tx]	
y = TopMBIndexOfTile[Ty + 1] − 1	
arrayLocal[ ] = {MbDCLP[x][y][i][8], MbDCLP[x][y][i][12],	

FirstLevelOverlapFilteringPrimary(i) {	Reference
MbDCLP[x][y+1][i][0], MbDCLP[x][y+1][i][4]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
MbDCLP[x][y][i][8] = arrayLocal[0]	
MbDCLP[x][y][i][12] = arrayLocal[1]	
MbDCLP[x][y+1][i][0] = arrayLocal[2]	
MbDCLP[x][y+1][i][4] = arrayLocal[3]	
arrayLocal[ ] = {MbDCLP[x][y][i][9], MbDCLP[x][y][i][13], MbDCLP[x][y+1][i][1], MbDCLP[x][y+1][i][5]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
MbDCLP[x][y][i][9] = arrayLocal[0]	
MbDCLP[x][y][i][13] = arrayLocal[1]	
MbDCLP[x][y+1][i][1] = arrayLocal[2]	
MbDCLP[x][y+1][i][5] = arrayLocal[3]	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Tx != NumTileCols − 1) && (Ty = = 0)) { /* Top edge for soft tiles */	
x = LeftMBIndexOfTile[Tx + 1] − 1	
y = TopMBIndexOfTile[Ty]	
arrayLocal[ ] = {MbDCLP[x][y][i][2], MbDCLP[x][y][i][3], MbDCLP[x+1][y][i][0], MbDCLP[x+1][y][i][1]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
MbDCLP[x][y][i][2] = arrayLocal[0]	
MbDCLP[x][y][i][3] = arrayLocal[1]	
MbDCLP[x+1][y][i][0] = arrayLocal[2]	
MbDCLP[x+1][y][i][1] = arrayLocal[3]	
arrayLocal[ ] = {MbDCLP[x][y][i][6], MbDCLP[x][y][i][7], MbDCLP[x+1][y][i][4], MbDCLP[x+1][y][i][5]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
MbDCLP[x][y][i][6] = arrayLocal[0]	
MbDCLP[x][y][i][7] = arrayLocal[1]	
MbDCLP[x+1][y][i][4] = arrayLocal[2]	
MbDCLP[x+1][y][i][5] = arrayLocal[3]	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Tx = = NumTileCols − 1) && (Ty != NumTileRows − 1)) { /* Right edge for soft tiles */	
x = LeftMBIndexOfTile[Tx + 1] − 1	
y = TopMBIndexOfTile[Ty + 1] − 1	
arrayLocal[ ] = {MbDCLP[x][y][i][10], MbDCLP[x][y][i][14], MbDCLP[x][y+1][i][2], MbDCLP[x][y+1][i][6]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
MbDCLP[x][y][i][10] = arrayLocal[0]	
MbDCLP[x][y][i][14] = arrayLocal[1]	
MbDCLP[x][y+1][i][2] = arrayLocal[2]	
MbDCLP[x][y+1][i][6] = arrayLocal[3]	
arrayLocal[ ] = {MbDCLP[x][y][i][11], MbDCLP[x][y][i][15], MbDCLP[x][y+1][i][4], MbDCLP[x][y+1][i][7]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
MbDCLP[x][y][i][11] = arrayLocal[0]	
MbDCLP[x][y][i][15] = arrayLocal[1]	
MbDCLP[x][y+1][i][4] = arrayLocal[2]	
MbDCLP[x][y+1][i][7] = arrayLocal[3]	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Tx != NumTileCols − 1) && (Ty = = NumTileRows − 1)) { /* Bottom edge for soft tiles */	
x = LeftMBIndexOfTile[Tx + 1] − 1	
y = TopMBIndexOfTile[Ty + 1] − 1	
arrayLocal[ ] = {MbDCLP[x][y][i][10], MbDCLP[x][y][i][11], MbDCLP[x+1][y][i][8], MbDCLP[x+1][y][i][9]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
MbDCLP[x][y][i][10] = arrayLocal[0]	

FirstLevelOverlapFilteringPrimary(i) {	Reference
MbDCLP[x][y][i][11] = arrayLocal[1]	
MbDCLP[x+1][y][i][8] = arrayLocal[2]	
MbDCLP[x+1][y][i][9] = arrayLocal[3]	
arrayLocal[ ] = {MbDCLP[x][y][i][14], MbDCLP[x][y][i][15], MbDCLP[x+1][y][i][12], MbDCLP[x+1][y][i][13]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
MbDCLP[x][y][i][14] = arrayLocal[0]	
MbDCLP[x][y][i][15] = arrayLocal[1]	
MbDCLP[x+1][y][i][12] = arrayLocal[2]	
MbDCLP[x+1][y][i][13] = arrayLocal[3]	
}	
}	
}	
}	

9.9.3.3 FirstLevelOverlapFiltering422( )

Pseudocode for the function FirstLevelOverlapFiltering422( ) is specified in Table 154.

Table 154 — Pseudocode for function FirstLevelOverlapFiltering422( )

FirstLevelOverlapFiltering422(i) {	Reference
for (Ty = 0; Ty <= (NumTileRows – 1); Ty ++) {	
if ((Ty = = 0) | | (HARD_TILING_FLAG = = TRUE)) { /* Top edge */	
/* OverlapPostFilter1 */	
y = TopMBIndexOfTile[Ty]	
MbDCLP[LeftMBIndexOfTile[0]][y][i][0] –= MbDCLP[LeftMBIndexOfTile[0]][y][i][1] /* Upper left corner difference */	
MbDCLP[LeftMBIndexOfTile[NumTileCols] – 1][y][i][1] –= MbDCLP[LeftMBIndexOfTile[NumTileCols] – 1][y][i][0] /* Upper right corner difference */	
if (HARD_TILING_FLAG = = TRUE)	
for (Tx = 1; Tx < (NumTileCols – 1); Tx++) {	
MbDCLP[LeftMBIndexOfTile[Tx]][y][i][0] –= MbDCLP[LeftMBIndexOfTile[Tx]][y][i][1]	
MbDCLP[LeftMBIndexOfTile[Tx] – 1][y][i][1] –= MbDCLP[LeftMBIndexOfTile[Tx] – 1][y][i][0]	
}	
}	
if ((Ty = = NumTileRows – 1) | | (HARD_TILING_FLAG = = TRUE)) { /* Bottom edge */	
/* OverlapPostFilter1 */	
y = TopMBIndexOfTile[Ty + 1] – 1	
MbDCLP[LeftMBIndexOfTile[0]][y][i][6] –= MbDCLP[LeftMBIndexOfTile[0]][y][i][7] /* Bottom left corner difference */	
MbDCLP[LeftMBIndexOfTile[NumTileCols] – 1][y][i][7] –= MbDCLP[LeftMBIndexOfTile[NumTileCols] – 1][y][i][6] /* Bottom right corner difference */	
if (HARD_TILING_FLAG = = TRUE)	
for (Tx = 1; Tx < (NumTileCols – 1); Tx++) {	
MbDCLP[LeftMBIndexOfTile[Tx]][y][i][6] –= MbDCLP[LeftMBIndexOfTile[Tx]][y][i][7]	
MbDCLP[LeftMBIndexOfTile[Tx] – 1][y][i][7] –= MbDCLP[LeftMBIndexOfTile[Tx] – 1][y][i][6]	
}	
}	
for (Tx = 0; Tx <= (NumTileCols – 1); Tx++) {	
for (y = TopMBIndexOfTile[Ty]; y <= (TopMBIndexOfTile[Ty + 1] – 1); y++)	
for (x = LeftMBIndexOfTile[Tx]; x <= (LeftMBIndexOfTile[Tx + 1] – 2); x++) {	
arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x+1][y][i][2], MbDCLP[x][y][i][5], MbDCLP[x+1][y][i][4]}	
OverlapPostFilter2x2(arrayLocal[ ])	9.9.8.3
MbDCLP[x][y][i][3] = arrayLocal[0]	
MbDCLP[x+1][y][i][2] = arrayLocal[1]	
MbDCLP[x][y][i][5] = arrayLocal[2]	

<table>
<tr><th>FirstLevelOverlapFiltering422(i) {</th><th>Reference</th></tr>
<tr><td>MbDCLP[x+1][y][i][4] = arrayLocal[3]</td><td></td></tr>
<tr><td>if (y != (TopMBIndexOfTile[Ty + 1] − 1)) {</td><td></td></tr>
<tr><td>arrayLocal[ ] = {MbDCLP[x][y][i][7], MbDCLP[x+1][y][i][6],
MbDCLP[x][y+1][i][1], MbDCLP[x+1][y+1][i][0]}</td><td></td></tr>
<tr><td>OverlapPostFilter2x2(arrayLocal[ ])</td><td>9.9.8.3</td></tr>
<tr><td>MbDCLP[x][y][i][7] = arrayLocal[0]</td><td></td></tr>
<tr><td>MbDCLP[x+1][y][i][6] = arrayLocal[1]</td><td></td></tr>
<tr><td>MbDCLP[x][y+1][i][1] = arrayLocal[2]</td><td></td></tr>
<tr><td>MbDCLP[x+1][y+1][i][0] = arrayLocal[3]</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>if ((Tx = = 0) | | (HARD_TILING_FLAG = = TRUE)) { /* Left edge */</td><td></td></tr>
<tr><td>x = LeftMBIndexOfTile[Tx]</td><td></td></tr>
<tr><td>for (y = TopMBIndexOfTile[Ty]; y <= (TopMBIndexOfTile[Ty + 1] − 1); y++) {</td><td></td></tr>
<tr><td>arrayLocal[ ] = {MbDCLP[x][y][i][2], MbDCLP[x][y][i][4]}</td><td></td></tr>
<tr><td>OverlapPostFilter2(arrayLocal[ ])</td><td>9.9.8.4</td></tr>
<tr><td>MbDCLP[x][y][i][2] = arrayLocal[0]</td><td></td></tr>
<tr><td>MbDCLP[x][y][i][4] = arrayLocal[1]</td><td></td></tr>
<tr><td>if (y != (TopMBIndexOfTile[Ty + 1] − 1)) {</td><td></td></tr>
<tr><td>arrayLocal[ ] = {MbDCLP[x][y][i][6], MbDCLP[x][y+1][i][0]}</td><td></td></tr>
<tr><td>OverlapPostFilter2(arrayLocal[ ])</td><td>9.9.8.4</td></tr>
<tr><td>MbDCLP[x][y][i][6] = arrayLocal[0]</td><td></td></tr>
<tr><td>MbDCLP[x][y+1][i][0] = arrayLocal[1]</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>if ((Tx = = NumTileCols − 1) | | (HARD_TILING_FLAG = = TRUE)) { /* Right edge */</td><td></td></tr>
<tr><td>x = LeftMBIndexOfTile[Tx + 1] − 1</td><td></td></tr>
<tr><td>for (y = TopMBIndexOfTile[Ty]; y <= (TopMBIndexOfTile[Ty + 1] − 1); y++) {</td><td></td></tr>
<tr><td>arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x][y][i][5]}</td><td></td></tr>
<tr><td>OverlapPostFilter2(arrayLocal[ ])</td><td>9.9.8.4</td></tr>
<tr><td>MbDCLP[x][y][i][3] = arrayLocal[0]</td><td></td></tr>
<tr><td>MbDCLP[x][y][i][5] = arrayLocal[1]</td><td></td></tr>
<tr><td>if (y != (TopMBIndexOfTile[Ty + 1] − 1)) {</td><td></td></tr>
<tr><td>arrayLocal[ ] = {MbDCLP[x][y][i][7], MbDCLP[x][y+1][i][1]}</td><td></td></tr>
<tr><td>OverlapPostFilter2(arrayLocal[ ])</td><td>9.9.8.4</td></tr>
<tr><td>MbDCLP[x][y][i][7] = arrayLocal[0]</td><td></td></tr>
<tr><td>MbDCLP[x][y+1][i][1] = arrayLocal[1]</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>if ((Ty = = 0) | | (HARD_TILING_FLAG = = TRUE)) { /* Top edge */</td><td></td></tr>
<tr><td>y = TopMBIndexOfTile[Ty]</td><td></td></tr>
<tr><td>for (x = LeftMBIndexOfTile[Tx]; x <= (LeftMBIndexOfTile[Tx + 1] − 2); x++) {</td><td></td></tr>
<tr><td>arrayLocal[ ] = {MbDCLP[x][y][i][1], MbDCLP[x+1][y][i][0]}</td><td></td></tr>
<tr><td>OverlapPostFilter2(arrayLocal[ ])</td><td>9.9.8.4</td></tr>
<tr><td>MbDCLP[x][y][i][1] = arrayLocal[0]</td><td></td></tr>
<tr><td>MbDCLP[x+1][y][i][0] = arrayLocal[1]</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>if ((Ty = = NumTileRows − 1) | | (HARD_TILING_FLAG = = TRUE)) { /* Bottom edge */</td><td></td></tr>
<tr><td>y = TopMBIndexOfTile[Ty + 1] − 1</td><td></td></tr>
<tr><td>for (x = LeftMBIndexOfTile[Tx]; x <= (LeftMBIndexOfTile[Tx + 1] − 2); x++) {</td><td></td></tr>
<tr><td>arrayLocal[ ] = {MbDCLP[x][y][i][7], MbDCLP[x+1][y][i][6]}</td><td></td></tr>
<tr><td>OverlapPostFilter2(arrayLocal[ ])</td><td>9.9.8.4</td></tr>
<tr><td>MbDCLP[x][y][i][7] = arrayLocal[0]</td><td></td></tr>
<tr><td>MbDCLP[x+1][y][i][6] = arrayLocal[1]</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>if ((HARD_TILING_FLAG = = FALSE) && (Tx != NumTileCols − 1)) {</td><td></td></tr>
</table>

FirstLevelOverlapFiltering422(i) {	Reference
/* Right across for soft tiles */	
x = LeftMBIndexOfTile[Tx + 1] − 1	
for (y = TopMBIndexOfTile[Ty]; y <= (TopMBIndexOfTile[Ty + 1] − 2); y++) {	
arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x+1][y][i][2], MbDCLP[x][y][i][5], MbDCLP[x+1][y][i][4]}	
OverlapPostFilter2x2(arrayLocal[ ])	9.9.8.3
MbDCLP[x][y][i][3] = arrayLocal[0]	
MbDCLP[x+1][y][i][2] = arrayLocal[1]	
MbDCLP[x][y][i][5] = arrayLocal[2]	
MbDCLP[x+1][y][i][4] = arrayLocal[3]	
arrayLocal[ ] = {MbDCLP[x][y][i][7], MbDCLP[x+1][y][i][6], MbDCLP[x][y+1][i][1], MbDCLP[x+1][y+1][i][0]}	
OverlapPostFilter2x2(arrayLocal[ ])	9.9.8.3
MbDCLP[x][y][i][7] = arrayLocal[0]	
MbDCLP[x+1][y][i][6] = arrayLocal[1]	
MbDCLP[x][y+1][i][1] = arrayLocal[2]	
MbDCLP[x+1][y+1][i][0] = arrayLocal[3]	
}	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Ty != NumTileRows − 1)) { /* Bottom across for soft tiles */	
y = TopMBIndexOfTile[Ty + 1] − 1	
for (x = LeftMBIndexOfTile[Tx]; x <= (LeftMBIndexOfTile[Tx + 1] − 2); x++) {	
arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x+1][y][i][2], MbDCLP[x][y][i][5], MbDCLP[x+1][y][i][4]}	
OverlapPostFilter2x2(arrayLocal[ ])	9.9.8.3
MbDCLP[x][y][i][3] = arrayLocal[0]	
MbDCLP[x+1][y][i][2] = arrayLocal[1]	
MbDCLP[x][y][i][5] = arrayLocal[2]	
MbDCLP[x+1][y][i][4] = arrayLocal[3]	
arrayLocal[ ] = {MbDCLP[x][y][i][7], MbDCLP[x+1][y][i][6], MbDCLP[x][y+1][i][1], MbDCLP[x+1][y+1][i][0]}	
OverlapPostFilter2x2(arrayLocal[ ])	9.9.8.3
MbDCLP[x][y][i][7] = arrayLocal[0]	
MbDCLP[x+1][y][i][6] = arrayLocal[1]	
MbDCLP[x][y+1][i][1] = arrayLocal[2]	
MbDCLP[x+1][y+1][i][0] = arrayLocal[3]	
}	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Tx != NumTileCols − 1) && (Ty != NumTileRows − 1)) {	
x = LeftMBIndexOfTile[Tx + 1] − 1	
y = TopMBIndexOfTile[Ty + 1] − 1	
arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x+1][y][i][2], MbDCLP[x][y][i][5], MbDCLP[x+1][y][i][4]}	
OverlapPostFilter2x2(arrayLocal[ ])	9.9.8.3
MbDCLP[x][y][i][3] = arrayLocal[0]	
MbDCLP[x+1][y][i][2] = arrayLocal[1]	
MbDCLP[x][y][i][5] = arrayLocal[2]	
MbDCLP[x+1][y][i][4] = arrayLocal[3]	
arrayLocal[ ] = {MbDCLP[x][y][i][7], MbDCLP[x+1][y][i][6], MbDCLP[x][y+1][i][1], MbDCLP[x+1][y+1][i][0]}	
OverlapPostFilter2x2(arrayLocal[ ])	9.9.8.3
MbDCLP[x][y][i][7] = arrayLocal[0]	
MbDCLP[x+1][y][i][6] = arrayLocal[1]	
MbDCLP[x][y+1][i][1] = arrayLocal[2]	
MbDCLP[x+1][y+1][i][0] = arrayLocal[3]	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Tx = = 0) && (Ty != NumTileRows − 1)) {	
x = LeftMBIndexOfTile[Tx]	

FirstLevelOverlapFiltering422(i) {	**Reference**
y = TopMBIndexOfTile[Ty + 1] – 1	
arrayLocal[ ] = {MbDCLP[x][y][i][2], MbDCLP[x][y][i][4]}	
OverlapPostFilter2(arrayLocal[ ])	9.9.8.4
MbDCLP[x][y][i][2] = arrayLocal[0]	
MbDCLP[x][y][i][4] = arrayLocal[1]	
arrayLocal[ ] = {MbDCLP[x][y][i][6], MbDCLP[x][y+1][i][0]}	
OverlapPostFilter2(arrayLocal[ ])	9.9.8.4
MbDCLP[x][y][i][6] = arrayLocal[0]	
MbDCLP[x][y+1][i][0] = arrayLocal[1]	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Tx = = NumTileCols – 1) && (Ty != NumTileRows – 1)) {	
x = LeftMBIndexOfTile[Tx + 1] – 1	
y = TopMBIndexOfTile[Ty + 1] – 1	
arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x][y][i][5]}	
OverlapPostFilter2(arrayLocal[ ])	9.9.8.4
MbDCLP[x][y][i][3] = arrayLocal[0]	
MbDCLP[x][y][i][5] = arrayLocal[1]	
arrayLocal[ ] = {MbDCLP[x][y][i][7], MbDCLP[x][y+1][i][1]}	
OverlapPostFilter2(arrayLocal[ ])	9.9.8.4
MbDCLP[x][y][i][7] = arrayLocal[0]	
MbDCLP[x][y+1][i][1] = arrayLocal[1]	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Tx != NumTileCols – 1) && (Ty = = 0)) {	
x = LeftMBIndexOfTile[Tx + 1] – 1	
y = TopMBIndexOfTile[Ty]	
arrayLocal[ ] = {MbDCLP[x][y][i][1], MbDCLP[x+1][y][i][0]}	
OverlapPostFilter2(arrayLocal[ ])	9.9.8.4
MbDCLP[x][y][i][1] = arrayLocal[0]	
MbDCLP[x+1][y][i][0] = arrayLocal[1]	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Tx != NumTileCols – 1) && (Ty = = NumTileRows – 1)) {	
x = LeftMBIndexOfTile[Tx + 1] – 1	
y = TopMBIndexOfTile[Ty + 1] – 1	
arrayLocal[ ] = {MbDCLP[x][y][i][7], MbDCLP[x+1][y][i][6]}	
OverlapPostFilter2(arrayLocal[ ])	9.9.8.4
MbDCLP[x][y][i][7] = arrayLocal[0]	
MbDCLP[x+1][y][i][6] = arrayLocal[1]	
}	
}	
if ((Ty = = 0) | | (HARD_TILING_FLAG = = TRUE)) { /*Top edge */	
/* OverlapPostFilter1 */	
y = TopMBIndexOfTile[Ty]	
MbDCLP[LeftMBIndexOfTile[0]][y][i][0] += MbDCLP[LeftMBIndexOfTile[0]][y][i][1] /* Upper left corner addition */	
MbDCLP[LeftMBIndexOfTile[NumTileCols] – 1][y][i][1] += MbDCLP[LeftMBIndexOfTile[NumTileCols] – 1][y][i][0] /* Upper right corner addition */	
if (HARD_TILING_FLAG = = TRUE)	
for (Tx = 1; Tx < (NumTileCols – 1); Tx++) {	
MbDCLP[LeftMBIndexOfTile[Tx]][y][i][0] += MbDCLP[LeftMBIndexOfTile[Tx]][y][i][1]	
MbDCLP[LeftMBIndexOfTile[Tx] – 1][y][i][1] += MbDCLP[LeftMBIndexOfTile[Tx] – 1][y][i][0]	
}	
}	
if ((Ty = = NumTileRows – 1) | | (HARD_TILING_FLAG = = TRUE)) { /* Bottom edge */	
/* OverlapPostFilter1 */	
y = TopMBIndexOfTile[Ty + 1] – 1	
MbDCLP[LeftMBIndexOfTile[0]][y][i][6] += MbDCLP[LeftMBIndexOfTile[0]][y][i][7] /* Bottom left corner addition */	

FirstLevelOverlapFiltering422(i) {	Reference
MbDCLP[LeftMBIndexOfTile[NumTileCols] − 1][y][i][7] += MbDCLP[LeftMBIndexOfTile[NumTileCols] − 1][y][i][6] /* Bottom right corner addition */	
if (HARD_TILING_FLAG = = TRUE)	
for (Tx = 1; Tx < (NumTileCols − 1); Tx++) {	
MbDCLP[LeftMBIndexOfTile[Tx]][y][i][6] += MbDCLP[LeftMBIndexOfTile[Tx]][y][i][7]	
MbDCLP[LeftMBIndexOfTile[Tx] − 1][y][i][7] += MbDCLP[LeftMBIndexOfTile[Tx] − 1][y][i][6]	
}	
}	
}	
}	

9.9.3.4 FirstLevelOverlapFiltering420( )

Pseudocode for the function FirstLevelOverlapFiltering420( ) is specified in Table 155.

Table 155 — Pseudocode for function FirstLevelOverlapFiltering420( )

FirstLevelOverlapFiltering420(i) {	Reference
for (Ty = 0; Ty <= (NumTileRows − 1); Ty ++) {	
if ((Ty = = 0) | | (HARD_TILING_FLAG = = TRUE)) { /* Top edge */	
/* OverlapPostFilter1 */	
y = TopMBIndexOfTile[Ty]	
MbDCLP[LeftMBIndexOfTile[0]][y][i][0] −= MbDCLP[LeftMBIndexOfTile[0]][y][i][1] /* Upper left corner difference */	
MbDCLP[LeftMBIndexOfTile[NumTileCols] − 1][y][i][1] −= MbDCLP[LeftMBIndexOfTile[NumTileCols] − 1][y][i][0] /* Upper right corner difference */	
if (HARD_TILING_FLAG = = TRUE)	
for (Tx = 1; Tx < (NumTileCols − 1); Tx++) {	
MbDCLP[LeftMBIndexOfTile[Tx]][y][i][0] −= MbDCLP[LeftMBIndexOfTile[Tx]][y][i][1]	
MbDCLP[LeftMBIndexOfTile[Tx] − 1][y][i][1] −= MbDCLP[LeftMBIndexOfTile[Tx] − 1][y][i][0]	
}	
}	
if ((Ty = = NumTileRows − 1) | | (HARD_TILING_FLAG = = TRUE)) { /* Bottom edge */	
/* OverlapPostFilter1 */	
y = TopMBIndexOfTile[Ty + 1] − 1	
MbDCLP[LeftMBIndexOfTile[0]][y][i][2] −= MbDCLP[LeftMBIndexOfTile[0]][y][i][3] /* Bottom left corner difference */	
MbDCLP[LeftMBIndexOfTile[NumTileCols] − 1][y][i][3] −= MbDCLP[LeftMBIndexOfTile[NumTileCols] − 1][y][i][2] /* Bottom right corner difference */	
if (HARD_TILING_FLAG = = TRUE)	
for (Tx = 1; Tx < (NumTileCols − 1); Tx++) {	
MbDCLP[LeftMBIndexOfTile[Tx]][y][i][2] −= MbDCLP[LeftMBIndexOfTile[Tx]][y][i][3]	
MbDCLP[LeftMBIndexOfTile[Tx] − 1][y][i][3] −= MbDCLP[LeftMBIndexOfTile[Tx] − 1][y][i][2]	
}	
}	
for (Tx = 0; Tx <= (NumTileCols − 1); Tx++) {	
for (y = TopMBIndexOfTile[Ty]; y <= (TopMBIndexOfTile[Ty + 1] − 2); y++)	
for (x = LeftMBIndexOfTile[Tx]; x <= (LeftMBIndexOfTile[Tx + 1] − 2); x++) {	
arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x+1][y][i][2], MbDCLP[x][y+1][i][1], MbDCLP[x+1][y+1][i][0]}	
OverlapPostFilter2x2(arrayLocal[ ])	9.9.8.3
MbDCLP[x][y][i][3] = arrayLocal[0]	
MbDCLP[x+1][y][i][2] = arrayLocal[1]	
MbDCLP[x][y+1][i][1] = arrayLocal[2]	
MbDCLP[x+1][y+1][i][0] = arrayLocal[3]	
}	
if ((Tx = = 0) | | (HARD_TILING_FLAG = = TRUE)) { /* Left edge */	
x = LeftMBIndexOfTile[Tx]	
for (y = TopMBIndexOfTile[Ty]; y <= (TopMBIndexOfTile[Ty + 1] − 2); y++) {	

FirstLevelOverlapFiltering420(i) {	**Reference**
arrayLocal[ ] = {MbDCLP[x][y][i][2], MbDCLP[x][y+1][i][0]}	
OverlapPostFilter2(arrayLocal[ ])	9.9.8.4
MbDCLP[x][y][i][2] = arrayLocal[0]	
MbDCLP[x][y+1][i][0] = arrayLocal[1]	
}	
}	
if ((Tx = = NumTileCols – 1) | | (HARD_TILING_FLAG = = TRUE)) { /* Right edge */	
x = LeftMBIndexOfTile[Tx + 1] – 1	
for (y = TopMBIndexOfTile[Ty]; y <= (TopMBIndexOfTile[Ty + 1] – 2); y++) {	
arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x][y+1][i][1]}	
OverlapPostFilter2(arrayLocal[ ])	9.9.8.4
MbDCLP[x][y][i][3] = arrayLocal[0]	
MbDCLP[x][y+1][i][1] = arrayLocal[1]	
}	
}	
if ((Ty = = 0) | | (HARD_TILING_FLAG = = TRUE)) { /* Top edge */	
y = TopMBIndexOfTile[Ty]	
for (x = LeftMBIndexOfTile[Tx]; x <= (LeftMBIndexOfTile[Tx + 1] – 2); x++) {	
arrayLocal[ ] = {MbDCLP[x][y][i][1], MbDCLP[x+1][y][i][0]}	
OverlapPostFilter2(arrayLocal[ ])	9.9.8.4
MbDCLP[x][y][i][1] = arrayLocal[0]	
MbDCLP[x+1][y][i][0] = arrayLocal[1]	
}	
}	
if ((Ty = = NumTileRows – 1) | | (HARD_TILING_FLAG = = TRUE)) { /* Bottom edge */	
y = TopMBIndexOfTile[Ty + 1] – 1	
for (x = LeftMBIndexOfTile[Tx]; x <= (LeftMBIndexOfTile[Tx + 1] – 2); x++) {	
arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x+1][y][i][2]}	
OverlapPostFilter2(arrayLocal[ ])	9.9.8.4
MbDCLP[x][y][i][3] = arrayLocal[0]	
MbDCLP[x+1][y][i][2] = arrayLocal[1]	
}	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Tx != NumTileCols – 1)) { /* Right across for soft tiles */	
x = LeftMBIndexOfTile[Tx + 1] – 1	
for (y = TopMBIndexOfTile[Ty]; y <= (TopMBIndexOfTile[Ty + 1] – 2); y++) {	
arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x+1][y][i][2], MbDCLP[x][y+1][i][1], MbDCLP[x+1][y+1][i][0]}	
OverlapPostFilter2x2(arrayLocal[ ])	9.9.8.3
MbDCLP[x][y][i][3] = arrayLocal[0]	
MbDCLP[x+1][y][i][2] = arrayLocal[1]	
MbDCLP[x][y+1][i][1] = arrayLocal[2]	
MbDCLP[x+1][y+1][i][0] = arrayLocal[3]	
}	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Ty != NumTileRows – 1)) { /* Bottom across for soft tiles */	
y = TopMBIndexOfTile[Ty + 1] – 1	
for (x = LeftMBIndexOfTile[Tx]; x <= (LeftMBIndexOfTile[Tx + 1] – 2); x++) {	
arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x+1][y][i][2], MbDCLP[x][y+1][i][1], MbDCLP[x+1][y+1][i][0]}	
OverlapPostFilter2x2(arrayLocal[ ])	9.9.8.3
MbDCLP[x][y][i][3] = arrayLocal[0]	
MbDCLP[x+1][y][i][2] = arrayLocal[1]	
MbDCLP[x][y+1][i][1] = arrayLocal[2]	
MbDCLP[x+1][y+1][i][0] = arrayLocal[3]	
}	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Tx != NumTileCols – 1) && (Ty != NumTileRows – 1)) {	
x = LeftMBIndexOfTile[Tx + 1] – 1	

FirstLevelOverlapFiltering420(i) {	**Reference**
y = TopMBIndexOfTile[Ty + 1] – 1	
arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x+1][y][i][2], MbDCLP[x][y+1][i][1], MbDCLP[x+1][y+1][i][0]}	
OverlapPostFilter2x2(arrayLocal[ ])	9.9.8.3
MbDCLP[x][y][i][3] = arrayLocal[0]	
MbDCLP[x+1][y][i][2] = arrayLocal[1]	
MbDCLP[x][y+1][i][1] = arrayLocal[2]	
MbDCLP[x+1][y+1][i][0] = arrayLocal[3]	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Tx = = 0) && (Ty != NumTileRows – 1)) {	
x = LeftMBIndexOfTile[Tx]	
y = TopMBIndexOfTile[Ty + 1] – 1	
arrayLocal[ ] = {MbDCLP[x][y][i][2], MbDCLP[x][y+1][i][0]}	
OverlapPostFilter2(arrayLocal[ ])	9.9.8.4
MbDCLP[x][y][i][2] = arrayLocal[0]	
MbDCLP[x][y+1][i][0] = arrayLocal[1]	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Tx = = NumTileCols – 1) && (Ty != NumTileRows – 1)) {	
x = LeftMBIndexOfTile[Tx + 1] – 1	
y = TopMBIndexOfTile[Ty + 1] – 1	
arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x][y+1][i][1]}	
OverlapPostFilter2(arrayLocal[ ])	9.9.8.4
MbDCLP[x][y][i][3] = arrayLocal[0]	
MbDCLP[x][y+1][i][1] = arrayLocal[1]	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Tx != NumTileCols – 1) && (Ty = = 0)) {	
x = LeftMBIndexOfTile[Tx + 1] – 1	
y = TopMBIndexOfTile[Ty]	
arrayLocal[ ] = {MbDCLP[x][y][i][1], MbDCLP[x+1][y][i][0]}	
OverlapPostFilter2(arrayLocal[ ])	9.9.8.4
MbDCLP[x][y][i][1] = arrayLocal[0]	
MbDCLP[x+1][y][i][0] = arrayLocal[1]	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Tx != NumTileCols – 1) && (Ty = = NumTileRows – 1)) {	
x = LeftMBIndexOfTile[Tx + 1] – 1	
y = TopMBIndexOfTile[Ty + 1] – 1	
arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x+1][y][i][2]}	
OverlapPostFilter2(arrayLocal[ ])	9.9.8.4
MbDCLP[x][y][i][3] = arrayLocal[0]	
MbDCLP[x+1][y][i][2] = arrayLocal[1]	
}	
}	
if ((Ty = = 0) | | (HARD_TILING_FLAG = = TRUE)) { /*Top edge */	
/* OverlapPostFilter1 */	
y = TopMBIndexOfTile[Ty]	
MbDCLP[LeftMBIndexOfTile[0]][y][i][0] += MbDCLP[LeftMBIndexOfTile[0]][y][i][1] /* Upper left corner addition */	
MbDCLP[LeftMBIndexOfTile[NumTileCols] – 1][y][i][1] += MbDCLP[LeftMBIndexOfTile[NumTileCols] – 1][y][i][0] /* Upper right corner addition */	
if (HARD_TILING_FLAG = = TRUE)	
for (Tx = 1; Tx < (NumTileCols – 1); Tx++) {	
MbDCLP[LeftMBIndexOfTile[Tx]][y][i][0] += MbDCLP[LeftMBIndexOfTile[Tx]][y][i][1]	
MbDCLP[LeftMBIndexOfTile[Tx] – 1][y][i][1] += MbDCLP[LeftMBIndexOfTile[Tx] – 1][y][i][0]	
}	
}	
if ((Ty = = NumTileRows – 1) | | (HARD_TILING_FLAG = = TRUE)) { /* Bottom edge */	
/* OverlapPostFilter1 */	
y = TopMBIndexOfTile[Ty + 1] – 1	
MbDCLP[LeftMBIndexOfTile[0]][y][i][2] += MbDCLP[LeftMBIndexOfTile[0]][y][i][3]	

FirstLevelOverlapFiltering420(i) {	Reference
/* Bottom left corner addition */	
MbDCLP[LeftMBIndexOfTile[NumTileCols] − 1][y][i][3] += MbDCLP[LeftMBIndexOfTile[NumTileCols] − 1][y][i][2] /* Bottom right corner addition */	
if (HARD_TILING_FLAG = = TRUE)	
for (Tx = 1; Tx < (NumTileCols − 1); Tx++) {	
MbDCLP[LeftMBIndexOfTile[Tx]][y][i][2] += MbDCLP[LeftMBIndexOfTile[Tx]][y][i][3]	
MbDCLP[LeftMBIndexOfTile[Tx] − 1][y][i][3] += MbDCLP[LeftMBIndexOfTile[Tx] − 1][y][i][2]	
}	
}	
}	
}	

9.9.3.5 FirstLevelCallOverlapPostFilterx4x4( )

Pseudocode for the function FirstLevelCallOverlapPostFilter4x4( ) is specified in Table 156.

Table 156 — Pseudocode for function FirstLevelCallOverlapPostFilter4x4( )

FirstLevelCallOverlapPostFilter4x4(i, x, y) {	Reference
arrayLocal[ ] = {MbDCLP[x][y][i][10], MbDCLP[x][y][i][11], MbDCLP[x+1][y][i][8], MbDCLP[x+1][y][i][9], MbDCLP[x][y][i][14], MbDCLP[x][y][i][15], MbDCLP[x+1][y][i][12], MbDCLP[x+1][y][i][13], MbDCLP[x][y+1][i][2], MbDCLP[x][y+1][i][3], MbDCLP[x+1][y+1][i][0], MbDCLP[x+1][y+1][i][1], MbDCLP[x][y+1][i][6], MbDCLP[x][y+1][i][7], MbDCLP[x+1][y+1][i][4], MbDCLP[x+1][y+1][i][5]}	
OverlapPostFilter4x4(arrayLocal[ ])	9.9.8.1
MbDCLP[x][y][i][10] = arrayLocal[0]	
MbDCLP[x][y][i][11] = arrayLocal[1]	
MbDCLP[x+1][y][i][8] = arrayLocal[2]	
MbDCLP[x+1][y][i][9] = arrayLocal[3]	
MbDCLP[x][y][i][14] = arrayLocal[4]	
MbDCLP[x][y][i][15] = arrayLocal[5]	
MbDCLP[x+1][y][i][12] = arrayLocal[6]	
MbDCLP[x+1][y][i][13] = arrayLocal[7]	
MbDCLP[x][y+1][i][2] = arrayLocal[8]	
MbDCLP[x][y+1][i][3] = arrayLocal[9]	
MbDCLP[x+1][y+1][i][0] = arrayLocal[10]	
MbDCLP[x+1][y+1][i][1] = arrayLocal[11]	
MbDCLP[x][y+1][i][6] = arrayLocal[12]	
MbDCLP[x][y+1][i][7] = arrayLocal[13]	
MbDCLP[x+1][y+1][i][4] = arrayLocal[14]	
MbDCLP[x+1][y+1][i][5] = arrayLocal[15]	
}	

9.9.4 Second level coefficient combination

NOTE At this point in the process, the DC-LP array coefficients have gone through the first-level transform and overlap filtering. The DC-LP array coefficients and the HP coefficients are then combined in an image plane represented by the values ImagePlane[i][x][y], where the colour component is specified by i, and x and y mark the location of the sample in the image plane.

Inputs to this process are the values MbDCLP[MBx][MBy][i][j], and MBBuffer[MBx][MBy][i][j], for the current macroblock.

Outputs to this process are the values of ImagePlane[i][x][y], for the current macroblock.

The second level coefficient combination process proceeds as in Table 157.

Table 157 — Pseudocode for function SecondLevelCoefficientCombination( )

<table>
<tr><th>SecondLevelCoefficientCombination( ) {</th><th>Reference</th></tr>
<tr><td>for (i = 0; i < NumComponents; i++) {</td><td></td></tr>
<tr><td>for (MBy = 0; MBy < MBHeight; MBy++)</td><td></td></tr>
<tr><td>for (MBx = 0; MBx < MBWidth; MBx++)</td><td></td></tr>
<tr><td>if ((i = = 0) | | ((INTERNAL_CLR_FMT != YUV420) &&
(INTERNAL_CLR_FMT != YUV422))) {</td><td></td></tr>
<tr><td>for (j = 0; j <= 15; j++) {</td><td></td></tr>
<tr><td>x = 16 * MBx + 4 * (j % 4)</td><td></td></tr>
<tr><td>y = 16 * MBy + 4 * (j / 4)</td><td></td></tr>
<tr><td>ImagePlane[i][x][y] = MbDCLP[MBx][MBy][i][j]</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>for (j = 0; j <= 255; j++) {</td><td></td></tr>
<tr><td>x = 16 * MBx + 4 * ((j / 16) % 4) + (j % 4)</td><td></td></tr>
<tr><td>y = 16 * MBy + 4 * (j / 64) + ((j / 4) % 4)</td><td></td></tr>
<tr><td>k = j % 16</td><td></td></tr>
<tr><td>if (k != 0) /* only the HP coefficients are copied */</td><td></td></tr>
<tr><td>ImagePlane[i][x][y] = MBBuffer[MBx][MBy][i][j]</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>} else if (INTERNAL_CLR_FMT = = YUV422) {</td><td></td></tr>
<tr><td>for (j = 0; j <= 7; j++) {</td><td></td></tr>
<tr><td>x = 8 * MBx + 4 * (j % 2)</td><td></td></tr>
<tr><td>y = 16 * MBy + 4 * (j / 2)</td><td></td></tr>
<tr><td>ImagePlane[i][x][y] = MbDCLP[MBx][MBy][i][j]</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>for (j = 0; j <= 127; j++) {</td><td></td></tr>
<tr><td>x = 8 * MBx + 4 * ((j % 32) / 16) + ((j % 32) % 4)</td><td></td></tr>
<tr><td>y = 16 * MBy + 4 * (j / 32) + ((j / 4) % 4)</td><td></td></tr>
<tr><td>k = j % 16</td><td></td></tr>
<tr><td>if (k != 0) /* only the HP coefficients are copied */</td><td></td></tr>
<tr><td>ImagePlane[i][x][y] = MBBuffer[MBx][MBy][i][j]</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>} else if (INTERNAL_CLR_FMT = = YUV420) {</td><td></td></tr>
<tr><td>for (j = 0; j <= 3; j++) {</td><td></td></tr>
<tr><td>x = 8 * MBx + 4 * (j % 2)</td><td></td></tr>
<tr><td>y = 8 * MBy + 4 * (j / 2)</td><td></td></tr>
<tr><td>ImagePlane[i][x][y] = MbDCLP[MBx][MBy][i][j]</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>for (j = 0; j <= 63; j++) {</td><td></td></tr>
<tr><td>x = 8 * MBx + 4 * ((j % 32) / 16) + ((j % 32) % 4)</td><td></td></tr>
<tr><td>y = 8 * MBy + 4 * (j / 32) + ((j / 4) % 4)</td><td></td></tr>
<tr><td>k = j % 16</td><td></td></tr>
<tr><td>if (k != 0) /* only the HP coefficients are copied */</td><td></td></tr>
<tr><td>ImagePlane[i][x][y] = MBBuffer[MBx][MBy][i][j]</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>}</td><td></td></tr>
</table>

9.9.5 Second level inverse transform

Inputs to this process are the values ImagePlane[i][x][y] for the entire image plane.

Outputs to this process are the modified values ImagePlane[i][x][y] for the current macroblock.

The second level inverse transform process is specified as in Table 158.

Table 158 — Pseudocode for function SecondLevelInverseTransform( )

SecondLevelInverseTransform( ) {	Reference
for (i = 0; i < NumComponents; i++)	
for (x = 0; x < ExtendedWidth[i]; x += 4)	
for (y = 0; y < ExtendedHeight[i]; y += 4) {	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y], ImagePlane[i][x+2][y], ImagePlane[i][x+3][y], ImagePlane[i][x][y+1], ImagePlane[i][x+1][y+1], ImagePlane[i][x+2][y+1], ImagePlane[i][x+3][y+1], ImagePlane[i][x][y+2], ImagePlane[i][x+1][y+2], ImagePlane[i][x+2][y+2], ImagePlane[i][x+3][y+2], ImagePlane[i][x][y+3], ImagePlane[i][x+1][y+3], ImagePlane[i][x+2][y+3], ImagePlane[i][x+3][y+3]}	
ICT4x4(arrayLocal[ ])	9.9.7.1
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x+1][y] = arrayLocal[1]	
ImagePlane[i][x+2][y] = arrayLocal[2]	
ImagePlane[i][x+3][y] = arrayLocal[3]	
ImagePlane[i][x][y+1] = arrayLocal[4]	
ImagePlane[i][x+1][y+1] = arrayLocal[5]	
ImagePlane[i][x+2][y+1] = arrayLocal[6]	
ImagePlane[i][x+3][y+1] = arrayLocal[7]	
ImagePlane[i][x][y+2] = arrayLocal[8]	
ImagePlane[i][x+1][y+2] = arrayLocal[9]	
ImagePlane[i][x+2][y+2] = arrayLocal[10]	
ImagePlane[i][x+3][y+2] = arrayLocal[11]	
ImagePlane[i][x][y+3] = arrayLocal[12]	
ImagePlane[i][x+1][y+3] = arrayLocal[13]	
ImagePlane[i][x+2][y+3] = arrayLocal[14]	
ImagePlane[i][x+3][y+3] = arrayLocal[15]	
}	
}	

9.9.6 Second level overlap filtering

Inputs to this process are the values ImagePlane[i][x][y] for the entire image plane.

Outputs to this process are the modified values ImagePlane[i][x][y] for the entire image plane.

Outputs to this process are the modified values ImagePlane[i][x][y] for the current macroblock.

NOTE The process specification below formalizes the geometric nature of the overlap filtering process. There are 4 cases:

(1) (interior): at every point in the image plane where 4 blocks meet in a corner, the 4×4 overlap filter is applied to the 4×4 block straddling these 4 blocks evenly (i.e., overlapping with a 2×2 corner of each block).

(2) (top and bottom 2 rows): along both the top two sample rows and the bottom two sample rows of the image plane, the 4-point overlap filter is applied evenly across adjacent block boundaries (overlapping with a 1×2 strip of each block).

(3) (right-most and left-most columns): along both the left-most two sample columns and the right-most two sample columns, the 4-point overlap filter is applied evenly across adjacent block boundaries (overlapping with a 2×1 strip of each block).

(4) (four corners of the image plane): over the corner 2×2 blocks in the top-left, top-right, bottom-left and bottom-right, the 4-point overlap filter process is applied in a raster scan order (top-left, top-right, bottom-left, then bottom-right).

The second-level overlap filtering process is specified in Table 159.

Table 159 — Pseudocode for function SecondLevelOverlapFiltering( )

SecondLevelOverlapFiltering( ) {	**Reference**
for (i = 0; i < NumComponents; i++) {	
if ((i != 0) && ((INTERNAL_CLR_FMT = = YUV422) | | (INTERNAL_CLR_FMT = = YUV420)))	
dx = 2	
else	
dx = 1	
if ((i != 0) && (INTERNAL_CLR_FMT = = YUV420))	
dy = 2	
else	
dy = 1	
for (Tx = 0; Tx <= (NumTileCols − 1); Tx++) {	
for (Ty = 0; Ty <= (NumTileRows − 1); Ty++) {	
for (x = 16 * LeftMBIndexOfTile[Tx] / dx + 2; x < (16 * LeftMBIndexOfTile[Tx + 1] / dx − 2); x += 4)	
for (y = 16 * TopMBIndexOfTile[Ty] / dy + 2; y < (16 * TopMBIndexOfTile[Ty + 1] / dy − 2); y += 4) {	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y], ImagePlane[i][x+2][y], ImagePlane[i][x+3][y], ImagePlane[i][x][y+1], ImagePlane[i][x+1][y+1], ImagePlane[i][x+2][y+1], ImagePlane[i][x+3][y+1], ImagePlane[i][x][y+2], ImagePlane[i][x+1][y+2], ImagePlane[i][x+2][y+2], ImagePlane[i][x+3][y+2], ImagePlane[i][x][y+3], ImagePlane[i][x+1][y+3], ImagePlane[i][x+2][y+3], ImagePlane[i][x+3][y+3]}	
OverlapPostFilter4x4(arrayLocal[ ])	9.9.8.1
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x+1][y] = arrayLocal[1]	
ImagePlane[i][x+2][y] = arrayLocal[2]	
ImagePlane[i][x+3][y] = arrayLocal[3]	
ImagePlane[i][x][y+1] = arrayLocal[4]	
ImagePlane[i][x+1][y+1] = arrayLocal[5]	
ImagePlane[i][x+2][y+1] = arrayLocal[6]	
ImagePlane[i][x+3][y+1] = arrayLocal[7]	
ImagePlane[i][x][y+2] = arrayLocal[8]	
ImagePlane[i][x+1][y+2] = arrayLocal[9]	
ImagePlane[i][x+2][y+2] = arrayLocal[10]	
ImagePlane[i][x+3][y+2] = arrayLocal[11]	
ImagePlane[i][x][y+3] = arrayLocal[12]	
ImagePlane[i][x+1][y+3] = arrayLocal[13]	
ImagePlane[i][x+2][y+3] = arrayLocal[14]	
ImagePlane[i][x+3][y+3] = arrayLocal[15]	
}	
if ((Tx = = 0) | | (HARD_TILING_FLAG = = TRUE)) { /* Left edge */	
for (y = 16 * TopMBIndexOfTile[Ty] / dy + 2; y < (16 * TopMBIndexOfTile[Ty + 1] / dy − 2); y += 4) {	
x = 16 * LeftMBIndexOfTile[Tx] / dx	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x][y+1], ImagePlane[i][x][y+2], ImagePlane[i][x][y+3]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x][y+1] = arrayLocal[1]	
ImagePlane[i][x][y+2] = arrayLocal[2]	
ImagePlane[i][x][y+3] = arrayLocal[3]	
x = 16 * LeftMBIndexOfTile[Tx] / dx + 1	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x][y+1], ImagePlane[i][x][y+2], ImagePlane[i][x][y+3]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2

SecondLevelOverlapFiltering( ) {	**Reference**
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x][y+1] = arrayLocal[1]	
ImagePlane[i][x][y+2] = arrayLocal[2]	
ImagePlane[i][x][y+3] = arrayLocal[3]	
}	
}	
if ((Ty = = 0) | | (HARD_TILING_FLAG = = TRUE)) { /* Top edge */	
for (x = 16 * LeftMBIndexOfTile[Tx] / dx + 2; x < (16 * LeftMBIndexOfTile[Tx + 1] / dx – 2); x += 4) {	
y = 16 * TopMBIndexOfTile[Ty] / dy	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y], ImagePlane[i][x+2][y], ImagePlane[i][x+3][y]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x+1][y] = arrayLocal[1]	
ImagePlane[i][x+2][y] = arrayLocal[2]	
ImagePlane[i][x+3][y] = arrayLocal[3]	
y = 16 * TopMBIndexOfTile[Ty] / dy + 1	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y], ImagePlane[i][x+2][y], ImagePlane[i][x+3][y]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x+1][y] = arrayLocal[1]	
ImagePlane[i][x+2][y] = arrayLocal[2]	
ImagePlane[i][x+3][y] = arrayLocal[3]	
}	
}	
if ((Tx = = NumTileCols – 1) | | (HARD_TILING_FLAG = = TRUE)) { /* Right edge */	
for (y = 16 * TopMBIndexOfTile[Ty] / dy + 2; y < (16 * TopMBIndexOfTile[Ty + 1] / dy – 2); y += 4) {	
x = 16 * LeftMBIndexOfTile[Tx + 1] / dx – 2	
arrayLocal[ ] = { ImagePlane[i][x][y], ImagePlane[i][x][y+1], ImagePlane[i][x][y+2], ImagePlane[i][x][y+3]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x][y+1] = arrayLocal[1]	
ImagePlane[i][x][y+2] = arrayLocal[2]	
ImagePlane[i][x][y+3] = arrayLocal[3]	
x = 16 * LeftMBIndexOfTile[Tx + 1] / dx – 1	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x][y+1], ImagePlane[i][x][y+2], ImagePlane[i][x][y+3]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x][y+1] = arrayLocal[1]	
ImagePlane[i][x][y+2] = arrayLocal[2]	
ImagePlane[i][x][y+3] = arrayLocal[3]	
}	
}	
if ((Ty = = NumTileRows – 1) | | (HARD_TILING_FLAG = = TRUE)) { /* Bottom edge */	
for (x = 16 * LeftMBIndexOfTile[Tx] / dx + 2; x < (16 * LeftMBIndexOfTile[Tx + 1] / dx – 2); x += 4) {	
y = 16 * TopMBIndexOfTile[Ty + 1] / dy – 2	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y], ImagePlane[i][x+2][y], ImagePlane[i][x+3][y]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x+1][y] = arrayLocal[1]	
ImagePlane[i][x+2][y] = arrayLocal[2]	
ImagePlane[i][x+3][y] = arrayLocal[3]	
y = 16 * TopMBIndexOfTile[Ty + 1] / dy – 1	

<table>
<tr><th>SecondLevelOverlapFiltering( ) {</th><th>Reference</th></tr>
<tr><td>arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y],
ImagePlane[i][x+2][y], ImagePlane[i][x+3][y]}</td><td></td></tr>
<tr><td>OverlapPostFilter4(arrayLocal[ ])</td><td>9.9.8.2</td></tr>
<tr><td>ImagePlane[i][x][y] = arrayLocal[0]</td><td></td></tr>
<tr><td>ImagePlane[i][x+1][y] = arrayLocal[1]</td><td></td></tr>
<tr><td>ImagePlane[i][x+2][y] = arrayLocal[2]</td><td></td></tr>
<tr><td>ImagePlane[i][x+3][y] = arrayLocal[3]</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>if (((Tx = = 0) && (Ty = = 0)) | | (HARD_TILING_FLAG = = TRUE)) { /* Top left edge */</td><td></td></tr>
<tr><td>x = 16 * LeftMBIndexOfTile[Tx] / dx</td><td></td></tr>
<tr><td>y = 16 * TopMBIndexOfTile[Ty] / dy</td><td></td></tr>
<tr><td>arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y],
ImagePlane[i][x][y+1], ImagePlane[i][x+1][y+1]}</td><td></td></tr>
<tr><td>OverlapPostFilter4(arrayLocal[ ])</td><td>9.9.8.2</td></tr>
<tr><td>ImagePlane[i][x][y] = arrayLocal[0]</td><td></td></tr>
<tr><td>ImagePlane[i][x+1][y] = arrayLocal[1]</td><td></td></tr>
<tr><td>ImagePlane[i][x][y+1] = arrayLocal[2]</td><td></td></tr>
<tr><td>ImagePlane[i][x+1][y+1] = arrayLocal[3]</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>if (((Tx = = NumTileCols − 1) && (Ty = = 0)) | | (HARD_TILING_FLAG = = TRUE)) {
/* Top right edge */</td><td></td></tr>
<tr><td>x = 16 * LeftMBIndexOfTile[Tx + 1] / dx − 2</td><td></td></tr>
<tr><td>y = 16 * TopMBIndexOfTile[Ty] / dy</td><td></td></tr>
<tr><td>arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y],
ImagePlane[i][x][y+1], ImagePlane[i][x+1][y+1]}</td><td></td></tr>
<tr><td>OverlapPostFilter4(arrayLocal[ ])</td><td>9.9.8.2</td></tr>
<tr><td>ImagePlane[i][x][y] = arrayLocal[0]</td><td></td></tr>
<tr><td>ImagePlane[i][x+1][y] = arrayLocal[1]</td><td></td></tr>
<tr><td>ImagePlane[i][x][y+1] = arrayLocal[2]</td><td></td></tr>
<tr><td>ImagePlane[i][x+1][y+1] = arrayLocal[3]</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>if (((Tx = = 0) && (Ty = = NumTileRows − 1)) | | (HARD_TILING_FLAG = = TRUE)) {
/* Bottom left edge */</td><td></td></tr>
<tr><td>x = 16 * LeftMBIndexOfTile[Tx] / dx</td><td></td></tr>
<tr><td>y = 16 * TopMBIndexOfTile[Ty + 1] / dy − 2</td><td></td></tr>
<tr><td>arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y],
ImagePlane[i][x][y+1], ImagePlane[i][x+1][y+1]}</td><td></td></tr>
<tr><td>OverlapPostFilter4(arrayLocal[ ])</td><td>9.9.8.2</td></tr>
<tr><td>ImagePlane[i][x][y] = arrayLocal[0]</td><td></td></tr>
<tr><td>ImagePlane[i][x+1][y] = arrayLocal[1]</td><td></td></tr>
<tr><td>ImagePlane[i][x][y+1] = arrayLocal[2]</td><td></td></tr>
<tr><td>ImagePlane[i][x+1][y+1] = arrayLocal[3]</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>if (((Tx = = NumTileCols − 1) && (Ty = = NumTileRows − 1)) | |
(HARD_TILING_FLAG = = TRUE)) { /* Bottom right edge */</td><td></td></tr>
<tr><td>x = 16 * LeftMBIndexOfTile[Tx + 1] / dx − 2</td><td></td></tr>
<tr><td>y = 16 * TopMBIndexOfTile[Ty + 1] / dy − 2</td><td></td></tr>
<tr><td>arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y],
ImagePlane[i][x][y+1], ImagePlane[i][x+1][y+1]}</td><td></td></tr>
<tr><td>OverlapPostFilter4(arrayLocal[ ])</td><td>9.9.8.2</td></tr>
<tr><td>ImagePlane[i][x][y] = arrayLocal[0]</td><td></td></tr>
<tr><td>ImagePlane[i][x+1][y] = arrayLocal[1]</td><td></td></tr>
<tr><td>ImagePlane[i][x][y+1] = arrayLocal[2]</td><td></td></tr>
<tr><td>ImagePlane[i][x+1][y+1] = arrayLocal[3]</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>if ((HARD_TILING_FLAG = = FALSE) && (Tx != NumTileCols − 1)) {
/* Right across for soft tiles */</td><td></td></tr>
<tr><td>x = 16 * LeftMBIndexOfTile[Tx + 1] / dx − 2</td><td></td></tr>
<tr><td>for (y = 16 * TopMBIndexOfTile[Ty] / dy + 2; y < (16 * TopMBIndexOfTile[Ty + 1] / dy − 2);</td><td></td></tr>
</table>

SecondLevelOverlapFiltering( ) {	Reference
y += 4) {	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y], ImagePlane[i][x+2][y],ImagePlane[i][x+3][y], ImagePlane[i][x][y+1], ImagePlane[i][x+1][y+1], ImagePlane[i][x+2][y+1], ImagePlane[i][x+3][y+1], ImagePlane[i][x][y+2], ImagePlane[i][x+1][y+2], ImagePlane[i][x+2][y+2], ImagePlane[i][x+3][y+2], ImagePlane[i][x][y+3], ImagePlane[i][x+1][y+3], ImagePlane[i][x+2][y+3], ImagePlane[i][x+3][y+3]}	
OverlapPostFilter4x4(arrayLocal[ ])	9.9.8.1
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x+1][y] = arrayLocal[1]	
ImagePlane[i][x+2][y] = arrayLocal[2]	
ImagePlane[i][x+3][y] = arrayLocal[3]	
ImagePlane[i][x][y+1] = arrayLocal[4]	
ImagePlane[i][x+1][y+1] = arrayLocal[5]	
ImagePlane[i][x+2][y+1] = arrayLocal[6]	
ImagePlane[i][x+3][y+1] = arrayLocal[7]	
ImagePlane[i][x][y+2] = arrayLocal[8]	
ImagePlane[i][x+1][y+2] = arrayLocal[9]	
ImagePlane[i][x+2][y+2] = arrayLocal[10]	
ImagePlane[i][x+3][y+2] = arrayLocal[11]	
ImagePlane[i][x][y+3] = arrayLocal[12]	
ImagePlane[i][x+1][y+3] = arrayLocal[13]	
ImagePlane[i][x+2][y+3] = arrayLocal[14]	
ImagePlane[i][x+3][y+3] = arrayLocal[15]	
}	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Ty != NumTileRows – 1)) { /* Bottom across for soft tiles */	
y = 16 * TopMBIndexOfTile[Ty + 1] / dy – 2	
for (x = 16 * LeftMBIndexOfTile[Tx] / dx + 2; x < (16 * LeftMBIndexOfTile[Tx + 1] / dx – 2); x += 4) {	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y], ImagePlane[i][x+2][y], ImagePlane[i][x+3][y], ImagePlane[i][x][y+1], ImagePlane[i][x+1][y+1], ImagePlane[i][x+2][y+1], ImagePlane[i][x+3][y+1], ImagePlane[i][x][y+2], ImagePlane[i][x+1][y+2], ImagePlane[i][x+2][y+2], ImagePlane[i][x+3][y+2], ImagePlane[i][x][y+3], ImagePlane[i][x+1][y+3], ImagePlane[i][x+2][y+3], ImagePlane[i][x+3][y+3]}	
OverlapPostFilter4x4(arrayLocal[ ])	9.9.8.1
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x+1][y] = arrayLocal[1]	
ImagePlane[i][x+2][y] = arrayLocal[2]	
ImagePlane[i][x+3][y] = arrayLocal[3]	
ImagePlane[i][x][y+1] = arrayLocal[4]	
ImagePlane[i][x+1][y+1] = arrayLocal[5]	
ImagePlane[i][x+2][y+1] = arrayLocal[6]	
ImagePlane[i][x+3][y+1] = arrayLocal[7]	
ImagePlane[i][x][y+2] = arrayLocal[8]	
ImagePlane[i][x+1][y+2] = arrayLocal[9]	
ImagePlane[i][x+2][y+2] = arrayLocal[10]	
ImagePlane[i][x+3][y+2] = arrayLocal[11]	
ImagePlane[i][x][y+3] = arrayLocal[12]	
ImagePlane[i][x+1][y+3] = arrayLocal[13]	
ImagePlane[i][x+2][y+3] = arrayLocal[14]	
ImagePlane[i][x+3][y+3] = arrayLocal[15]	
}	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Tx != NumTileCols – 1) &&	

SecondLevelOverlapFiltering( ) {	**Reference**
(Ty != NumTileRows – 1)) {	
x = 16 * LeftMBIndexOfTile[Tx + 1] / dx – 2	
y = 16 * TopMBIndexOfTile[Ty + 1] / dy – 2	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y], ImagePlane[i][x+2][y], ImagePlane[i][x+3][y], ImagePlane[i][x][y+1], ImagePlane[i][x+1][y+1], ImagePlane[i][x+2][y+1], ImagePlane[i][x+3][y+1], ImagePlane[i][x][y+2], ImagePlane[i][x+1][y+2], ImagePlane[i][x+2][y+2], ImagePlane[i][x+3][y+2], ImagePlane[i][x][y+3], ImagePlane[i][x+1][y+3], ImagePlane[i][x+2][y+3], ImagePlane[i][x+3][y+3]}	
OverlapPostFilter4x4(arrayLocal[ ])	9.9.8.1
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x+1][y] = arrayLocal[1]	
ImagePlane[i][x+2][y] = arrayLocal[2]	
ImagePlane[i][x+3][y] = arrayLocal[3]	
ImagePlane[i][x][y+1] = arrayLocal[4]	
ImagePlane[i][x+1][y+1] = arrayLocal[5]	
ImagePlane[i][x+2][y+1] = arrayLocal[6]	
ImagePlane[i][x+3][y+1] = arrayLocal[7]	
ImagePlane[i][x][y+2] = arrayLocal[8]	
ImagePlane[i][x+1][y+2] = arrayLocal[9]	
ImagePlane[i][x+2][y+2] = arrayLocal[10]	
ImagePlane[i][x+3][y+2] = arrayLocal[11]	
ImagePlane[i][x][y+3] = arrayLocal[12]	
ImagePlane[i][x+1][y+3] = arrayLocal[13]	
ImagePlane[i][x+2][y+3] = arrayLocal[14]	
ImagePlane[i][x+3][y+3] = arrayLocal[15]	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Tx = = 0) && (Ty != NumTileRows – 1)) { /* Left edge for soft tiles */	
y = 16 * TopMBIndexOfTile[Ty + 1] / dy – 2	
x = 16 * LeftMBIndexOfTile[Tx] / dx	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x][y+1], ImagePlane[i][x][y+2], ImagePlane[i][x][y+3]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x][y+1] = arrayLocal[1]	
ImagePlane[i][x][y+2] = arrayLocal[2]	
ImagePlane[i][x][y+3] = arrayLocal[3]	
x = 16 * LeftMBIndexOfTile[Tx] / dx + 1	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x][y+1], ImagePlane[i][x][y+2], ImagePlane[i][x][y+3]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x][y+1] = arrayLocal[1]	
ImagePlane[i][x][y+2] = arrayLocal[2]	
ImagePlane[i][x][y+3] = arrayLocal[3]	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Tx != NumTileCols – 1) && (Ty = = 0)) { /* Top edge for soft tiles */	
x = 16 * LeftMBIndexOfTile[Tx + 1] / dx – 2	
y = 16 * TopMBIndexOfTile[Ty] / dy	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y], ImagePlane[i][x+2][y], ImagePlane[i][x+3][y]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x+1][y] = arrayLocal[1]	
ImagePlane[i][x+2][y] = arrayLocal[2]	
ImagePlane[i][x+3][y] = arrayLocal[3]	

SecondLevelOverlapFiltering( ) {	**Reference**
y = 16 * TopMBIndexOfTile[Ty] / dy + 1	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y], ImagePlane[i][x+2][y], ImagePlane[i][x+3][y]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x+1][y] = arrayLocal[1]	
ImagePlane[i][x+2][y] = arrayLocal[2]	
ImagePlane[i][x+3][y] = arrayLocal[3]	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Tx = = NumTileCols − 1) && (Ty != NumTileRows − 1)) { /* Right edge for soft tiles */	
y = 16 * TopMBIndexOfTile[Ty + 1] / dy − 2	
x = 16 * LeftMBIndexOfTile[Tx + 1] / dx − 2	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x][y+1], ImagePlane[i][x][y+2], ImagePlane[i][x][y+3]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x][y+1] = arrayLocal[1]	
ImagePlane[i][x][y+2] = arrayLocal[2]	
ImagePlane[i][x][y+3] = arrayLocal[3]	
x = 16 * LeftMBIndexOfTile[Tx + 1] / dx − 1	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x][y+1], ImagePlane[i][x][y+2], ImagePlane[i][x][y+3]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x][y+1] = arrayLocal[1]	
ImagePlane[i][x][y+2] = arrayLocal[2]	
ImagePlane[i][x][y+3] = arrayLocal[3]	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Tx != NumTileCols − 1) && (Ty = = NumTileRows − 1)) { /* Bottom edge for soft tiles */	
x = 16 * LeftMBIndexOfTile[Tx + 1] / dx − 2	
y = 16 * TopMBIndexOfTile[Ty + 1] / dy − 2	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y], ImagePlane[i][x+2][y], ImagePlane[i][x+3][y]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x+1][y] = arrayLocal[1]	
ImagePlane[i][x+2][y] = arrayLocal[2]	
ImagePlane[i][x+3][y] = arrayLocal[3]	
y = 16 * TopMBIndexOfTile[Ty + 1] / dy − 1	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y], ImagePlane[i][x+2][y], ImagePlane[i][x+3][y]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x+1][y] = arrayLocal[1]	
ImagePlane[i][x+2][y] = arrayLocal[2]	
ImagePlane[i][x+3][y] = arrayLocal[3]	
}	
}	
}	
}	
}	

9.9.7 Inverse transform basic operations

9.9.7.1 ICT4x4( )

NOTE 1 The 2D ICT4x4( ) is built using the three operators: T2x2h, InvTodd and InvToddodd, preceded by the permutation function InvPermute. After the initial permutation, the transform operation consists of two stages, where each stage operates on all 16 of the input values.

The function ICT4x4( ) is specified by the pseudocode in Table 160.

Table 160 — Pseudocode for function ICT4x4( )

ICT4x4(iCoeff[ ]) {	**Reference**
/* Permute the coefficients */	
InvPermute(iCoeff[ ])	9.9.7.5
/* First stage */	
arrayLocal[ ] = {iCoeff[0], iCoeff[1], iCoeff[4], iCoeff[5]}	
T2x2h(arrayLocal[ ], 1)	9.9.7.2
iCoeff[0] = arrayLocal[0]	
iCoeff[1] = arrayLocal[1]	
iCoeff[4] = arrayLocal[2]	
iCoeff[5] = arrayLocal[3]	
arrayLocal[ ] = {iCoeff[2], iCoeff[3], iCoeff[6], iCoeff[7]}	
InvTodd(arrayLocal[ ])	9.9.7.3
iCoeff[2] = arrayLocal[0]	
iCoeff[3] = arrayLocal[1]	
iCoeff[6] = arrayLocal[2]	
iCoeff[7] = arrayLocal[3]	
arrayLocal[ ] = {iCoeff[8], iCoeff[12], iCoeff[9], iCoeff[13]}	
InvTodd(arrayLocal[ ])	9.9.7.3
iCoeff[8] = arrayLocal[0]	
iCoeff[12] = arrayLocal[1]	
iCoeff[9] = arrayLocal[2]	
iCoeff[13] = arrayLocal[3]	
arrayLocal[ ] = {iCoeff[10], iCoeff[11], iCoeff[14], iCoeff[15]}	
InvToddodd(arrayLocal[ ])	9.9.7.4
iCoeff[10] = arrayLocal[0]	
iCoeff[11] = arrayLocal[1]	
iCoeff[14] = arrayLocal[2]	
iCoeff[15] = arrayLocal[3]	
/* Second stage */	
arrayLocal[ ] = {iCoeff[0], iCoeff[3], iCoeff[12], iCoeff[15]}	
T2x2h(arrayLocal[ ], 0)	9.9.7.2
iCoeff[0] = arrayLocal[0]	
iCoeff[3] = arrayLocal[1]	
iCoeff[12] = arrayLocal[2]	
iCoeff[15] = arrayLocal[3]	
arrayLocal[ ] = {iCoeff[5], iCoeff[6], iCoeff[9], iCoeff[10]}	
T2x2h(arrayLocal[ ], 0)	9.9.7.2
iCoeff[5] = arrayLocal[0]	
iCoeff[6] = arrayLocal[1]	
iCoeff[9] = arrayLocal[2]	
iCoeff[10] = arrayLocal[3]	
arrayLocal[ ] = {iCoeff[1], iCoeff[2], iCoeff[13], iCoeff[14]}	
T2x2h(arrayLocal[ ], 0)	9.9.7.2
iCoeff[1] = arrayLocal[0]	
iCoeff[2] = arrayLocal[1]	
iCoeff[13] = arrayLocal[2]	
iCoeff[14] = arrayLocal[3]	
arrayLocal[ ] = {iCoeff[4], iCoeff[7], iCoeff[8], iCoeff[11]}	

ICT4x4(iCoeff[ ]) {	Reference
T2x2h(arrayLocal[ ], 0)	9.9.7.2
iCoeff[4] = arrayLocal[0]	
iCoeff[7] = arrayLocal[1]	
iCoeff[8] = arrayLocal[2]	
iCoeff[11] = arrayLocal[3]	
}	

NOTE 2 Each stage consists of four 2×2 transforms which can be done in any arbitrary sequence, or concurrently, within the stage. However, the first-stage transforms must be complete before any of the second-stage transforms are initiated.

9.9.7.2 T2x2h( )

The function T2x2h( ) is specified in Table 161.

NOTE The variable valRound is a rounding control variable. The value of valRound is set to 0 or 1 by the function that invokes T2x2h( ). The inverse of T2x2Th( ) is two successive applications of T2x2Th, operating on variables of the array iCoeff[ ] with the same value of valRound.

Table 161 — Pseudocode for function T2x2h( )

T2x2h(iCoeff[ ], valRound) {	Reference
iCoeff[0] += iCoeff[3]	
iCoeff[1] −= iCoeff[2]	
valT1 = ((iCoeff[0] − iCoeff[1] + valRound) >> 1)	
valT2 = iCoeff[2]	
iCoeff[2] = valT1 − iCoeff[3]	
iCoeff[3] = valT1 − valT2	
iCoeff[0] −= iCoeff[3]	
iCoeff[1] += iCoeff[2]	
}	

9.9.7.3 InvTodd( )

The function InvTodd( ) is specified by the pseudocode in Table 162.

Table 162 — Pseudocode for function InvTodd( )

InvTodd(iCoeff[ ]) {	Reference
iCoeff[1] += iCoeff[3]	
iCoeff[0] −= iCoeff[2]	
iCoeff[3] −= (iCoeff[1] >> 1)	
iCoeff[2] += ((iCoeff[0] + 1) >> 1)	
iCoeff[0] −= ((3* iCoeff[1] + 4) >> 3)	
iCoeff[1] += ((3* iCoeff[0] + 4) >> 3)	
iCoeff[2] −= ((3* iCoeff[3] + 4) >> 3)	
iCoeff[3] += ((3* iCoeff[2] + 4) >> 3	
iCoeff[2] −= ((iCoeff[1] + 1) >> 1)	
iCoeff[3] = ((iCoeff[0] + 1) >> 1) − iCoeff[3]	
iCoeff[1] += iCoeff[2]	
iCoeff[0] −= iCoeff[3]	
}	

9.9.7.4 InvToddodd( )

The function, InvToddodd( ) is specified by the pseudocode in Table 163.

Table 163 — Pseudocode for function InvToddodd

InvToddodd(iCoeff[ ]**)** {	**Reference**
iCoeff[3] += iCoeff[0]	
iCoeff[2] −= iCoeff[1]	
valT1 = iCoeff[3] >> 1	
valT2 = iCoeff[2] >> 1	
iCoeff[0] −= valT1	
iCoeff[1] += valT2	
iCoeff[0] −= ((iCoeff[1] * 3 + 3) >> 3)	
iCoeff[1] += ((iCoeff[0] * 3 + 3) >> 2)	
iCoeff[0] −= ((iCoeff[1] * 3 + 4) >> 3)	
iCoeff[1] −= valT2	
iCoeff[0] += valT1	
iCoeff[2] += iCoeff[1]	
iCoeff[3] −= iCoeff[0]	
iCoeff[1] = −iCoeff[1]	
iCoeff[2] = −iCoeff[2]	
}	

9.9.7.5 InvPermute( )

The function InvPermute( ) operates on an ordered array of 16 sample values, producing a permuted list. The input to this function is the ordered array arrayInput[i], for i ranging from 0 to 15. The output of this function is the re-ordered array arrayInput[i].

To define the permutation, the array InvPermArr[i] is specified, for i ranging from 0 to 15, in Table 164.

Table 164 — Inverse permutation

i	**InvPermArr[i]**
0	0
1	8
2	4
3	13
4	2
5	15
6	3
7	14
8	1
9	12
10	5
11	9
12	7
13	11
14	6
15	10

The function InvPermute( ) is specified as in Table 165.

Table 165 — Pseudocode for function InvPermute( )

InvPermute(arrayInput[ ]) {	**Reference**
for (i = 0; i <= 15; i++)	
arrayTemp[InvPermArr[i]] = arrayInput[i]	
for (i = 0; i <= 15; i++)	
arrayInput[i] = arrayTemp[i]	
}	

9.9.7.6 InvPermute2pt( )

The function InvPermute2pt( ) operates on an ordered array of 2 sample values, producing a permuted list. The input to this function is the ordered array arrayInput[i], for i ranging from 0 to 1. The output of this function is the re-ordered array arrayInput[i].

The function InvPermute2pt( ) is specified as in Table 166.

Table 166 — Pseudocode for function InvPermute2pt( )

InvPermute2pt( ) {	**Reference**
arrayTemp[0] = arrayInput[1]	
arrayTemp[1] = arrayInput[0]	
for (i = 0; i <= 1; i++)	
arrayInput[i] = arrayTemp[i]	
}	

9.9.7.7 T2pt( )

The function T2pt( ) is specified by the pseudocode in Table 167.

Table 167 — Pseudocode for function T2pt( )

T2pt(iCoeff[ ]) {	**Reference**
iCoeff[0] -= (iCoeff[1]+1) >> 1	
iCoeff[1] += iCoeff[0]	
}	

9.9.8 Overlap filtering functions

9.9.8.1 OverlapPostFilter4x4( )

The function OverlapPostFilter4x4( ) is specified in Table 168.

Table 168 — Pseudocode for function OverlapPostFilter4x4( )

OverlapPostFilter4x4(iCoeff[ ]) {	**Reference**
arrayLocal[ ] = {iCoeff[0], iCoeff[3], iCoeff[12], iCoeff[15]}	
T2x2h(arrayLocal[ ], 0)	9.9.7.2
iCoeff[0] = arrayLocal[0]	
iCoeff[3] = arrayLocal[1]	
iCoeff[12] = arrayLocal[2]	
iCoeff[15] = arrayLocal[3]	
arrayLocal[ ] = {iCoeff[1], iCoeff[2], iCoeff[13], iCoeff[14]}	
T2x2h(arrayLocal[ ], 0)	9.9.7.2
iCoeff[1] = arrayLocal[0]	
iCoeff[2] = arrayLocal[1]	
iCoeff[13] = arrayLocal[2]	
iCoeff[14] = arrayLocal[3]	
arrayLocal[ ] = {iCoeff[4], iCoeff[7], iCoeff[8], iCoeff[11]}	
T2x2h(arrayLocal[ ], 0)	9.9.7.2
iCoeff[4] = arrayLocal[0]	
iCoeff[7] = arrayLocal[1]	
iCoeff[8] = arrayLocal[2]	
iCoeff[11] = arrayLocal[3]	
arrayLocal[ ] = {iCoeff[5], iCoeff[6], iCoeff[9], iCoeff[10]}	
T2x2h(arrayLocal[ ], 0)	9.9.7.2
iCoeff[5] = arrayLocal[0]	
iCoeff[6] = arrayLocal[1]	
iCoeff[9] = arrayLocal[2]	
iCoeff[10] = arrayLocal[3]	
arrayLocal[ ] = {iCoeff[13], iCoeff[12]}	
InvRotate(arrayLocal[ ])	9.9.8.5
iCoeff[13] = arrayLocal[0]	
iCoeff[12] = arrayLocal[1]	
arrayLocal[ ] = {iCoeff[9], iCoeff[8]}	
InvRotate(arrayLocal[ ])	9.9.8.5
iCoeff[9] = arrayLocal[0]	
iCoeff[8] = arrayLocal[1]	
arrayLocal[ ] = {iCoeff[7], iCoeff[3]}	
InvRotate(arrayLocal[ ])	9.9.8.5
iCoeff[7] = arrayLocal[0]	
iCoeff[3] = arrayLocal[1]	
arrayLocal[ ] = {iCoeff[6], iCoeff[2]}	
InvRotate(arrayLocal[ ])	9.9.8.5
iCoeff[6] = arrayLocal[0]	
iCoeff[2] = arrayLocal[1]	
arrayLocal[ ] = {iCoeff[10], iCoeff[11], iCoeff[14], iCoeff[15]}	
InvToddoddPOST(arrayLocal[ ])	9.9.8.8
iCoeff[10] = arrayLocal[0]	
iCoeff[11] = arrayLocal[1]	
iCoeff[14] = arrayLocal[2]	
iCoeff[15] = arrayLocal[3]	

OverlapPostFilter4x4(iCoeff[ ]) {	**Reference**
arrayLocal[ ] = {iCoeff[0], iCoeff[15]}	
InvScale(arrayLocal[ ])	9.9.8.6
iCoeff[0] = arrayLocal[0]	
iCoeff[15] = arrayLocal[1]	
arrayLocal[ ] = {iCoeff[1], iCoeff[14]}	
InvScale(arrayLocal[ ])	9.9.8.6
iCoeff[1] = arrayLocal[0]	
iCoeff[14] = arrayLocal[1]	
arrayLocal[ ] = {iCoeff[4], iCoeff[11]}	
InvScale(arrayLocal[ ])	9.9.8.6
iCoeff[4] = arrayLocal[0]	
iCoeff[11] = arrayLocal[1]	
arrayLocal[ ] = {iCoeff[5], iCoeff[10]}	
InvScale(arrayLocal[ ])	9.9.8.6
iCoeff[5] = arrayLocal[0]	
iCoeff[10] = arrayLocal[1]	
arrayLocal[ ] = {iCoeff[0], iCoeff[3], iCoeff[12], iCoeff[15]}	
T2x2hPOST(arrayLocal[ ])	9.9.8.7
iCoeff[0] = arrayLocal[0]	
iCoeff[3] = arrayLocal[1]	
iCoeff[12] = arrayLocal[2]	
iCoeff[15] = arrayLocal[3]	
arrayLocal[ ] = {iCoeff[1], iCoeff[2], iCoeff[13], iCoeff[14]}	
T2x2hPOST(arrayLocal[ ])	9.9.8.7
iCoeff[1] = arrayLocal[0]	
iCoeff[2] = arrayLocal[1]	
iCoeff[13] = arrayLocal[2]	
iCoeff[14] = arrayLocal[3]	
arrayLocal[ ] = {iCoeff[4], iCoeff[7], iCoeff[8], iCoeff[11]}	
T2x2hPOST(arrayLocal[ ])	9.9.8.7
iCoeff[4] = arrayLocal[0]	
iCoeff[7] = arrayLocal[1]	
iCoeff[8] = arrayLocal[2]	
iCoeff[11] = arrayLocal[3]	
arrayLocal[ ] = {iCoeff[5], iCoeff[6], iCoeff[9], iCoeff[10]}	
T2x2hPOST(arrayLocal[ ])	9.9.8.7
iCoeff[5] = arrayLocal[0]	
iCoeff[6] = arrayLocal[1]	
iCoeff[9] = arrayLocal[2]	
iCoeff[10] = arrayLocal[3]	
}	

9.9.8.2 OverlapPostFilter4( )

The function OverlapPostFilter4( ) is specified in Table 169.

Table 169 — Pseudocode for function OverlapPostFilter4( )

OverlapPostFilter4(iCoeff[ ]) {	Reference
iCoeff[0] += iCoeff[3]	
iCoeff[1] += iCoeff[2]	
iCoeff[3] −= ((iCoeff[0] + 1) >> 1)	
iCoeff[2] −= ((iCoeff[1] + 1) >> 1)	
InvScale(iCoeff[0], iCoeff[3])	9.9.8.6
InvScale(iCoeff[1], iCoeff[2])	9.9.8.6
iCoeff[0] += ((iCoeff[3] * 3+ 4) >> 3)	
iCoeff[1] += ((iCoeff[2] * 3 + 4) >> 3)	
iCoeff[3] −= (iCoeff[0] >> 1)	
iCoeff[2] −= (iCoeff[1] >> 1)	
iCoeff[0] += iCoeff[3]	
iCoeff[1] += iCoeff[2]	
iCoeff[3] = −iCoeff[3]	
iCoeff[2] = −iCoeff[2]	
InvRotate(iCoeff[2], iCoeff[3])	9.9.8.5
iCoeff[3] += ((iCoeff[0] + 1) >> 1)	
iCoeff[2] += ((iCoeff[1] + 1) >> 1)	
iCoeff[0] −= iCoeff[3]	
iCoeff[1] −= iCoeff[2]	
}	

9.9.8.3 OverlapPostFilter2x2( )

The function OverlapPostFilter2x2( ) is specified in Table 170.

Table 170 — Pseudocode for function OverlapPostFilter2x2( )

OverlapPostFilter2x2(iCoeff[ ]) {	Reference
iCoeff[0] += iCoeff[3]	
iCoeff[1] += iCoeff[2]	
iCoeff[3] −= ((iCoeff[0] + 1) >> 1)	
iCoeff[2] −= ((iCoeff[1] + 1) >> 1)	
iCoeff[1] += ((iCoeff[0] + 2) >> 2)	
iCoeff[0] += ((iCoeff[1] + 1) >> 1)	
iCoeff[0] += (iCoeff[1] >> 5)	
iCoeff[0] += (iCoeff[1] >> 9)	
iCoeff[0] += (iCoeff[1] >> 13)	
iCoeff[1] += ((iCoeff[0] + 2) >> 2)	
iCoeff[3] += ((iCoeff[0] + 1) >> 1)	
iCoeff[2] += ((iCoeff[1] + 1) >> 1)	
iCoeff[0] −= iCoeff[3]	
iCoeff[1] −= iCoeff[2]	
}	

9.9.8.4 OverlapPostFilter2( )

The function OverlapPostFilter2( ) is specified in Table 171.

Table 171 — Pseudocode for function OverlapPostFilter2( )

OverlapPostFilter2(iCoeff[ ]) {	**Reference**
iCoeff[1] += ((iCoeff[0] + 2) >> 2)	
iCoeff[0] += ((iCoeff[1] + 1) >> 1)	
iCoeff[0] += (iCoeff[1] >> 5)	
iCoeff[0] += (iCoeff[1] >> 9)	
iCoeff[0] += (iCoeff[1] >> 13)	
iCoeff[1] += ((iCoeff[0] + 2) >> 2)	
}	

9.9.8.5 InvRotate( )

The function InvRotate( ) is specified by the pseudocode in Table 172.

Table 172 — Pseudocode for function InvRotate( )

InvRotate(iCoeff[ ]) {	**Reference**
iCoeff[0] −= ((iCoeff[1] + 1) >> 1)	
iCoeff[1] += ((iCoeff[0] + 1) >> 1)	
}	

9.9.8.6 InvScale( )

The function InvScale( ) is specified by the pseudocode in Table 173.

Table 173 — Pseudocode for function InvScale( )

InvScale(iCoeff[ ]) {	**Reference**
iCoeff[0] += iCoeff[1]	
iCoeff[1] = (iCoeff[0] >> 1) − iCoeff[1]	
iCoeff[0] += (iCoeff[1] * 3 + 0) >> 3	
iCoeff[1] += (iCoeff[0] * 3 + 0) >> 4	
iCoeff[1] += (iCoeff[0] >> 7)	
iCoeff[1] −= (iCoeff[0] >> 10)	
}	

9.9.8.7 T2x2hPOST( )

The function T2x2hPOST( ) is specified by the pseudocode in Table 174.

Table 174 — Pseudocode for function T2x2hPOST( )

T2x2hPOST(iCoeff[ ]) {	**Reference**
iCoeff[1] −= iCoeff[2]	
iCoeff[0] += (iCoeff[3] * 3 + 4) >> 3	
iCoeff[3] −= (iCoeff[1] >> 1)	
iCoeff[2] = ((iCoeff[0] − iCoeff[1]) >> 1) − iCoeff[2]	
valT1 = iCoeff[2]	
iCoeff[2] = iCoeff[3]	
iCoeff[3] = valT1	
iCoeff[0] −= iCoeff[3]	
iCoeff[1] += iCoeff[2]	
}	

9.9.8.8 InvToddoddPOST( )

The function InvToddoddPOST( ) is specified by the pseudocode in Table 175.

Table 175 — Pseudocode for function InvToddoddPOST( )

InvToddoddPOST(iCoeff[ ]) {	**Reference**
iCoeff[3] += iCoeff[0]	
iCoeff[2] −= iCoeff[1]	
valT1 = iCoeff[3] >> 1	
valT2 = iCoeff[2] >> 1	
iCoeff[0] −= valT1	
iCoeff[1] += valT2	
iCoeff[0] −= (iCoeff[1] * 3 + 6) >> 3	
iCoeff[1] += (iCoeff[0] * 3 + 2) >> 2	
iCoeff[0] −= (iCoeff[1] * 3 + 4) >> 3	
iCoeff[1] −= valT2	
iCoeff[0] += valT1	
iCoeff[2] += iCoeff[1]	
iCoeff[3] −= iCoeff[0]	
}	

9.10 Output formatting

The final stage of the decoder consists of converting the sample values reconstructed in the internal format to the intended output format as specified in subclause 9.10.2.

9.10.1 Overview

First, the decoder may be required to perform upsampling to obtain an intermediate YUV444 format. Next, colour format conversion is applied to convert the internal colour formats to output formats. The colour format conversions are specified below. A bias is added to the sample values, to re-centre the values around the nominal value for a neutral or zero intensity representation. When the scaling mode is used, on the decoder side, the values are rounded down after colour conversion. For high numerical range formats (BD16, BD16S, BD32S and BD32F), the internal integer representations need to be converted to output representations. Finally, the values are clipped to fit the appropriate range.

9.10.2 Output formatting stage

At the completion of the transform and overlap filtering, the sample values for the image are reconstructed in an internal colour format and an internal two's complement integer representation. The output formatting stage converts the decoded image plane data into a representation specified by the OUTPUT_CLR_FMT and the output bit depth. In the specification of output formatting, the term INTERNAL_CLR_FMT refers to the corresponding syntax element of the primary image plane.

The output formatting process is specified for the combinations of OUTPUT_BITDEPTH and OUTPUT_CLR_FMT that are listed in Table 176.

In this table, "+" indicates that output formatting is specified for the corresponding combinations of OUTPUT_BIT_DEPTH and OUTPUT_CLR_FMT. The combination of OUTPUT_BIT_DEPTH and OUTPUT_CLR_FMT shall not have the value corresponding to empty cells.

Table 176 — Conformance-specified output formatting combinations of OUTPUT_BITDEPTH and OUTPUT_CLR_FMT

OUTPUT_BITDEPTH	OUTPUT_CLR_FMT								
	YONLY	YUV420	YUV422	YUV444	RGB	RGBE	CMYK	CMYKDIRECT	NCOMPONENT
BD1WHITE1	+								
BD1BLACK1	+								
BD5					+				
BD565					+				
BD8	+	+	+	+	+	+	+	+	+
BD10			+	+	+				
BD16	+		+	+	+		+	+	+
BD16S	+			+	+				
BD16F	+				+				
BD32S	+				+				
BD32F	+				+				

The output formatting stage consists of several sub-processes that are performed as specified in Table 177.

Table 177 — Pseudocode for function OutputFormatting( )

OutputFormatting( ) {	**Reference**
SamplingConversion( )	9.10.3
ConvertInternalToOutputClrFmt( )	9.10.4.1
AddBias( )	9.10.5
ComputeScaling( )	9.10.6
PostscalingProcess( )	9.10.7.1
ClippingAndPackingStage( )	9.10.8
}	

9.10.3 Sampling conversion

9.10.3.1 General

The sampling conversion process is specified in Table 178.

The combinations of INTERNAL_CLR_FMT and OUTPUT_CLR_FMT for which sampling conversions are specified for conformance purposes are specified in Table 179. In this table, "+" indicates that no sampling conversion is required. It is a requirement of codestream conformance to this document that the combination of INTERNAL_CLR_FMT and OUTPUT_CLR_FMT shall not have a value corresponding to any empty cell in Table 179.

In the illustrated case in Table 178 in which upsampling is specified both vertically and horizontally for INTERNAL_CLR_FMT equal to YUV420, the upsampling process to be performed by the decoder shall produce an array of two-dimensionally upsampled samples at the index values for which such samples are produced as specified by Table 178. However, decoders may use alternative upsampling methods (different from that specified by Table 178) – the actual filtering method used to produce the values of the entries in the upsampled array is outside the scope of this document. The particular filtering method specified by Table 178 is an example of how such upsampled array values may be produced. For example, upsampling may be applied both vertically and horizontally as a single process, or the relative ordering of the vertical and horizontal upsampling may be switched.

NOTE When TILING_FLAG is equal to TRUE and the transform processing does not cross tile boundaries (due either to HARD_TILING_FLAG being equal to TRUE or OVERLAP_MODE being equal to 0), the example upsampling method illustrated in Table 178 for cases with INTERNAL_CLR_FMT equal to YUV422 or YUV420 will produce an upsampled image in which the output samples next to tile boundaries may be affected by the values of decoded samples in other tiles. For many applications, it would be desirable to instead design the upsampling process to be performed separately within each tile in order to avoid this cross-tile dependency.

Table 178 — Pseudocode for function SamplingConversion( )

SamplingConversion( ) {	**Reference**
if (((INTERNAL_CLR_FMT = =YUV422) | | (INTERNAL_CLR_FMT = =YUV420)) && ((OUTPUT_CLR_FMT = = YUV444) | | (OUTPUT_CLR_FMT = =RGB)) {	
if (INTERNAL_CLR_FMT = = YUV420)	
Upsample( ) in the vertical direction	9.10.3.2
if ((INTERNAL_CLR_FMT = = YUV422) | | (INTERNAL_CLR_FMT = = YUV420))	
Upsample( ) in the horizontal direction	9.10.3.2
}	
}	

Table 179 — Conformance-specified sampling conversions

OUTPUT_CLR_FMT	INTERNAL_CLR_FMT					
	YONLY	YUV420	YUV422	YUV444	YUVK	NCOMPONENT
YONLY	+					
YUV420		+				
YUV422			+			
YUV444	+	Upsample( ) in the vertical and horizontal directions	Upsample( ) in the horizontal direction	+		
RGB with OUTPUT_BITDEPTH equal to BD5, BD565, BD8, BD10, BD16, BD16S or BD32S	+	Upsample( ) in the vertical and horizontal directions	Upsample( ) in the horizontal direction	+		
RGB with OUTPUT_BITDEPTH equal to BD16F or BD32F				+		
RGBE				+		
CMYK					+	
CMYKDIRECT					+	
NCOMPONENT						+

9.10.3.2 Upsample( )

In the chroma upsampling function, for the chroma component i (1 <= i < NumComponents), let iOriArray[ ] be the original sample array before upsampling, and iIntArray[ ] be the upsampled array. If upsampling is performed in the horizontal direction, iOriArray[ ] is one input sample row of length ExtendedWidth[i] and iIntArray[ ] is one output sample row of length ExtendedWidth[0], and the variable iOriLength is set equal to ExtendedWidth[i]. Otherwise, iOriArray[ ] is one input sample column of length ExtendedHeight[i] and iIntArray[ ] is one output sample column of length ExtendedHeight[0], and iOriLength is set equal to ExtendedHeight[i].

The upsampling process to be performed by the decoder shall produce an array of upsampled samples at the index values for which such samples are produced as specified by Table 180. However, decoders may use alternative upsampling methods (different from that specified by Table 180) – the actual filtering method used to produce the values of the entries in the upsampled array is outside the scope of this document. The particular filtering method specified by Table 180 is an example of how such upsampled array values may be produced. For example, a different type of filtering or a different number of taps may be used during the upsampling process than the two-tap filtering specified by Table 180.

Table 180 — Pseudocode for function Upsample( )

Upsample( ) {	Reference
for (k = 0; k < iOriLength; k++) {	
iIntArray[2 * k] = ((iH[2] * iOriArray[Max(0, k − 1)] + iH[3] * iOriArray[k] + 4) >> 3)	
iIntArray[2 * k + 1] = ((iH[0] * iOriArray[k] + iH[1] * iOriArray[Min(iOriLength − 1, k + 1)] + 4) >> 3)	
}	
}	

The values of the filter coefficients iH[0], iH[1], iH[2], iH[3] for the chroma positions are specified by Table 181 as a function of the variable chromaCentering. If Upsample( ) is applied in the horizontal direction, chromaCentering is set equal to CHROMA_CENTERING_X; otherwise, it is set equal to CHROMA_CENTERING_Y.

Table 181 — Upsampling filter coefficient for different chroma positions

chromaCentering	iH[0]	iH[1]	iH[2]	iH[3]
0	4	4	0	8
1	5	3	1	7
2	6	2	2	6
3	7	1	3	5
4	8	0	4	4

9.10.4 Conversion from INTERNAL_CLR_FMT to OUTPUT_CLR_FMT

9.10.4.1 Overview

The conversion process proceeds as specified in Table 182.

Table 182 — Pseudocode for function ConvertInternalToOutputClrFmt( )

ConvertInternalToOutputClrFmt( ) {	Reference
if ((INTERNAL_CLR_FMT = = YUVK) && (OUTPUT_CLR_FMT = = CMYK))	
InvColorFmtConvert3( )	9.10.4.4
else if ((INTERNAL_CLR_FMT = = YUVK) && (OUTPUT_CLR_FMT = = CMYKDIRECT))	
InvColorFmtConvert4( )	9.10.4.5
else if ((INTERNAL_CLR_FMT = = YONLY) && (OUTPUT_CLR_FMT = = RGB))	
InvColorFmtConvert1( )	9.10.4.2
else if (((INTERNAL_CLR_FMT = = YUV444) | | (INTERNAL_CLR_FMT = = YUV422) | | (INTERNAL_CLR_FMT = = YUV420)) && ((OUTPUT_CLR_FMT = = RGB) | | (OUTPUT_CLR_FMT = = RGBE)))	
InvColorFmtConvert2( )	9.10.4.3
else if ((INTERNAL_CLR_FMT = = YONLY) && ((OUTPUT_CLR_FMT = = YUV444) | | (OUTPUT_CLR_FMT = = YUV422) | | (OUTPUT_CLR_FMT = = YUV420))) {	
if (OUTPUT_CLR_FMT = = YUV420)	
chromaHeight = ExtendedHeight[0] / 2	
else	
chromaHeight = ExtendedHeight[0]	
if ((OUTPUT_CLR_FMT = = YUV422) | | (OUTPUT_CLR_FMT = = YUV420))	
chromaWidth = ExtendedWidth[0] / 2	
else	
chromaWidth = ExtendedWidth[0]	
for (i = 1; i < 3; i ++)	
for (y = 0; y < chromaHeight; y++)	
for (x = 0; x < chromaWidth; x++)	

ConvertInternalToOutputClrFmt( ) {	Reference
ImagePlane[i][x][y] = 0 /* Ensure that chroma is inferred as zero */	
}	
}	

The combinations of INTERNAL_CLR_FMT and OUTPUT_CLR_FMT for which colour format conversions are specified for conformance purposes are specified in Table 183. In this table, "+" indicates that no colour format conversion is required. For cases that require colour format conversion, the function name for the conversion process is specified in the table cell. It is a requirement of codestream conformance to this document that the combination of INTERNAL_CLR_FMT and OUTPUT_CLR_FMT shall not have a value corresponding to any empty cell in Table 183.

Table 183 — Conformance-specified colour format conversions

OUTPUT_CLR_FMT	INTERNAL_CLR_FMT					
	YONLY	YUV420	YUV422	YUV444	YUVK	NCOMPONENT
YONLY	+					
YUV420		+				
YUV422			+			
YUV444	+	+	+	+		
RGB with OUTPUT_BITDEPTH equal to BD5, BD565, BD8, BD10, BD16, BD16S or BD32S	InvColorFmt Convert1( )	InvColorFmt Convert2( )	InvColorFmt Convert2( )	InvColorFmt Convert2( )		
RGB with OUTPUT_BITDEPTH equal to BD16F or BD32F				InvColorFmt Convert2( )		
RGBE				InvColorFmt Convert2( )		
CMYK					InvColorFmt Convert3( )	
CMYKDIRECT					InvColorFmt Convert4( )	
NCOMPONENT						+

The pseudocode functions InvColorFmtConvert1( ), InvColorFmtConvert2( ), InvColorFmtConvert3( ), and InvColorFmtConvert4( ) that are referred to in Table 183 are specified in subclause 9.10.4.2, subclause 9.10.4.3, subclause 9.10.4.4, and subclause 9.10.4.5, respectively.

9.10.4.2 InvColorFmtConvert1( )

The operations in InvColorFmtConvert1( ) are specified in Table 184.

Table 184 — Pseudocode for function InvColorFmtConvert1( )

InvColorFmtConvert1( ) {	Reference
for (y = 0; y < ExtendedHeight[0]; y++)	
for (x = 0; x < ExtendedWidth[0]; x++) {	
/* ImagePlane[0][x][y] G = Y */	
ImagePlane[1][x][y] = ImagePlane[0][x][y] /* R = Y */	
ImagePlane[2][x][y] = ImagePlane[0][x][y] /* B = Y */	
}	
}	

9.10.4.3 InvColorFmtConvert2( )

The operations in InvColorFmtConvert2( ) are specified in Table 185.

Table 185 — Pseudocode for function InvColorFmtConvert2( )

InvColorFmtConvert2( ) {	**Reference**
for (y = 0; y < ExtendedHeight[0]; y++)	
for (x = 0; x < ExtendedWidth[0]; x++) {	
tempT = −ImagePlane[1][x][y] /* t = −U */	
arrayOut[1] = ImagePlane[0][x][y] − Floor(tempT ÷ 2) /* G = Y − Floor(t ÷ 2) */	
arrayOut[0] = tempT + arrayOut[1] − Ceiling(ImagePlane[2][x][y] ÷ 2) /* R = t + G − Ceiling(V ÷ 2) */	
arrayOut[2] = ImagePlane[2][x][y] + arrayOut[0] /* B = V + R */	

if ((OUTPUT_BITDEPTH = = BD5 | | OUTPUT_BITDEPTH = = BD565 | |
OUTPUT_BITDEPTH = = BD10) &&
!RED_BLUE_NOT_SWAPPED_FLAG) {	

tempT = arrayOut[0]	
arrayOut[0] = arrayOut[2]	
arrayOut[2] = tempT	
}	
for (i = 0; i < 3; i++)	
ImagePlane[i][x][y] = arrayOut[i]	
}	
}	

9.10.4.4 InvColorFmtConvert3( )

The operations in InvColorFmtConvert3( ) are specified in Table 186.

Table 186 — Pseuocode for function InvColorFmtConvert3( )

InvColorFmtConvert3( ) {	**Reference**
for (y = 0; y < ExtendedHeight[0]; y++)	
for (x = 0; x < ExtendedWidth[0]; x++) {	
arrayOut[3] = ImagePlane[3][x][y] + Floor(ImagePlane[0][x][y] ÷ 2) /* k = K + Floor(Y ÷ 2) */	
arrayOut[1] = arrayOut[3] − ImagePlane[0][x][y] − Floor(ImagePlane[1][x][y] ÷ 2) /* m = k − Y − Floor(U ÷ 2) */	
arrayOut[0] = ImagePlane[1][x][y] + arrayOut[1] + Floor(ImagePlane[2][x][y] ÷ 2) /* c = U + m + Floor(V ÷ 2) */	
arrayOut[2] = arrayOut[0] − ImagePlane[2][x][y] /* y = c − V */	
for (i = 0; i < 4; i++)	
ImagePlane[i][x][y] = arrayOut[i]	
}	
}	

9.10.4.5 InvColorFmtConvert4( )

The operations in InvColorFmtConvert4( ) are specified in Table 187.

Table 187 — Pseudocode for function InvColorFmtConvert4( )

InvColorFmtConvert4( ) {	**Reference**
for (y = 0; y < ExtendedHeight[0]; y++)	
for (x = 0; x < ExtendedWidth[0]; x++) {	
arrayOut[3] = ImagePlane[0][x][y] /* k = Y */	
arrayOut[1] = ImagePlane[2][x][y] /* m = V */	
arrayOut[0] = ImagePlane[1][x][y] /* c = U */	
arrayOut[2] = ImagePlane[3][x][y] /* y = K */	
for (i = 0; i < 4; i++)	
ImagePlane[i][x][y] = arrayOut[i]	
}	
}	

9.10.5 AddBias( )

The function AddBias( ) specified in Table 188 performs the computation and addition of bias to the sample values.

Table 188 — Pseudocode for function AddBias( )

AddBias( ) {	**Reference**
if (SCALED_FLAG)	
iScale = 3	
else	
iScale = 0	
if (OUTPUT_BITDEPTH = = BD5)	
iBias = (1 << 4)	
else if (OUTPUT_BITDEPTH = = BD565)	
iBias = (1 << 5)	
else if (OUTPUT_BITDEPTH = = BD8)	
iBias = (1 << 7)	
else if (OUTPUT_BITDEPTH = = BD10)	
iBias = (1 << 9)	
else if (OUTPUT_BITDEPTH = = BD16)	
iBias = (1 << 15)	
else	
iBias = 0	
if ((OUTPUT_BITDEPTH = = BD16) | | (OUTPUT_BITDEPTH = = BD16S) | | (OUTPUT_BITDEPTH = = BD32S))	
iBias = (iBias >> SHIFT_BITS)	
if ((OUTPUT_CLR_FMT = = RGB) | | (OUTPUT_CLR_FMT = = YUV444) | | (OUTPUT_CLR_FMT = = YUV422) | | (OUTPUT_CLR_FMT = = YUV420) | | (OUTPUT_CLR_FMT = = YONLY) | | (OUTPUT_CLR_FMT = = NCOMPONENT) | | (OUTPUT_CLR_FMT = = CMYKDIRECT)) {	
if ((OUTPUT_CLR_FMT = = RGB) | | (OUTPUT_CLR_FMT = = YUV444) | | (OUTPUT_CLR_FMT = = YUV422) | | (OUTPUT_CLR_FMT = = YUV420))	
outputComponents = 3	
else	
outputComponents = NumComponents	
for (i = 0; i < outputComponents; i++) {	
if ((i > 0) && (OUTPUT_CLR_FMT = = YUV420))	
outputHeight = ExtendedHeight[0] / 2	
else	
outputHeight = ExtendedHeight[0]	
if ((i > 0) && ((OUTPUT_CLR_FMT = = YUV422) | | (OUTPUT_CLR_FMT = = YUV420)))	

<table>
<tr><th>AddBias( ) {</th><th>Reference</th></tr>
<tr><td>outputWidth = ExtendedWidth[0] / 2</td><td></td></tr>
<tr><td>else</td><td></td></tr>
<tr><td>outputWidth = ExtendedWidth[0]</td><td></td></tr>
<tr><td>for (y = 0; y < outputHeight; y++)</td><td></td></tr>
<tr><td>for (x = 0; x < outputWidth; x++)</td><td></td></tr>
<tr><td>ImagePlane[i][x][y] += (iBias << iScale)</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>} else if (OUTPUT_CLR_FMT = = CMYK) {</td><td></td></tr>
<tr><td>for (i = 0; i < 3; i++)</td><td></td></tr>
<tr><td>for (y = 0; y < ExtendedHeight[0]; y++)</td><td></td></tr>
<tr><td>for (x = 0; x < ExtendedWidth[0]; x++)</td><td></td></tr>
<tr><td>ImagePlane[i][x][y] += ((iBias >> 1) << iScale) /* c, m, y */</td><td></td></tr>
<tr><td>for (y = 0; y < ExtendedHeight[0]; y++)</td><td></td></tr>
<tr><td>for (x = 0; x < ExtendedWidth[0]; x++)</td><td></td></tr>
<tr><td>ImagePlane[3][x][y] −= ((iBias >> 1) << iScale) /* k */</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>}</td><td></td></tr>
</table>

9.10.6 ComputeScaling( )

The function ComputeScaling( ) specified in Table 189 performs the computation of the scaling factor iScale, and the rounding factor iRoundingFactor, and modifies sample values based on these two factors.

Table 189 — Pseudocode for function ComputeScaling( )

<table>
<tr><th>ComputeScaling( ) {</th><th>Reference</th></tr>
<tr><td>iScale = 0</td><td></td></tr>
<tr><td>iRoundingFactor = 0</td><td></td></tr>
<tr><td>if (SCALED_FLAG) {</td><td></td></tr>
<tr><td>iScale = 3</td><td></td></tr>
<tr><td>if ((OUTPUT_BITDEPTH = = BD5) | | (OUTPUT_BITDEPTH = = BD565) | |
(OUTPUT_BITDEPTH = = BD8) | | (OUTPUT_BITDEPTH = = BD10) | |
(OUTPUT_BITDEPTH = = BD16S) | | (OUTPUT_BITDEPTH = = BD16F) | |
(OUTPUT_BITDEPTH = = BD32S) | | (OUTPUT_BITDEPTH = = BD32F))</td><td></td></tr>
<tr><td>iRoundingFactor = 3</td><td></td></tr>
<tr><td>else if ((OUTPUT_BITDEPTH = = BD1WHITE1) | |
(OUTPUT_BITDEPTH = = BD1BLACK1) | | (OUTPUT_BITDEPTH = = BD16))</td><td></td></tr>
<tr><td>iRoundingFactor = 4</td><td></td></tr>
<tr><td>}</td><td></td></tr>
<tr><td>if ((OUTPUT_CLR_FMT = = RGB) | | (OUTPUT_CLR_FMT = = RGBE) | |
(OUTPUT_CLR_FMT = = YUV444) | | (OUTPUT_CLR_FMT = = YUV422) | |
(OUTPUT_CLR_FMT = = YUV420))</td><td></td></tr>
<tr><td>outputComponents = 3</td><td></td></tr>
<tr><td>else</td><td></td></tr>
<tr><td>outputComponents = NumComponents</td><td></td></tr>
<tr><td>for (i = 0; i < outputComponents; i++) {</td><td></td></tr>
<tr><td>if ((i > 0) && (OUTPUT_CLR_FMT = = YUV420))</td><td></td></tr>
<tr><td>outputHeight = ExtendedHeight[0] / 2</td><td></td></tr>
<tr><td>else</td><td></td></tr>
<tr><td>outputHeight = ExtendedHeight[0]</td><td></td></tr>
<tr><td>if ((i > 0) &&
((OUTPUT_CLR_FMT = = YUV422) | | (OUTPUT_CLR_FMT = = YUV420)))</td><td></td></tr>
<tr><td>outputWidth = ExtendedWidth[0] / 2</td><td></td></tr>
<tr><td>else</td><td></td></tr>
<tr><td>outputWidth = ExtendedWidth[0]</td><td></td></tr>
<tr><td>if ((OUTPUT_BITDEPTH = = BD565)) && (i != 1))</td><td></td></tr>
<tr><td>jScale = iScale + 1</td><td></td></tr>
<tr><td>else</td><td></td></tr>
<tr><td>jScale = iScale</td><td></td></tr>
<tr><td>for (y = 0; y < outputHeight; y++)</td><td></td></tr>
</table>

ComputeScaling( ) {	Reference
for (x = 0; x < outputWidth; x++)	
ImagePlane[i][x][y] = ((ImagePlane[i][x][y] + iRoundingFactor) >> jScale)	
}	
}	

9.10.7 Postscaling process

9.10.7.1 Overview

The function PostscalingProcess( ) is specified in Table 190.

Table 190 — Pseudocode for function PostscalingProcess( )

PostscalingProcess( ) {	Reference
if (OUTPUT_CLR_FMT = = RGBE)	
for (y = 0; y < ExtendedHeight[0]; y++)	
for (x = 0; x < ExtendedWidth[0]; x++) {	
for (k = 0; k < 3; k++)	
localArrayIn[k] = ImagePlane[k][x][y]	
PostScalingF2(localArrayOut[ ], localArrayIn[ ]) /* Produces 4 outputs for 3 inputs */	9.10.7.4
for (k = 0; k < 4; k++)	
ImagePlane[k][x][y] = localArrayOut[k]	
}	
else {	
if ((OUTPUT_CLR_FMT = = RGB) | | (OUTPUT_CLR_FMT = = YUV444) | | (OUTPUT_CLR_FMT = = YUV422) | | (OUTPUT_CLR_FMT = = YUV420))	
outputComponents = 3	
else	
outputComponents = NumComponents	
for (i = 0; i < outputComponents; i++) {	
if ((i > 0) && (OUTPUT_CLR_FMT = = YUV420))	
outputHeight = ExtendedHeight[0] / 2	
else	
outputHeight = ExtendedHeight[0]	
if ((i > 0) && ((OUTPUT_CLR_FMT = = YUV422) | | (OUTPUT_CLR_FMT = = YUV420)))	
outputWidth = ExtendedWidth[0] / 2	
else	
outputWidth = ExtendedWidth[0]	
if ((OUTPUT_BITDEPTH = = BD16) | | (OUTPUT_BITDEPTH = = BD16S) | | (OUTPUT_BITDEPTH = = BD32S))	
for (y = 0; y < outputHeight; y++)	
for (x = 0; x < outputWidth; x++)	
ImagePlane[i][x][y] = PostScalingInt(ImagePlane[i][x][y])	9.10.7.2
else if ((OUTPUT_BITDEPTH = = BD32F) | | (OUTPUT_BITDEPTH = = BD16F))	
for (y = 0; y < outputHeight; y++)	
for (x = 0; x < outputWidth; x++)	
ImagePlane[i][x][y] = PostScalingFl(ImagePlane[i][x][y])	9.10.7.3
}	
}	
}	

9.10.7.2 PostScalingInt( )

The sample values are left-shifted by the amount determined by SHIFT_BITS. For input value inX, the output shifted value outX is determined as specified in Table 191.

Table 191 — Pseudocode for function PostScalingInt( )

PostScalingInt(inX) {	**Reference**
outX = inX << SHIFT_BITS	
return outX	
}	

NOTE In this manner, the output is moved from a 27-bit or 24-bit nominal range scaling to the range scaling specified for image reconstruction. The 27-bit range scaling applies when the data is scaled, and the 24-bit range scaling applies when the data is unscaled.

9.10.7.3 PostScalingFl( )

When OUTPUT_BITDEPTH is equal to BD32F or BD16F, the integer sample value iX is converted to a value fV that can be interpreted as a floating point representation.

The PostScalingFl( ) process computes the value fV as specified in Table 192.

Table 192 — Pseudocode for function PostScalingFl( )

PostScalingFl(iX) {	**Reference**
if (iX < 0)	
iS = 1	
else	
iS = 0	
if (OUTPUT_BITDEPTH = = BD16F) {	
iEM = Min(Abs(iX), 32767)	
fV = ((iS << 15)	iEM) /* Concatenate these fields*/	
} else { /* OUTPUT_BITDEPTH = = BD32F */	
iX = Abs(iX)	
iE = (iX >> LEN_MANTISSA)	
iM = ((iX & ((1 << LEN_MANTISSA) − 1))	(1 << LEN_MANTISSA))	
if (iE = = 0) {	
iM ^= (1 << LEN_MANTISSA)	
iE = 1	
}	
iE = iE − EXP_BIAS + 127	
while ((iM < (1 << LEN_MANTISSA)) && (iE > 1) && (iM > 0)) {	
iE −= 1	
iM <<= 1	
}	
if (iM < (1 << LEN_MANTISSA))	
iE = 0	
else	
iM ^= (1 << LEN_MANTISSA)	
iM <<= (23 − LEN_MANTISSA)	
fV = ((iS << 31)	(iE << 23)	iM) /* Concatenate these fields */	
}	
return fV	
}	

9.10.7.4 PostScalingF2( )

When OUTPUT_CLR_FMT is equal to RGBE, the three integer sample values of array arrayIn[ ] (R, G, and B) are converted to an array arrayOut[ ] of four integer values forming the RGBE representation (Rrgbe, Grgbe, Brgbe and Ergbe). The function PostScalingF2( ) specified in Table 193 performs the conversion.

Table 193 — Pseudocode for function PostScalingF2( )

PostScalingF2(arrayOut[ ], arrayIn[ ]) {	**Reference**
/* arrayIn[ ]= {R, G, B} */	
/* arrayOut[ ]= {Rrgbe, Grgbe, Brgbe, Ergbe} */	
if (arrayIn[0] <= 0) {	
arrayOut[0] = 0	
iEr = 0	
} else if ((arrayIn[0] >> 7) > 1) {	
arrayOut[0] = (arrayIn[0] & 0x7F) + 128	
iEr = (arrayIn[0] >> 7)	
} else {	
arrayOut[0] = arrayIn[0]	
iEr = 1	
}	
if (arrayIn[1] <= 0) {	
arrayOut[1] = 0	
iEg = 0	
} else if ((arrayIn[1] >> 7) > 1) {	
arrayOut[1] = (arrayIn[1] & 0x7F) + 128	
iEg = (arrayIn[1] >> 7)	
} else {	
arrayOut[1] = arrayIn[1]	
iEg = 1	
}	
if (arrayIn[2] <= 0) {	
arrayOut[2] = 0	
iEb = 0	
} else if ((arrayIn[2] >> 7) > 1) {	
arrayOut[2] = (arrayIn[2] & 0x7F) + 128	
iEb = (arrayIn[2] >> 7)	
} else {	
arrayOut[2] = arrayIn[2]	
iEb = 1	
}	
arrayOut[3] = Max(iEr, Max(iEg, iEb))	
if (arrayOut[3] > iEr) {	
iShift = (arrayOut[3] − iEr)	
arrayOut[0] = ((2 * arrayOut[0] + 1) >> (iShift + 1))	
}	
if (arrayOut[3] > iEg) {	
iShift = (arrayOut[3] − iEg)	
arrayOut[1] = ((2 * arrayOut[1] + 1) >> (iShift + 1))	
}	
if (arrayOut[3] > iEb) {	
iShift = (arrayOut[3] − iEb)	
arrayOut[2] = ((2 * arrayOut[2] + 1) >> (iShift + 1))	
}	
}	

9.10.8 Clipping and packing stage

9.10.8.1 General

The ClippingAndPackingStage( ) process by which clipping, packing, and windowing are performed is specified in Table 194. The clipping ensures that the sample values are constrained to the appropriate range. The packing process packs multiple samples into single variables for some values of OUTPUT_BITDEPTH. The windowing process uses the LEFT_MARGIN, TOP_MARGIN, WIDTH_MINUS1 and HEIGHT_MINUS1 syntax elements to discard the data outside of the image area that is to be output.

Table 194 — Pseudocode for function ClippingAndPackingStage( )

ClippingAndPackingStage( ) {	**Reference**
if((OUTPUT_CLR_FMT = = RGB) && ((OUTPUT_BITDEPTH = = BD5) | | (OUTPUT_BITDEPTH = = BD565) | | (OUTPUT_BITDEPTH = = BD10))) /* Packed RGB */	
outputArrays = 1	
else if ((OUTPUT_CLR_FMT = = RGB) | | (OUTPUT_CLR_FMT = = YUV444) | | (OUTPUT_CLR_FMT = = YUV422) | | (OUTPUT_CLR_FMT = = YUV420))	
outputArrays = 3	
else if (OUTPUT_CLR_FMT = = RGBE)	
outputArrays = 4	
else	
outputArrays = NumComponents	
for (i = 0; i < outputArrays; i++) {	
if ((i > 0) && (OUTPUT_CLR_FMT = = YUV420)) {	
outputHeight = (HEIGHT_MINUS1 + 1) / 2	
n = TOP_MARGIN / 2	
} else {	
outputHeight = HEIGHT_MINUS1 + 1	
n = TOP_MARGIN	
}	
if ((i > 0) && ((OUTPUT_CLR_FMT = = YUV422) | | (OUTPUT_CLR_FMT = = YUV420))) {	
outputWidth = (WIDTH_MINUS1 + 1) / 2	
m = LEFT_MARGIN / 2	
} else {	
outputWidth = WIDTH_MINUS1 + 1	
m = LEFT_MARGIN	
}	
if ((OUTPUT_BITDEPTH = = BD8) | | (OUTPUT_BITDEPTH = = BD16) | | (OUTPUT_BITDEPTH = = BD16S))	
for (y = 0; y < outputHeight; y++)	
for (x = 0; x < outputWidth; x++)	
ImagePlane[i][x][y] = ClippingBasic(ImagePlane[i][x + m][y + n])	9.10.8.2
else if (OUTPUT_BITDEPTH = = BD565)	
for (y = 0; y < outputHeight; y++)	
for (x = 0; x < outputWidth; x++)	
ImagePlane[i][x][y] = ClipAndPackBD565(ImagePlane[0][x + m][y + n], ImagePlane[1][x + m][y + n], ImagePlane[2][x + m][y + n])	9.10.8.3
else if (OUTPUT_BITDEPTH = = BD5)	
for (y = 0; y < outputHeight; y++)	
for (x = 0; x < outputWidth; x++)	
ImagePlane[i][x][y] = ClipAndPackBD5(ImagePlane[0][x + m][y + n], ImagePlane[1][x + m][y + n], ImagePlane[2][x + m][y + n])	9.10.8.4
else if (OUTPUT_BITDEPTH = = BD10)	
if (OUTPUT_CLR_FMT = = RGB)	
for (y = 0; y < outputHeight; y++)	
for (x = 0; x < outputWidth; x++)	
ImagePlane[i][x][y] = ClipAndPackBD10(ImagePlane[0][x + m][y + n], ImagePlane[1][x + m][y + n], ImagePlane[2][x + m][y + n])	9.10.8.5

ClippingAndPackingStage( ) {	Reference
else	
for (y = 0; y < outputHeight; y++)	
for (x = 0; x < outputWidth; x++)	
ImagePlane[i][x][y] = ClipAndPackBD10(ImagePlane[i][x + m][y + n], 0, 0)	9.10.8.5
else if ((OUTPUT_BITDEPTH = = BD1WHITE1) | | (OUTPUT_BITDEPTH = = BD1BLACK1))	
for (y = 0; y < outputHeight; y++)	
for (x = 0; x < outputWidth; x+=8) { /* Up to 8 samples are packed into each output byte */	
pNum = Min(outputWidth − x, 8) /* Number of values to pack into current output byte */	
for (p = pNum; m < 8; p++) /* Prevent junk beyond valid image data in array */	
valList[p] = 0 /* Actual value does not matter in this region */	
for (p = 0; p < pNum; p++)	
valList[p] = ImagePlane[i][x + m + p][y + n]	
ImagePlane[i][x >> 3][y] = ClipAndPackBD1BorW(valList)	9.10.8.6
}	
else /* OUTPUT_BITDEPTH equal to BD16F, BD32F, or BD32S */	
for (y = 0; y < outputHeight; y++)	
for (x = 0; x < outputWidth; x++)	
ImagePlane[i][x][y] = ImagePlane[i][x + m][y + n]	
}	
}	

9.10.8.2 ClippingBasic( )

The pseudocode function ClippingBasic( ) is specified in Table 195.

Table 195 — Pseudocode for function ClippingBasic( )

ClippingBasic(iSample) {
if (OUTPUT_BITDEPTH = = BD8) {
iLow = 0
iHigh = 255
} else if (OUTPUT_BITDEPTH = = BD16) {
iLow = 0
iHigh = 65535
} else if (OUTPUT_BITDEPTH = = BD16S) {
iLow = −32768
iHigh = 32767
}
iResult = Clip(iSample, iLow, iHigh) /* Clip within the range iLow to iHigh */
return iResult
}

9.10.8.3 ClipAndPackBD565( )

The pseudocode function ClipAndPackBD565( ) is specified in Table 196.

Table 196 — Pseudocode for function ClipAndPackBD565( )

ClipAndPackBD565(i0, i1, i2) {
iLow = 0
iHigh = 31
i0 = Clip(i0, iLow, iHigh)
i2 = Clip(i2, iLow, iHigh)
iLow = 0
iHigh = 63
i1 = Clip(i1, iLow, iHigh)
iResult = i0 + (i1 << 5) + (i2 << 11)
return iResult
}

9.10.8.4 ClipAndPackBD5( )

The pseudocode function ClipAndPackBD5( )is specified in Table 197.

Table 197 — Pseudocode for function ClipAndPackBD5( )

ClipAndPackBD5(i0, i1, i2) {
iLow = 0
iHigh = 31
i0 = Clip(i0, iLow, iHigh)
i2 = Clip(i2, iLow, iHigh)
i1 = Clip(i1, iLow, iHigh)
iResult = i0 + (i1 << 5) + (i2 << 10)
return iResult
}

9.10.8.5 ClipAndPackBD10( )

The pseudocode function ClipAndPackBD10( ) is specified in Table 198.

Table 198 — Pseudocode for function ClipAndPackBD10( )

ClipAndPackBD10(iSample0, iSample1, iSample2) {
iLow = 0
iHigh = 1023
if (OUTPUT_CLR_FMT = = RGB) {
i0 = Clip(iSample0, iLow, iHigh)
i1 = Clip(iSample1, iLow, iHigh)
i2 = Clip(iSample2, iLow, iHigh)
iResult = i0 + (i1 << 10) + (i2 << 20)
} else
iResult = Clip(iSample0, iLow, iHigh)
return iResult
}

9.10.8.6 ClipAndPackBD1BorW( )

The pseudocode function ClipAndPackBD1BorW( ) is specified in Table 199.

Table 199 — Pseudocode for function ClipAndPackBD1BorW( )

```
ClipAndPackBD1BorW(valList) {
    /* valList[0] holds the value associated with the first sample value in the scan order, and
       valList[7] holds the value associated with the last sample value in the scan order */
    valList[0] = Clip(valList[0], 0, 1)
    valList[1] = Clip(valList[1], 0, 1)
    valList[2] = Clip(valList[2], 0, 1)
    valList[3] = Clip(valList[3], 0, 1)
    valList[4] = Clip(valList[4], 0, 1)
    valList[5] = Clip(valList[5], 0, 1)
    valList[6] = Clip(valList[6], 0, 1)
    valList[7] = Clip(valList[7], 0, 1)
    if (OUTPUT_BITDEPTH = = BD1BLACK1)
       iResult = (1 − valList[7]) + ((1 − valList[6]) << 1) + ((1 − valList[5]) << 2) +
          ((1 − valList[4]) << 3) + ((1 − valList[3]) << 4) + ((1 − valList[2]) << 5) +
          ((1 − valList[1]) << 6) + ((1 − valList[0]) << 7)
    else /* OUTPUT_BITDEPTH = = BD1WHITE1 */
       iResult = valList[7] + (valList[6] << 1) + (valList[5] << 2) +
          (valList[4] << 3) + (valList[3] << 4) + (valList[2] << 5) +
          (valList[1] << 6) + (valList[0] << 7)
    return iResult
}
```

Annex A
(normative)

Tag-based file format

A.1 General

This annex specifies a format for files containing JPEG XR images. It uses syntax structures (IFD_ENTRY( ) structures as specified in subclause A.4) that each contain a syntax element (FIELD_TAG as specified in subclause A.4.2) that can be referred to as a tag. Therefore, this file format is referred to as being tag-based. The value of the tag serves as an identifier of the type of data contained in the syntax structure that is associated with the tag.

NOTE 1 The file format specified in this annex is based on that specified for use in ISO 12234-2 [4], ISO 12639 [5], Adobe TIFF 6.0 [7], and EXIF 2.2 [6], and is intended to provide a form of consistency and compatibility with those specifications – e.g. to enable the sharing of some functional components designed for reading, writing, and otherwise making use of such files.

NOTE 2 This specification of this file format does not preclude the existence of alternative file format specifications for files containing JPEG XR images.

NOTE 3 When a file is formatted as specified in this annex, in addition to the syntax structures that are specified by this annex, arbitrary data (formatted in a manner not specified by this annex) may also be present at locations within the file that lie between or beyond the locations in the file that contain the syntax structures specified by this annex.

NOTE 4 The use of the filename extension ".jxr" is suggested for files conforming to the file format specified in this annex.

The FILE_HEADER( ) syntax structure specified in subclause A.2 shall be present at the beginning of the file (at byte position 0).

The variable FileSizeInBytes is considered to be equal to the total number of bytes in the file. The method of determining the value of FileSizeInBytes is determined by the application and is not specified in this document. The value of FileSizeInBytes shall not exceed $2^{32} - 1$.

For purposes of this document, a decoder is assumed to be capable of either storing the entire file in random access memory or performing random access seek operations to access the data at arbitrary specified positions in the file.

A.2 FILE_HEADER( )

A.2.1 Syntax structure

The FILE_HEADER( ) syntax structure is specified by Table A.1.

Table A.1 — FILE_HEADER( ) syntax structure

FILE_HEADER( ) {	Descriptor	Reference
FIXED_FILE_HEADER_II_2BYTES	u(16)	A.2.2
FIXED_FILE_HEADER_0XBC_BYTE	u(8)	A.2.3
FILE_VERSION_ID	u(8)	A.2.4
FIRST_IFD_OFFSET	le(32)	A.2.5
}		

A.2.2 FIXED_FILE_HEADER_II_2BYTES

FIXED_FILE_HEADER_II_2BYTES shall be equal to 0x4949.

A.2.3 FIXED_FILE_HEADER_0XBC_BYTE

FIXED_FILE_HEADER_0XBC_BYTE shall be equal to 0xBC.

A.2.4 FILE_VERSION_ID

FILE_VERSION_ID shall be equal to 1. Other values of FILE_VERSION_ID are reserved for future use, as modified in additional parts or amendments, by ITU-T | ISO/IEC.

A.2.5 FIRST_IFD_OFFSET

FIRST_IFD_OFFSET specifies the byte position, relative to the beginning of the file, of the first IMAGE_FILE_DIRECTORY( ) syntax structure (subclause A.3) in the file. The value of FIRST_IFD_OFFSET shall be an integer multiple of 2.

A.3 IMAGE_FILE_DIRECTORY( )

A.3.1 Syntax structure

The IMAGE_FILE_DIRECTORY( ) syntax structure is specified by Table A.2.

Table A.2 — IMAGE_FILE_DIRECTORY( ) syntax structure

IMAGE_FILE_DIRECTORY( ) {	Descriptor	Reference
NUM_ENTRIES	le(16)	A.3.2
for (iNumEntries = 0; iNumEntries < NUM_ENTRIES; iNumEntries++)		
IFD_ENTRY( )		A.4
ZERO_OR_NEXT_IFD_OFFSET	le(32)	A.3.3
}		

A.3.2 NUM_ENTRIES

NUM_ENTRIES specifies the number of entries in the IMAGE_FILE_DIRECTORY( ) syntax structure. NUM_ENTRIES shall not be equal to 0. The value 0 for NUM_ENTRIES is reserved for future use by ITU-T | ISO/IEC.

A.3.3 ZERO_OR_NEXT_IFD_OFFSET

ZERO_OR_NEXT_IFD_OFFSET is interpreted as follows:

— If ZERO_OR_NEXT_IFD_OFFSET is equal to 0, this indicates that no additional IMAGE_FILE_DIRECTORY( ) syntax structures are present in the file.

— Otherwise, ZERO_OR_NEXT_IFD_OFFSET specifies the byte position, relative to the beginning of the file, to the next IMAGE_FILE_DIRECTORY( ) syntax structure in the file.

The value of ZERO_OR_NEXT_IFD_OFFSET shall be an integer multiple of 2.

Decoders may ignore any IMAGE_FILE_DIRECTORY( ) syntax structures at locations in the file specified by any ZERO_OR_NEXT_IFD_OFFSET syntax element.

A.4 IFD_ENTRY( )

A.4.1 Syntax structure

The IFD_ENTRY( ) syntax structure is specified by Table A.3.

Table A.3 — IFD_ENTRY( ) syntax structure

IFD_ENTRY( ) {	Descriptor	Reference
FIELD_TAG	le(16)	A.4.2
ELEMENT_TYPE	le(16)	A.4.3
NUM_ELEMENTS	le(32)	A.4.4
VALUES_OR_OFFSET	le(32)	A.4.5
}		

The interpretation and presence of syntax elements of the IFD_ENTRY( ) syntax structure is specified in Table A.4. The data associated with a FIELD_TAG value is interpreted as the value of the syntax element or syntax structure in the column of the table with the heading "Syntax element or syntax structure". The term "variable" is used in the table to indicate cases in which NUM_ELEMENTS may have any value corresponding to the quantity of associated data. The column of the table with the heading "Presence" is interpreted as follows.

— "Required" specifies that the FIELD_TAG value shall be present in an IFD_ENTRY( ) syntax structure of each IMAGE_FILE_DIRECTORY( ) syntax structure in the file.

— "Optional" indicates that the FIELD_TAG value may or may not be present in an IFD_ENTRY( ) syntax structure of each IMAGE_FILE_DIRECTORY( ) syntax structure in the file.

IFD entries with combinations of FIELD_TAG, ELEMENT_TYPE, and NUM_ELEMENTS that do not appear in Table A.4, with the FIELD_TAG value in the range of 0x1000 to 0x3FFF and 0x5000 to 0x7FFF are reserved for future use by ITU-T | ISO/IEC. Decoders shall ignore (parse and discard) any IFD_ENTRY( ) syntax structures in which such combinations appear.

NOTE 1 The purpose of the specification for decoders to ignore IFD entries with such combinations of FIELD_TAG, ELEMENT_TYPE, and NUM_ELEMENTS is to enable the future definition of a backward-compatible usage of additional combinations.

IFD entries with combinations of FIELD_TAG, ELEMENT_TYPE, and NUM_ELEMENTS that do not appear in Table A.4, with the FIELD_TAG value in the range of 0x0000 to 0x0FFF, 0x4000 to 0x4FFF, and 0x8000 to 0xFFFF are available for unspecified use and interpretation as determined by the application. Decoders shall parse any IFD_ENTRY( ) syntax structures in which such combinations appear and, for purposes relevant to determining conformance to this document, shall ignore these syntax structures. Any use of such FIELD_TAG values shall not affect the expressed requirements for conformance to this document. Additionally, ITU-T and ISO/IEC reserve the ability to potentially specify uses for such FIELD_TAG values in future revisions of this document.

NOTE 2 Since interpretation of such FIELD_TAG values can be application-specific, it is suggested to remove any such IFD_ENTRY( ) syntax structures that have unknown interpretations when transferring files between differing application domains.

NOTE 3 The use of a field tag value equal to 0x02BC in tag-based encoded files (such as files formatted according to ISO 12234-2 [4], ISO 12639 [5], Adobe TIFF 6.0 [7], or EXIF 2.2 [6]), is specified in Adobe Extensible Metadata Platform (XMP):2005, section 5 [8]. The use of a field tag value equal to 0x8769 is specified in EXIF 2.2:2002, section 4.6.3. The use of a field tag value equal to 0x8773 is specified in ICC ICC.1:2001, section B.3 [12] and in ISO 15076-1:2010, B.4, which specify two versions of ICC profile data. The provision to allow these field tag values to be present is intended to allow the use of the XMP, EXIF 2.2, ICC.1 and ISO 15076-1 specifications with files encoded according to this document (without imposing normative conformance requirements related to such use). The use of ICC profile data is further discussed in Annex C.

Table A.4 — Interpretation, allowed combinations, and presence of syntax elements of the IFD_ENTRY( ) syntax structure

FIELD_TAG	ELEMENT_TYPE	NUM_ELEMENTS	Syntax element or syntax structure	Presence
0x010D	UTF8	variable	DOCUMENT_NAME	Optional
0x010E	UTF8	variable	IMAGE_DESCRIPTION	Optional
0x010F	UTF8	variable	EQUIPMENT_MAKE	Optional
0x0110	UTF8	variable	EQUIPMENT_MODEL	Optional
0x011D	UTF8	variable	PAGE_NAME	Optional
0x0129	USHORT	2	PAGE_NUMBER	Optional
0x0131	UTF8	variable	SOFTWARE_NAME_VERSION	Optional
0x0132	UTF8	20	DATE_TIME	Optional
0x013B	UTF8	variable	ARTIST_NAME	Optional
0x013C	UTF8	variable	HOST_COMPUTER	Optional
0x8298	UTF8	variable	COPYRIGHT_NOTICE	Optional
0xA001	USHORT	1	COLOR_SPACE	Optional
0xBC01	BYTE	16	PIXEL_FORMAT	Required
0xBC02	BYTE, USHORT, or ULONG	1	SPATIAL_XFRM_PRIMARY	Optional
0xBC04	ULONG	1	IMAGE_TYPE	Optional
0xBC05	BYTE	4	PTM_COLOR_INFO( )	Optional
0xBC06	BYTE	variable	PROFILE_LEVEL_CONTAINER( )	Optional
0xBC80	BYTE, USHORT, or ULONG	1	IMAGE_WIDTH	Required
0xBC81	BYTE, USHORT, or ULONG	1	IMAGE_HEIGHT	Required
0xBC82	FLOAT	1	WIDTH_RESOLUTION	Optional
0xBC83	FLOAT	1	HEIGHT_RESOLUTION	Optional
0xBCC0	BYTE, USHORT, or ULONG	1	IMAGE_OFFSET	Required
0xBCC1	BYTE, USHORT, or ULONG	1	IMAGE_BYTE_COUNT	Required
0xBCC2	BYTE, USHORT, or ULONG	1	ALPHA_OFFSET	Optional
0xBCC3	BYTE, USHORT, or ULONG	1	ALPHA_BYTE_COUNT	Optional
0xBCC4	BYTE	1	IMAGE_BAND_PRESENCE	Optional
0xBCC5	BYTE	1	ALPHA_BAND_PRESENCE	Optional
0xEA1C	UNDEFINED	variable	PADDING_DATA	Optional

A.4.2 FIELD_TAG

FIELD_TAG identifies the data contained in the IFD_ENTRY( ) syntax structure. When the IFD_ENTRY( ) syntax structure is not the first IFD_ENTRY( ) syntax structure of the IMAGE_FILE_DIRECTORY( ) syntax structure, the value of FIELD_TAG shall be greater than the value of FIELD_TAG in the preceding IFD_ENTRY( ) syntax structure of the IMAGE_FILE_DIRECTORY( ) syntax structure.

A.4.3 ELEMENT_TYPE

ELEMENT_TYPE identifies the type of data contained in the IFD_ENTRY( ) syntax structure as specified by Table A.5.

Table A.5 — ELEMENT_TYPE

ELEMENT_TYPE	Mnemonic	SizeOfElement
0	RESERVED	Not specified
1	BYTE	1
2	UTF8	1
3	USHORT	2
4	ULONG	4
5	URATIONAL	8
6	SBYTE	1
7	UNDEFINED	1
8	SSHORT	2
9	SLONG	4
10	SRATIONAL	8
11	FLOAT	4
12	DOUBLE	8
13-65535	RESERVED	Not specified

The value of ELEMENT_TYPE shall not be equal to RESERVED in files conforming to this edition of this annex. Such values are reserved for use in future editions of this document. Decoders that encounter files containing ELEMENT_TYPE equal to RESERVED shall ignore the associated data.

When ELEMENT_TYPE is equal to UNDEFINED, the value of FIELD_TAG shall be equal to 0x8773 if it is not equal to 0xEA1C in files conforming to this edition of this annex. Other values of ELEMENT_TYPE are reserved for use in future editions of this document.

The quantity of data associated with each syntax element VALUES_OR_OFFSET, as specified by ELEMENT_TYPE in units of bytes, is specified in the SizeOfElement column of Table A.5. The interpretation of the data elements associated with each value of ELEMENT_TYPE is specified as follows.

— If ELEMENT_TYPE is equal to BYTE, USHORT, or ULONG, each data element is interpreted as an unsigned integer of the specified length in little-endian form.

— Otherwise, if ELEMENT_TYPE is equal to SBYTE, SSHORT, or SLONG, each data element is interpreted as a two's complement signed integer of the specified length in little-endian form.

— Otherwise, if ELEMENT_TYPE is equal to UTF8, each data element is interpreted as a UTF-8 character set code as specified by ISO/IEC 10646:2017, Annex D, and the value of the last data element of the IFD_ENTRY( ) shall be equal to 0 (null). Any such field may contain multiple strings of UTF-8 characters, each terminated with a 0-valued character. The NUM_ELEMENTS for such multi-string payloads is the total number of bytes in all of the associated strings including the 0-valued byte at the end of each such string. Within the associated NUM_ELEMENTS bytes, there shall not be any two consecutive bytes equal to 0.

— Otherwise, if ELEMENT_TYPE is equal to UNDEFINED, the interpretation of the data elements depends on the FIELD_TAG value as follows:

  — If the value of FIELD_TAG is equal to 0xEA1C (PADDING_DATA), the interpretation of each data element is specified in subclause A.4.33.

  — Otherwise, the interpretation of the data elements is not specified by this document.

— Otherwise, if ELEMENT_TYPE is equal to URATIONAL, each data element is interpreted as a rational number having a numerator equal to the first four bytes of the data element value, interpreted as an unsigned integer of the specified length in little-endian form, and a denominator equal to the remaining four bytes of the data element value, interpreted as an unsigned integer of the specified length in little-endian form.

— Otherwise, if ELEMENT_TYPE is equal to SRATIONAL, each data element is interpreted as a rational number having a numerator equal to the first four bytes of the data element value, interpreted as a two's complement signed integer of the specified length in little-endian form, and a denominator equal to the remaining four bytes of the data element value, interpreted as a two's complement signed integer of the specified length in little-endian form.

— Otherwise, if ELEMENT_TYPE is equal to FLOAT, each data element is interpreted as a four-byte ISO/IEC/IEEE 60559 floating-point number in little-endian form.

— Otherwise (ELEMENT_TYPE is equal to DOUBLE), each data element is interpreted as an eight-byte ISO/IEC/IEEE 60559 floating-point number in little-endian form.

A.4.4 NUM_ELEMENTS

NUM_ELEMENTS specifies the number of data elements associated with the IFD_ENTRY( ) syntax structure. Let iSizeOfElement be equal to the value in the SizeOfElement column of Table A.5 for the value of ELEMENT_TYPE. The total number of bytes of data associated with the IFD_ENTRY( ) syntax structure is NUM_ELEMENTS * iSizeOfElement.

NOTE NUM_ELEMENTS must be set to the total quantity of data elements, including termination indicator values such as null-valued terminators as applicable.

A.4.5 VALUES_OR_OFFSET

VALUES_OR_OFFSET is interpreted as follows:

— Let iSizeOfElement be equal to the value in the SizeOfElement column of Table A.5 for the value of ELEMENT_TYPE. If NUM_ELEMENTS * iSizeOfElement is less than or equal to 4, VALUES_OR_OFFSET contains the data elements associated with the IFD_ENTRY( ) syntax structure.

— Otherwise, VALUES_OR_OFFSET specifies the byte position, relative to the beginning of the file, of the data elements associated with the IFD_ENTRY( ) syntax structure. In this case, the value of the VALUES_OR_OFFSET syntax element shall be an integer multiple of 2.

A.4.6 DOCUMENT_NAME

DOCUMENT_NAME (when present) provides, as a UTF-8 character string, a name for the image. There are no specific conformance requirements for the content of the associated character string (other than that it follows the format conventions of the ELEMENT_TYPE definition of the UTF8 data type as specified in subclause A.4.3).

NOTE 1 The name of this syntax element alludes to the intended interpretation instead of any normative meaning.

NOTE 2 DOCUMENT_NAME is specified here in a manner that is intended to be consistent with the "DocumentName" tag specified in ISO 12639 [5] and TIFF 6.0 [7].

A.4.7 IMAGE_DESCRIPTION

IMAGE_DESCRIPTION (when present) provides, as a UTF-8 character string, a title or description of the image subject matter. There are no specific conformance requirements for the content of the associated character string (other than that it follows the format conventions of the ELEMENT_TYPE definition of the UTF8 data type as specified in subclause A.4.3).

NOTE 1 The name of this syntax element alludes to the intended interpretation instead of any normative meaning.

NOTE 2 IMAGE_DESCRIPTION is specified here in a manner that is intended to be consistent with the "ImageDescription" tag specified in ISO 12234-2 [4], ISO 12639 [5], TIFF 6.0 [7], and EXIF 2.2 [6].

A.4.8 EQUIPMENT_MAKE

EQUIPMENT_MAKE (when present) provides, as a UTF-8 character string, a name for the manufacturer or vendor of the camera, image-capturing device, scanner, or other equipment that generated the image. There are no specific conformance requirements for the content of the associated character string (other than that it follows the format conventions of the ELEMENT_TYPE definition of the UTF8 data type as specified in subclause A.4.3).

NOTE 1 The name of this syntax element alludes to the intended interpretation instead of any normative meaning.

NOTE 2 EQUIPMENT_MAKE is specified here in a manner that is intended to be consistent with the "Make" tag specified in ISO 12234-2 [4], ISO 12639 [5], TIFF 6.0 [7], and EXIF 2.2 [6].

A.4.9 EQUIPMENT_MODEL

When present, EQUIPMENT_MODEL provides, as a UTF-8 character string, a model name or model number of the camera, image-capturing device, scanner, or other equipment that generated the image. There are no specific conformance requirements for the content of the associated character string (other than that it follows the format conventions of the ELEMENT_TYPE definition of the UTF8 data type as specified in subclause A.4.3).

NOTE 1 The name of this syntax element alludes to the intended interpretation instead of any normative meaning.

NOTE 2 EQUIPMENT_MODEL is specified here in a manner that is intended to be consistent with the "Model" tag specified in ISO 12234-2 [4], ISO 12639 [5], TIFF 6.0 [7], and EXIF 2.2 [6].

A.4.10 PAGE_NAME

PAGE_NAME (when present) provides, as a UTF-8 character string, a name to be considered to be the name of a page to which the image is considered to belong. There are no specific conformance requirements for the content of the associated character string (other than that it follows the format conventions of the ELEMENT_TYPE definition of the UTF8 data type as specified in subclause A.4.3).

NOTE 1 The name of this syntax element alludes to the intended interpretation instead of any normative meaning.

NOTE 2 PAGE_NAME is specified here in a manner that is intended to be consistent with the "PageName" tag specified in ISO 12639 [5] and TIFF 6.0 [7].

A.4.11 PAGE_NUMBER

PAGE_NUMBER (when present) provides, as a pair of unsigned integers, a "page number" (the first number) to be considered the page number of a page from a sequence of pages (the second number) to which the image is considered to belong. There are no specific conformance requirements for the content of the associated character string (other than that it follows the format conventions of the ELEMENT_TYPE definition of the USHORT data type as specified in subclause A.4.3).

NOTE 1 The name of this syntax element alludes to the intended interpretation instead of any normative meaning.

NOTE 2 PAGE_NUMBER is specified here in a manner that is intended to be consistent with the "PageNumber" tag specified in TIFF 6.0 [7].

A.4.12 SOFTWARE_NAME_VERSION

SOFTWARE_NAME_VERSION (when present) provides, as a UTF-8 character string, name and version number information for one or more software packages used in generating the image or file. There are no specific conformance requirements for the content of the associated character string (other than that it follows the format conventions of the ELEMENT_TYPE definition of the UTF8 data type as specified in subclause A.4.3).

NOTE 1 The name of this syntax element alludes to the intended interpretation instead of any normative meaning.

NOTE 2 SOFTWARE_NAME_VERSION is specified here in a manner that is intended to be consistent with the "Software" tag specified in ISO 12234-2 [4], ISO 12639 [5], TIFF 6.0 [7], and EXIF 2.2 [6].

A.4.13 DATE_TIME

DATE_TIME (when present) provides, as a UTF-8 character string, the date and time of the creation or last modification of the image.

The value of NUM_ELEMENTS for the DATE_TIME tag shall be equal to 20.

The interpretation of the UTF-8 character string associated with the DATE_TIME tag shall be as follows:

— If the UTF-8 character string associated with the DATE_TIME tag is the string "0000:00:00 00:00:00" or this string with all "0" characters replaced by space (0x20) characters as in the string "   : :   : : ", followed by a byte equal to 0, this indicates that the date and time are unknown.

— Otherwise, the UTF-8 character string associated with the DATE_TIME tag shall be formatted as "YYYY:MM:DD HH:MM:SS", followed by a byte equal to 0, where the formatting of this string shall be as follows.

  — The "YYYY" part of the string shall contain a character string representation of a four-digit decimal number for a calendar year.

  — Each colon character ":" part of the string shall be a colon ":" character (a byte equal to 0x3A).

  — The "MM" part of the string shall contain a character string representation of a two-digit decimal number in the range from "01" to "12" for a month within the year (where "01" indicates January, "02" indicates February, etc.) or two space (0x20) characters.

  — The "DD" part of the string shall contain a character string representation of a two-digit decimal number in the range from "01" to "31" for a day within the month (where "01" indicates the first day of the month, "02" indicates the second day of the month, etc.).

  — The character separating the "DD" and "HH" parts of the string shall be a space character " " (a byte equal to 0x20).

  — The "HH" part of the string shall contain a character string representation of a two-digit decimal number in the range from "00" to "23" for an hour within the day, following a 24-hour time of day convention (where "00" indicates a time in the first hour of the day, "01" indicates a time in the second hour of the day, etc.).

  NOTE 1 The time 00:00:00 is used to indicate midnight. Similarly, 00:00:01 indicates one second after midnight and 23:59:59 indicates one second prior to midnight.

  — The "MM" part of the string shall contain a character string representation of a two-digit decimal number in the range from "00" to "59" for a minute within the hour (where "00" indicates a time in the first minute of the hour, "01" indicates a time within the second minute of the hour, etc.).

  — The "SS" part of the string shall contain a character string representation of a two-digit decimal number in the range from "00" to "59" for a second within the minute (where "00" indicates a time in the first second of the minute, "01" indicates a time within the second second of the minute, etc.).

The time zone for the date and time are not specified in this edition of this document.

NOTE 2 For example, the date and time may be referenced to the local time zone in which the encoding equipment is operated or is ordinarily kept.

There are no additional specific conformance requirements for the content of the associated character string.

NOTE 3 The name of this syntax element alludes to the intended interpretation instead of any normative meaning.

NOTE 4 DATE_TIME is specified here in a manner that is intended to be consistent with the "DateTime" tag specified in ISO 12234-2, ISO 12639, TIFF 6.0, and EXIF 2.2.

A.4.14 ARTIST_NAME

ARTIST_NAME (when present) provides, as a UTF-8 character string, one or more names and possibly additional information relating to persons involved in the creation of the image. There are no specific conformance requirements for the content of the associated character string (other than that it follows the format conventions of the ELEMENT_TYPE definition of the UTF8 data type as specified in subclause A.4.3).

NOTE 1 The name of this syntax element alludes to the intended interpretation instead of any normative meaning.

NOTE 2 ARTIST_NAME is specified here in a manner that is intended to be consistent with the "Artist" tag specified in ISO 12234-2 [4], ISO 12639 [5], TIFF 6.0 [7], and EXIF 2.2 [6]. An example preferred use of ARTIST_NAME is the string value "Camera owner, John Smith; Photographer, Michael Brown; Image creator, Ken James" (followed by a byte equal to 0).

A.4.15 HOST_COMPUTER

HOST_COMPUTER (when present) provides, as a UTF-8 character string, an indication of a computer, operating system, or other system used in generating the image or file. There are no specific conformance requirements for the content of the associated character string (other than that it follows the format conventions of the ELEMENT_TYPE definition of the UTF8 data type as specified in subclause A.4.3).

NOTE 1 The name of this syntax element alludes to the intended interpretation instead of any normative meaning.

NOTE 2 HOST_COMPUTER is specified here in a manner that is intended to be consistent with the "HostComputer" tag specified in ISO 12639 [5] and TIFF 6.0 [7].

A.4.16 COPYRIGHT_NOTICE

COPYRIGHT_NOTICE (when present) provides, as a UTF-8 character string, an indication of copyright rights associated with the image or file. There are no specific conformance requirements for the content of the associated character string (other than that it follows the format conventions of the ELEMENT_TYPE definition of the UTF8 data type as specified in subclause A.4.3).

NOTE 1 The name of this syntax element alludes to the intended interpretation instead of any normative meaning.

NOTE 2 COPYRIGHT_NOTICE is specified here in a manner that is intended to be consistent with the "Copyright" tag specified in ISO 12234-2 [4], ISO 12639 [5], TIFF 6.0 [7], and EXIF 2.2 [6]. A complete copyright statement can be listed in the character string including any dates and statements of claims – for example, "Copyright, John Smith, 2009. All rights reserved" (followed by a byte equal to 0). When desired, this character string can also list a royalty clearing house. The string can be used to indicate either or both photographer and editor copyrights as well as any additional or alternative copyrights that apply to the image or file. When a photographer statement is included, it would appear first. When no photographer copyright statement is included and an editor or additional or alternative copyright statement is included, the absence of a photographer copyright statement would be indicated by a space character (0x20) followed by a byte equal to 0 (null) at the beginning of the character string. When an editor copyright statement is included, it would appear next. When no editor copyright statement is included and some additional or alternative copyright statement is included, the absence of an editor copyright statement would be indicated by a space character (0x20) followed by a byte equal to 0 (null) at the position in the character string at which the editor copyright statement would otherwise appear. When some additional or alternative copyright statement is included, it would appear next. When present, each additional or alternative copyright statement would end with a byte equal to 0 (null). When a royalty clearing house listing is included, this listing would appear last.

A.4.17 COLOR_SPACE

COLOR_SPACE (when present) equal to 1, when the value in the Num column in Table A.6 for the line corresponding to the value of PIXEL_FORMAT is equal to UINT, indicates that the associated image should be interpreted according to the sRGB specification IEC 61966-2-1. COLOR_SPACE equal to 0xFFFF, when used with an unsigned pixel format as specified by PIXEL_FORMAT, indicates that the sRGB specification may not be the preferred method of interpretation of the associated image.

COLOR_SPACE shall not be equal to 1 when PIXEL_FORMAT specifies a fixed or floating point pixel format.

When present, COLOR_SPACE shall be equal to 1 or 0xFFFF. All other values are reserved for future use by ITU-T | ISO/IEC. All values other than 1 and 0xFFFF shall be treated the same as the value 0xFFFF by decoders.

NOTE sRGB image data is used in many systems, being interpreted directly by various receiving devices.

A.4.18 PIXEL_FORMAT

PIXEL_FORMAT consists of a string of 16 BYTE values. The values of the bytes ordered from first to last in this string are interpreted as digit pairs appearing from left to right in the hexadecimal integer values specified in Table A.6.

NOTE 1 The PIXEL_FORMAT values specified in Table A.6 originated as UUID values as specified by ISO/IEC 11578. UUID values are commonly shown as a string of hexadecimal digits separated by dash "-" characters. The bytes of such a UUID value are typically stored in a file using a rearranged form of such a typical dashed-form description. In Table A.6, that convention is not followed. Instead, the values are specified in a manner intended to ease the interpretation of the value as a string of 16 BYTE values that can be directly compared to the provided strings of hexadecimal digits. The rearrangement of the common form of UUID specification to produce the equivalent ordered string of BYTE values as shown in Table A.6 can be performed as follows:

— The first (left-most) four bytes (which precede the first dash character in the common form of UUID value specification) are ordered first, represented in little-endian (byte-reversed) form.

— The next two bytes (which precede the second dash character in the common form of UUID value specification) are ordered next, represented in little-endian (byte-reversed) form.

— The next two bytes (which precede the third dash character in the common form of UUID value specification) are ordered next, represented in little-endian (byte-reversed) form.

— The remaining bytes in the common form of UUID value specification follow, in left-to-right order.

The headings of Table A.6 are interpreted as follows:

— "Mnemonic name" indicates a string associated with the syntax element corresponding to the value of PIXEL_FORMAT.

— "NC" specifies the following:

  — If the value of PIXEL_FORMAT specifies that no alpha channel is present (as specified by the entry in the "Alpha" column), it is a requirement for file format conformance to this annex that the value of NumComponents as calculated in Table 31 shall be equal to the value of the entry in the "NC" column.

  — Otherwise, if the alpha channel is present as a separate alpha image plane (as specified by the entry in the "Alpha" column and the presence of ALPHA_OFFSET), it is a requirement for file format conformance to this annex that the value of NumComponents as calculated in Table 31 for the CODED_IMAGE( ) syntax structure at the position specified by IMAGE_OFFSET shall be equal to the value of the entry in the "NC" column minus 1, and the value of NumComponents as calculated in Table 31 for the CODED_IMAGE( ) syntax structure at the position specified by ALPHA_OFFSET shall be equal to 1.

  — Otherwise (the alpha channel is present as an interleaved alpha image plane), it is a requirement for file format conformance to this annex that the value of NumComponents as calculated in Table 31 for the primary image plane shall be equal to the value of the entry in the "NC" column minus 1, and the value of NumComponents as calculated in Table 31 for the alpha image plane shall be equal to 1.

— "Alpha" specifies whether the PIXEL_FORMAT specifies the presence of an alpha channel. An entry marked with "✓" indicates that an alpha channel is present and an entry not marked with "✓" indicates that an alpha channel is not present.

— "BPC" specifies a requirement for file format conformance to this annex that the value of the OUTPUT_BITDEPTH syntax element of the CODED_IMAGE( ) syntax structure at the position specified by IMAGE_OFFSET shall be equal to the value of the entry in the "BPC" column and, when present, the value of the OUTPUT_BITDEPTH syntax element of the CODED_IMAGE( ) syntax structure at the position specified by ALPHA_OFFSET shall be equal to value of the entry in the "BPC" column.

— "Num" specifies the numeric interpretation of values in the decoded image.

— "Colour" specifies a requirement for file format conformance to this annex that the value of the OUTPUT_CLR_FMT syntax element of the CODED_IMAGE( ) syntax structure at the position specified by IMAGE_OFFSET shall be equal to the value of the entry in the "Colour" column.

NOTE 2 Image data can be structured as single channel monochrome "Gray", three-channel "RGB", four-channel "CMYK", or n-Channel ("NCOMPONENT") with n in the range of 2 to 16, inclusive.

Table A.6 — Interpretation of PIXEL_FORMAT

PIXEL_FORMAT value	Mnemonic	NC	Alpha	BPC	Num	Colour
0x24C3DD6F034EFE4BB1853D77768DC90D	24bppRGB	3		BD8	UINT	RGB
0x24C3DD6F034EFE4BB1853D77768DC90C	24bppBGR	3		BD8	UINT	RGB
0x24C3DD6F034EFE4BB1853D77768DC90E	32bppBGR	3		BD8	UINT	RGB
0x24C3DD6F034EFE4BB1853D77768DC915	48bppRGB	3		BD16	UINT	RGB
0x24C3DD6F034EFE4BB1853D77768DC912	48bppRGBFixedPoint	3		BD16S	SINT	RGB
0x24C3DD6F034EFE4BB1853D77768DC93B	48bppRGBHalf	3		BD16F	Float	RGB
0x24C3DD6F034EFE4BB1853D77768DC918	96bppRGBFixedPoint	3		BD32S	SINT	RGB
0x24C3DD6F034EFE4BB1853D77768DC940	64bppRGBFixedPoint	3		BD16S	SINT	RGB
0x24C3DD6F034EFE4BB1853D77768DC942	64bppRGBHalf	3		BD16F	Float	RGB
0x24C3DD6F034EFE4BB1853D77768DC941	128bppRGBFixedPoint	3		BD32S	SINT	RGB
0x24C3DD6F034EFE4BB1853D77768DC91B	128bppRGBFloat	3		BD32F	Float	RGB
0x24C3DD6F034EFE4BB1853D77768DC90F	32bppBGRA	4	✓	BD8	UINT	RGB
0x24C3DD6F034EFE4BB1853D77768DC916	64bppRGBA	4	✓	BD16	UINT	RGB
0x24C3DD6F034EFE4BB1853D77768DC91D	64bppRGBAFixedPoint	4	✓	BD16S	SINT	RGB
0x24C3DD6F034EFE4BB1853D77768DC93A	64bppRGBAHalf	4	✓	BD16F	Float	RGB
0x24C3DD6F034EFE4BB1853D77768DC91E	128bppRGBAFixedPoint	4	✓	BD32S	SINT	RGB
0x24C3DD6F034EFE4BB1853D77768DC919	128bppRGBAFloat	4	✓	BD32F	Float	RGB
0x24C3DD6F034EFE4BB1853D77768DC910	32bppPBGRA	4	✓	BD8	UINT	RGB
0x24C3DD6F034EFE4BB1853D77768DC917	64bppPRGBA	4	✓	BD16	UINT	RGB
0x24C3DD6F034EFE4BB1853D77768DC91A	128bppPRGBAFloat	4	✓	BD32F	Float	RGB
0x24C3DD6F034EFE4BB1853D77768DC91C	32bppCMYK	4		BD8	UINT	CMYK
0x24C3DD6F034EFE4BB1853D77768DC92C	40bppCMYKAlpha	5	✓	BD8	UINT	CMYK
0x24C3DD6F034EFE4BB1853D77768DC91F	64bppCMYK	4		BD16	UINT	CMYK
0x24C3DD6F034EFE4BB1853D77768DC92D	80bppCMYKAlpha	5	✓	BD16	UINT	CMYK
0x24C3DD6F034EFE4BB1853D77768DC920	24bpp3Channels	3		BD8	UINT	NCOMPONENT
0x24C3DD6F034EFE4BB1853D77768DC921	32bpp4Channels	4		BD8	UINT	NCOMPONENT
0x24C3DD6F034EFE4BB1853D77768DC922	40bpp5Channels	5		BD8	UINT	NCOMPONENT
0x24C3DD6F034EFE4BB1853D77768DC923	48bpp6Channels	6		BD8	UINT	NCOMPONENT
0x24C3DD6F034EFE4BB1853D77768DC924	56bpp7Channels	7		BD8	UINT	NCOMPONENT
0x24C3DD6F034EFE4BB1853D77768DC925	64bpp8Channels	8		BD8	UINT	NCOMPONENT
0x24C3DD6F034EFE4BB1853D77768DC92E	32bpp3ChannelsAlpha	4	✓	BD8	UINT	NCOMPONENT
0x24C3DD6F034EFE4BB1853D77768DC92F	40bpp4ChannelsAlpha	5	✓	BD8	UINT	NCOMPONENT
0x24C3DD6F034EFE4BB1853D77768DC930	48bpp5ChannelsAlpha	6	✓	BD8	UINT	NCOMPONENT

PIXEL_FORMAT value	Mnemonic	NC	Alpha	BPC	Num	Colour
0x24C3DD6F034EFE4BB1853D77768DC931	56bpp6ChannelsAlpha	7	✓	BD8	UINT	NCOMPONENT
0x24C3DD6F034EFE4BB1853D77768DC932	64bpp7ChannelsAlpha	8	✓	BD8	UINT	NCOMPONENT
0x24C3DD6F034EFE4BB1853D77768DC933	72bpp8ChannelsAlpha	9	✓	BD8	UINT	NCOMPONENT
0x24C3DD6F034EFE4BB1853D77768DC926	48bpp3Channels	3		BD16	UINT	NCOMPONENT
0x24C3DD6F034EFE4BB1853D77768DC927	64bpp4Channels	4		BD16	UINT	NCOMPONENT
0x24C3DD6F034EFE4BB1853D77768DC928	80bpp5Channels	5		BD16	UINT	NCOMPONENT
0x24C3DD6F034EFE4BB1853D77768DC929	96bpp6Channels	6		BD16	UINT	NCOMPONENT
0x24C3DD6F034EFE4BB1853D77768DC92A	112bpp7Channels	7		BD16	UINT	NCOMPONENT
0x24C3DD6F034EFE4BB1853D77768DC92B	128bpp8Channels	8		BD16	UINT	NCOMPONENT
0x24C3DD6F034EFE4BB1853D77768DC934	64bpp3ChannelsAlpha	4	✓	BD16	UINT	NCOMPONENT
0x24C3DD6F034EFE4BB1853D77768DC935	80bpp4ChannelsAlpha	5	✓	BD16	UINT	NCOMPONENT
0x24C3DD6F034EFE4BB1853D77768DC936	96bpp5ChannelsAlpha	6	✓	BD16	UINT	NCOMPONENT
0x24C3DD6F034EFE4BB1853D77768DC937	112bpp6ChannelsAlpha	7	✓	BD16	UINT	NCOMPONENT
0x24C3DD6F034EFE4BB1853D77768DC938	128bpp7ChannelsAlpha	8	✓	BD16	UINT	NCOMPONENT
0x24C3DD6F034EFE4BB1853D77768DC939	144bpp8ChannelsAlpha	9	✓	BD16	UINT	NCOMPONENT
0x24C3DD6F034EFE4BB1853D77768DC908	8bppGray	1		BD8	UINT	YONLY
0x24C3DD6F034EFE4BB1853D77768DC90B	16bppGray	1		BD16	UINT	YONLY
0x24C3DD6F034EFE4BB1853D77768DC913	16bppGrayFixedPoint	1		BD16S	SINT	YONLY
0x24C3DD6F034EFE4BB1853D77768DC93E	16bppGrayHalf	1		BD16F	Float	YONLY
0x24C3DD6F034EFE4BB1853D77768DC93F	32bppGrayFixedPoint	1		BD32S	SINT	YONLY
0x24C3DD6F034EFE4BB1853D77768DC911	32bppGrayFloat	1		BD32F	Float	YONLY
0x24C3DD6F034EFE4BB1853D77768DC905	BlackWhite	1		BD1WHITE1 or BD1BLACK1	UINT	YONLY
0x24C3DD6F034EFE4BB1853D77768DC909	16bppBGR555	3		BD5	UINT	RGB
0x24C3DD6F034EFE4BB1853D77768DC90A	16bppBGR565	3		BD565	UINT	RGB
0x24C3DD6F034EFE4BB1853D77768DC914	32bppBGR101010	3		BD10	UINT	RGB
0x24C3DD6F034EFE4BB1853D77768DC93D	32bppRGBE	3		BD8	Float	RGBE
0x24C3DD6F034EFE4BB1853D77768DC954	32bppCMYKDIRECT	4		BD8	UINT	CMYKDIRECT
0x24C3DD6F034EFE4BB1853D77768DC955	64bppCMYKDIRECT	4		BD16	UINT	CMYKDIRECT
0x24C3DD6F034EFE4BB1853D77768DC956	40bppCMYKDIRECTAlpha	5	✓	BD8	UINT	CMYKDIRECT
0x24C3DD6F034EFE4BB1853D77768DC943	80bppCMYKDIRECTAlpha	5	✓	BD16	UINT	CMYKDIRECT
0x24C3DD6F034EFE4BB1853D77768DC944	12bppYCC420	3		BD8	UINT	YUV420
0x24C3DD6F034EFE4BB1853D77768DC945	16bppYCC422	3		BD8	UINT	YUV422
0x24C3DD6F034EFE4BB1853D77768DC946	20bppYCC422	3		BD10	UINT	YUV422
0x24C3DD6F034EFE4BB1853D77768DC947	32bppYCC422	3		BD16	UINT	YUV422
0x24C3DD6F034EFE4BB1853D77768DC948	24bppYCC444	3		BD8	UINT	YUV444
0x24C3DD6F034EFE4BB1853D77768DC949	30bppYCC444	3		BD10	UINT	YUV444
0x24C3DD6F034EFE4BB1853D77768DC94A	48bppYCC444	3		BD16	UINT	YUV444
0x24C3DD6F034EFE4BB1853D77768DC94B	48bppYCC444FixedPoint	3		BD16S	SINT	YUV444
0x24C3DD6F034EFE4BB1853D77768DC94C	20bppYCC420Alpha	4	✓	BD8	UINT	YUV420
0x24C3DD6F034EFE4BB1853D77768DC94D	24bppYCC422Alpha	4	✓	BD8	UINT	YUV422
0x24C3DD6F034EFE4BB1853D77768DC94E	30bppYCC422Alpha	4	✓	BD10	UINT	YUV422
0x24C3DD6F034EFE4BB1853D77768DC94F	48bppYCC422Alpha	4	✓	BD16	UINT	YUV422
0x24C3DD6F034EFE4BB1853D77768DC950	32bppYCC444Alpha	4	✓	BD8	UINT	YUV444
0x24C3DD6F034EFE4BB1853D77768DC951	40bppYCC444Alpha	4	✓	BD10	UINT	YUV444
0x24C3DD6F034EFE4BB1853D77768DC952	64bppYCC444Alpha	4	✓	BD16	UINT	YUV444
0x24C3DD6F034EFE4BB1853D77768DC953	64bppYCC444AlphaFixedPoint	4	✓	BD16S	SINT	YUV444

The manner of output for the decoded image data produced from a conforming decoder may be determined by the application. However, each PIXEL_FORMAT value specified in Table A.6 has a corresponding defined output format that is specified for reference purposes as the output of a hypothetical reference decoder. This reference output format is in the form of an ordered string of bytes defined as follows:

— If PIXEL_FORMAT is equal to 24bppRGB, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is R, then G, then B, and each sample value for each channel is output as a single byte.

— Otherwise, if PIXEL_FORMAT is equal to 32bppRGBE, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is B, then G, then R, then E, and each sample value for each channel is output as a single byte.

— Otherwise, if PIXEL_FORMAT is equal to 48bppRGB, 48bppRGBFixedPoint, or 48bppRGBHalf, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is R, then G, then B, and each sample value for each channel is output using 16 bits in little-endian form.

— Otherwise, if PIXEL_FORMAT is equal to 96bppRGBFixedPoint, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is R, then G, then B, and each sample value for each channel is output using 32 bits in little-endian form.

— Otherwise, if PIXEL_FORMAT is equal to 64bppRGBFixedPoint or 64bppRGBHalf, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is R, then G, then B, then a padding channel, and each sample value for each channel is output using 16 bits in little-endian form.

— Otherwise, if PIXEL_FORMAT is equal to 128bppRGBFixedPoint or 128bppRGBFloat, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is R, then G, then B, then a padding channel, and each sample value for each channel is output using 32 bits in little-endian form.

— Otherwise, if PIXEL_FORMAT is equal to 64bppRGBA, 64bppRGBAFixedPoint, 64bppRGBAHalf, or 64bppPRGBA, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is R, then G, then B, then Alpha, and each sample value for each channel is output using 16 bits in little-endian form.

— Otherwise, if PIXEL_FORMAT is equal to 128bppRGBAFixedPoint, 128bppRGBAFloat, or 128bppPRGBAFloat, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is R, then G, then B, then Alpha, and each sample value for each channel is output using 32 bits in little-endian form.

— Otherwise, if PIXEL_FORMAT is equal to 24bppBGR, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is B, then G, then R, and each sample value for each channel is output as a single byte.

— Otherwise, if PIXEL_FORMAT is equal to 32bppBGR, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is B, then G, then R, then a padding channel, and each sample value for each channel is output as a single byte.

— Otherwise, if PIXEL_FORMAT is equal to 32bppPBGRA, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is B, then G, then R, then Alpha, and each sample value for each channel is output as a single byte.

— Otherwise, if PIXEL_FORMAT is equal to 16bppBGR565, the output is in the form of packed bit fields within the integer values specified by Table 196, and these integer values are output in raster scan order using 16 bits in little-endian form. (See NOTE 3 below.)

— Otherwise, if PIXEL_FORMAT is equal to 16bppBGR555, the output is in the form of packed bit fields within the integer values specified by Table 197, and these integer values are output in raster scan order using 16 bits in little-endian form. (See NOTE 3 below.)

— Otherwise, if PIXEL_FORMAT is equal to 32bppBGR101010, the output is in the form of packed bit fields within the integer values specified by Table 198, and these integer values are output in raster scan order using 32 bits in little-endian form.

NOTE 3 When the PIXEL_FORMAT is equal to 16bppBGR565, 16bppBGR555, or 32bppBGR101010, the positions of the bits representing the actual R and B components depend on the value of the RED_BLUE_NOT_SWAPPED_FLAG. In these cases, if RED_BLUE_NOT_SWAPPED_FLAG is equal to 0, the R component bits are in the MSBs of the packed bit fields and the B component bits are in the LSBs; otherwise, in these cases the R component bits are in the LSBs of the packed bit fields and the B component bits are in the MSBs.

— Otherwise, if PIXEL_FORMAT is equal to BlackWhite, the output is in the form of packed bit fields within the integer values specified by Table 199, and these integer values are output as bytes in raster scan order.

— Otherwise, if PIXEL_FORMAT is equal to 32bppCMYK or 32bppCMYKDIRECT, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is C, then M, then Y, then K, and each sample value for each channel is output as a single byte.

— Otherwise, if PIXEL_FORMAT is equal to 40bppCMYKAlpha or 40bppCMYKDIRECTAlpha, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is C, then M, then Y, then K, then Alpha, and each sample value for each channel is output as a single byte.

— Otherwise, if PIXEL_FORMAT is equal to 64bppCMYK or 64bppCMYKDIRECT, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is C, then M, then Y, then K, and each sample value for each channel is output using 16 bits in little-endian form.

— Otherwise, if PIXEL_FORMAT is equal to 80bppCMYKAlpha or 80bppCMYKDIRECTAlpha, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is C, then M, then Y, then K, then Alpha, and each sample value for each channel is output using 16 bits in little-endian form.

— Otherwise, if PIXEL_FORMAT is equal to 12bppYCC420, 16bppYCC422, or 24bppYCC444, the Y samples are output first in raster scan order, then the U samples are output in raster scan order, then the V samples are output in raster scan order, and each sample value for each channel is output as a single byte.

— Otherwise, if PIXEL_FORMAT is equal to 20bppYCC422, 32bppYCC422, 30bppYCC444, 48bppYCC444, or 48bppYCC444FixedPoint, the Y samples are output first in raster scan order, then the U samples are output in raster scan order, then the V samples are output in raster scan order, and each sample value for each channel is output using 16 bits in little-endian form.

— Otherwise, if PIXEL_FORMAT is equal to 20bppYCC420Alpha or 24bppYCC422Alpha, the Y samples are output first in raster scan order, then the U samples are output in raster scan order, then the V samples are output in raster scan order, then the Alpha samples are output in raster scan order, and each sample value for each channel is output as a single byte.

— Otherwise, if PIXEL_FORMAT is equal to 32bppYCC444Alpha, the Y samples are output first in raster scan order, then the U samples are output in raster scan order, then the V samples are output in raster scan order, then the Alpha samples are output in raster scan order, and each sample value for each channel is output as a single byte.

— Otherwise, if PIXEL_FORMAT is equal to 30bppYCC422Alpha, 48bppYCC422Alpha, 40bppYCC444Alpha, 64bppYCC444Alpha, or 64bppYCC444AlphaFixedPoint, the Y samples are output first in raster scan order, then the U samples are output in raster scan order, then the V samples are output in raster scan order, then the Alpha samples are output in raster scan order, and each sample value for each channel is output using 16 bits in little-endian form.

— Otherwise, if PIXEL_FORMAT is equal to 24bpp3Channels, 32bpp4Channels, 40bpp5Channels, 48bpp6Channels, 56bpp7Channels, or 64bpp8Channels, the decoded channels are ordered by increasing channel number, and the samples of each channel are output in raster scan order, and each sample value for each channel is output as a single byte.

— Otherwise, if PIXEL_FORMAT is equal to 48bpp3Channels, 64bpp4Channels, 80bpp5Channels, 96bpp6Channels, 112bpp7Channels, or 128bpp8Channels, the decoded channels are ordered by increasing channel number, and the samples of each channel are output in raster scan order, and each sample value for each channel is output using 16 bits in little-endian form.

— Otherwise, if PIXEL_FORMAT is equal to 32bpp3ChannelsAlpha, 40bpp4ChannelsAlpha, 48bpp5ChannelsAlpha, 56bpp6ChannelsAlpha, 64bpp7ChannelsAlpha, or 72bpp8ChannelsAlpha, the decoded channels are ordered by increasing channel number with the Alpha channel being considered as having the highest channel number and the samples of each channel are output in raster scan order, and each sample value for each channel is output as a single byte.

— Otherwise, if PIXEL_FORMAT is equal to 64bpp3ChannelsAlpha, 80bpp4ChannelsAlpha, 96bpp5ChannelsAlpha, 112bpp6ChannelsAlpha, 128bpp7ChannelsAlpha, or 144bpp8ChannelsAlpha, the decoded channels are ordered by increasing channel number and the last channel is considered the Alpha channel, and the samples of each channel are output in raster scan order, and each sample value for each channel is output using 16 bits in little-endian form.

— Otherwise, if PIXEL_FORMAT is equal to 8bppGray, there is only one decoded channel, and the samples of that channel are output in raster scan order, and each sample value is output as a single byte.

— Otherwise, if PIXEL_FORMAT is equal to 16bppGray, 16bppGrayFixedPoint, or 16bppGrayHalf, , there is only one decoded channel, and the samples of that channel are output in raster scan order, and each sample value is output using 16 bits in little-endian form.

— Otherwise (PIXEL_FORMAT is equal to 32bppGrayFixedPoint or 32bppGrayFloat), there is only one decoded channel, and the samples of that channel are output in raster scan order, and each sample value is output using 32 bits in little-endian form.

NOTE 4 Particular care should be taken in regard to the interpretation of this defined output format for PIXEL_FORMAT values with mnemonic names that include "RGB", "BGR", or "CMYK" as part of the mnemonic name due to the specification of little-endian form for packed bit fields. In particular, the first byte of each packed bit field for the PIXEL_FORMAT values 16bppBGR565, 16bppBGR555, and 32bppBGR101010 actually contains the LSBs of the packed bit fields due to the use of little-endian form. Similarly, if the four bytes that contain the decoded channel samples for the PIXEL_FORMAT values 32bppRGBE, 32bppBGR, 32bppPBGRA, 32bppCMYK, and 32bppCMYKDIRECT are written or read as 32-bit integers in little-endian form rather than as ordered strings of four bytes in which each byte represents a sample value, the ordering of the channels will be swapped relative to the ordering described herein.

This defined format is specified for reference purposes and may be used by some decoders as an interface format for the output of decoded pictures. However, the use of this defined format is not a requirement for conformance to this document.

Interpretation of alpha channel information (when present) is considered pre-multiplied or not pre-multiplied as follows:

— If PIXEL_FORMAT is equal to 32bppPBGRA, 64bppPRGBA, or 128bppPRGBAFloat, the channels other than the alpha channel are considered to be in pre-multiplied form in relation to the alpha channel.

— Otherwise, the channels other than the alpha channel are considered not to be in pre-multiplied form in relation to the alpha channel. In these cases, the value of PREMULTIPLIED_ALPHA_FLAG in the associated IMAGE_HEADER( ) of the coded image that contains the alpha channel shall be equal to 0.

NOTE 5 The designation of an alpha channel as pre-multiplied indicates that the decoded sample values do not require multiplication by the alpha channel values when performing compositing (as any necessary such multiplication process was performed as a pre-processing step prior to encoding).

JPEG XR supports three types of numerical encoding: unsigned integer, fixed point, and floating point, each at a variety of bit depths.

PIXEL_FORMAT values having the "Num" column of Table A.6 indicating "UINT" are unsigned integer formats as follows.

— If BPC is equal to BD8, the minimum value is 0 and the maximum value is 255, providing 256 unique values.

— Otherwise (BPC is equal to BD16), the minimum value is 0 and the maximum value is 65 535, providing 65 536 unique values.

In unsigned integer cases, a value of zero ordinarily represents the minimum level or the encoding black for the specific channel and the maximum possible value represents the maximum value for that channel. When all viewable channels for a pixel format are at their maximum numerical value, this corresponds to the brightest representable colour, or the encoding white. Exceptions to this general rule include the following unsigned integer cases:

— When OUTPUT_BITDEPTH is BD1BLACK1, a value of zero represents the maximum level or white and a value of one represents the minimum level or black.

— When the OUTPUT_CLR_FMT is YUV444, YUV422, or YUV420, the U and V components are interpreted as colour difference representations that are offset by a constant value, such that the middle value of the range of possible integer values is used for the representation of both the encoding black and the encoding white.

— When the OUTPUT_CLR_FMT is CMYK or CMYKDIRECT, the K component is interpreted as a degree of proximity toward black, such that the maximum value for the K component is used for the encoding black and the minimum value for the K component is used for the encoding white.

— When the OUTPUT_CLR_FMT is NCOMPONENT and the number of encoded components other than the Alpha channel (when present) is greater than 3, in the absence of other information to assist in the colour channel interpretation, the fourth channel should be interpreted as a K channel indicating a degree of proximity toward black as in the cases when the OUTPUT_CLR_FMT is CMYK or CMYKDIRECT (see also Annex C).

PIXEL_FORMAT values having mnemonic names that end in "FixedPoint" specify fixed point representations.

NOTE 6 A fixed point numerical representation is not commonly supported in prior image file formats. It is supported in JPEG XR as a way to encode an extended range of numerical values more directly while retaining the performance advantages of integer processing.

JPEG XR supports fixed point numerical encoding for 16-bit and 32-bit signed values. In this document, the abbreviation SINT is used to refer to signed integer or fixed point values.

Fixed point values having BPC equal to 16 or 32 are interpreted as follows.

— 16-bit Fixed Point, a format referred to as s2.13: The 16 bits that make up an individual value are interpreted as a sign bit, two integer bits and thirteen fractional bits. Using this interpretation, a numerical range of −4.0 to +3.999… can be represented, with the value of 1.0 represented by the signed integer value 8 192 (0x2000).

— 32-bit Fixed Point, a format referred to as s7.24: The 32 bits that make up an individual value are interpreted as a sign bit, seven integer bits and twenty-four fractional bits. Using this interpretation, a numerical range of −128.0 to +127.999… can be represented, with the value of 1.0 represented by the signed integer value 16 777 216 (0x01000000).

NOTE 7 JPEG XR does not enable fully lossless compression for 32-bit data in general. The encoding and decoding algorithms use 32-bit computations, and some dynamic range is lost to necessary headroom for signal processing calculations such as overlap and core transform computations. A minimum of 22 bits and typically 24 bits or more precision is retained through the end-to-end encoding and decoding process.

