
**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 29199-2:2012(E)

Documents
Document Sharing Hub
© ISO/IEC 2012



COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2012

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Page

Foreword	v
Introduction	vi
1 Scope.....	1
2 Normative references	1
3 Terms and definitions.....	1
4 Abbreviations	8
5 Conventions	8
5.1 Conformance language	8
5.2 Operators.....	8
5.3 Syntax and semantics notation	14
5.4 Formatting conventions	16
5.5 Global variables	17
5.6 Adaptive VLC deltaDisc tables.....	23
5.7 Adaptive inverse scanning tables	23
6 Image and codestream structures	24
6.1 General.....	24
6.2 Image planes and component arrays	24
6.3 Image windowing	24
6.4 Image partitioning	25
6.5 Transform coefficients and frequency bands	26
6.6 Codestream structure	26
6.7 Precision and word length.....	27
7 Overview of decoder	27
7.1 General.....	27
7.2 Overview of parsing process.....	28
7.3 Overview of the decoding process	29
8 Syntax, semantics, and parsing process.....	30
8.1 General.....	30
8.2 CODED_IMAGE()	31
8.3 IMAGE_HEADER()	33
8.4 IMAGE_PLANE_HEADER()	42
8.5 INDEX_TABLE_TILES()	50
8.6 PROFILE_LEVEL_INFO()	51
8.7 CODED_TILES()	52
8.8 Adaptive VLC code table selection	85
8.9 Adaptation of CBPLP state variables.....	92
8.10 Adaptive CBPHP prediction	93
8.11 Adaptive inverse scanning	94
8.12 Adaptive coefficient normalization.....	97
9 Decoding process	99
9.1 General.....	99
9.2 Image decoding.....	99
9.3 Image plane decoding	100
9.4 Tile transform coefficient processing	101
9.5 Coefficient remapping	102
9.6 Transform coefficient prediction	104
9.7 Derivation of quantization parameters	114

9.8	Dequantization	117
9.9	Sample reconstruction.....	120
9.10	Output formatting	151
Annex A (normative) Tag-based file format		168
Annex B (normative) Profiles and Levels		194
Annex C (informative) Color imagery representation and color management.....		197
Annex D (informative) Encoder processing.....		200
Annex E (informative) Patent Rights		215
Bibliography.....		216

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

ISO/IEC 29199-2 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.

This part of ISO/IEC 29199 is technically aligned with ITU-T Rec. T.832 but is not published as identical text.

This third edition cancels and replaces the second edition (ISO/IEC 29199-2:2010), which has been technically revised.

ISO/IEC 29199 consists of the following parts, under the general title *Information technology — JPEG XR image coding system*:

- *Part 1: System architecture* [Technical Report]
- *Part 2: Image coding specification*
- *Part 3: Motion JPEG XR*
- *Part 4: Conformance testing*
- *Part 5: Reference software*

Introduction

This part of ISO/IEC 29199 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 color 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 color 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} color 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 color 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 color types by converting these color formats to an internal luma-dominant format through the use of a reversible color transform. In addition, YUV, monochrome and arbitrary n-channel color 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 part of ISO/IEC 29199 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 color imagery representations and color management are discussed in Annex C. The typical expected encoding process is described in Annex D.

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 companies listed in Annex E.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights other than those identified above. 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 part of ISO/IEC 29199 specifies a coding format, referred to as JPEG XR, which is designed primarily for continuous-tone photographic content.

2 Normative references

Normative references having a scope that is limited to the use of the file format specified in Annex A are listed in A.2.

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

NOTE Definitions of terms having a scope that is limited to the use of the **file format** specified in Annex A are listed in A.2.

3.1

adaptive coefficient normalization

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

NOTE 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** associated with a set of **transform coefficients** 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** 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 associated with an image of the same dimensions as the luma component of the primary image plane

NOTE The alpha image plane has one component, a luma component.

3.5

block

$m \times n$ array of **samples**, or an $m \times n$ array of **transform coefficients**

3.6

block index

integer in the range 0 to 15 identifying, by its position in **raster scan order**, a particular 4×4 **block** 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** is **byte-aligned** if 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 of the **primary image plane** with non-zero index, or the **transform coefficients** and sample values associated with this **component**

3.10

coded block pattern high-pass

coded block pattern high-pass is a **syntax element** indicating the **coded block status**, i.e. the presence or absence of non-zero **high-pass transform coefficients**, for each of the **blocks** in the **macroblock**

3.11

coded block pattern low-pass

coded block pattern low-pass is a **syntax element** indicating the presence or absence of non-zero **low-pass transform coefficients** in the **macroblock**

3.12

coded block status

coded block status is an indication of the presence or absence of non-zero **transform coefficients** in that **block**

3.13

codestream

sequence of bits contained in a sequence of **bytes** from which syntax elements are parsed, such that 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**

3.15

context

possible value of a specific instance of a **context variable**

3.16

context variable

variable used in the **parsing process** to select which data structure is to be used for the **adaptive VLC** parsing of a given syntax element

3.17

DC coefficient

first subset when the **transform coefficients**, that are contained in a specific **macroblock** and a specific **component**, are partitioned into 3 subsets

3.18**DC-LP array**

array of all DC and low-pass **transform coefficients**, for all **macroblocks** associated with a specific **component**

3.19**decoder**

embodiment of a **parsing process** and **decoding process**

3.20**decoding process**

process of computing output sample values from the parsed syntax elements of the **codestream**

3.21**dequantization**

process of rescaling the quantized **transform coefficients** after their value has been parsed from the **codestream** and before they are presented to the **inverse transform process**

3.22**discriminant**

collective term for one of DiscrimVal1 or DiscrimVal2, which are the two member variables of an instance of the **adaptive VLC** data structure specified in subclause 5.5.5.

3.23**encoder**

embodiment of an **encoding process**

3.24**encoding process**

process of converting source sample values into a **codestream** conforming to this part of ISO/IEC 29199

3.25**extended image**

image produced by the **decoding process** prior to **windowing**

NOTE The **extended image** has a **luma** array that is an integer multiple of 16 in width and height.

3.26**file**

finite-length sequence of **bytes** that is accessible to a **decoder** in a manner such that the **decoder** can obtain access to the data at specified positions within the sequence of **bytes** (e.g. 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**

3.28**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 part of a **DC coefficient** or **low-pass coefficient** that is parsed using adaptive **fixed-length codes**

3.30**flexbits**

fixed-length-coded part of the **high-pass coefficient** information which is parsed using adaptive **fixed-length codes**

3.31

frequency band

collective term for one of the following three subsets of the **transform coefficients** for an **image**, which are separately parsed: **DC coefficients**, **low-pass coefficients**, and **high-pass coefficients**

3.32

frequency mode

codestream structure mode where the DC, low-pass, high-pass and **flexbits frequency bands** for each **tile** are grouped separately

3.33

hard tiles

codestream 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** that are contained in a specific **macroblock** and a specific **component** are partitioned into 3 subsets

3.35

image

result of the **decoding process**, consisting of a **primary image plane** and an optional **alpha image plane**

3.36

image plane

collective term for a grouping of the **components** of the **image**

3.37

initial level value

one of two values used to compute the **VLC-coded** part of a **transform coefficient**

3.38

internal color format

color format associated with the spatial-domain samples obtained through the **inverse transform process** and the **sample reconstruction process**, and distinguished from the **output color format** associated with the **output formatting process**

3.39

inverse core transform (ICT)

two steps of the **inverse transform process** that involve processing of **transform coefficients** associated with each **macroblock** independently, with no **overlap filtering**

3.40

inverse transform process

part of the **decoding process** by which a set of **dequantized transform coefficients** are converted into spatial-domain values

3.41

inverse scanning

process of reordering an ordered set of parsed **syntax elements** from the **codestream** to form an array of **transform coefficients** associated with a specific **component** and **macroblock**

3.42

little-endian form

ordering of the **bytes** that represent a numerical value as an integer number of **bytes** in which the **bytes** representing the number are in ascending order of significance, i.e. with the least significant **byte** first, followed by the next least significant **byte**, etc.

3.43**low-pass coefficient**

second subset, when the **transform coefficients** that are contained in a specific **macroblock** and a specific **component** are partitioned into 3 subsets

3.44**luma**

component of an **image plane** with index zero, and the **transform coefficients** and sample values associated with this **component**

NOTE Although this term is commonly associated with a signal that conveys perceptual brightness information, as used in this International Standard the term is primarily an identifier of a particular array of samples or **transform coefficients** for an **image**.

3.45**macroblock**

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

3.46**macroblock partition**

partitioning of each **component**, into 16×16, 8×8, or 16×8 **blocks**, depending on the **internal color format**

3.47**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** that are the result of the **decoding process**

3.48**output color format**

color format associated with the output **image** that is the result of the **decoding process**

3.49**output formatting process**

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

NOTE This specifies a conversion (if necessary) into the appropriate **output color format** and **output bit depth**.

3.50**output image height**

height of the sub-array of the **luma component** of the **primary image plane** that is output by the **decoding process**

3.51**output image width**

width of the sub-array, of the **luma component** of the **primary image plane** that is output by the **decoding process**

3.52**overlap filtering**

steps of the **inverse transform process** that involve processing of **transform coefficients** across adjacent **blocks** and **macroblocks**

NOTE When **overlap filtering** is applied, it is applied across **macroblock** boundaries as well as **block** boundaries. When the **codestream** uses **soft tiles**, the **overlap filtering** is also applied across **tile** boundaries. Otherwise, **overlap filtering** does not occur across **tile** boundaries.

3.53**parsing process**

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

3.54

prediction

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

3.55

prediction residual

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

3.56

primary image plane

image plane that consists of all **image components** that are not a part of the **alpha image plane**

3.57

QP index

integer, which for a particular **frequency band** and **macroblock** specifies the index into the table of **quantization parameters** available for this **frequency band** and **tile**

NOTE The **QP index** thereby selects, for this **macroblock**, the **quantization parameter** used for the **dequantization** of the **transform coefficients** in the specific **frequency band**.

3.58

QP set

set of **quantization parameters** associated with a particular **frequency band**, corresponding to the **luma** and **chroma components**

3.59

quantization parameter (QP)

value used to compute the scaling factor for the **dequantization** of a **transform coefficient**, before the **inverse transform process** is applied

3.60

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

3.61

refinement

process of modifying a predicted or partially-computed **transform coefficient**

3.62

run

number of zero valued coefficient levels that precede a non-zero valued coefficient level in the **zigzag scan order** during the **inverse scanning** process

3.63

sample reconstruction process

process of converting dequantized **transform coefficients** into samples of the **image**

3.64

soft tiles

codestream structure mode where the overlap operators are applied across tile boundaries

3.65

spatial co-location

sub-arrays of samples are **spatially co-located** across **components** when they correspond to the same spatial region of the decoded **image**

NOTE The **macroblock partition** of the **image** ensures that the i -th **macroblock** horizontally and j -th **macroblock** vertically across all **components** are **spatially co-located**.

3.66**spatial mode**

codestream structure mode where the **DC**, **low-pass**, **high-pass** and **flexbits frequency bands** for each specific **macroblock** are grouped together

3.67**spatial transformation**

element in the **codestream** indicating the preferred final displayed orientation of the decoded **image**, as specified in subclause 8.3.8

NOTE The **spatial transformation** is only a suggestion, and **decoder** conformance is checked only for the decoded **image** prior to the application of this transformation (i.e. for orientation 0).

3.68**start code**

bit pattern that specifies the beginning of a **tile packet** or other distinguished, contiguous set of syntax elements in the **codestream**

3.69**tile**

collection of **macroblocks** that have the same indices i and j with respect to a **tile partition**

NOTE Each **tile** corresponds to the **macroblocks** for a rectangular region of the **image**.

3.70**tile packet**

contiguous subset of the **codestream**, which contains the coded **syntax elements** associated with a specific **tile**

3.71**tile partition**

partition of the **image** into rectangular arrays of **macroblocks**, as specified in subclause 6.4

3.72**transform coefficients**

values, associated with each specific **macroblock** and specific **component**, that — after **dequantization** — form the input arrays into the **inverse transform process**

3.73**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

3.74**VLC refinement**

one of two values used to compute the **VLC-coded** part of a **transform coefficient**

NOTE The number of bits required to specify the **VLC-refinement** is dependent on the value of the **initial level value**. The **VLC refinement** is added to the **initial level value** to produce the **VLC-coded** part of the transform coefficient.

3.75**windowing**

selection of **spatially co-located** sub-arrays of the **components** of all present **image planes** associated with an **image** that are output by the **decoding process**

3.76**zigzag scan order**

adaptive ordering for the **inverse scanning** process, which assigns array indices to each subsequent **transform coefficient** parsed from the **codestream**

4 Abbreviations

For the purposes of this document, the following abbreviations apply. Abbreviations having a scope that is limited to the use of the file format specified in Annex A are listed in subclause A.4.

CBPHP	Coded block pattern high-pass
CBPLP	Coded block pattern low-pass
FCT	Forward core transform
FLC	Fixed-length code
HP	High-pass
ICT	Inverse core transform
JPEG	Joint Photographic Experts Group
LP	Low-pass
LSB	Least significant bit
MSB	Most significant bit
QP	Quantization parameter
VLC	Variable-length code

5 Conventions

5.1 Conformance language

This International Standard consists of normative and informative text.

Normative text is that text which expresses mandatory requirements. The word "shall" is used to express mandatory requirements to be followed strictly in order to conform to this Specification and from which no deviation is permitted. A conforming implementation is one that fulfils all mandatory requirements.

Informative text is text that is potentially helpful to the user, but not indispensable and can be removed, changed or added editorially without affecting interoperability. All text in this International Standard is normative, with the following exceptions: the Introduction, any parts of the text that are explicitly labelled as "informative", statements appearing with the preamble "NOTE", behaviour described using the word "should", and pseudocode comments delimited as specified in subclause 5.2.7. The word "should" is used to describe behaviour that is preferred but is not necessarily required for conformance to this Specification.

The keywords "may" and "need not" indicate a course of action that is permissible in a conforming implementation.

The keyword "reserved" indicates a provision that is not specified at this time, shall not be used in implementations conforming to this version of this Specification 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 Specification 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 $x \geq 0$ 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 Specification.

NOTE 2 – The division operators used in this Specification 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.
$x \gg b$	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.
$x \ll b$	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 $x = x - a$
$\wedge=$	$x \wedge= a$ is defined as $x = x \wedge a$
$\ast=$	$x \ast= a$ is defined as $x = x \ast a$
$\ll=$	$x \ll= a$ is defined as $x = (x \ll a)$
$\gg=$	$x \gg= a$ is defined as $x = (x \gg a)$

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

ExamplePseudocodeArrayInitialization() {	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.
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 is 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 Specification.

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, DONLY	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 $0 \leq i < \text{NumComponents}$ (subclause 8.4.11), $0 \leq x < \text{ExtendedWidth}[i]$, $0 \leq y < \text{ExtendedHeight}[i]$, 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 $i = 0$, $0 \leq x < \text{ExtendedWidth}[i]$, $0 \leq y < \text{ExtendedHeight}[i]$, 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 color 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 color 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 color 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 color 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 color component indexed by i. The index j ranges from 0 to 15 for luma components of all color formats and chroma components of all color 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 color 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 color 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 color 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 color component i, located at the position indicated by the values x and y, where $0 \leq i < \text{NumComponents}$, $0 \leq x < \text{ExtendedWidth}[i]$, $0 \leq y < \text{ExtendedHeight}[i]$, 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 $0 \leq \text{TileIndexx} < \text{NumTileCols}$.

TileIndexy: This variable holds the row index of the current tile. The value of TileIndexy is in the range $0 \leq \text{TileIndexy} < \text{NumTileRows}$.

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 color component i of the current tile.

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

HPQuantParam[i][j]: This variable holds the HP quantization parameter for the color 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 color component i.

LPInput[i][j]: This variable holds the j-th LP transform coefficient value for each color 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 color 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 color 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 color 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 Image and codestream structures

6.1 General

In Clause 6, italic font formatting is used to identify all 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 $1 \leq \text{NumComponents} \leq 4111$. For this *primary image plane*, each *component* is an $\text{ExtendedHeight}[i] \times \text{ExtendedWidth}[i]$ array of samples, where $\text{ExtendedWidth}[i]$ and $\text{ExtendedHeight}[i]$ specify (respectively) the width and height of the array for the i -th *component*, for the index i , $0 \leq i < \text{NumComponents}$.

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.

$\text{ExtendedHeight}[0]$ 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 $\text{HEIGHT_MINUS1} + 1 + \text{TOP_MARGIN} + \text{BOTTOM_MARGIN}$.

$\text{ExtendedWidth}[0]$ 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 $\text{WIDTH_MINUS1} + 1 + \text{LEFT_MARGIN} + \text{RIGHT_MARGIN}$.

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

Table 17 – Pseudocode to calculate chroma component array sizes

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

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 $0 \leq j < (\text{ExtendedWidth}[0] / 16)$ and $0 \leq k < (\text{ExtendedHeight}[0] / 16)$ are the *block indices* defined by this partitioning. MBHeight is defined to be equal to $\text{ExtendedHeight}[0] / 16$, and MBWidth is defined to be equal to $\text{ExtendedWidth}[0] / 16$.

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* arrayComponentMB[i][j][k], for $0 \leq i < \text{NumComponents}$. Across all *components* (all values of i), arrayComponent[i][j][k] and arrayChromaMB[i][j][k] are *spatially co-located*.

Let $0 = \text{LeftMBIndexofTile}[0] < \text{LeftMBIndexofTile}[1] < \dots < \text{LeftMBIndexofTile}[\text{NumTileCols}] = \text{MBWidth}$, and $0 = \text{TopMBIndexofTile}[0] < \text{TopMBIndexofTile}[1] < \dots < \text{TopMBIndexofTile}[\text{NumTileRows}] = \text{MBHeight}$ be two increasing sequences of integers, where the sequences are of length $\text{NumTileCols} + 1$ and $\text{NumTileRows} + 1$, respectively. $\text{LeftMBIndexofTile}[]$ is calculated by calling DetermineLeftBoundaryofTile() (subclause 8.3.25) and $\text{TopMBIndexofTile}[]$ 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 $0 \leq m < \text{NumTileCols}$, and $0 \leq n < \text{NumTileRows}$, where arrayTile[m][n] is defined to be the set of all *macroblocks* MB[j][k] $\text{LeftMBIndexofTile}[m] \leq j < \text{LeftMBIndexofTile}[m+1]$ and $\text{TopMBIndexofTile}[n] \leq k < \text{TopMBIndexofTile}[n+1]$.

The *i*-th *tile column* corresponds to the set of all *tiles* of the form arrayTile[i][n], for $0 \leq n < \text{NumTileRows}$; similarly, the *j*-th *tile row* corresponds to the set of all *tiles* arrayTile[m][j], for $0 \leq m < \text{NumTileCols}$. The *i*-th *tile width* is defined to be $\text{LeftMBIndexofTile}[i+1] - \text{LeftMBIndexofTile}[i]$; likewise, the *i*-th *tile height* is defined to be $\text{TopMBIndexofTile}[i+1] - \text{TopMBIndexofTile}[i]$. 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 $1 \leq \text{NumTileCols} \leq 4096$, and $1 \leq \text{NumTileRows} \leq 4096$.

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

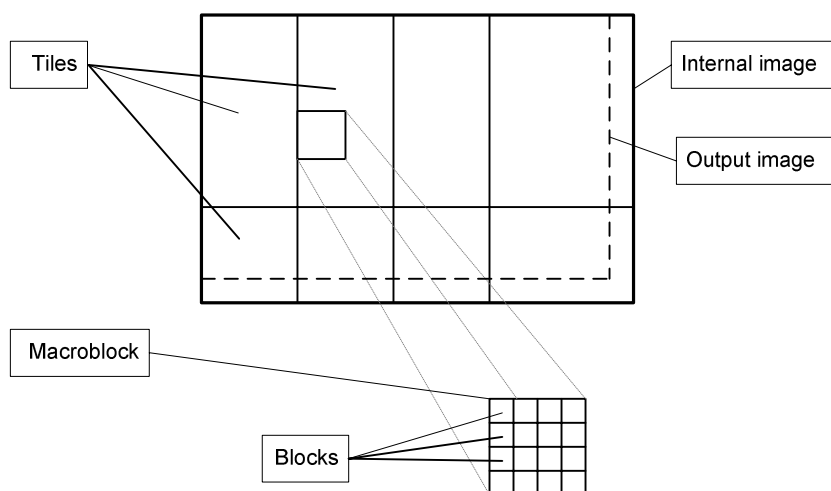


Figure 1 – Informative overview of Image partitions and internal windowing

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 color *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 $\frac{1}{2}$ 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 $\frac{1}{2}$ 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.

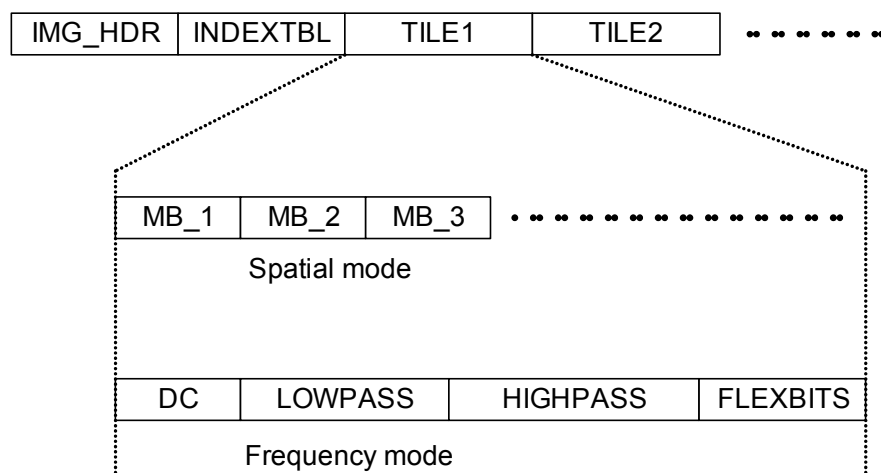


Figure 2 – Informative overview of codestream structure. Image header is followed by a sequence of tiles which are in Spatial or Frequency mode

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 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

This subclause is informative: It is not an integral part of this Specification.

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 precision 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

This clause is informative: it is not an integral part of this Specification.

7.1 General

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

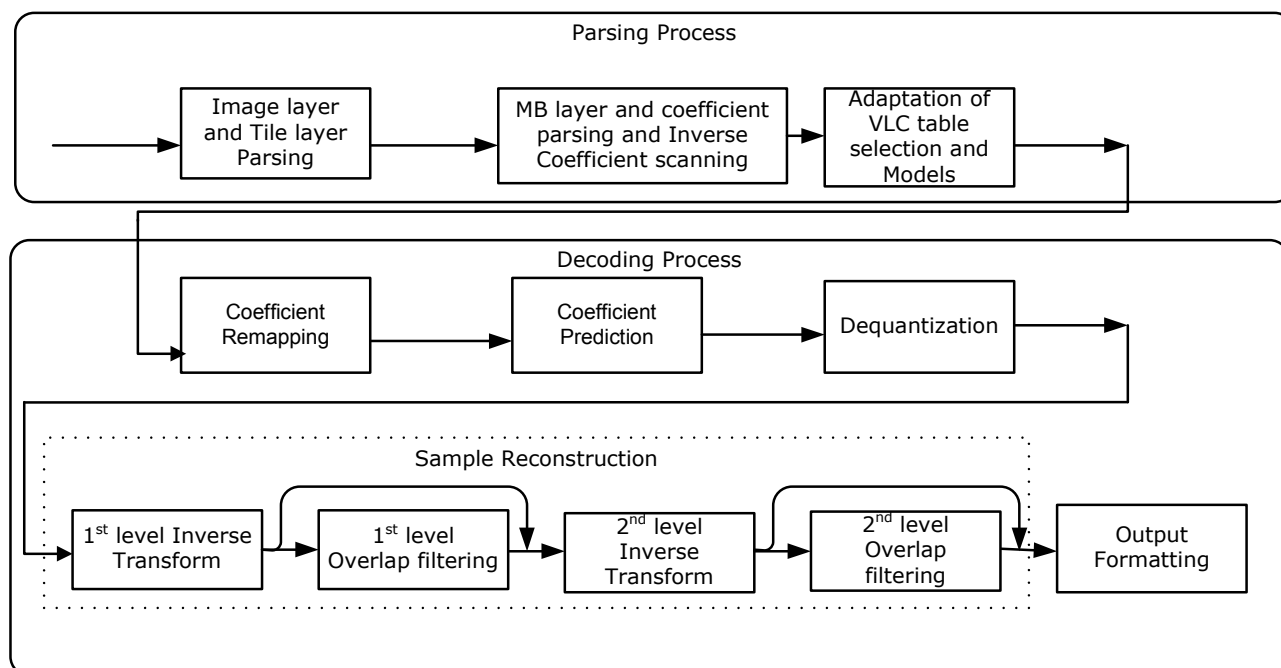


Figure 3 – Informative decoding process block diagram

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 flow chart for the sample reconstruction process is shown in Figure 4.

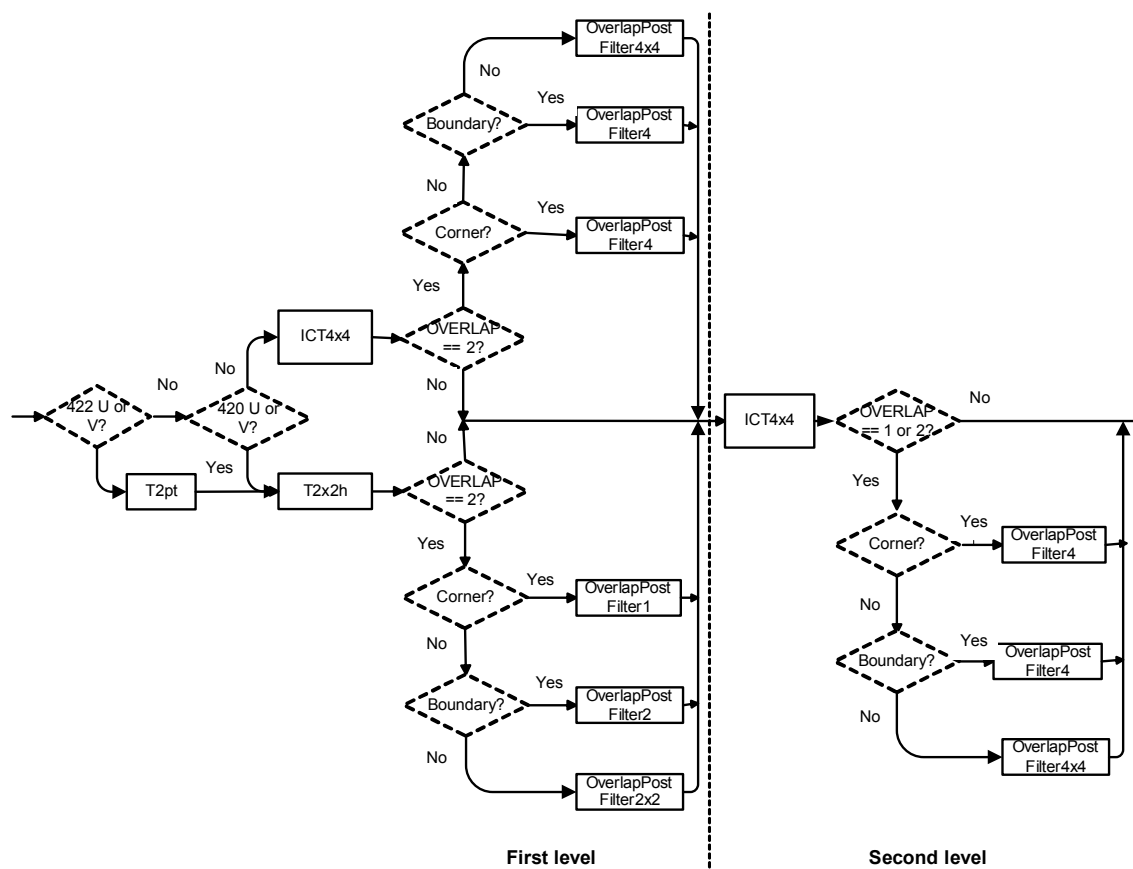


Figure 4 – Informative overview of sample reconstruction process

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 color 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 version of this Specification. 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 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 version of this Specification. All other values are reserved.

NOTE – Alternative values for RESERVED_B may 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 version of this Specification. All other values are reserved. Decoders conforming to this version of this Specification 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		TL
1	0	0	1		BL
2	0	1	0		TR
3	0	1	1		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 the informative 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 Specification 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 Specification (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 Specification 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 Specification.

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 version of this Specification. The value 1 is reserved. Decoders conforming to this version of this Specification 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 Specification. 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 Specification have operated in a manner corresponding to the value 0.

8.3.17 PREMULIPLIED_ALPHA_FLAG

PREMULIPLIED_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 PREMULIPLIED_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, PREMULIPLIED_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, PREMULIPLIED_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 PREMULIPLIED_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 Specification 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 PREMULIPLIED_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 PREMULIPLIED_ALPHA_FLAG was not included in the original edition of this Specification. The specification of PREMULIPLIED_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 Specification, the bit corresponding to PREMULIPLIED_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 PREMULIPLIED_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 PREMULIPLIED_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 color 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 color components: the component corresponding to index 0 is the Y component, while the U and V correspond to color 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. 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 Specification that the value of WIDTH_MINUS1 + 1 shall be an integer multiple of 2.

It is a requirement of codestream conformance to this Specification 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 Specification 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 Specification that the value of HEIGHT_MINUS1 + 1 shall be an integer multiple of 2.

It is a requirement of codestream conformance to this Specification 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 a 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 a 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
<code>TopMBIndexOfTile[0] = 0</code>	
<code>for (n = 0; n < NUM_HOR_TILES_MINUS1; n++)</code>	
<code>TopMBIndexOfTile[n+1] = TopMBIndexOfTile[n] + TILE_HEIGHT_IN_MB[n]</code>	
<code>TopMBIndexOfTile[NUM_HOR_TILES_MINUS1 + 1] = MBHeight</code>	
<code>}</code>	

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

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 Specification 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 Specification 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 $16 - ((\text{HEIGHT_MINUS1} + 1) \% 16)$.

When OUTPUT_CLR_FMT is equal to YUV420, it is a requirement of codestream conformance to this Specification 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 $16 - ((\text{WIDTH_MINUS1} + 1) \% 16)$.

When OUTPUT_CLR_FMT is equal to YUV420 or YUV422, it is a requirement of codestream conformance to this Specification 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

IMAGE_PLANE_HEADER() {	Descriptor	Reference
INTERNAL_CLR_FMT	u(3)	8.4.2
SCALED_FLAG	u(1)	8.4.3
BANDS_PRESENT	u(4)	8.4.4
if (INTERNAL_CLR_FMT == YUV444 INTERNAL_CLR_FMT == YUV420 INTERNAL_CLR_FMT == YUV422) {		
if (INTERNAL_CLR_FMT == YUV420 INTERNAL_CLR_FMT == YUV422) {		
RESERVED_E_BIT	u(1)	8.4.5
CHROMA_CENTERING_X	u(3)	8.4.6
} else /* INTERNAL_CLR_FMT == YUV444 */		
RESERVED_F	u(4)	8.4.7
if (INTERNAL_CLR_FMT == YUV420) {		
RESERVED_G_BIT	u(1)	8.4.8
CHROMA_CENTERING_Y	u(3)	8.4.9
} else		
RESERVED_H	u(4)	8.4.10
} else if (INTERNAL_CLR_FMT == NCOMPONENT) {		
NUM_COMPONENTS_MINUS1	u(4)	8.4.11
if (NUM_COMPONENTS_MINUS1 == 0xF)		
NUM_COMPONENTS_EXTENDED_MINUS16	u(12)	8.4.12
else		
RESERVED_H	u(4)	8.4.10
}		
if (OUTPUT_BITDEPTH == BD16 OUTPUT_BITDEPTH == BD16S OUTPUT_BITDEPTH == BD32S)		
SHIFT_BITS	u(8)	8.4.13
if (OUTPUT_BITDEPTH == BD32F) {		
LEN_MANTISSA	u(8)	8.4.14
EXP_BIAS	i(8)	8.4.15
}		
DC_IMAGE_PLANE_UNIFORM_FLAG	u(1)	8.4.16
if (DC_IMAGE_PLANE_UNIFORM_FLAG)		
DC_QP()		8.4.22
if (BANDS_PRESENT != DONLY) {		
RESERVED_I_BIT	u(1)	8.4.17
LP_IMAGE_PLANE_UNIFORM_FLAG	u(1)	8.4.18
if (LP_IMAGE_PLANE_UNIFORM_FLAG) {		
NumLPQPs = 1		
LP_QP()		8.4.23
}		
if (BANDS_PRESENT != NOHIGHPASS) {		
RESERVED_J_BIT	u(1)	8.4.19
HP_IMAGE_PLANE_UNIFORM_FLAG	u(1)	8.4.20
if (HP_IMAGE_PLANE_UNIFORM_FLAG) {		
NumHPQPs = 1		
HP_QP()		8.4.24
}		
}		
}		
while (!IS_BYTE_ALIGNED())		
BYTE_ALIGNMENT_BIT	u(1)	8.4.21
}		