International Telecommunication Union

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

T.832

Corrigendum 2
(05/2011)

SERIES T: TERMINALS FOR TELEMATIC SERVICES

Still-image compression – JPEG XR

Information technology – JPEG XR image coding system – Image coding specification

Corrigendum 2: Corrections and clarifications

Recommendation ITU-T T.832 (2009) – Corrigendum 2

ITU-T T-SERIES RECOMMENDATIONS

TERMINALS FOR TELEMATIC SERVICES

Facsimile – Framework	T.0–T.19
Still-image compression – Test charts	T.20–T.29
Facsimile – Group 3 protocols	T.30–T.39
Colour representation	T.40–T.49
Character coding	T.50–T.59
Facsimile – Group 4 protocols	T.60–T.69
Telematic services – Framework	T.70–T.79
Still-image compression – JPEG-1, Bi-level and JBIG	T.80–T.89
Telematic services – ISDN Terminals and protocols	T.90–T.99
Videotext – Framework	T.100–T.109
Data protocols for multimedia conferencing	T.120–T.149
Telewriting	T.150–T.159
Multimedia and hypermedia framework	T.170–T.189
Cooperative document handling	T.190–T.199
Telematic services – Interworking	T.300–T.399
Open document architecture	T.400–T.429
Document transfer and manipulation	T.430–T.449
Document application profile	T.500–T.509
Communication application profile	T.510–T.559
Telematic services – Equipment characteristics	T.560–T.649
Still-image compression – JPEG 2000	T.800–T.829
Still-image compression – JPEG XR	**T.830–T.849**
Still-image compression – JPEG-1 extensions	T.850–T.899

For further details, please refer to the list of ITU-T Recommendations.

Recommendation ITU-T T.832

Information technology – JPEG XR image coding system – Image coding specification

Corrigendum 2

Corrections and clarifications

Summary

Recommendation ITU-T T.832 specifies a coded image format, referred to as JPEG XR, which is designed primarily for storage and interchange of continuous-tone photographic content. The main body of the text specifies the syntax and semantics of JPEG XR coded images and the associated decoding process. 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.

Corrigendum 1 to the 2009 edition of Recommendation ITU-T T.832 (approved 12/2009) contained~~s~~ a set of various minor corrections.

Corrigendum 2 contains additional minor corrections approved 05/2011.

The text was developed as a twin text Recommendation corresponding to ISO/IEC 29199-2 in collaboration with ISO/IEC JTC 1/SC 29/WG 1 (JPEG).

History

Edition	Recommendation	Approval	Study Group
1.0	ITU-T T.832	2009-03-16	16
1.1	ITU-T T.832 (2009) Cor.1	2009-12-14	16
1.2	ITU-T T.832 (2009) Cor. 2	2011-05-14	16

Table of Contents

Page

Foreword

This Recommendation | International Standard specifies a coded image format, referred to as JPEG XR, which is designed primarily for storage and interchange of continuous-tone photographic content. The main body of the text specifies the syntax and semantics of JPEG XR coded images and the associated decoding process. 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.

Introduction

This Recommendation | International S~~s~~tandard 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 Rec. ITU-T 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 Specification 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.

Corrigendum 1 (2009) and Corrigendum 2 (2011) introduce~~s~~ a set of various minor corrections to Rec. ITU-T T.832 (2009).

Recommendation ITU-T T.832

Information technology – JPEG XR image coding system – Image coding specification

Corrigendum 2

Corrections and clarifications

Modifications introduced by this corrigendum are shown in revision marks. Unchanged text is replaced by ellipsis (…). Some parts of unchanged texts (clause numbers, etc.) may be kept to indicate the correct insertion points.

1 Scope

This Recommendation | International S~~s~~tandard specifies a coding format, referred to as JPEG XR, which is designed primarily for continuous-tone photographic content.

• • •

3 Definitions

• • •

3.1 adaptive coefficient normalization: A 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. 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*.

• • •

3.28 fixed-length code (FLC): A 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: The *fixed-~~–~~length coded* part of a *DC coefficient* or *low-pass coefficient* that is parsed using adaptive fixed-length codes.

3.30 flexbits: The *fixed-~~–~~length coded* part of the *high-pass coefficient* information which is parsed using adaptive fixed-length codes.

• • •

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

• • •

3.59 quantization parameter (QP): A value used to compute the scaling factor for the *dequantization* of a *transform coefficient*, before the *inverse transform process* is applied.

• • •

3.73 variable-~~–~~length code (VLC): A 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~~, to a specific set of values~~.

• • •

4 Abbreviations

For the purposes of this Recommendation | International Standard, 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.

• • •

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

• • •

5.3.1 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.

• • •

6.2 Image planes and component arrays

• • •

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

• • •

6.4 Image partitioning

• • •

Let 0 = LeftMBIndexOfTile[0] < LeftMBIndexOfTile[1] < … < LeftMBIndexOfTile[NumTileCols] = MBWidth, and 0 = TopMBIndexOfTile[0] < TopMBIndexOfTile[1] < … < TopMBIndexOfTile[NumTileRows] = MBHeight be two increasing sequences of integers, where the sequences are of length NumTileCols + 1 and NumTileRows + 1, respectively. LeftMBIndexOfTile[ ] is calculated by calling DetermineLeftBoundaryofTile( ) (subclause 8.3.24~~8.3.23~~) and TopMBIndexOfTile[ ] is calculated by calling DetermineTopBoundaryofTile( ) (subclause 8.3.25~~8.3.24~~). Associated with any such pair of sequences, a *tile partition* may be defined: partition the *macroblocks* of each into *tiles* arrayTile[m][n], for 0 <= m < NumTileCols, and 0 <= n < NumTileRows, where arrayTile[m][n] is defined to be the set of all *macroblocks* MB[j][k] LeftMBIndexOfTile[m] <= j < LeftMBIndexOfTile[m+1] and TopMBIndexOfTile[n] <= k < TopMBIndexOfTile[n+1].

• • •

6.5 Transform coefficients and frequency bands

• • •

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

• • •

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

• • •

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 ~~necessarily~~ 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.

• • •

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(2~~3~~)	8.3.15
PREMULTIPLIED_ALPHA_FLAG	u(1)	8.3.16
ALPHA_IMAGE_PLANE_FLAG	u(1)	8.3.17~~8.3.16~~
OUTPUT_CLR_FMT	u(4)	8.3.18~~8.3.17~~
OUTPUT_BITDEPTH	u(4)	8.3.19~~8.3.18~~
if (SHORT_HEADER_FLAG) {		
WIDTH_MINUS1	u(16)	8.3.20~~8.3.19~~
HEIGHT_MINUS1	u(16)	8.3.21~~8.3.20~~
} else {		
WIDTH_MINUS1	u(32)	8.3.20~~8.3.19~~
HEIGHT_MINUS1	u(32)	8.3.21~~8.3.20~~
}		
if (TILING_FLAG) {		
NUM_VER_TILES_MINUS1	u(12)	8.3.22~~8.3.21~~
NUM_HOR_TILES_MINUS1	u(12)	8.3.23~~8.3.22~~
}		
for (n = 0; n < NUM_VER_TILES_MINUS1; n++)		
if (SHORT_HEADER_FLAG)		
TILE_WIDTH_IN_MB[n]	u(8)	8.3.24~~8.3.23~~
else		
TILE_WIDTH_IN_MB[n]	u(16)	8.3.24~~8.3.23~~
for (n = 0; n < NUM_HOR_TILES_MINUS1; n++)		
if (SHORT_HEADER_FLAG)		
TILE_HEIGHT_IN_MB[n]	u(8)	8.3.25~~8.3.24~~
else		

Table 20 – IMAGE_HEADER( ) syntax structure

IMAGE_HEADER( ) {	Descriptor	Reference
TILE_HEIGHT_IN_MB[n]	u(16)	8.3.25~~8.3.24~~
if (WINDOWING_FLAG) {		
TOP_MARGIN	u(6)	8.3.26~~8.3.25~~
LEFT_MARGIN	u(6)	8.3.27~~8.3.26~~
BOTTOM_MARGIN	u(6)	8.3.28~~8.3.27~~
RIGHT_MARGIN	u(6)	8.3.29~~8.3.28~~
}		
}		

• • •

8.3.8 SPATIAL_XFRM_SUBORDINATE

• • •

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

NOTE ~~3~~– 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.

• • •

8.3.10 OVERLAP_MODE

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

When~~If~~ 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.

• • •

8.3.13 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 8.3.26~~8.3.25~~ through 8.3.29~~8.3.28~~) 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.

• • •

8.3.15 RESERVED_D

RESERVED_D is a 2~~3~~-bit syntax element. The value of RESERVED_D shall be equal to 0. Other values are reserved. Decoders shall ignore the value of this syntax element.

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

8.3.16 PREMULTIPLIED_ALPHA_FLAG

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

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

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

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

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

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

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

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

8.3.17~~6~~ ALPHA_IMAGE_PLANE_FLAG

• • •

8.3.18~~7~~ OUTPUT_CLR_FMT

• • •

8.3.19~~8~~ OUTPUT_BITDEPTH

• • •

8.3.20~~19~~ WIDTH_MINUS1

• • •

8.3.2~~0~~1 HEIGHT_MINUS1

• • •

8.3.2~~1~~2 NUM_VER_TILES_MINUS1

• • •

8.3.2~~2~~3 NUM_HOR_TILES_MINUS1

• • •

8.3.24~~3~~ TILE_WIDTH_IN_MB[n]

• • •

8.3.25~~4~~ TILE_HEIGHT_IN_MB[n]

• • •

8.3.26~~5~~ TOP_MARGIN

• • •

8.3.27~~6~~ LEFT_MARGIN

• • •

8.3.28~~7~~ BOTTOM_MARGIN

• • •

8.3.29~~8~~ RIGHT_MARGIN

• • •

8.6 PROFILE_LEVEL_INFO( )

8.6.1 Syntax structure

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

Table 37 – PROFILE_LEVEL_INFO( ) syntax structure

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

• • •

8.7.5 TILE_LOWPASS( )

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

Table 42 – TILE_LOWPASS( ) syntax structure

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

8.7.6 TILE_HEADER_LOWPASS( )

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

Table 43 – TILE_HEADER_LOWPASS( ) syntax structure

TILE_HEADER_LOWPASS( ) {	**Descriptor**	**Reference**
if (LP_IMAGE_PLANE_UNIFORM_FLAG = = FALSE) {		
USE_DC_QP_FLAG	u(1)	8.7.10.4
if (USE_DC_QP_FLAG~~= = FALSE) {~~		
NumLPQPs = 1		

else {		
NUM_LP_QPS_MINUS1	u(4)	8.7.10.5
NumLPQPs = NUM_LP_QPS_MINUS1 + 1		
LP_QP( )		8.4.23
}		
}		
}		

8.7.7 TILE_HIGHPASS( )

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

Table 44 – TILE_HIGHPASS( ) syntax structure

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

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

8.7.8 TILE_HEADER_HIGHPASS( )

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

Table 45 – TILE_HEADER_HIGHPASS( ) syntax structure

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

• • •

8.7.10.8 LP_QP_INDEX[n]

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

8.7.10.9 HP_QP_INDEX[n]

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

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

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

8.7.10.10 DECODE_QP_INDEX( )

• • •

8.7.11 MB_DC( )

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

Table 48 – MB_DC( ) syntax structure

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

• • •

8.7.14.1 IS_DC_CH_FLAG

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

• • •

8.7.16.1 MB_LP( )

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

Table 53 – MB_LP( ) syntax structure

MB_LP( ) {	**Descriptor**	**Reference**
/* IsCurrPlaneAlphaFlag is equal to TRUE for parsing alpha image plane, and IsCurrPlaneAlphaFlag is equal to FALSE for parsing primary image plane */		
iBand = 1 /* 0 = DC 1 = LP, 2 = HP */		
iTranspose444[ ] = {0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15}		
iTranspose422[ ] = {0, 2, 1, 3, 4, 6, 5, 7}		
iTranspose420[ ] = {0, 2, 1, 3}		
bInitializeContext = (IsMBLeftEdgeOfTileFlag && IsMBTopEdgeOfTileFlag)		
if (bInitializeContext) {		
InitializeCountCBPLP( )		8.9.2
InitializeLPVLC( )		8.8.3.2
InitializeAdaptiveScanLP( )		8.11.2
InitializeModelMB(ModelLP, iBand)		8.12.1
}		
bResetTotals = ((MBx − LeftMBIndexOfTile[TileIndexx]) % 16) = = 0)		

Table 53 – MB_LP( ) syntax structure

MB_LP( ) {	**Descriptor**	**Reference**
if (bResetTotals)		
ResetTotalsAdaptiveScanLP( )		8.11.4
iLapMean[ ] = {0, 0}		
if (INTERNAL_CLR_FMT = = YUV422 | | INTERNAL_CLR_FMT = = YUV420)		
iFullPlanes = 2		
else		
iFullPlanes = NumComponents		
if (INTERNAL_CLR_FMT = = YUV420 | | INTERNAL_CLR_FMT = = YUV422 | | INTERNAL_CLR_FMT = = YUV444) {		
iMax = iFullPlanes * 4 − 5 /* Max value of CBPLP */		
if (CountZeroCBPLP <= 0 | | CountMaxCBPLP < 0) {		
CBPLP_YUV1	e(v)	8.7.16.3.1
if (CountMaxCBPLP < CountZeroCBPLP)		
iCBPLP = iMax − CBPLP_YUV1		
else		
iCBPLP = CBPLP_YUV1		
} else {		
CBPLP_YUV2	u(iFullPlanes)	8.7.16.3.2
iCBPLP = CBPLP_YUV2		
}		
UpdateCountCBPLP(iCBPLP, iMax)		8.9.3
} else {		
iCBPLP = 0		
for (n=0; n < NumComponents; n++) {		
CBPLP_CH_BIT	u(1)	8.7.16.3.3
iCBPLP |= (CBPLP_CH_BIT << n)		
}		
}		
for (n = 0; n < NumComponents; n++) {		
if (INTERNAL_CLR_FMT = = YUV420)		
jMax = 3		
else if (INTERNAL_CLR_FMT = = YUV422)		
jMax = 7		
else		
jMax = 15		
for (j = 0; j <= jMax; j++)		
LPInput[k][j] = 0		
}		
for (n = 0; n < iFullPlanes; n++) {		
if (n = = 0)		
iIndex = 0		
else		
iIndex = 1		
iNumNonZero = 0		
if ((iCBPLP >> n) & 1) {		
for (i = 0; i < 32; i++)		

Table 53 – MB_LP( ) syntax structure

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

Table 53 – MB_LP( ) syntax structure

MB_LP( ) {	**Descriptor**	**Reference**
LPInput[2][iTranspose420[k]] = REFINE_LP(LPInput[2][iTranspose420[k]], iModelBits)		8.7.16.2
}		
else if ((INTERNAL_CLR_FMT= =YUV422) && n)		
for (k = 1; k < 8; k++) {		
LPInput[1][iTranspose422[k]]= REFINE_LP(LPInput[1][iTranspose422[k]], iModelBits)		8.7.16.2
LPInput[2][iTranspose422[k]] = REFINE_LP(LPInput[2][iTranspose422[k]], iModelBits)		8.7.16.2
}		
else		
for (k = 1; k < 16; k++)		
LPInput[n][iTranspose444[k]] = REFINE_LP(LPInput[n][iTranspose444[k]], iModelBits)		8.7.16.2
} /* for (n=0 … */		
UpdateModelMB(iLapMean[ ], ModelLP, iBand)		8.12.2
bResetContext = (MBx = = (LeftMBIndexOfTile[TileIndexx + 1] − 1) | | (MBx − LeftMBIndexOfTile[TileIndexx]) % 16 = = 0)		
if (bResetContext)		
AdaptLP( )		8.8.4.2
}		

• • •

8.7.18.2 MB_HP( )

Table 69 – MB_HP( ) syntax structure

MB_HP( ) {	**Descriptor**	**Reference**
/* IsCurrPlaneAlphaFlag is equal to TRUE for parsing alpha image plane, and IsCurrPlaneAlphaFlag is equal to FALSE for parsing primary image plane */		
iBand = 2		
iHierScanOrder[ ] = {0, 1, 4, 5, 2, 3, 6, 7, 8, 9, 12, 13, 10, 11, 14, 15}		
bInitializeContext = (IsMBLeftEdgeOfTileFlag && IsMBTopEdgeOfTileFlag)		
if (bInitializeContext) {		
InitializeHPVLC( )		8.8.3.3
InitializeAdaptiveScanHP( )		8.11.3
InitializeModelMB(ModelHP, iBand)		8.12.1
}		
bResetTotals = (((MBx − LeftMBIndexOfTile[TileIndexx]) % 16) = = 0)		
if (bResetTotals)		
ResetTotalsAdaptiveScanHP( )		8.11.5
iLapMean[~~2~~] = {0, 0}		
for (i = 0; i < NumComponents; i++) {		
bChroma = (i > 0)		
iNBlocks = 4		
if (bChroma && INTERNAL_CLR_FMT = = YUV420)		
iNBlocks = 1		
else if (bChroma && INTERNAL_CLR_FMT = = YUV422)		

Table 69 – MB_HP( ) syntax structure

MB_HP( ) {	**Descriptor**	**Reference**
iNBlocks = 2		
iCBPHP = MBCBPHP[MBx][MBy][i]		
for (iBlock = 0; iBlock < iNBlocks * 4; iBlock++) {		
iBlockMap = iBlock		
if (iNBlocks = = 4)		
iBlockMap = iHierScanOrder[iBlock]		
for (k = 0; k < 16; k++)		
HPInputVLC[i][iBlock][k] = 0		
iNumNonZero = DECODE_BLOCK_ADAPTIVE(iCBPHP & 1, bChroma, i, iBlockMap)		8.7.18.4
iLapMean[bChroma] += iNumNonZero		
iCBPHP >>= 1		
}		
}		
ModelBitsMBHP[MBx][MBy][0] = ModelHP.MB~~b~~its[0]		
ModelBitsMBHP[MBx][MBy][1] = ModelHP.MB~~b~~its[1]		
UpdateModelMB(iLapMean[ ], ModelHP, iBand)		8.12.2
bResetContext = (MBx = = (LeftMBIndexOfTile[TileIndexx + 1] − 1) || (MBx − LeftMBIndexOfTile[TileIndexx]) % 16 = = 0)		
if (bResetContext)		
AdaptHP( )		8.8.4.3
}		

8.7.18.3 MB_HP_FLEX( )

Table 70 – MB_HP_FLEX( ) syntax structure

MB_HP_FLEX( ) {	**Descriptor**	**Reference**
/* IsCurrPlaneAlphaFlag is equal to TRUE for parsing alpha image plane, and IsCurrPlaneAlphaFlag is equal to FALSE for parsing primary image plane */		
iBand = 2		
iHierScanOrder[ ] = {0, 1, 4, 5, 2, 3, 6, 7, 8, 9, 12, 13, 10, 11, 14, 15}		
bInitializeContext = (IsMBLeftEdgeOfTileFlag && IsMBTopEdgeOfTileFlag)		
if (bInitializeContext) {		
InitializeHPVLC( )		8.8.3.3
InitializeAdaptiveScanHP( )		8.11.3
InitializeModelMB(ModelHP, iBand)		8.12.1
}		
bResetTotals = (((MBx − LeftMBIndexOfTile[TileIndexx]) % 16) = = 0)		
if (bResetTotals)		
ResetTotalsAdaptiveScanHP( )		8.11.5
iLapMean[~~2~~] = {0, 0}		
for (i = 0; i < NumComponents; i++) {		
iIndex = 0		
bChroma = i > 0		
if (i> 0)		
iIndex = 1		
iModelBits = ModelHP.MBits[iIndex]		

Table 70 – MB_HP_FLEX( ) syntax structure

MB_HP_FLEX( ) {	**Descriptor**	**Reference**
iNBlocks = 4		
if (bChroma && INTERNAL_CLR_FMT = = YUV420)		
iNBlocks = 1		
else if (bChroma && INTERNAL_CLR_FMT = = YUV422)		
iNBlocks = 2		
iCBPHP = MBCBPHP[MBx][MBy][i]		
for (iBlock = 0; iBlock < iNBlocks*4; iBlock++) {		
iBlockMap = iBlock		
if (iNBlocks = = 4)		
iBlockMap = iHierScanOrder[iBlock]		
for (k = 0; k < 16; k++)		
HPInputVLC[i][iBlock][k] = 0		
iNumNonZero = DECODE_BLOCK_ADAPTIVE(iCBPHP & 1, bChroma, i, iBlockMap)		8.7.18.4
BLOCK_FLEXBITS(i, iBlockMap, iModelBits, TRIM_FLEXBITS)		8.7.19.2
iLapMean[bChroma] += iNumNonZero		
iCBPHP >>= 1		
}		
}		
ModelBitsMBHP[MBx][MBy][0] = ModelHP.MB~~b~~its[0]		
ModelBitsMBHP[MBx][MBy][1] = ModelHP.MB~~b~~its[1]		
UpdateModelMB(iLapMean[ ], ModelHP, iBand)		8.12.2
bResetContext = (MBx = = (LeftMBIndexOfTile[TileIndexx + 1] − 1) | | (MBx − LeftMBIndexOfTile[TileIndexx]) % 16 = = 0)		
if (bResetContext)		
AdaptHP( )		8.8.4.3
}		

8.7.18.4 DECODE_BLOCK_ADAPTIVE( )

Table 71 – DECODE_BLOCK_ADAPTIVE( ) syntax structure

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

return iNumNonZero		
}		

8.7.18.5 DECODE_BLOCK( )

Table 72 – DECODE_BLOCK( ) syntax structure

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

8.7.18.6 DECODE_RUN( )

• • •

Table 75 – DECODE_FIRST_INDEX( ) syntax structure

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

• • •

8.7.18.9.5 INDEX_B

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

Table 81 – Code table for INDEX_B

Code	**Value**
0	0
10	1
110	2
111	3

INDEX_B jointly codes the following two events:

– The binary event of whether the magnitude of the next non-zero coefficient is equal to 1 or greater than 1 as follows:
  - If (INDEX_B & 1) is equal to 0, this magnitude is equal to 1.
  - Otherwise, this magnitude is greater than 1.

– The binary ~~of~~ event of whether this coefficient is the last coefficient in the block or if there are more non-zero coefficients, as follows:
  - If (INDEX_B >> 1) is equal to 0, this coefficient is the last coefficient in the block.
  - Otherwise, the run before the next non-zero coefficient is zero.

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

• • •

8.8.4.5 AdaptVLCTable2( )

AdaptVLCTable2( ) is used for choosing VLC code tables when there are more than two possible code tables. In this case, the index TableIndex can take values between 0 and the maximum table index for that set of VLC Code tables. This maximum table index is contained in the parameter iMaxTableIndex. For AdaptVLCTable2( ), there are two parameters (DiscrimVal1 and DiscrimVal2) which determines the selection of VLC code tables. DiscrimVal1 determines whether the code table index should be decreased, while DiscrimVal2 determines whether the code table index should be increased.

Table 102 – Pseudocode for function AdaptVLCTable2( )

AdaptVLCTable2(sAdaptVLC, iMaxTableIndex) {	**Reference**
/* **sAdaptVLC** is an instance of the AdaptiveVLC struct */	
/* iMaxTableIndex − max table index possible for this struct instance */	
bChange = FALSE	
iDiscrimLow = **sAdaptVLC**.DiscrimVal1	
iDiscrimHigh = **sAdaptVLC**.DiscrimVal2	
cLowerBound = −8	
cUpperBound = 8	
if (iDiscrimLow < cLowerBound && **sAdaptVLC**.TableIndex != 0) {	
sAdaptVLC.TableIndex− −	
bChange = TRUE	
} else if (iDiscrimHigh > cUpperBound && **sAdaptVLC**.TableIndex != iMaxTableIndex) {	
sAdaptVLC.TableIndex++	
bChange = TRUE	
}	
if (bChange) {	
sAdaptVLC.DiscrimVal1 = 0	
sAdaptVLC. DiscrimVal2 = 0	
if (**sAdaptVLC**.TableIndex = = iMaxTableIndex) {	
sAdaptVLC.DeltaTableIndex = **sAdaptVLC**.TableIndex − 1	
sAdaptVLC.Delta2TableIndex = **sAdaptVLC**.TableIndex − 1	
} else if (**sAdaptVLC**.TableIndex = = 0) {	
sAdaptVLC.DeltaTableIndex = **sAdaptVLC**.TableIndex	
sAdaptVLC.Delta2TableIndex = **sAdaptVLC**.TableIndex	
} else {	
sAdaptVLC.DeltaTableIndex = **sAdaptVLC**.TableIndex − 1	
sAdaptVLC.Delta2TableIndex = **sAdaptVLC**.TableIndex	
}	
} else { /* no change to table, but clip the discriminant */	
if (**sAdaptVLC**.DiscrimVal1 < −64)	
sAdaptVLC.DiscrimVal1 = −64	
if (**sAdaptVLC**.DiscrimVal1 > 64)	
sAdaptVLC.DiscrimVal1 = 64	
if (**sAdaptVLC**.DiscrimVal2 < −64)	
sAdaptVLC.DiscrimVal2 = −64	
if (**sAdaptVLC**.DiscrimVal2 > 64)	
sAdaptVLC.DiscrimVal2 = 64	
}	
return **sAdaptVLC**	
}	

• • •

Table 107 – Definitions of ScanOrder0 and ScanOrder1

• • •

9 Decoding process

9.1 General

This clause specifies the decoding process. The decoding process is interdependent with the initialization of variables and parsing of syntax elements as specified in clause 8.

The decoding process specified in this clause is distinguished from the codestream parsing process in the following manner: the codestream parsing process manages all control flow regarding the correct parsing of codestream syntax elements. This includes maintaining state variables for adaptive VLC selection, adaptive coefficient normalization, and other related information. The processes in this clause therefore are written with the assumption that, when they are invoked, the input variables required for this process have been correctly parsed from the codestream.

The decoding process is specified so that the decoded samples from any two JPEG XR decoders will be numerically identical. Any decoder which produces results that match the process specified here conforms to the requirements of this S~~codestream s~~pecification.

The image decoding process proceeds as specified in subclause 9.2.

9.2 Image decoding

The outputs of this process are the output samples of the image.

The image decoding process proceeds as in Table 117.

Table 117 – Pseudocode for function ImageDecoding( )

ImageDecoding( ) {	**Reference**
ImagePlaneDecoding( ) /* resulting sample values are stored in the variables ImagePlane[i][x][y] */	9.3
if ((OUTPUT_CLR_FMT = = RGB) && ((OUTPUT_BITDEPTH = = BD5) | | (OUTPUT_BITDEPTH = = BD565) | | (OUTPUT_BITDEPTH = = BD10))) /* Packed RGB */	
outputArrays = 1	
else if ((OUTPUT_CLR_FMT = = RGB) | | (OUTPUT_CLR_FMT = = YUV444) | | (OUTPUT_CLR_FMT = = YUV422) | | (OUTPUT_CLR_FMT = = YUV420))	
outputArrays = 3	
else if (OUTPUT_CLR_FMT = = RGBE)	
outputArrays = 4	
else	
outputArrays = NumComponents	
for (i = 0; i < outputArrays; i++) {	
if ((i > 0) && (OUTPUT_CLR_FMT = = YUV420))	
outputHeight = (HEIGHT_MINUS1 + 1) / 2	
else	
outputHeight = HEIGHT_MINUS1 + 1	
if ((OUTPUT_BITDEPTH = = BD1WHITE1) | | (OUTPUT_BITDEPTH = = BD1BLACK1)) /* Horizonally packed flags */	
outputWidth = WIDTH_MINUS1 / 8 + 1	
else if ((i > 0) && ((OUTPUT_CLR_FMT = = YUV422) | | (OUTPUT_CLR_FMT = = YUV420)))	

outputWidth = (WIDTH_MINUS1 + 1) / 2	
else	
outputWidth = WIDTH_MINUS1 + 1	
for (y = 0; y < outputHeight; y++)	
for (x = 0; x < outputWidth; x++)	
ImagePrimary[i][x][y] = ImagePlane[i][x][y]	
}	
~~for (i = 0; i < NumComponents; i++)~~	
~~for (x = 0; x < ExtendedWidth[i]; x++)~~	
~~for (y = 0; y < ExtendedHeight[i]; y++)~~	
~~ImagePrimary[i][x][y] = ImagePlane[i][x][y]~~	
if (ALPHA_IMAGE_PLANE_FLAG = = TRUE)	
ImagePlaneDecoding( ) /* resulting sample values, corresponding to the alpha image plane, are stored in the variables ImagePlane[0][x][y] */	9.3
for (y = 0; y <= HEIGHT_MINUS1; y++)	
for (x = 0; x <= WIDTH_MINUS1~~ExtendedWidth[i]~~; x++)	
~~for (y = 0; y < ExtendedHeight[i]; y++)~~	
ImageAlpha[0][x][y] = ImagePlane[0][x][y]	
}	

NOTE – Throughout the parsing of syntax elements in clause 8, it is assumed that~~,~~ if ALPHA_IMAGE_PLANE_FLAG is equal to TRUE, there are two sets of parsed syntax elements: one set corresponding to the primary image plane, and one set corresponding to the alpha image plane. In the same manner, this subclause assumes that there are two sets of global variables being used in the decoding process, corresponding to the primary and alpha image planes, respectively.

• • •

9.5.3 High-pass macroblock coefficient remapping

The HP coefficient remapping process proceeds as in Table 125.

Table 125 – Pseudocode for function HPMBCoefficientRemap( )

HPMBCoefficientRemap( ) {	**Reference**
for (i = 0; i < NumComponents; i++) {	
if (i != 0 && INTERNAL_CLR_FMT = = YUV420)	
jMax = 3	
else if (i != 0 && INTERNAL_CLR_FMT = = YUV422)	
jMax = 7	
else	
jMax = 15	
for (j = 0; j <= jMax~~15~~; j++)	
HPBlockCoefficientRemap(i, j)	9.5.4
}	
}	

• • •

9.7.2.3 Assignment of low-pass quantization parameters

The assignment process of LP quantization parameters proceeds as in Table 142.

Table 142 – Pseudocode for function AssignLPQuantizationParameters( )

AssignLPQuantizationParameters( ) {	**Reference**
for (j = 0; j < NumLPQPs~~ − 1~~; j++) {	
if (COMPONENT_MODE = = UNIFORM)	
for (i = 0; i <=NumComponents−1; i++)	
LPQuantParam[i][j] = LP_QUANT[j]	
else if (COMPONENT_MODE = = SEPARATE) {	
LPQuantParam[0][j] = LP_QUANT_LUMA[j]	
for (i = 1; i <=NumComponents−1; i++)	
LPQuantParam[i][j] = LP_QUANT_CHROMA[j]	
} else if (COMPONENT_MODE = = INDEPENDENT)	
for (i = 0; i <=NumComponents−1; i++)	
LPQuantParam[i][j] = LP_QUANT_CH[i][j]	
}	
}	

• • •

9.7.3.3 Assignment of high-pass quantization parameters

The assignment process of HP quantization parameters proceeds as in Table 145.

Table 145 – Pseudocode for function AssignHPQuantizationParameters( )

AssignHPQuantizationParameters( ) {	**Reference**
for (j = 0; j < NumHPQPs~~ − 1~~; j++) {	
if (COMPONENT_MODE = = UNIFORM)	
for (i = 0; i <=NumComponents−1; i++)	
HPQuantParam[i][j] = HP_QUANT[j]	
else if (COMPONENT_MODE = = SEPARATE) {	
HPQuantParam[0][j] = HP_QUANT_LUMA[j]	
for (i = 1; i <=NumComponents−1; i++)	
HPQuantParam[i][j] = HP_QUANT_CHROMA[j]	
} else if (COMPONENT_MODE = = INDEPENDENT)	
for (i = 0; i <=NumComponents−1; i++)	
HPQuantParam[i][j] = HP_QUANT_CH[i][j]	
}	
}	

• • •

9.8.3 Dequantization of high-pass coefficients

This process is applied for the HP coefficients of an entire macroblock, for all color components.

Inputs to this process are the values HPQuantParam[i][j] of HP quantization parameters, for each color component i and index j; the quantization parameter index MBQPIndexHP[MBx][MBy]; the array MBBuffer[MBx][MBy][i][j] of HP transform coefficients, where i and j are indices, with i representing the color component, and j ranging from 1 to 255.

Output of this process is an array of scaled HP transform coefficients MBBuffer[MBx][MBy][i][j], for each color component i and index j ranging from 1 to 255, referencing the respective HP transform coefficient.

The dequantization process for HP coefficients proceeds as in Table 148.

Table 148 – Pseudocode for function DequantizeHPCoefficients( )

DequantizeHPCoefficients( ) {	**Reference**
for (i = 0; i < NumComponents; i++) {	
k = MBQPIndexHP[MBx][MBy]	
valueQP[i] = HPQuantParam[i][k]	
iQuantScalingFactor[i] = QuantMap(valueQP[i], 1)	9.8.4
if (i = = 0) /* Luma Component */	
for (blkIndex = 0; blkIndex <= 15; blkIndex++)	
for (j = 1; j <= 15; j++)	
~~MbDCLP~~MBBuffer[MBx][MBy][i][16*blkIndex + j] = ~~MbDCLP~~MBBuffer[MBx][MBy][i][16*blkIndex + j] * iQuantScalingFactor[i]	
else if ((INTERNAL_CLR_FMT != YUV422) && (INTERNAL_CLR_FMT != YUV420))	
for (blkIndex = 0; blkIndex <= 15; blkIndex++)	
for (j = 1; j <= 15; j++)	
~~MbDCLP~~MBBuffer[MBx][MBy][i][16*blkIndex + j] = ~~MbDCLP~~MBBuffer[MBx][MBy][i][16*blkIndex + j] * iQuantScalingFactor[i]	
else if (INTERNAL_CLR_FMT = = YUV422)	
for (blkIndex = 0; blkIndex <= 7; blkIndex++)	
for (j = 1; j <= 15; j++)	
~~MbDCLP~~MBBuffer[MBx][MBy][i][16*blkIndex + j] = ~~MbDCLP~~MBBuffer[MBx][MBy][i][16*blkIndex + j] * iQuantScalingFactor[i]	
else /* if (INTERNAL_CLR_FMT = = YUV420) */	
for (blkIndex = 0; blkIndex <= 3; blkIndex++)	
for (j = 1; j <= 15; j++)	
~~MbDCLP~~MBBuffer[MBx][MBy][i][16*blkIndex + j] = ~~MbDCLP~~MBBuffer[MBx][MBy][i][16*blkIndex + j] * iQuantScalingFactor[i]	
}	
}	

9.8.4 QuantMap( )

• • •

NOTE – The input parameter iScaledShift takes the value of either 0 or 1, dependent on the component and band. When SCALED_FLAG is equal to TRUE, the ~~iShift parameter can modify the resulting~~ quantization scaling factor value can be modified by a power of 2. See the note at the end of subclause 9.9.2.

• • •

9.9.2 First level inverse transform

• • •

NOTE – The purpose of the multiplication by 2 for Chroma components, in circumstances where scaling is involved, is to re-normalize the chroma with respect to the Y component. Due to possible conversion from RGB to YUV during encoding, the U and V components may have a numerical range that has increased by one bit. If SCALED_FLAG is equal to TRUE, the dynamic range of the (DC and LP) U and ~~,~~V component values could potentially grow beyond 16 bits, due to the numerical range expansion associated with the two levels of transform on the encode side (for the DC and LP coefficients). Therefore, the quantization parameter for these chroma components is set to half the value used for luma components. The coefficients are scaled by this factor of two at the end of the first-level transform process.

• • •

9.9.3.2 FirstLevelOverlapFilteringPrimary( )

Pseudocode for the function FirstLevelOverlapFilteringPrimary( ) is specified in Table 153.

Table 153 – Pseudocode for function FirstLevelOverlapFilteringPrimary( )

FirstLevelOverlapFilteringPrimary(i) {	**Reference**
for (Ty = 0; Ty <= (NumTileRows − 1); Ty++) {	
for (Tx = 0; Tx <= (NumTileCols − 1); Tx++) {	
for (y = TopMBIndexOfTile[Ty]; y <= (TopMBIndexOfTile[Ty + 1] − 2); y++)	
for (x = LeftMBIndexOfTile[Tx]; x <= (LeftMBIndexOfTile[Tx + 1] − 2); x++)	
FirstLevelCallOverlapPostFilter4x4(i, x, y)	9.9.3.5
if ((Tx = = 0) | | (HARD_TILING_FLAG = = TRUE)) { /* Left edge */	
x = LeftMBIndexOfTile[Tx]	
for (y = TopMBIndexOfTile[Ty]; y <= TopMBIndexOfTile[Ty + 1] − 2; y++) {	
arrayLocal[ ] = {MbDCLP[x][y][i][8], MbDCLP[x][y][i][12], MbDCLP[x][y+1][i][0], MbDCLP[x][y+1][i][4]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
MbDCLP[x][y][i][8] = arrayLocal[0]	
MbDCLP[x][y][i][12] = arrayLocal[1]	
MbDCLP[x][y+1][i][0] = arrayLocal[2]	
MbDCLP[x][y+1][i][4] = arrayLocal[3]	
arrayLocal[ ] = {MbDCLP[x][y][i][9], MbDCLP[x][y][i][13], MbDCLP[x][y+1][i][1], MbDCLP[x][y+1][i][5]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
MbDCLP[x][y][i][9] = arrayLocal[0]	
MbDCLP[x][y][i][13] = arrayLocal[1]	
MbDCLP[x][y+1][i][1] = arrayLocal[2]	
MbDCLP[x][y+1][i][5] = arrayLocal[3]	
}	
}	
if ((Ty = = 0) | | (HARD_TILING_FLAG = = TRUE)) { /* Top edge */	
y = TopMBIndexOfTile[Ty]	
for (x = LeftMBIndexOfTile[Tx]; x <= LeftMBIndexOfTile[Tx + 1] − 2; x++) {	
arrayLocal[ ] = {MbDCLP[x][y][i][2], MbDCLP[x][y][i][3], MbDCLP[x+1][y][i][0], MbDCLP[x+1][y][i][1]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
MbDCLP[x][y][i][2] = arrayLocal[0]	
MbDCLP[x][y][i][3] = arrayLocal[1]	
MbDCLP[x+1][y][i][0] = arrayLocal[2]	
MbDCLP[x+1][y][i][1] = arrayLocal[3]	
arrayLocal[ ] = {MbDCLP[x][y][i][6], MbDCLP[x][y][i][7], MbDCLP[x+1][y][i][4], MbDCLP[x+1][y][i][5]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
MbDCLP[x][y][i][6] = arrayLocal[0]	
MbDCLP[x][y][i][7] = arrayLocal[1]	
MbDCLP[x+1][y][i][4] = arrayLocal[2]	
MbDCLP[x+1][y][i][5] = arrayLocal[3]	
}	
}	
if ((Tx = = NumTileCols − 1) | | (HARD_TILING_FLAG = = TRUE)) { /* Right edge */	
x = LeftMBIndexOfTile[Tx + 1] − 1	
for (y = TopMBIndexOfTile[Ty]; y <= TopMBIndexOfTile[Ty + 1] − 2; y++) {	

Table 153 – Pseudocode for function FirstLevelOverlapFilteringPrimary( )

FirstLevelOverlapFilteringPrimary(i) {	**Reference**
arrayLocal[ ] = {MbDCLP[x][y][i][10], MbDCLP[x][y][i][14], MbDCLP[x][y+1][i][2], MbDCLP[x][y+1][i][6]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
MbDCLP[x][y][i][10] = arrayLocal[0]	
MbDCLP[x][y][i][14] = arrayLocal[1]	
MbDCLP[x][y+1][i][2] = arrayLocal[2]	
MbDCLP[x][y+1][i][6] = arrayLocal[3]	
arrayLocal[ ] = {MbDCLP[x][y][i][11], MbDCLP[x][y][i][15], MbDCLP[x][y+1][i][3~~4~~], MbDCLP[x][y+1][i][7]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
MbDCLP[x][y][i][11] = arrayLocal[0]	
MbDCLP[x][y][i][15] = arrayLocal[1]	
MbDCLP[x][y+1][i][3~~4~~] = arrayLocal[2]	
MbDCLP[x][y+1][i][7] = arrayLocal[3]	
}	
}	
● ● ●	

9.9.3.3 FirstLevelOverlapFiltering422( )

Pseudocode for the function FirstLevelOverlapFiltering422( ) is specified in Table 154.

Table 154 – Pseudocode for function FirstLevelOverlapFiltering422( )

FirstLevelOverlapFiltering422(i) {	**Reference**
for (Ty = 0; Ty <= (NumTileRows − 1); Ty ++) {	
if ((Ty = = 0) | | (HARD_TILING_FLAG = = TRUE)) { /* Top edge */	
/* OverlapPostFilter1 */	
y = TopMBIndexOfTile[Ty]	
MbDCLP[LeftMBIndexOfTile[0]][y][i][0] −= MbDCLP[LeftMBIndexOfTile[0]][y][i][1] /* Upper left corner difference */	
MbDCLP[LeftMBIndexOfTile[NumTileCols] − 1][y][i][1] −= MbDCLP[LeftMBIndexOfTile[NumTileCols] − 1][y][i][0] /* Upper right corner difference */	
if (HARD_TILING_FLAG = = TRUE)	
for (Tx = 1; Tx < (NumTileCols − 1); Tx++) {	
MbDCLP[LeftMBIndexOfTile[Tx]][y][i][0] −= MbDCLP[LeftMBIndexOfTile[Tx]][y][i][1]	
MbDCLP[LeftMBIndexOfTile[Tx] − 1][y][i][1] −= MbDCLP[LeftMBIndexOfTile[Tx] − 1][y][i][0]	
}	
}	
if ((Ty = = NumTileRows − 1) | | (HARD_TILING_FLAG = = TRUE)) { /* Bottom edge */	
/* OverlapPostFilter1 */	
y = TopMBIndexOfTile[Ty + 1] − 1	
MbDCLP[LeftMBIndexOfTile[0]][y][i][6] −= MbDCLP[LeftMBIndexOfTile[0]][y][i][7] /* Bottom left corner difference */	
MbDCLP[LeftMBIndexOfTile[NumTileCols] − 1][y][i][7] −= MbDCLP[LeftMBIndexOfTile[NumTileCols] − 1][y][i][6] /* Bottom right corner difference */	
if (HARD_TILING_FLAG = = TRUE)	
for (Tx = 1; Tx < (NumTileCols − 1); Tx++) {	
MbDCLP[LeftMBIndexOfTile[Tx]][y][i][6] −= MbDCLP[LeftMBIndexOfTile[Tx]][y][i][7]	

Table 154 – Pseudocode for function FirstLevelOverlapFiltering422( )

FirstLevelOverlapFiltering422(i) {	**Reference**
MbDCLP[LeftMBIndexOfTile[Tx] − 1][y][i][7] −= MbDCLP[LeftMBIndexOfTile[Tx] − 1][y][i][6]	
}	
}	
for (Tx = 0; Tx <= (NumTileCols − 1); Tx++) {	
for (y = TopMBIndexOfTile[Ty]; y <= (TopMBIndexOfTile[Ty + 1] − 1~~2~~); y++)	
for (x = LeftMBIndexOfTile[Tx]; x <= (LeftMBIndexOfTile[Tx + 1] − 2); x++) {	
arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x+1][y][i][2], MbDCLP[x][y][i][5], MbDCLP[x+1][y][i][4]}	
OverlapPostFilter2x2(arrayLocal[ ])	9.9.8.3
MbDCLP[x][y][i][3] = arrayLocal[0]	
MbDCLP[x+1][y][i][2] = arrayLocal[1]	
MbDCLP[x][y][i][5] = arrayLocal[2]	
MbDCLP[x+1][y][i][4] = arrayLocal[3]	
if (y != (TopMBIndexOfTile[Ty + 1] − 1)) {	
arrayLocal[ ] = {MbDCLP[x][y][i][7], MbDCLP[x+1][y][i][6], MbDCLP[x][y+1][i][1], MbDCLP[x+1][y+1][i][0]}	
OverlapPostFilter2x2(arrayLocal[ ])	9.9.8.3
MbDCLP[x][y][i][7] = arrayLocal[0]	
MbDCLP[x+1][y][i][6] = arrayLocal[1]	
MbDCLP[x][y+1][i][1] = arrayLocal[2]	
MbDCLP[x+1][y+1][i][0] = arrayLocal[3]	
}	
}	
if ((Tx = = 0) | | (HARD_TILING_FLAG = = TRUE)) { /* Left edge */	
x = LeftMBIndexOfTile[Tx]	
for (y = TopMBIndexOfTile[Ty]; y <= (TopMBIndexOfTile[Ty + 1] − 1~~2~~); y++) {	
arrayLocal[ ] = {MbDCLP[x][y][i][2], MbDCLP[x][y][i][4]}	
OverlapPostFilter2(arrayLocal[ ])	9.9.8.4
MbDCLP[x][y][i][2] = arrayLocal[0]	
MbDCLP[x][y][i][4] = arrayLocal[1]	
if (y != (TopMBIndexOfTile[Ty + 1] − 1)) {	
arrayLocal[ ] = {MbDCLP[x][y][i][6], MbDCLP[x][y+1][i][0]}	
OverlapPostFilter2(arrayLocal[ ])	9.9.8.4
MbDCLP[x][y][i][6] = arrayLocal[0]	
MbDCLP[x][y+1][i][0] = arrayLocal[1]	
}	
}	
}	
if ((Tx = = NumTileCols − 1) | | (HARD_TILING_FLAG = = TRUE)) { /* Right edge */	
x = LeftMBIndexOfTile[Tx + 1] − 1	
for (y = TopMBIndexOfTile[Ty]; y <= (TopMBIndexOfTile[Ty + 1] − 1~~2~~); y++) {	
arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x][y][i][5]}	
OverlapPostFilter2(arrayLocal[ ])	9.9.8.4
MbDCLP[x][y][i][3] = arrayLocal[0]	
MbDCLP[x][y][i][5] = arrayLocal[1]	
if (y != (TopMBIndexOfTile[Ty + 1] − 1)) {	
arrayLocal[ ] = {MbDCLP[x][y][i][7], MbDCLP[x][y+1][i][1]}	

Table 154 – Pseudocode for function FirstLevelOverlapFiltering422( )

FirstLevelOverlapFiltering422(i) {	**Reference**
OverlapPostFilter2(arrayLocal[ ])	9.9.8.4
MbDCLP[x][y][i][7] = arrayLocal[0]	
MbDCLP[x][y+1][i][1] = arrayLocal[1]	
}	
}	
}	
• • •	

• • •

9.9.4 Second level coefficient combination

• • •

Table 157 – Pseudocode for function SecondLevelCoefficientCombination( )

SecondLevelCoefficientCombination( ) {	**Reference**
for (i = 0; i < NumComponents; i++) {	
for (MBy = 0; MBy < MBHeight; MBy++)	
for (MBx = 0; MBx < MBWidth; MBx++)	
if ((i = = 0) | | ((INTERNAL_CLR_FMT != YUV420) && (INTERNAL_CLR_FMT != YUV422))) {	
for (j = 0; j <= 15; j++) {	
x = 16 * MBx + 4 * (j % 4)	
y = 16 * MBy + 4 * (j / 4)	
ImagePlane[i][x][y] = MbDCLP[MBx][MBy][i][j]	
}	
for (j = 0; j <= 255; j++) {	
x = 16 * MBx + 4 * ((j / 16) % 4) + (j % 4)	
y = 16 * MBy + 4 * (j / 64) + ((j / 4) % 4)	
k = j % 16	
if (k != 0) /* only the HP coefficients are copied */	
ImagePlane[i][x][y] = MBBuffer~~MbDCLP~~[MBx][MBy][i][j]	
}	
} else if (INTERNAL_CLR_FMT = = YUV422) {	
for (j = 0; j <= 7; j++) {	
x = 8~~16~~ * MBx + 4 * (j % 2)	
y = 16 * MBy + 4 * (j / 2)	
ImagePlane[i][x][y] = MbDCLP[MBx][MBy][i][j]	
}	
for (j = 0; j <= 127; j++) {	
x = 8~~16~~ * MBx + 4 * ((j % 32) / 16) + ((j % 32) % 4)	
y = 16 * MBy + 4 * (j / 32) + ((j / 4) % 4)	
k = j % 16	
if (k != 0) /* only the HP coefficients are copied */	
ImagePlane[i][x][y] = MBBuffer~~MbDCLP~~[MBx][MBy][i][j]	

Table 157 – Pseudocode for function SecondLevelCoefficientCombination( )

SecondLevelCoefficientCombination( ) {	**Reference**
}	
} else if (INTERNAL_CLR_FMT = = YUV420) {	
for (j = 0; j <= 3; j++) {	
x = 8~~16~~ * MBx + 4 * (j % 2)	
y = 8~~16~~ * MBy + 4 * (j / 2)	
ImagePlane[i][x][y] = MbDCLP[MBx][MBy][i][j]	
}	
for (j = 0; j <= 63; j++) {	
x = 8~~16~~ * MBx + 4 * ((j % 32) / 16) + ((j % 32) % 4)	
y = 8~~16~~ * MBy + 4 * (j / 32) + ((j / 4) % 4)	
k = j % 16	
if (k != 0) /* only the HP coefficients are copied */	
ImagePlane[i][x][y] = MBBuffer~~MbDCLP~~[MBx][MBy][i][j]	
}	
}	
}	

• • •

9.9.6 Second level overlap filtering

• • •

Table 159 – Pseudocode for function SecondLevelOverlapFiltering( )

SecondLevelOverlapFiltering( ) {	**Reference**
for (i = 0; i < NumComponents; i++) {	
if ((i != 0) && ((INTERNAL_CLR_FMT = = YUV422) | | (INTERNAL_CLR_FMT = = YUV420)))	
dx = 2	
else	
dx = 1	
if ((i != 0) && (INTERNAL_CLR_FMT = = YUV420))	
dy = 2	
else	
dy = 1	
for (Tx = 0; Tx <= (NumTileCols − 1); Tx++) {	
for (Ty = 0; Ty <= (NumTileRows − 1); Ty++) {	
for (x = 16 * LeftMBIndexOfTile[Tx] / dx + 2; x < (16 * LeftMBIndexOfTile[Tx + 1] / dx − 2); x += 4)	
for (y = 16 * TopMBIndexOfTile[Ty] / dy + 2; y < (16 * TopMBIndexOfTile[Ty + 1] / dy − 2); -y += 4) {	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y], ImagePlane[i][x+2][y], ImagePlane[i][x+3][y], ImagePlane[i][x][y+1], ImagePlane[i][x+1][y+1], ImagePlane[i][x+2][y+1], ImagePlane[i][x+3][y+1], ImagePlane[i][x][y+2], ImagePlane[i][x+1][y+2], ImagePlane[i][x+2][y+2], ImagePlane[i][x+3][y+2], ImagePlane[i][x][y+3], ImagePlane[i][x+1][y+3], ImagePlane[i][x+2][y+3], ImagePlane[i][x+3][y+3]}	
OverlapPostFilter4x4(arrayLocal[ ])	9.9.8.1

Table 159 – Pseudocode for function SecondLevelOverlapFiltering( )

SecondLevelOverlapFiltering( ) {	**Reference**
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x+1][y] = arrayLocal[1]	
ImagePlane[i][x+2][y] = arrayLocal[2]	
ImagePlane[i][x+3][y] = arrayLocal[3]	
ImagePlane[i][x][y+1] = arrayLocal[4]	
ImagePlane[i][x+1][y+1] = arrayLocal[5]	
ImagePlane[i][x+2][y+1] = arrayLocal[6]	
ImagePlane[i][x+3][y+1] = arrayLocal[7]	
ImagePlane[i][x][y+2] = arrayLocal[8]	
ImagePlane[i][x+1][y+2] = arrayLocal[9]	
ImagePlane[i][x+2][y+2] = arrayLocal[10]	
ImagePlane[i][x+3][y+2] = arrayLocal[11]	
ImagePlane[i][x][y+3] = arrayLocal[12]	
ImagePlane[i][x+1][y+3] = arrayLocal[13]	
ImagePlane[i][x+2][y+3] = arrayLocal[14]	
ImagePlane[i][x+3][y+3] = arrayLocal[15]	
}	
if ((Tx = = 0) | | (HARD_TILING_FLAG = = TRUE)) { /* Left edge */	
for (y = 16 * TopMBIndexOfTile[Ty] / dy + 2; y < (16 * TopMBIndexOfTile[Ty + 1] / dy − 2); y += 4) {	
x = 16 * LeftMBIndexOfTile[Tx] / dx	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x][y+1], ImagePlane[i][x][y+2], ImagePlane[i][x][y+3]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x][y+1] = arrayLocal[1]	
ImagePlane[i][x][y+2] = arrayLocal[2]	
ImagePlane[i][x][y+3] = arrayLocal[3]	
x = 16 * LeftMBIndexOfTile[Tx] / dx + 1	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x][y+1], ImagePlane[i][x][y+2], ImagePlane[i][x][y+3]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x][y+1] = arrayLocal[1]	
ImagePlane[i][x][y+2] = arrayLocal[2]	
ImagePlane[i][x][y+3] = arrayLocal[3]	
}	
}	
if ((Ty = = 0) | | (HARD_TILING_FLAG = = TRUE)) { /* Top edge */	
for (x = 16 * LeftMBIndexOfTile[Tx] / dx + 2; x < (16 * LeftMBIndexOfTile[Tx + 1] / dx − 2); x += 4) {	
y = 16 * TopMBIndexOfTile[Ty] / dy	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y], ImagePlane[i][x+2][y], ImagePlane[i][x+3][y]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x+1][y] = arrayLocal[1]	
ImagePlane[i][x+2][y] = arrayLocal[2]	

Table 159 – Pseudocode for function SecondLevelOverlapFiltering( )

SecondLevelOverlapFiltering( ) {	**Reference**
ImagePlane[i][x+3][y] = arrayLocal[3]	
y = 16 * TopMBIndexOfTile[Ty] / dy + 1	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y], ImagePlane[i][x+2][y], ImagePlane[i][x+3][y]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x+1][y] = arrayLocal[1]	
ImagePlane[i][x+2][y] = arrayLocal[2]	
ImagePlane[i][x+3][y] = arrayLocal[3]	
}	
}	
if ((Tx = = NumTileCols − 1) | | (HARD_TILING_FLAG = = TRUE)) { /* Right edge */	
for (y = 16 * TopMBIndexOfTile[Ty] / dy + 2; y < (16 * TopMBIndexOfTile[Ty + 1] / dy − 2); y += 4) {	
x = 16 * LeftMBIndexOfTile[Tx + 1] / dx − 2	
arrayLocal[ ] = { ImagePlane[i][x][y], ImagePlane[i][x][y+1], ImagePlane[i][x][y+2], ImagePlane[i][x][y+3]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x][y+1] = arrayLocal[1]	
ImagePlane[i][x][y+2] = arrayLocal[2]	
ImagePlane[i][x][y+3] = arrayLocal[3]	
x = 16 * LeftMBIndexOfTile[Tx + 1] / dx − 1	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x][y+1], ImagePlane[i][x][y+2], ImagePlane[i][x][y+3]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x][y+1] = arrayLocal[1]	
ImagePlane[i][x][y+2] = arrayLocal[2]	
ImagePlane[i][x][y+3] = arrayLocal[3]	
}	
}	
if ((Ty = = NumTileRows − 1) | | (HARD_TILING_FLAG = = TRUE)) { /* Bottom edge */	
for (x = 16 * LeftMBIndexOfTile[Tx] / dx + 2; x < (16 * LeftMBIndexOfTile[Tx + 1] / dx − 2); x += 4) {	
y = 16 * TopMBIndexOfTile[Ty + 1] / dy − 2	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y], ImagePlane[i][x+2][y], ImagePlane[i][x+3][y]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x+1][y] = arrayLocal[1]	
ImagePlane[i][x+2][y] = arrayLocal[2]	
ImagePlane[i][x+3][y] = arrayLocal[3]	
y = 16 * TopMBIndexOfTile[Ty + 1] / dy − 1	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y], ImagePlane[i][x+2][y], ImagePlane[i][x+3][y]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x+1][y] = arrayLocal[1]	

Table 159 – Pseudocode for function SecondLevelOverlapFiltering( )

SecondLevelOverlapFiltering( ) {	**Reference**
ImagePlane[i][x+2][y] = arrayLocal[2]	
ImagePlane[i][x+3][y] = arrayLocal[3]	
}	
}	
if (((Tx = = 0) && (Ty = = 0)) | | (HARD_TILING_FLAG = = TRUE)) { /* Top left edge */	
x = 16 * LeftMBIndexOfTile[Tx] / dx	
y = 16 * Top~~Left~~MBIndexOfTile[Ty] / dy	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y], ImagePlane[i][x][y+1], ImagePlane[i][x+1][y+1]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x+1][y] = arrayLocal[1]	
ImagePlane[i][x][y+1] = arrayLocal[2]	
ImagePlane[i][x+1][y+1] = arrayLocal[3]	
}	
if (((Tx = = NumTileCols − 1) && (Ty = = 0)) | | (HARD_TILING_FLAG = = TRUE)) { /* Top right edge */	
x = 16 * LeftMBIndexOfTile[Tx + 1] / dx − 2	
y = 16 * Top~~Left~~MBIndexOfTile[Ty] / dy	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y], ImagePlane[i][x][y+1], ImagePlane[i][x+1][y+1]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x+1][y] = arrayLocal[1]	
ImagePlane[i][x][y+1] = arrayLocal[2]	
ImagePlane[i][x+1][y+1] = arrayLocal[3]	
}	
if (((Tx = = 0) && (Ty = = NumTileRows − 1)) | | (HARD_TILING_FLAG = = TRUE)) { /* Bottom left edge */	
x = 16 * LeftMBIndexOfTile[Tx] / dx	
y = 16 * Top~~Left~~MBIndexOfTile[Ty + 1] / dy − 2	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y], ImagePlane[i][x][y+1], ImagePlane[i][x+1][y+1]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x+1][y] = arrayLocal[1]	
ImagePlane[i][x][y+1] = arrayLocal[2]	
ImagePlane[i][x+1][y+1] = arrayLocal[3]	
}	
if (((Tx = = NumTileCols − 1) && (Ty = = NumTileRows − 1)) | | (HARD_TILING_FLAG = = TRUE)) { /* Bottom right edge */	
x = 16 * LeftMBIndexOfTile[Tx + 1] / dx − 2	
y = 16 * Top~~Left~~MBIndexOfTile[Ty + 1] / dy − 2	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y], ImagePlane[i][x][y+1], ImagePlane[i][x+1][y+1]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x+1][y] = arrayLocal[1]	
ImagePlane[i][x][y+1] = arrayLocal[2]	

Table 159 – Pseudocode for function SecondLevelOverlapFiltering( )

SecondLevelOverlapFiltering( ) {	**Reference**
ImagePlane[i][x+1][y+1] = arrayLocal[3]	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Tx != NumTileCols − 1)) { /* Right across for soft tiles */	
x = 16 * LeftMBIndexOfTile[Tx + 1] / dx − 2	
for (y = 16 * TopMBIndexOfTile[Ty] / dy + 2; y < (16 * TopMBIndexOfTile[Ty + 1] / dy − 2); -y += 4) {	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y], ImagePlane[i][x+2][y],ImagePlane[i][x+3][y], ImagePlane[i][x][y+1], ImagePlane[i][x+1][y+1], ImagePlane[i][x+2][y+1], ImagePlane[i][x+3][y+1], ImagePlane[i][x][y+2], ImagePlane[i][x+1][y+2], ImagePlane[i][x+2][y+2], ImagePlane[i][x+3][y+2], ImagePlane[i][x][y+3], ImagePlane[i][x+1][y+3], ImagePlane[i][x+2][y+3], ImagePlane[i][x+3][y+3]}	
OverlapPostFilter4x4(arrayLocal[ ])	9.9.8.1
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x+1][y] = arrayLocal[1]	
ImagePlane[i][x+2][y] = arrayLocal[2]	
ImagePlane[i][x+3][y] = arrayLocal[3]	
ImagePlane[i][x][y+1] = arrayLocal[4]	
ImagePlane[i][x+1][y+1] = arrayLocal[5]	
ImagePlane[i][x+2][y+1] = arrayLocal[6]	
ImagePlane[i][x+3][y+1] = arrayLocal[7]	
ImagePlane[i][x][y+2] = arrayLocal[8]	
ImagePlane[i][x+1][y+2] = arrayLocal[9]	
ImagePlane[i][x+2][y+2] = arrayLocal[10]	
ImagePlane[i][x+3][y+2] = arrayLocal[11]	
ImagePlane[i][x][y+3] = arrayLocal[12]	
ImagePlane[i][x+1][y+3] = arrayLocal[13]	
ImagePlane[i][x+2][y+3] = arrayLocal[14]	
ImagePlane[i][x+3][y+3] = arrayLocal[15]	
}	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Ty != NumTileRows − 1)) { /* Bottom across for soft tiles */	
y = 16 * TopMBIndexOfTile[Ty + 1] / dy − 2	
for (x = 16 * LeftMBIndexOfTile[Tx] / dx + 2; x < (16 * LeftMBIndexOfTile[Tx + 1] / dx − 2); -x += 4) {	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y], ImagePlane[i][x+2][y], ImagePlane[i][x+3][y], ImagePlane[i][x][y+1], ImagePlane[i][x+1][y+1], ImagePlane[i][x+2][y+1], ImagePlane[i][x+3][y+1], ImagePlane[i][x][y+2], ImagePlane[i][x+1][y+2], ImagePlane[i][x+2][y+2], ImagePlane[i][x+3][y+2], ImagePlane[i][x][y+3], ImagePlane[i][x+1][y+3], ImagePlane[i][x+2][y+3], ImagePlane[i][x+3][y+3]}	
OverlapPostFilter4x4(arrayLocal[ ])	9.9.8.1
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x+1][y] = arrayLocal[1]	
ImagePlane[i][x+2][y] = arrayLocal[2]	
ImagePlane[i][x+3][y] = arrayLocal[3]	
ImagePlane[i][x][y+1] = arrayLocal[4]	

Table 159 – Pseudocode for function SecondLevelOverlapFiltering( )

SecondLevelOverlapFiltering( ) {	**Reference**
ImagePlane[i][x+1][y+1] = arrayLocal[5]	
ImagePlane[i][x+2][y+1] = arrayLocal[6]	
ImagePlane[i][x+3][y+1] = arrayLocal[7]	
ImagePlane[i][x][y+2] = arrayLocal[8]	
ImagePlane[i][x+1][y+2] = arrayLocal[9]	
ImagePlane[i][x+2][y+2] = arrayLocal[10]	
ImagePlane[i][x+3][y+2] = arrayLocal[11]	
ImagePlane[i][x][y+3] = arrayLocal[12]	
ImagePlane[i][x+1][y+3] = arrayLocal[13]	
ImagePlane[i][x+2][y+3] = arrayLocal[14]	
ImagePlane[i][x+3][y+3] = arrayLocal[15]	
}	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Tx != NumTileCols − 1) && (Ty != NumTileRows − 1)) {	
x = 16 * LeftMBIndexOfTile[Tx + 1] / dx – 2	
y = 16 * TopMBIndexOfTile[Ty + 1] / dy – 2	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y], ImagePlane[i][x+2][y], ImagePlane[i][x+3][y], ImagePlane[i][x][y+1], ImagePlane[i][x+1][y+1], ImagePlane[i][x+2][y+1], ImagePlane[i][x+3][y+1], ImagePlane[i][x][y+2], ImagePlane[i][x+1][y+2], ImagePlane[i][x+2][y+2], ImagePlane[i][x+3][y+2], ImagePlane[i][x][y+3], ImagePlane[i][x+1][y+3], ImagePlane[i][x+2][y+3], ImagePlane[i][x+3][y+3]}	
OverlapPostFilter4x4(arrayLocal[ ])	9.9.8.1
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x+1][y] = arrayLocal[1]	
ImagePlane[i][x+2][y] = arrayLocal[2]	
ImagePlane[i][x+3][y] = arrayLocal[3]	
ImagePlane[i][x][y+1] = arrayLocal[4]	
ImagePlane[i][x+1][y+1] = arrayLocal[5]	
ImagePlane[i][x+2][y+1] = arrayLocal[6]	
ImagePlane[i][x+3][y+1] = arrayLocal[7]	
ImagePlane[i][x][y+2] = arrayLocal[8]	
ImagePlane[i][x+1][y+2] = arrayLocal[9]	
ImagePlane[i][x+2][y+2] = arrayLocal[10]	
ImagePlane[i][x+3][y+2] = arrayLocal[11]	
ImagePlane[i][x][y+3] = arrayLocal[12]	
ImagePlane[i][x+1][y+3] = arrayLocal[13]	
ImagePlane[i][x+2][y+3] = arrayLocal[14]	
ImagePlane[i][x+3][y+3] = arrayLocal[15]	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Tx = = 0) && (Ty != NumTileRows − 1)) { /* Left edge for soft tiles */	
y = 16 * TopMBIndexOfTile[Ty + 1] / dy − 2	
x = 16 * LeftMBIndexOfTile[Tx] / dx	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x][y+1], ImagePlane[i][x][y+2], ImagePlane[i][x][y+3]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2

Table 159 – Pseudocode for function SecondLevelOverlapFiltering( )

SecondLevelOverlapFiltering( ) {	**Reference**
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x][y+1] = arrayLocal[1]	
ImagePlane[i][x][y+2] = arrayLocal[2]	
ImagePlane[i][x][y+3] = arrayLocal[3]	
x = 16 * LeftMBIndexOfTile[Tx] / dx + 1	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x][y+1], ImagePlane[i][x][y+2], ImagePlane[i][x][y+3]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x][y+1] = arrayLocal[1]	
ImagePlane[i][x][y+2] = arrayLocal[2]	
ImagePlane[i][x][y+3] = arrayLocal[3]	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Tx != NumTileCols − 1) && (Ty = = 0)) { /* Top edge for soft tiles */	
x = 16 * LeftMBIndexOfTile[Tx + 1] / dx − 2	
y = 16 * TopMBIndexOfTile[Ty] / dy	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y], ImagePlane[i][x+2][y], ImagePlane[i][x+3][y]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x+1][y] = arrayLocal[1]	
ImagePlane[i][x+2][y] = arrayLocal[2]	
ImagePlane[i][x+3][y] = arrayLocal[3]	
y = 16 * TopMBIndexOfTile[Ty] / dy + 1	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y], ImagePlane[i][x+2][y], ImagePlane[i][x+3][y]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x+1][y] = arrayLocal[1]	
ImagePlane[i][x+2][y] = arrayLocal[2]	
ImagePlane[i][x+3][y] = arrayLocal[3]	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Tx = = NumTileCols − 1) && (Ty != NumTileRows − 1)) { /* Right edge for soft tiles */	
y = 16 * TopMBIndexOfTile[Ty + 1] / dy − 2	
x = 16 * LeftMBIndexOfTile[Tx + 1] / dx − 2	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x][y+1], ImagePlane[i][x][y+2], ImagePlane[i][x][y+3]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x][y+1] = arrayLocal[1]	
ImagePlane[i][x][y+2] = arrayLocal[2]	
ImagePlane[i][x][y+3] = arrayLocal[3]	
x = 16 * LeftMBIndexOfTile[Tx + 1] / dx − 1	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x][y+1], ImagePlane[i][x][y+2], ImagePlane[i][x][y+3]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	

Table 159 – Pseudocode for function SecondLevelOverlapFiltering( )

SecondLevelOverlapFiltering( ) {	**Reference**
ImagePlane[i][x][y+1] = arrayLocal[1]	
ImagePlane[i][x][y+2] = arrayLocal[2]	
ImagePlane[i][x][y+3] = arrayLocal[3]	
}	
if ((HARD_TILING_FLAG = = FALSE) && (Tx != NumTileCols − 1) && (Ty = = NumTileRows − 1)) { /* Bottom edge for soft tiles */	
x = 16 * LeftMBIndexOfTile[Tx + 1] / dx − 2	
y = 16 * TopMBIndexOfTile[Ty + 1] / dy − 2	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y], ImagePlane[i][x+2][y], ImagePlane[i][x+3][y]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x+1][y] = arrayLocal[1]	
ImagePlane[i][x+2][y] = arrayLocal[2]	
ImagePlane[i][x+3][y] = arrayLocal[3]	
y = 16 * TopMBIndexOfTile[Ty + 1] / dy − 1	
arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y], ImagePlane[i][x+2][y], ImagePlane[i][x+3][y]}	
OverlapPostFilter4(arrayLocal[ ])	9.9.8.2
ImagePlane[i][x][y] = arrayLocal[0]	
ImagePlane[i][x+1][y] = arrayLocal[1]	
ImagePlane[i][x+2][y] = arrayLocal[2]	
ImagePlane[i][x+3][y] = arrayLocal[3]	
}	
}	
}	
}	
}	

• • •

9.9.7.3 InvTodd( )

The function InvTodd( ) is specified by the pseudocode in Table 162.

Table 162 – Pseudocode for function InvTodd( )

InvTodd(iCoeff[ ]) {	**Reference**
iCoeff[1] += iCoeff[3]	
iCoeff[0] −= iCoeff[2]	
iCoeff[3] −= (iCoeff[1] >> 1)	
iCoeff[2] += ((iCoeff[0] + 1) >> 1)	
iCoeff[0] −= ((3* iCoeff[1] + 4) >> 3)	
iCoeff[1] += ((3* iCoeff[0] + 4) >> 3)	
iCoeff[2] −= ((3* iCoeff[3] + 4) >> 3)	
iCoeff[3] += ((3* iCoeff[2] + 4) >> 3	
iCoeff[2] −= ((iCoeff[1] + 1) >> 1)	
iCoeff[3] = ((iCoeff[0] + 1) >> 1) − iCoeff[3]~~d~~	
iCoeff[1] += iCoeff[2]	

iCoeff[0] −= iCoeff[3]	
}	

9.9.7.4 InvToddodd( )

The function, InvToddodd( ) is specified by the pseudocode in Table 163.

Table 163 – Pseudocode for function InvToddodd

InvToddodd(iCoeff[ ]) {	**Reference**
iCoeff[3] += iCoeff[0]	
iCoeff[2] −= iCoeff[1]	
valT1 = iCoeff[3] >> 1	
valT2 = iCoeff[2] >> 1	
iCoeff[0] −= valT~~t~~1	
iCoeff[1] += valT~~t~~2	
iCoeff[0] −= ((iCoeff[1] * 3 + 3) >> 3)	
iCoeff[1] += ((iCoeff[0] * 3 + 3) >> 2)	
iCoeff[0] −= ((iCoeff[1] * 3 + 4) >> 3)	
iCoeff[1] −= valT2	
iCoeff[0] += valT1	
iCoeff[2] += iCoeff[1]	
iCoeff[3] −= iCoeff[0]	
iCoeff[1] = −iCoeff[1]	
iCoeff[2] = −iCoeff[2]	
}	

• • •

9.9.8 Overlap f~~F~~iltering f~~F~~unctions

• • •

9.9.8.5 InvRotate( )

The function InvRotate( ) is specified by the pseudocode in Table 172.

Table 172 – Pseudocode for function InvRotate( )

InvRotate(iCoeff[ ]) {	**Reference**
iCoeff[0] −= ((-iCoeff[1] + 1) >> 1)	
iCoeff[1] += ((iCoeff[0] + 1) >> 1)	
}	

9.9.8.6 InvScale( )

The function InvScale( ) is specified by the pseudocode in Table 173.

Table 173 – Pseudocode for function InvScale( )

InvScale(iCoeff[ ]) {	**Reference**
iCoeff[0] += iCoeff[1]	
iCoeff[1] = (iCoeff[0] >> 1) − iCoeff[1]	
iCoeff[0] += (-iCoeff[1] * 3 + 0) >> 3	
iCoeff[1] += (~~(~~iCoeff[0] * 3 + 0) >> 4	
iCoeff[1] += (iCoeff[0] >> 7)	
iCoeff[1] −= (iCoeff[0] >> 10)	
}	

• • •

9.10.1 Overview

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

First, the decoder may be required to perform upsampling to obtain an intermediate YUV444 format. Next, color format conversion is applied to convert the internal color formats to output formats. The color format conversions are specified below. A bias is added to the sample values, to re-center the values around the nominal value for a neutral or zero intensity representation. When the scaling mode is used, on the decoder side, the values are rounded down after color conversion. For high numerical range formats (BD16, BD16S, BD32S and BD32F), the internal integer representations need to be converted to output representations. Finally, the values are clipped to fit the appropriate range.

9.10.2 Output formatting stages

At the completion of the transform and overlap filtering, the sample values for the image are reconstructed in an internal color format and an internal two's complement integer representation. The output formatting stage converts the decoded image plane data into a representation specified by the OUTPUT_C~~OLOR_FORMAT~~LR_FMT and the output bit depth. In the specification of output formatting, the term INTERNAL_CLR_FMT refers to the corresponding syntax element of the primary image plane.

The output formatting process is specified for the combinations of OUTPUT_BITDEPTH and OUTPUT_CLR_FMT that are listed in Table 176.

In this table, "+" indicates that output formatting is specified for the corresponding combinations of OUTPUT_BIT_DEPTH and OUTPUT_CLR_FMT. The combination of OUTPUT_BIT_DEPTH and OUTPUT_CLR_FMT shall not have the value corresponding to empty cells.

Table 176 – Conformance-specified output formatting combinations of OUTPUT_BITDEPTH and OUTPUT_CLR_FMT

OUTPUT_ BITDEPTH	**OUTPUT_CLR_FMT**								
	YONLY	**YUV420**	**YUV422**	**YUV444**	**RGB**	**RGBE**	**CMYK**	**CMYKDIRECT**	**NCOMPONENT**
• • •									

The output formatting stage consists of several sub-processes that are performed ~~in the sequence shown~~as specified in Table 177.

• • •

9.10.3 Sampling conversion

9.10.3.1 ~~Overview~~General

The sampling conversion process is specified in Table 178.

The combinations of INTERNAL_CLR_FMT and OUTPUT_C~~OLOR_FORMAT~~LR_FMT~~ and INTERNAL_CLR_FMT ~~for which sampling conversions are specified for conformance purposes ~~is ~~are specified in Table 179. In this table, "+" indicates that no sampling conversion is required. It is a requirement of codestream conformance to this Specification that the~~The~~ combination of INTERNAL_CLR_FMT and OUTPUT_CLR_FMT~~ and INTERNAL_CLR_FMT ~~shall not have ~~the ~~a value corresponding to any empty cell in Table 179~~s~~.

~~The sampling conversion process is specified in Table 178.~~

In the illustrated case in Table 178 in which upsampling is specified both vertically and horizontally for INTERNAL_CLR_FMT equal to YUV420, the upsampling process to be performed by the decoder shall produce an array of two-dimensionally upsampled samples at the index values for which such samples are produced as specified by Table 178. However, decoders may use alternative upsampling methods (different from that specified by Table 178) – the actual filtering method used to produce the values of the entries in the upsampled array is outside the scope of this Specification. The particular filtering method specified by Table 178 is an example of how such upsampled array values may be produced. For example, upsampling may be applied both vertically and horizontally as a single process, or the relative ordering of the vertical and horizontal upsampling may be switched.

NOTE – When TILING_FLAG is equal to TRUE and the transform processing does not cross tile boundaries (due either to HARD_TILING_FLAG being equal to TRUE or OVERLAP_MODE being equal to 0), the example upsampling method illustrated in Table 178 for cases with INTERNAL_CLR_FMT equal to YUV422 or YUV420 will produce an upsampled image in which the output samples next to tile boundaries may be affected by the values of decoded samples in other tiles. For many applications, it may be desirable to instead design the upsampling process to be performed separately within each tile in order to avoid this cross-tile dependency.

Table 178 – Pseudocode for function SamplingConversion( )

SamplingConversion( ) {	**Reference**
if (((INTERNAL_CLR_FMT = =YUV422) | | (INTERNAL_CLR_FMT = =YUV420)) && ((OUTPUT_CLR_FMT = = YUV444) | | (OUTPUT_CLR_FMT = =RGB)) {~~if (OUTPUT_CLR_FMT != INTERNAL_CLR_FMT) {~~	
~~if (((INTERNAL_CLR_FMT = =YUV420) | | (INTERNAL_CLR_FMT = =YUV422)) && ((OUTPUT_CLR_FMT = = YUV444) | | ((OUTPUT_CLR_FMT = =RGB) && ((OUTPUT_BITDEPTH = = BD5) | | (OUTPUT_BITDEPTH = = BD565) | | (OUTPUT_BITDEPTH = = BD10) | | (OUTPUT_BITDEPTH = = BD8) | | (OUTPUT_BITDEPTH = = BD16) | | (OUTPUT_BITDEPTH = = BD16S) | | (OUTPUT_BITDEPTH = = BD32S))))) {~~	
if (INTERNAL_CLR_FMT = = YUV420)	
Upsample( ) in the vertical direction	9.10.3.2
~~1.1.1~~ if ((INTERNAL_CLR_FMT = = YUV422) | | (INTERNAL_CLR_FMT = = YUV420))	
Upsample( ) in the horizontal direction	9.10.3.2
~~1.1.1.1.1.1.1.1.1. if (INTERNAL_CLR_FMT = = YUV420) {~~	
~~Upsample( ) in the horizontal direction~~	~~9.10.3.2~~
~~Upsample( ) in the vertical direction~~	~~9.10.3.2~~
~~}~~	
}	
~~}~~	
}	

Table 179 – Conformance-specified sampling conversions

OUTPUT_-CLR_FMT	**INTERNAL_CLR_FMT**					
	YONLY	**YUV420**	**YUV422**	**YUV444**	**YUVK**	**NCOMPONENT**
YONLY	+					
YUV420		+				
YUV422			+			
YUV444	+	Upsample( ) in the vertical and horizontal ~~and vertical~~ directions	Upsample( ) in the horizontal direction	+		
RGB with OUTPUT_BITDEPTH equal to BD5, BD565, ~~BD10,~~ BD8, BD10,	+	Upsample( ) in the vertical and horizontal ~~and vertical~~	Upsample( ) in the horizontal direction	+		

BD16, BD16S or BD32S		directions				
RGB with OUTPUT_BITDEPTH equal to BD16F or BD32F				+		
RGBE				+		
CMYK					+	
CMYKDIRECT					+	
NCOMPONENT						+

~~9.10.4.2~~9.10.3.2 Upsample( )

In the chroma upsampling function, for the chroma component i (1 <= i < NumComponents), let iOriArray[ ] be the original sample array before upsampling, and iIntArray[ ] be the upsampled array. If upsampling is performed in the horizontal direction, iOriArray[ ] is one input sample row of length ExtendedWidth[i] and iIntArray[ ] is one output sample row of length ExtendedWidth[0], and the variable iOriLength ~~upsampledLength~~ is set equal to ExtendedWidth[i~~0~~]. Otherwise, iOriArray[ ] is one input sample column of length ExtendedHeight[i] and iIntArray[ ] is one output sample column of length ExtendedHeight[0], and iOriLength ~~upsampledLength~~ is set equal to ExtendedHeight[~~0~~i].

The upsampling process to be performed by the decoder shall produce an array of upsampled samples ~~iIntArray[ ]~~ at the index values for which such samples are produced as specified by Table 180. However, decoders may use alternative upsampling methods (different from that specified by Table 180) – the actual filtering method used to produce the values of the entries in the upsampled array ~~iIntArray[ ]~~ is outside the scope of this Specification. The particular filtering method specified by Table 180 is an example of how such upsampled array values may be produced. For example, a different type of filtering or a different number of taps may be used during the upsampling process than the two-tap filtering specified by Table 180.

Table 180 – Pseudocode for function Upsample( )

Upsample(~~iOriArray[ ], iIntArray[ ], upsampledLength~~) {	**Reference**
for (k = 0; k <= iOriLength~~(upsampledLength − 2) / 2~~; k++) {	
iIntArray[2 * k] = ((iH[2] * iOriArray[Max(0, k − 1)] + iH[3] * iOriArray[k] + 4) >> 3)	
iIntArray[2 * k + 1] = ((iH[0] * iOriArray[k] + iH[1] * iOriArray[Min(iOriLength~~upsampledLength / 2~~ − 1, k + 1)] + 4) >> 3)	
~~for (k = −1; k <= (upsampledLength − 4) / 2; k++)~~	
~~iIntArray[2k+2] = ((iH[2] * iOriArray[Max(0, k)] + iH[3] * iOriArray[k+1] + 4) >> 3)~~	
}	
}	

The values of the filter coefficients iH[0], iH[1], iH[2], iH[3] for the chroma positions are specified by Table 181 as a function of the variable chromaCentering. If Upsample( ) is applied in the horizontal direction, chromaCentering is set equal to CHROMA_CENTERING_X; otherwise, it is set equal to CHROMA_CENTERING_Y.

Table 181 – Upsampling filter coefficient for different chroma positions

chromaCentering	iH[0]	iH[1]	iH[2]	iH[3]
0	4	4	0	8
1	5	3	1	7
2	6	2	2	6
3	7	1	3	5
4	8	0	4	4

~~9.10.5~~9.10.4 Conversion from INTERNAL_CLR_FMT to OUTPUT_CLR_FMT

~~9.10.5.1~~9.10.4.1 Overview

The conversion process proceeds as specified in Table 182.

Table 182 – Pseudocode for function ConvertInternalToOutputClrFmt( )

ConvertInternalToOutputClrFmt( ) {	**Reference**
if ((INTERNAL_CLR_FMT = = YUVK) && (OUTPUT_CLR_FMT = = CMYK))	
InvColorFmtConvert3( )	9.10.4.4
else if ((INTERNAL_CLR_FMT = = YUVK) && (OUTPUT_CLR_FMT = = CMYKDIRECT))	
InvColorFmtConvert4( )	9.10.4.5
else if ((INTERNAL_CLR_FMT = = YONLY) && ~~(~~(OUTPUT_CLR_FMT = = RGB)~~ && ((OUTPUT_BITDEPTH = = BD5) | | (OUTPUT_BITDEPTH = = BD565) | | (OUTPUT_BITDEPTH = = BD10) | | (OUTPUT_BITDEPTH = = BD8) | | (OUTPUT_BITDEPTH = = BD16) | | (OUTPUT_BITDEPTH = = BD16S) | | (OUTPUT_BITDEPTH = = BD32S)))~~)	
InvColorFmtConvert1( )	9.10.4.2
else if (((INTERNAL_CLR_FMT = = YUV444) | | (INTERNAL_CLR_FMT = = YUV422) | | (INTERNAL_CLR_FMT = = YUV420)) && ((OUTPUT_CLR_FMT = = RGB) ~~&& ((OUTPUT_BITDEPTH = = BD5) | | (OUTPUT_BITDEPTH = = BD565) | | (OUTPUT_BITDEPTH = = BD10) | | (OUTPUT_BITDEPTH = = BD8) | | (OUTPUT_BITDEPTH = = BD16) | | (OUTPUT_BITDEPTH = = BD16S) | | (OUTPUT_BITDEPTH = = BD32S)))~~| | (OUTPUT_CLR_FMT = = RGBE)))	
InvColorFmtConvert2( )	9.10.4.3
else if ((INTERNAL_CLR_FMT = = YONLY) && ((OUTPUT_CLR_FMT = = YUV444) | | (OUTPUT_CLR_FMT = = YUV422) | | (OUTPUT_CLR_FMT = = YUV420))) {~~(INTERNAL_CLR_FMT = = YUV444) && ((OUTPUT_CLR_FMT = = RGB) | | (OUTPUT_CLR_FMT = = RGBE)))~~	
if (OUTPUT_CLR_FMT = = YUV420)	
chromaHeight = ExtendedHeight[0] / 2	
else	
chromaHeight = ExtendedHeight[0]	
if ((OUTPUT_CLR_FMT = = YUV422) | | (OUTPUT_CLR_FMT = = YUV420))	
chromaWidth = ExtendedWidth[0] / 2	
else	
chromaWidth = ExtendedWidth[0]	
for (i = 1; i < 3; i ++)	
for (y = 0; y < chromaHeight; y++)	
for (x = 0; x < chromaWidth; x++)	
ImagePlane[i][x][y] = 0 /* Ensure that chroma is inferred as zero */	
}	
~~InvColorFmtConvert2( )~~	~~9.10.4.3~~
~~/* if ((INTERNAL_CLR_FMT != YUVK) && (OUTPUT_CLR_FMT = = INTERNAL_CLR_FMT)) no conversion is performed */~~	
}	

The combinations of INTERNAL_CLR_FMT and OUTPUT_CLR_FMT for which color format conversions are specified for conformance purposes are specified ~~The set of color format conversions between INTERNAL_CLR_FMT and OUTPUT_CLR_FMT that are specified for conformance purposes is listed~~ in Table 183. In this table, "+" indicates that no color format conversions ~~are~~ is required. For cases that require color format conversion, the function name for the conversion process is specified in the table cell. It is a requirement of codestream conformance to this Specification that the ~~The~~ combination of INTERNAL_CLR_FMT and OUTPUT_CLR_FMT shall not have a value corresponding~~OUTPUT_CLR_FMT and INTERNAL_CLR_FMT values shall not correspond~~ to any empty cell in Table 183.

Table 183 – Conformance-specified color format conversions

OUTPUT_-CLR_FMT	**INTERNAL_CLR_FMT**					
	YONLY	**YUV420**	**YUV422**	**YUV444**	**YUVK**	**NCOMPONENT**
YONLY	+					
YUV420		+				
YUV422			+			
YUV444	+	+	+	+		
RGB with OUTPUT_BITDEPTH equal to BD5, BD565, ~~BD10,~~ BD8, BD10, BD16, BD16S or BD32S	InvColorFmt Convert1( )	InvColorFmt Convert2( )	InvColorFmt Convert2( )	InvColorFmt Convert2( )		
RGB with OUTPUT_BITDEPTH equal to BD16F or BD32F				InvColorFmt Convert2( )		
RGBE				InvColorFmt Convert2( )		
CMYK					InvColorFmt Convert3( )	
CMYKDIRECT					InvColorFmt Convert4( )	
NCOMPONENT						+

The pseudocode functions InvColorFmtConvert1( ), InvColorFmtConvert2( ), InvColorFmtConvert3( ), and InvColorFmtConvert4( ) that are ~~(~~referred to in~~by~~ Table 183~~)~~ are specified in subclause 9.10.4.2, subclause 9.10.4.3, subclause 9.10.4.4, and subclause 9.10.4.5, respectively.

~~9.10.5.2~~9.10.4.2 InvColorFmtConvert1( )

The operations in InvColorFmtConvert1( ) are specified in Table 184.

Table 184 – Pseudocode for function InvColorFmtConvert1( )

InvColorFmtConvert1(~~valXPos, valYPos~~) {	**Reference**
for (y = 0; y < ExtendedHeight[0]; y++)	
for (x = 0; x < ExtendedWidth[0]; x++) {	
~~/* valXPos and valYPos are meant to indicate the x, y positions of the current sample */~~	
/* ImagePlane[0][~~valXPos~~x][~~valYPos~~y] G = Y */	
ImagePlane[1][~~valXPos~~x][~~valYPos~~y] = ImagePlane[0][~~valXPos~~x][~~valYPos~~y] /* R = Y */	
ImagePlane[2][~~valXPos~~x][~~valYPos~~y] = ImagePlane[0][~~valXPos~~x][~~valYPos~~y] /* B = Y */	
}	
}	

~~9.10.5.3~~9.10.4.3 InvColorFmtConvert2( )

The operations in InvColorFmtConvert2( ) are specified in Table 185.

Table 185 – Pseudocode for function InvColorFmtConvert2( )

InvColorFmtConvert2(~~valXPos, valYPos~~) {	**Reference**
~~/* valXPos and valYPos are meant to indicate the x, y positions of the current sample */~~ for (y = 0; y < ExtendedHeight[0]; y++)	
for (x = 0; x < ExtendedWidth[0]; x++) {	
tempT = −~~-~~ImagePlane[1][~~valXPos~~x][~~valYPos~~y] /* t = −U */	
arrayOut[1] = ImagePlane[0][~~valXPos~~x][~~valYPos~~y] − Floor(tempT ~~/~~÷ 2) /* G = Y − Floor(t ~~/~~÷ 2) */	
arrayOut[0] = tempT + arrayOut[1] − Ceiling(ImagePlane[2][~~valXPos~~x][~~valYPos~~y] ~~/~~÷ 2) /* R = t + G − Ceiling(V ÷~~/~~ 2) */	
arrayOut[2] = ImagePlane[2][~~valXPos~~x][~~valYPos~~y] + arrayOut[0] /* B = V + R */	
for (i = 0; i < 3; i++)	
ImagePlane[i][~~valXPos~~x][~~valYPos~~y] = arrayOut[i]	
}	
}	

~~9.10.5.4~~9.10.4.4 InvColorFmtConvert3( )

The operations in InvColorFmtConvert3( ) are specified in Table 186.

Table 186 – Pseudocode for function InvColorFmtConvert3( )

InvColorFmtConvert3(~~valXPos, valYPos~~) {	**Reference**
for (y = 0; y < ExtendedHeight[0]; y++)~~/* valXPos and valYPos are meant to indicate the x, y positions of the current sample */~~	
for (x = 0; x < ExtendedWidth[0]; x++) {	
arrayOut[3] = ImagePlane[3][~~valXPos~~x][~~valYPos~~y] + Floor(ImagePlane[0][~~valXPos~~x][~~valYPos~~y] ~~/~~÷ 2) /* k = K + Floor(Y ~~/~~÷ 2) */	
arrayOut[1] = arrayOut[3] − ImagePlane[0][~~valXPos~~x][~~valYPos~~y] − Floor(ImagePlane[1][~~valXPos~~x][~~valYPos~~y] ~~/~~÷ 2) /* m = k − Y − Floor(U ~~/~~÷ 2) */	
arrayOut[0] = ImagePlane[1][~~valXPos~~x][~~valYPos~~y] + arrayOut[1] + Floor(ImagePlane[2][~~valXPos~~x][~~valYPos~~y] ~~/~~÷ 2) /* c = U + m + Floor(V ~~/~~÷ 2) */	
arrayOut[2] = arrayOut[0] − ImagePlane[2][~~valXPos~~x][~~valYPos~~y] /* y = c − V */	
for (i = 0; i < 4; i++)	
ImagePlane[i][~~valXPos~~x][~~valYPos~~y] = arrayOut[i]	
}	
}	

~~9.10.5.5~~9.10.4.5 InvColorFmtConvert4( )

The operations in InvColorFmtConvert4( ) are specified in Table 187.

Table 187 – Pseudocode for function InvColorFmtConvert4( )

InvColorFmtConvert4(~~valXPos, valYPos~~) {	**Reference**
for (y = 0; y < ExtendedHeight[0]; y++)~~/* valXPos and valYPos are meant to indicate the x, y positions of the current sample */~~	
for (x = 0; x < ExtendedWidth[0]; x++) {	
arrayOut[3] = ImagePlane[0][~~valXPos~~x][~~valYPos~~y] /* k = Y */	
arrayOut[1] = ImagePlane[2][~~valXPos~~x][~~valYPos~~y] /* m = V */	
arrayOut[0] = ImagePlane[1][~~valXPos~~x][~~valYPos~~y] /* c = U */	

arrayOut[2] = ImagePlane[3][~~valXPos~~x][~~valYPos~~y] /* y = K */	
for (i = 0; i < 4; i++)	
ImagePlane[i][~~valXPos~~x][~~valYPos~~y] = arrayOut[i]	
}	
}	

~~9.10.6~~9.10.5 AddBias( )

The function AddBias( ) specified in Table 188 ~~specifies~~ performs the computation and addition of bias to the sample values.

Table 188 – Pseudocode for function AddBias( )

AddBias(~~valXPos, valYPos~~) {	**Reference**
if (SCALED_FLAG)~~iScale = 0 /* a scale factor when SCALED_FLAG is equal to TRUE */~~	
~~/* valXPos and valYPos are meant to indicate the x, y positions of the current sample */~~	
~~if (SCALED_FLAG)~~	
iScale = 3	
else	
iScale = 0	
~~if (OUTPUT_BITDEPTH = = BD1WHITE1)~~	
~~iBias = 0~~	
if (OUTPUT_BITDEPTH = = BD5)	
iBias = (1 << 4)	
else if (OUTPUT_BITDEPTH = = BD565)	
iBias = (1 << 5)	
else if (OUTPUT_BITDEPTH = = BD8)	
iBias = (1 << 7)	
else if (OUTPUT_BITDEPTH = = BD10)	
iBias = (1 << 9)	
else if (OUTPUT_BITDEPTH = = BD16)	
iBias = (1 << 15)	
~~else if (OUTPUT_BITDEPTH = = BD5)~~	
~~iBias = (1 << 4)~~	
~~else if (OUTPUT_BITDEPTH = = BD10)~~	
~~iBias = (1 << 9)~~	
~~else if (OUTPUT_BITDEPTH = = BD565)~~	
~~iBias = (1 << 5)~~	
else	
iBias = 0	
if ((OUTPUT_BITDEPTH = = BD16) | | (OUTPUT_BITDEPTH = = BD16S) | | (OUTPUT_BITDEPTH = = BD32S))	
iBias = (iBias >> SHIFT_BITS)	
if ((OUTPUT_CLR_FMT = = RGB) | | (OUTPUT_CLR_FMT = = YUV444) | | (OUTPUT_CLR_FMT = = YUV422) | | (OUTPUT_CLR_FMT = = YUV420) | | (OUTPUT_CLR_FMT = = YONLY) | | (OUTPUT_CLR_FMT = = NCOMPONENT) | | (OUTPUT_CLR_FMT = = CMYKDIRECT)) {~~OUTPUT_CLR_FMT = = RGB) {~~	
if ((OUTPUT_CLR_FMT = = RGB) | | (OUTPUT_CLR_FMT = = YUV444) | | (OUTPUT_CLR_FMT = = YUV422) | | (OUTPUT_CLR_FMT = = YUV420))	
outputComponents = 3	
else	
outputComponents = NumComponents	

Table 188 – Pseudocode for function AddBias( )

AddBias(~~valXPos, valYPos~~) {	**Reference**
for (i = 0; i < outputComponents; i++) {	
if ((i > 0) && (OUTPUT_CLR_FMT = = YUV420))	
outputHeight = ExtendedHeight[0] / 2	
else	
outputHeight = ExtendedHeight[0]	
if ((i > 0) && ((OUTPUT_CLR_FMT = = YUV422) | | (OUTPUT_CLR_FMT = = YUV420)))	
outputWidth = ExtendedWidth[0] / 2	
else	
outputWidth = ExtendedWidth[0]	
for (y = 0; y < outputHeight; y++)	
for (x = 0; x < outputWidth; x++)	
ImagePlane[i~~0~~][~~valXPos~~x][~~valYPos~~y] += (iBias << iScale) ~~/* r */~~	
}	
~~ImagePlane[1][valXPos][valYPos] += (iBias << iScale) /* g */~~	
~~ImagePlane[2][valXPos][valYPos] += (iBias << iScale) /* b */~~	
~~} else if (OUTPUT_CLR_FMT = = YUV420 | | OUTPUT_CLR_FMT = = YUV422 | | OUTPUT_CLR_FMT = = YUV444) {~~	
~~ImagePlane[0][valXPos][valYPos] += (iBias << iScale) /* y */~~	
~~ImagePlane[1][valXPos][valYPos] += (iBias << iScale) /* u */~~	
~~ImagePlane[2][valXPos][valYPos] += (iBias << iScale) /* v */~~	
~~} else if (OUTPUT_CLR_FMT = = YONLY)~~	
~~ImagePlane[0][valXPos][valYPos] += (iBias << iScale) /* y */~~	
~~else if (OUTPUT_CLR_FMT = = NCOMPONENT | | OUTPUT_CLR_FMT = = CMYKDIRECT)~~	
~~for (i = 0; i<= iComponent; i++)~~	
~~ImagePlane[i][valXPos][valYPos] += (iBias << iScale) /* Component i */~~	
} else if (~~(~~OUTPUT_CLR_FMT = = CMYK~~) &&~~ ~~((OUTPUT_BITDEPTH = = BD8) | | (OUTPUT_BITDEPTH = = BD16))~~) {	
for (i = 0; i < 3; i++)~~ /* output format CMYK */~~	
for (y = 0; y < ExtendedHeight[0]; y++)	
for (x = 0; x < ExtendedWidth[0]; x++)~~ ImagePlane[0][valXPos][valYPos] += ((iBias >> 1) << iScale) /* c */~~	
ImagePlane[i~~1~~][~~valXPos~~x][~~valYPos~~y] += ((iBias >> 1) << iScale) /* c, m, y */	
for (y = 0; y < ExtendedHeight[0]; y++)	
for (x = 0; x < ExtendedWidth[0]; x++)	
~~ImagePlane[2][valXPos][valYPos] += ((iBias >> 1) << iScale) /* y */~~	
ImagePlane[3][~~valXPos~~x][~~valYPos~~y] −= ((iBias >> 1) << iScale) /* k */	
}	
}	

~~9.10.7~~9.10.6 ComputeScaling( )

The function ComputeScaling( ) specified in Table 189 ~~specifies~~ performs the computation of the scaling factor iScale, and the rounding factor iRoundingFactor, and ~~specifies how they modify~~modifies sample values based on these two factors.

Table 189 – Pseudocode for function ComputeScaling( )

ComputeScaling(~~valXPos, valYPos~~) {	**Reference**
~~/* valXPos and valYPos are meant to indicate the x, y positions of the current sample */~~	
iScale = 0	
iRoundingFactor = 0	
if (SCALED_FLAG) {	
iScale = 3	
if ((OUTPUT_BITDEPTH = = BD5) | | (OUTPUT_BITDEPTH = = BD565) | | (OUTPUT_BITDEPTH = = BD8) | | (OUTPUT_BITDEPTH = = BD10) | | (OUTPUT_BITDEPTH = = BD16S) | | (OUTPUT_BITDEPTH = = BD16F) | | (OUTPUT_BITDEPTH = = BD32S) | | (OUTPUT_BITDEPTH = = BD32F)) ~~if ((OUTPUT_BITDEPTH = = BD8) | | (OUTPUT_BITDEPTH = = BD16S) | |~~ ~~(OUTPUT_BITDEPTH = = BD16F) | | (OUTPUT_BITDEPTH = = BD32S) | |~~ ~~(OUTPUT_BITDEPTH = = BD32F) | | (OUTPUT_BITDEPTH = = BD5) | |~~ ~~(OUTPUT_BITDEPTH = = BD565) | | (OUTPUT_BITDEPTH = = BD10))~~	
iRoundingFactor = 3	
else if (~~(OUTPUT_BITDEPTH = = BD16) | |~~ (OUTPUT_BITDEPTH = = BD1WHITE1) | | (OUTPUT_BITDEPTH = = BD1BLACK1) | | (OUTPUT_BITDEPTH = = BD16))	
iRoundingFactor = 4	
}	
~~if ((OUTPUT_CLR_FMT = = RGB) | | (OUTPUT_CLR_FMT = = RGBE)) {~~	
~~if (OUTPUT_BITDEPTH != BD565) {~~	
~~ImagePlane[0][valXPos][valYPos] =~~ ~~((ImagePlane[0][valXPos][valYPos] + iRoundingFactor) >> iScale) /* r */~~	
~~ImagePlane[1][valXPos][valYPos] =~~ ~~((ImagePlane[1][valXPos][valYPos] + iRoundingFactor) >> iScale) /* g */~~	
~~ImagePlane[2][valXPos][valYPos] =~~ ~~((ImagePlane[2][valXPos][valYPos] + iRoundingFactor) >> iScale) /* b */~~	
~~} else {~~	
~~ImagePlane[0][valXPos][valYPos] =~~ ~~((ImagePlane[0][valXPos][valYPos] + iRoundingFactor) >> (iScale + 1)) /* r */~~	
~~ImagePlane[1][valXPos][valYPos] =~~ ~~((ImagePlane[1][valXPos][valYPos] + iRoundingFactor) >> iScale) /* g */~~	
~~ImagePlane[2][valXPos][valYPos] =~~ ~~((ImagePlane[2][valXPos][valYPos] + iRoundingFactor) >> (iScale + 1)) /* b */~~	
~~}~~	
if ((OUTPUT_CLR_FMT = = RGB) | | (OUTPUT_CLR_FMT = = RGBE) | | (OUTPUT_CLR_FMT = = YUV444) | | (OUTPUT_CLR_FMT = = YUV422) | | (OUTPUT_CLR_FMT = = YUV420))	
outputComponents = 3	
else	
outputComponents = NumComponents	
for (i = 0; i < outputComponents; i++) {	
if ((i > 0) && (OUTPUT_CLR_FMT = = YUV420))	
outputHeight = ExtendedHeight[0] / 2	
else	
outputHeight = ExtendedHeight[0]	
if ((i > 0) && ((OUTPUT_CLR_FMT = = YUV422) | | (OUTPUT_CLR_FMT = = YUV420)))	
outputWidth = ExtendedWidth[0] / 2	
else	
outputWidth = ExtendedWidth[0]	

Table 189 – Pseudocode for function ComputeScaling( )

ComputeScaling(~~valXPos, valYPos~~) {	Reference
if ((OUTPUT_BITDEPTH = = BD565)) && (i != 1))	
jScale = iScale + 1	
else	
jScale = iScale	
for (y = 0; y < outputHeight; y++)	
for (x = 0; x < outputWidth; x++)	
ImagePlane[i][x][y] = ((ImagePlane[i][x][y] + iRoundingFactor) >> jScale)	
}	
~~} else if ((OUTPUT_CLR_FMT = = YUV420) | | (OUTPUT_CLR_FMT = = YUV422) | | (OUTPUT_CLR_FMT = = YUV444)) {~~	
~~ImagePlane[0][valXPos][valYPos] = ((ImagePlane[0][valXPos][valYPos] + iRoundingFactor) >> iScale) /* y */~~	
~~ImagePlane[1][valXPos][valYPos] = ((ImagePlane[1][valXPos][valYPos] + iRoundingFactor) >> iScale) /* u */~~	
~~ImagePlane[2][valXPos][valYPos] = ((ImagePlane[2][valXPos][valYPos] + iRoundingFactor) >> iScale) /* v */~~	
~~} else if (OUTPUT_CLR_FMT = = YONLY)~~	
~~ImagePlane[0][valXPos][valYPos] = ((ImagePlane[0][valXPos][valYPos] + iRoundingFactor) >> iScale)~~	
~~else if (OUTPUT_CLR_FMT = = NCOMPONENT)~~	
~~for (i = 0; i<= iComponent; i++)~~	
~~ImagePlane[i][valXPos][valYPos] = ((ImagePlane[i][valXPos][valYPos] + iRoundingFactor) >> iScale)~~	
~~else if ((OUTPUT_CLR_FMT = = CMYK) | | (OUTPUT_CLR_FMT = = CMYKDIRECT)) {~~	
~~ImagePlane[0][valXPos][valYPos] = ((ImagePlane[0][valXPos][valYPos] + iRoundingFactor) >> iScale) /* c */~~	
~~ImagePlane[1][valXPos][valYPos] = ((ImagePlane[1][valXPos][valYPos] + iRoundingFactor) >> iScale) /* m */~~	
~~ImagePlane[2][valXPos][valYPos] = ((ImagePlane[2][valXPos][valYPos] + iRoundingFactor) >> iScale) /* y */~~	
~~ImagePlane[3][valXPos][valYPos] = ((ImagePlane[3][valXPos][valYPos] + iRoundingFactor) >> iScale) /* k */~~	
~~}~~	
}	

~~9.10.8~~9.10.7 Postscaling process

~~9.10.8.1~~9.10.7.1 Overview

The function PostscalingProcess( ) ~~The Postscaling process~~ is specified in Table 190.

Table 190 – Pseudocode for function PostscalingProcess( )

PostscalingProcess(~~valXPos, valYPos~~) {	Reference
~~/* valXPos and valYPos are meant to indicate the x, y positions of the current sample */~~	
if (OUTPUT_CLR_FMT = = RGBE)	
for (y = 0; y < ExtendedHeight[0]; y++)	
for (x = 0; x < ExtendedWidth[0]; x++) {	
for (k = 0; k < 3; k++)	
localArrayIn[k] = ImagePlane[k][x][y]	
PostScalingF2(localArrayOut[ ], localArrayIn[ ]) /* Produces 4 outputs for 3	9.10.7.4

inputs */	
for (k = 0; k < 4; k++)	
ImagePlane[k][x][y] = localArrayOut[k]	
}	
else {	
if ((OUTPUT_CLR_FMT = = RGB) | | (OUTPUT_CLR_FMT = = YUV444) | | (OUTPUT_CLR_FMT = = YUV422) | | (OUTPUT_CLR_FMT = = YUV420))	
outputComponents = 3	
else	
outputComponents = NumComponents	
for (i = 0; i < outputComponents; i++) {	
if ((i > 0) && (OUTPUT_CLR_FMT = = YUV420))	
outputHeight = ExtendedHeight[0] / 2	
else	
outputHeight = ExtendedHeight[0]	
if ((i > 0) && ((OUTPUT_CLR_FMT = = YUV422) | | (OUTPUT_CLR_FMT = = YUV420)))	
outputWidth = ExtendedWidth[0] / 2	
else	
outputWidth = ExtendedWidth[0]	
if ((OUTPUT_BITDEPTH = = BD16) | | (OUTPUT_BITDEPTH = = BD16S) | | (OUTPUT_BITDEPTH = = BD32S))	
for (y = 0; y < outputHeight; y++)	
for (x = 0; x < outputWidth; x++)	
ImagePlane[i][x][y] = PostScalingInt(ImagePlane[i][x][y])	9.10.7.2
else if ((OUTPUT_BITDEPTH = = BD32F) | | (OUTPUT_BITDEPTH = = BD16F))	
for (y = 0; y < outputHeight; y++)	
for (x = 0; x < outputWidth; x++)	
ImagePlane[i][x][y] = PostScalingFl(ImagePlane[i][x][y])	9.10.7.3
}	
~~if ((OUTPUT_BITDEPTH = = BD16) | | (OUTPUT_BITDEPTH = = BD16S) | | (OUTPUT_BITDEPTH = = BD32S))~~	
~~for (i = 0; i < NumComponents; i++)~~	
~~ImagePlane[i][valXPos][valYPos] = PostScalingInt(ImagePlane[i][valXPos][valYPos])~~	~~9.10.7.2~~
~~if ((OUTPUT_BITDEPTH = = BD32F) | | (OUTPUT_BITDEPTH = = BD16F))~~	
~~for (i = 0; i < NumComponents; i++)~~	
~~ImagePlane[i][valXPos][valYPos] = PostScalingFl(ImagePlane[i][valXPos][valYPos])~~	~~9.10.7.3~~
~~if ((OUTPUT_CLR_FMT = = RGBE)) {~~	
~~localArrayIn[ ] = {ImagePlane[0][valXPos][valYPos], ImagePlane[1][valXPos][valYPos], ImagePlane[2][valXPos][valYPos]}~~	
~~PostScalingF2(localArrayOut[ ], localArrayIn[ ])~~	~~9.10.7.4~~
~~ImagePlane[0][valXPos][valYPos] = localArrayOut[0]~~	
~~ImagePlane[1][valXPos][valYPos] = localArrayOut[1]~~	
~~ImagePlane[2][valXPos][valYPos] = localArrayOut[2]~~	
~~ImagePlane[3][valXPos][valYPos] = localArrayOut[3]~~	
}	
}	

~~9.10.8.2~~9.10.7.2 PostScalingInt( )

The sample values are left-shifted by the amount determined by SHIFT_BITS. For input value~~,~~ inX, the output shifted value~~,~~ outX~~,~~ is determined as specified in Table 191.

Table 191 – Pseudocode for function PostScalingInt( )

PostScalingInt(inX) {	**Reference**
outX = inX << SHIFT_BITS	
return outX	
}	

NOTE – In this manner, the output is moved from a 27-bit or 24-bit nominal range scaling to the range scaling specified for image reconstruction. The 27-bit range scaling applies when the data is scaled, and the 24-bit range scaling applies when the data is unscaled.~~In this manner the output is moved from the 24/27 bit range to the original range. The 27 bit limit is used when data is scaled, and the 24 bit limit applies when the data is unscaled.~~

~~9.10.8.3~~9.10.7.3 PostScalingFl( )

~~If~~ When OUTPUT_BITDEPTH is equal to BD32F or BD16F, the integer sample value iX is converted to a value fV that can be interpreted as a floating point representation~~floating point value fV~~.

~~The inputs to this process are the syntax element LEN_MANTISSA and the syntax element EXP_BIAS.~~

The PostScalingFl( ) process computes the ~~floating point~~ value fV as specified in Table 192.

Table 192 – Pseudocode for function PostScalingFl( )

PostScalingFl(iX) {	**Reference**
if (iX < 0)	
iS = 1	
else	
iS = 0	
if (OUTPUT_BITDEPTH = = BD16F) {	
iEM = Min(Abs(iX), 32767)	
~~if (iEM > 0x7FFF)~~	
~~iEM = 0x7FFF~~	
fV = ((iS << 15) | iEM) /* Concatenate these fields*/	
} else { /* OUTPUT_BITDEPTH = = BD32F */	
iX = Abs(iX)	
iE = (iX >> LEN_MANTISSA)	
iM = ((iX & ((1 << LEN_MANTISSA) − 1)) | (1 << LEN_MANTISSA))	
if (iE = = 0) {	
iM ^= (1 << LEN_MANTISSA)	
iE = 1	
}	
iE = iE − EXP_BIAS + 127	
while ((iM < (1 << LEN_MANTISSA)) && (iE > 1) && (iM > 0)) {	
iE −= 1	
iM <<= 1	
}	
if (iM < (1 << LEN_MANTISSA))	
iE = 0	
else	
iM ^= (1 << LEN_MANTISSA)	
iM <<= (23 − LEN_MANTISSA)	

Table 192 – Pseudocode for function PostScalingFl( )

PostScalingFl(iX) {	**Reference**
fV = ((iS << 31) | (iE << 23) | iM) /* Concatenate these fields */	
}	
return fV	
}	

~~9.10.8.4~~9.10.7.4 PostScalingF2( )

~~If~~ When OUTPUT_CLR_FMT is equal to RGBE, the three integer sample values of array arrayIn[ ] (R, G, and B) are converted to an array arrayOut[ ] of four integer ~~RGBE~~ values forming the RGBE representation (Rrgbe, Grgbe, Brgbe and Ergbe). The function PostScalingF2( ) specified in Table 193 performs ~~specifies~~ the conversion.

Table 193 – Pseudocode for function PostScalingF2( )

PostScalingF2(arrayOut[ ], arrayIn[ ]) {	**Reference**
/* arrayIn[ ]= {R, G, B} */	
/* arrayOut[ ]= {Rrgbe, Grgbe, Brgbe, Ergbe} */	
if (arrayIn[0] <= 0) {	
arrayOut[0] = 0	
iEr = 0	
} else if ((arrayIn[0] >> 7) > 1) {	
arrayOut[0] = (arrayIn[0] & 0x7F) + ~~0x80~~128	
iEr = (arrayIn[0] >> 7)	
} else {	
arrayOut[0] = arrayIn[0]	
iEr = 1	
}	
if (arrayIn[1] <= 0) {	
arrayOut[1] = 0	
iEg = 0	
} else if ((arrayIn[1] >> 7) > 1) {	
arrayOut[1] = (arrayIn[1] & 0x7F) + ~~0x80~~128	
iEg = (arrayIn[1] >> 7)	
} else {	
arrayOut[1] = arrayIn[1]	
iEg = 1	
}	
if (arrayIn[2] <= 0) {	
arrayOut[2] = 0	
iEb = 0	
} else if ((arrayIn[2] >> 7) > 1) {	
arrayOut[2] = (arrayIn[2] & 0x7F) + ~~0x80~~128	
iEb = (arrayIn[2] >> 7)	
} else {	
arrayOut[2] = arrayIn[2]	
iEb = 1	
}	
arrayOut[3] = Max(iEr, Max(iEg, iEb))	
if (arrayOut[3] > iEr) {	
iShift = (arrayOut[3] − iEr)	

Table 193 – Pseudocode for function PostScalingF2( )

PostScalingF2(arrayOut[ ], arrayIn[ ]) {	**Reference**
arrayOut[0] = ((2 * arrayOut[0]~~ * 2~~ + 1) >> (iShift + 1))	
}	
if (arrayOut[3] > iEg) {	
iShift = (arrayOut[3] − iEg)	
arrayOut[1] = ((2 * arrayOut[1]~~ * 2~~ + 1) >> (iShift + 1))	
}	
if (arrayOut[3] > iEb) {	
iShift = (arrayOut[3] − iEb)	
arrayOut[2] = ((2 * arrayOut[2]~~ * 2~~ + 1) >> (iShift + 1))	
}	
}	

~~9.10.9~~9.10.8 Clipping and p~~P~~acking s~~S~~tage

9.10.8.1~~9.10.9.1~~ ~~Overview~~General

The ~~overall~~ ClippingAndPackingStage( ) process by which clipping, packing, and windowing are performed~~clipping process so that the sample value is constrained to the appropriate range~~ is specified in Table 194. The clipping ensures that the sample values are constrained to the appropriate range. The packing process packs multiple samples into single variables for some values of OUTPUT_BITDEPTH. The windowing process uses the LEFT_MARGIN, TOP_MARGIN, WIDTH_MINUS1 and HEIGHT_MINUS1 syntax elements to discard the data outside of the image area that is to be output.~~As part of the clipping process, for some values of OUTPUT_BITDEPTH (as in Table 194), this process also involves packing multiple samples into single variables.~~

Table 194 – Pseudocode for function ClippingAndPackingStage( )

ClippingAndPackingStage(~~valXPos, valYPos~~) {	**Reference**
if((OUTPUT_CLR_FMT = = RGB) && ((OUTPUT_BITDEPTH = = BD5) | | (OUTPUT_BITDEPTH = = BD565) | | (OUTPUT_BITDEPTH = = BD10))) /* Packed RGB */~~ /* valXPos and valYPos are meant to indicate the x, y positions of the current sample */~~	
outputArrays = 1	
else if ((OUTPUT_CLR_FMT = = RGB) | | (OUTPUT_CLR_FMT = = YUV444) | | (OUTPUT_CLR_FMT = = YUV422) | | (OUTPUT_CLR_FMT = = YUV420))	
outputArrays = 3	
else if (OUTPUT_CLR_FMT = = RGBE)	
outputArrays = 4	
else	
outputArrays = NumComponents	
for (i = 0; i < outputArrays; i++) {	
if ((i > 0) && (OUTPUT_CLR_FMT = = YUV420)) {	
outputHeight = (HEIGHT_MINUS1 + 1) / 2	
n = TOP_MARGIN / 2	
} else {	
outputHeight = HEIGHT_MINUS1 + 1	
n = TOP_MARGIN	
}	
if ((i > 0) && ((OUTPUT_CLR_FMT = = YUV422) | | (OUTPUT_CLR_FMT = = YUV420))) {	
outputWidth = (WIDTH_MINUS1 + 1) / 2	
m = LEFT_MARGIN / 2	

Table 194 – Pseudocode for function ClippingAndPackingStage( )

ClippingAndPackingStage(~~valXPos, valYPos~~) {	**Reference**
} else {	
outputWidth = WIDTH_MINUS1 + 1	
m = LEFT_MARGIN	
}	
if ((OUTPUT_BITDEPTH = = BD8) | | (OUTPUT_BITDEPTH = = BD16) | | (OUTPUT_BITDEPTH = = BD16S))	
~~for (i = 0; i < NumComponents; i++)~~	
for (y = 0; y < outputHeight; y++)	
for (x = 0; x < outputWidth; x++)	
ImagePlane[i][~~valXPos~~x][~~valYPos~~y] = -ClippingBasic(ImagePlane[i][~~valXPos~~x + m][~~valYPos~~y + n])	9.10.8.2
else if (OUTPUT_BITDEPTH = = BD565)	
for (y = 0; y < outputHeight; y++)	
for (x = 0; x < outputWidth; x++)	
ImagePlane[i][x][y]~~valueReturn~~ = ClipAndPackBD565(ImagePlane[0][x + m~~valXPos~~][~~valYPos~~y + n], ImagePlane[1][~~valXPos~~x + m][~~valYPos~~y + n], ImagePlane[2][~~valXPos~~x + m][~~valYPos~~y + n]) /* This function clips the range of r, g, and b, then the values are packed into a 16-bit value */	9.10.8.3
else if (OUTPUT_BITDEPTH = = BD5)	
for (y = 0; y < outputHeight; y++)	
for (x = 0; x < outputWidth; x++)	
ImagePlane[i][x][y] ~~valueReturn~~ = ClipAndPackBD5(ImagePlane[0][x + m~~valXPos~~][y + n~~valYPos~~], ImagePlane[1][x + m~~valXPos~~][y + n~~valYPos~~], ImagePlane[2][x + m~~valXPos~~][y + n~~valYPos~~])	9.10.8.4
else if (OUTPUT_BITDEPTH = = BD10)~~ {~~	
if (OUTPUT_CLR_FMT = = RGB)	
for (y = 0; y < outputHeight; y++)	
for (x = 0; x < outputWidth; x++)	
ImagePlane[i][x][y] ~~valueReturn~~ = ClipAndPackBD10(ImagePlane[0][x + m~~valXPos~~][y + n~~valYPos~~], ImagePlane[1][x + m~~valXPos~~][y + n~~valYPos~~], ImagePlane[2][x + m~~valXPos~~][y + n~~valYPos~~])	9.10.8.5
else~~ if (OUTPUT_CLR_FMT = = YUV444) {~~	
for (y = 0; y < outputHeight; y++)	
for (x = 0; x < outputWidth; x++)	
ImagePlane[i~~0~~][~~valXPos~~x][~~valYPos~~y] = ClipAndPackBD10(ImagePlane[i~~0~~][x + m~~valXPos~~][y + n~~valYPos~~], 0, 0)	9.10.8.5
~~ImagePlane[1][valXPos][valYPos] = ClipAndPackBD10(ImagePlane[1][valXPos][valYPos] , 0, 0)~~	~~9.10.8.5~~
~~ImagePlane[2][valXPos][valYPos] = ClipAndPackBD10(ImagePlane[2][valXPos][valYPos] , 0, 0)~~	~~9.10.8.5~~
~~} else if ((OUTPUT_CLR_FMT = = YUV422) && ((valXPos % 2) = = 0))~~ ~~/* There are two Y samples for every U and V sample. Input as U Y V Y */~~	
~~ImagePlane[1][valXPos >> 1][valYPos] =~~ ~~ClipAndPackBD10(ImagePlane[1][valXPos >> 1][valYPos], 0, 0)~~	~~9.10.8.5~~
~~ImagePlane[0][valXPos][valYPos] = ClipAndPackBD10(ImagePlane[0][valXPos][valYPos], 0, 0)~~	~~9.10.8.5~~
~~ImagePlane[2][valXPos >> 1][valYPos] =~~ ~~ClipAndPackBD10(ImagePlane[2][valXPos >> 1][valYPos], 0, 0)~~	~~9.10.8.5~~
~~ImagePlane[0][valXPos + 1][valYPos] =~~ ~~ClipAndPackBD10(ImagePlane[0][valXPos + 1][valYPos], 0, 0)~~	~~9.10.8.5~~

Table 194 – Pseudocode for function ClippingAndPackingStage( )

ClippingAndPackingStage(~~valXPos, valYPos~~) {	**Reference**
~~} else if (OUTPUT_CLR_FMT = = YONLY)~~	
~~ImagePlane[0][valXPos][valYPos] = ClipAndPackBD10(ImagePlane[0][valXPos][valYPos], 0, 0)~~	~~9.10.8.5~~
~~}~~ else if ((OUTPUT_BITDEPTH = = BD1WHITE1) | | (OUTPUT_BITDEPTH = = BD1BLACK1) ~~&& ((valXPos % 8) = = 0)~~) ~~/* Since 8 samples are packed into a single byte, only needs to be called for every 8th sample */~~	
for (y = 0; y < outputHeight; y++)	
for (x = 0; x < outputWidth; x+=8) { /* Up to 8 samples are packed into each output byte */	
pNum = Min(outputWidth − x, 8) /* Number of values to pack into current output byte */	
for (p = pNum; m < 8; p++) /* Prevent junk beyond valid image data in array */	
valList[p] = 0 /* Actual value does not matter in this region */	
for (p = 0; p < pNum; p++)	
valList[p] = ImagePlane[i][x + m + p][y + n]	
ImagePlane[i][x >> 3][y]~~valueReturn~~ = ClipAndPackBD1BorW(~~ImagePlane[0][valXPos][valYPos], ImagePlane[0][valXPos + 1][valYPos], ImagePlane[0][valXPos + 2][valYPos], ImagePlane[0][valXPos + 3][valYPos], ImagePlane[0][valXPos + 4][valYPos], ImagePlane[0][valXPos + 5][valYPos], ImagePlane[0][valXPos + 6][valYPos], ImagePlane[0][valXPos + 7][valYPos]~~valList)	9.10.8.6
}	
else /* OUTPUT_BITDEPTH equal to BD16F, BD32F, or BD32S */	
for (y = 0; y < outputHeight; y++)	
for (x = 0; x < outputWidth; x++)	
ImagePlane[i][x][y] = ImagePlane[i][x + m][y + n]	
}	
~~/* No corresponding processing is necessary for OUTPUT_BITDEPTH equal to BD16F, BD32F, or BD32S */~~	~~9.10.7~~
~~return valueReturn~~	
}	

9.10.8.2~~9.10.9.2~~ ClippingBasic( )

The pseudocode function ClippingBasic( ) is specified in Table 195.

Table 195 – Pseudocode for function ClippingBasic( )

ClippingBasic(iSample) {
if (OUTPUT_BITDEPTH = = BD8) {
iLow = 0
iHigh = 255
} else if (OUTPUT_BITDEPTH = = BD16) {
iLow = 0
iHigh = 65535
} else if (OUTPUT_BITDEPTH = = BD16S) {
iLow = −32768
iHigh = 32767
}
~~/* Now clip based on the range iLow to iHigh */~~
iResult = Clip(iSample, iLow, iHigh) /* Clip within the range iLow to iHigh */
return iResult
}

9.10.8.3~~9.10.9.3~~ ClipAndPackBD565( )

• • •

9.10.8.4~~9.10.9.4~~ ClipAndPackBD5( )

• • •

9.10.8.5~~9.10.9.5~~ ClipAndPackBD10( )

• • •

9.10.8.6~~9.10.9.6~~ ClipAndPackBD1BorW( )

The pseudocode function ClipAndPackBD1BorW( ) is specified in Table 199.

Table 199 – Pseudocode for function ClipAndPackBD1BorW( )

ClipAndPackBD1BorW(~~iSample0, iSample1, iSample2, iSample3, iSample4, iSample5, iSample6, iSample7~~valList) {
/* valList[0]~~iSample0~~ holds the value associated with the first sample value in the scan order, and valList[7]~~iSample7~~ holds the value associated with the last sample value in the scan order */
~~iSample~~valList[0] = Clip(~~iSample~~valList[0], 0, 1)
~~iSample~~valList[1] = Clip(~~iSample~~valList[1], 0, 1)
~~iSample~~valList[2] = Clip(~~iSample~~valList[2], 0, 1)
~~iSample~~valList[3] = Clip(~~iSample~~valList[3], 0, 1)
~~iSample~~valList[4] = Clip(~~iSample~~valList[4], 0, 1)
~~iSample~~valList[5] = Clip(~~iSample~~valList[5], 0, 1)
~~iSample~~valList[6] = Clip(~~iSample~~valList[6], 0, 1)
~~iSample~~valList[7] = Clip(~~iSample~~valList[7], 0, 1)
if (OUTPUT_BITDEPTH = = BD1BLACK1)
iResult = (1 − ~~iSample~~valList[7]) + ((1 − ~~iSample~~valList[6]) << 1) + ((1 − ~~iSample~~valList[5]) << 2) + ((1 − ~~iSample~~valList[4]) << 3) + ((1 − ~~iSample~~valList[3]) << 4) + ((1 − ~~iSample~~valList[2]) << 5) + ((1 − ~~iSample~~valList[1]) << 6) + ((1 − ~~iSample~~valList[0]) << 7)
else /* OUTPUT_BITDEPTH = = BD1WHITE1 */
iResult = ~~iSample~~valList[7] + (~~iSample~~valList[6] << 1) + (~~iSample~~valList[5] << 2) + (~~iSample~~valList[4] << 3) + (~~iSample~~valList[3] << 4) + (~~iSample~~valList[2] << 5) + (~~iSample~~valList[1] << 6) + (~~iSample~~valList[0] << 7)
return iResult
}

Annex A

Tag-based file format

(This annex forms an integral part of this Recommendation | International Standard.)

• • •

A.7.3 ELEMENT_TYPE

• • •

The quantity of data associated with each syntax element VALUES_OR_OFFSET, as specified by ELEMENT_TYPE in units of bytes, is specified in the SizeOfElement column of Table A.5. The interpretation of the data elements associated with each value of ELEMENT_TYPE is specified as follows.

– If ELEMENT_TYPE is equal to BYTE, USHORT, or ULONG, each data element is interpreted as an unsigned integer of the specified length in little-endian form.

– Otherwise, if ELEMENT_TYPE is equal to SBYTE, SSHORT, or SLONG, each data element is interpreted as a two's complement signed integer of the specified length in little-endian form.

– Otherwise, if ELEMENT_TYPE is equal to UTF8, each data element is interpreted as a UTF-8 character set code as specified by ISO/IEC 10646 Annex D, and the value of the last data element of the IFD_ENTRY( ) shall be equal to 0 (null). Any such field may contain multiple strings of UTF-8 characters, each terminated with a 0-valued character. The NUM_ELEMENTS for such multi-string payloads is the total number of bytes in all of the associated strings including the 0-valued byte at the end of each such string. Within the associated NUM_ELEMENTS bytes, there shall not be any two consecutive bytes equal to 0.

– Otherwise, if ELEMENT_TYPE is equal to UNDEFINED, the interpretation of the data elements depends on the FIELD_TAG value as follows:

  • If the value of FIELD_TAG is equal to 0xEA1C (PADDING_DATA), the interpretation of each data element is specified in subclause A.7.33~~A.7.32~~.

  • Otherwise, the interpretation of the data elements is not specified by this Specification.

• • •

A.7.18 PIXEL_FORMAT

• • •

~~PIXEL_FORMAT values with the "NC" column of Table A.6 being greater than 1 have an ordering of channels (other than the alpha channel, when present) as follows:~~

~~– If PIXEL_FORMAT is equal to 24bppRGB, 48bppRGB, 48bppRGBFixedPoint, 48bppRGBHalf, 96bppRGBFixedPoint, 64bppRGBA, 64bppRGBAFixedPoint, 64bppRGBAHalf, 128bppRGBAFixedPoint, 128bppRGBAFloat, 32bppPBGRA, 64bppPRGBA, or 128bppPRGBAFloat, the ordering of channels is R, then G, then B.~~

~~– Otherwise, if PIXEL_FORMAT is equal to 64bppRGBFixedPoint, 64bppRGBHalf, 128bppRGBFixedPoint, or 128bppRGBFloat, the ordering of channels is R, then G, then B, then a padding channel. Otherwise, if PIXEL_FORMAT is equal to 16bppBGR555, 16bppBGR565, or 32bppBGR101010, the data for each channel is in the form of packed bit fields within integer values.~~

~~– Otherwise, if PIXEL_FORMAT is equal to 32bppCMYK, 64bppCMYK, 40bppCMYKAlpha, 80bppCMYKAlpha, 32bppCMYKDIRECT, 64bppCMYKDIRECT, 40bppCMYKDIRECTAlpha or 80bppCMYKDIRECTAlpha the ordering of channels is C, then M, then Y, then K.~~

~~– Otherwise, if PIXEL_FORMAT is 12bppYCC420, 16bppYCC422, 20bppYCC422, 32bppYCC422, 24bppYCC444, 30bppYCC444, 48bppYCC444, 48bppYCC444FixedPoint, 20bppYCC420Alpha, 24bppYCC422Alpha, 30bppYCC422Alpha, 48bppYCC422Alpha, 32bppYCC444Alpha, 40bppYCC444Alpha, 64bppYCC444Alpha, 64bppYCC444AlphaFixedPoint, the ordering of channels is Planar Y, then Planar U, then Planar V.~~

~~– Otherwise, the channels are ordered by increasing channel number.~~

The manner of output for the decoded image data produced from a conforming decoder may be determined by the application. However, each PIXEL_FORMAT value specified in Table A.6 has a corresponding defined output format that is specified for reference purposes as the output of a hypothetical reference decoder. This reference output format is in the form of an ordered string of bytes defined as follows:

– If PIXEL_FORMAT is equal to 24bppRGB, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is R, then G, then B, and each sample value for each channel is output as a single byte.

– Otherwise, if PIXEL_FORMAT is equal to 32bppRGBE, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is B, then G, then R, then E, and each sample value for each channel is output as a single byte.

– Otherwise, if PIXEL_FORMAT is equal to 48bppRGB, 48bppRGBFixedPoint, or 48bppRGBHalf, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is R, then G, then B, and each sample value for each channel is output using 16 bits in little-endian form.

– Otherwise, if PIXEL_FORMAT is equal to 96bppRGBFixedPoint, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is R, then G, then B, and each sample value for each channel is output using 32 bits in little-endian form.

– Otherwise, if PIXEL_FORMAT is equal to 64bppRGBFixedPoint or 64bppRGBHalf, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is R, then G, then B, then a padding channel, and each sample value for each channel is output using 16 bits in little-endian form.

– Otherwise, if PIXEL_FORMAT is equal to 128bppRGBFixedPoint or 128bppRGBFloat, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is R, then G, then B, then a padding channel, and each sample value for each channel is output using 32 bits in little-endian form.

– Otherwise, if PIXEL_FORMAT is equal to 64bppRGBA, 64bppRGBAFixedPoint, 64bppRGBAHalf, or 64bppPRGBA, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is R, then G, then B, then Alpha, and each sample value for each channel is output using 16 bits in little-endian form.

– Otherwise, if PIXEL_FORMAT is equal to 128bppRGBAFixedPoint, 128bppRGBAFloat, or 128bppPRGBAFloat, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is R, then G, then B, then Alpha, and each sample value for each channel is output using 32 bits in little-endian form.

– Otherwise, if PIXEL_FORMAT is equal to 24bppBGR, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is B, then G, then R, and each sample value for each channel is output as a single byte.

– Otherwise, if PIXEL_FORMAT is equal to 32bppBGR, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is B, then G, then R, then a padding channel, and each sample value for each channel is output as a single byte.

– Otherwise, if PIXEL_FORMAT is equal to 32bppPBGRA, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is B, then G, then R, then Alpha, and each sample value for each channel is output as a single byte.

– Otherwise, if PIXEL_FORMAT is equal to 16bppBGR565, the output is in the form of packed bit fields within the integer values specified by Table 196, and these integer values are output in raster scan order using 16 bits in little-endian form.

– Otherwise, if PIXEL_FORMAT is equal to 16bppBGR555, the output is in the form of packed bit fields within the integer values specified by Table 197, and these integer values are output in raster scan order using 16 bits in little-endian form.

– Otherwise, if PIXEL_FORMAT is equal to 32bppBGR101010, the output is in the form of packed bit fields within the integer values specified by Table 198, and these integer values are output in raster scan order using 32 bits in little-endian form.

– Otherwise, if PIXEL_FORMAT is equal to BlackWhite, the output is in the form of packed bit fields within the integer values specified by Table 199, and these integer values are output as bytes in raster scan order.

– Otherwise, if PIXEL_FORMAT is equal to 32bppCMYK or 32bppCMYKDIRECT, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is C, then M, then Y, then K, and each sample value for each channel is output as a single byte.

– Otherwise, if PIXEL_FORMAT is equal to 40bppCMYKAlpha or 40bppCMYKDIRECTAlpha, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is C, then M, then Y, then K, then Alpha, and each sample value for each channel is output as a single byte.

– Otherwise, if PIXEL_FORMAT is equal to 64bppCMYK or 64bppCMYKDIRECT, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is C, then M, then Y, then K, and each sample value for each channel is output using 16 bits in little-endian form.

– Otherwise, if PIXEL_FORMAT is equal to 80bppCMYKAlpha or 80bppCMYKDIRECTAlpha, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is C, then M, then Y, then K, then Alpha, and each sample value for each channel is output using 16 bits in little-endian form.

– Otherwise, if PIXEL_FORMAT is equal to 12bppYCC420, 16bppYCC422, or 24bppYCC444, the Y samples are output first in raster scan order, then the U samples are output in raster scan order, then the V samples are output in raster scan order, and each sample value for each channel is output as a single byte.

– Otherwise, if PIXEL_FORMAT is equal to 20bppYCC422, 32bppYCC422, 30bppYCC444, 48bppYCC444, or 48bppYCC444FixedPoint, the Y samples are output first in raster scan order, then the U samples are output in raster scan order, then the V samples are output in raster scan order, and each sample value for each channel is output using 16 bits in little-endian form.

– Otherwise, if PIXEL_FORMAT is equal to 20bppYCC420Alpha or 24bppYCC422Alpha, the Y samples are output first in raster scan order, then the U samples are output in raster scan order, then the V samples are output in raster scan order, then the Alpha samples are output in raster scan order, and each sample value for each channel is output as a single byte.

– Otherwise, if PIXEL_FORMAT is equal to 32bppYCC444Alpha, the Y samples are output first in raster scan order, then the U samples are output in raster scan order, then the V samples are output in raster scan order, then the Alpha samples are output in raster scan order, and each sample value for each channel is output as a single byte.

– Otherwise, if PIXEL_FORMAT is equal to 30bppYCC422Alpha, 48bppYCC422Alpha, 40bppYCC444Alpha, 64bppYCC444Alpha, or 64bppYCC444AlphaFixedPoint, the Y samples are output first in raster scan order, then the U samples are output in raster scan order, then the V samples are output in raster scan order, then the Alpha samples are output in raster scan order, and each sample value for each channel is output using 16 bits in little-endian form.

– Otherwise, if PIXEL_FORMAT is equal to 24bpp3Channels, 32bpp4Channels, 40bpp5Channels, 48bpp6Channels, 56bpp7Channels, or 64bpp8Channels, the decoded channels are ordered by increasing channel number, and the samples of each channel are output in raster scan order, and each sample value for each channel is output as a single byte.

– Otherwise, if PIXEL_FORMAT is equal to 48bpp3Channels, 64bpp4Channels, 80bpp5Channels, 96bpp6Channels, 112bpp7Channels, or 128bpp8Channels, the decoded channels are ordered by increasing channel number, and the samples of each channel are output in raster scan order, and each sample value for each channel is output using 16 bits in little-endian form.

– Otherwise, if PIXEL_FORMAT is equal to 32bpp3ChannelsAlpha, 40bpp4ChannelsAlpha, 48bpp5ChannelsAlpha, 56bpp6ChannelsAlpha, 64bpp7ChannelsAlpha, or 72bpp8ChannelsAlpha, the decoded channels are ordered by increasing channel number with the Alpha channel being considered as having the highest channel number and the samples of each channel are output in raster scan order, and each sample value for each channel is output as a single byte.

– Otherwise, if PIXEL_FORMAT is equal to 64bpp3ChannelsAlpha, 80bpp4ChannelsAlpha, 96bpp5ChannelsAlpha, 112bpp6ChannelsAlpha, 128bpp7ChannelsAlpha, or 144bpp8ChannelsAlpha, the decoded channels are ordered by increasing channel number and the last channel is considered the Alpha channel, and the samples of each channel are output in raster scan order, and each sample value for each channel is output using 16 bits in little-endian form.

– Otherwise, if PIXEL_FORMAT is equal to 8bppGray, there is only one decoded channel, and the samples of that channel are output in raster scan order, and each sample value is output as a single byte.

- Otherwise, if PIXEL_FORMAT is equal to 16bppGray, 16bppGrayFixedPoint, or 16bppGrayHalf, , there is only one decoded channel, and the samples of that channel are output in raster scan order, and each sample value is output using 16 bits in little-endian form.
- Otherwise (PIXEL_FORMAT is equal to 32bppGrayFixedPoint or 32bppGrayFloat), there is only one decoded channel, and the samples of that channel are output in raster scan order, and each sample value is output using 32 bits in little-endian form.

NOTE 3 – Particular care should be taken in regard to the interpretation of this defined output format for PIXEL_FORMAT values with mnemonic names that include "RGB", "BGR", or "CMYK" as part of the mnemonic name due to the specification of little-endian form for packed bit fields. In particular, the first byte of each packed bit field for the PIXEL_FORMAT values 16bppBGR565, 16bppBGR555, and 32bppBGR101010 actually contains data for the R channel due to the use of little-endian form, which may be somewhat counter-intuitive. Similarly, if the four bytes that contain the decoded channel samples for the PIXEL_FORMAT values 32bppRGBE, 32bppBGR, 32bppPBGRA, 32bppCMYK, and 32bppCMYKDIRECT are written or read as 32-bit integers in little-endian form rather than as ordered strings of four bytes in which each byte represents a sample value, the ordering of the channels will be swapped relative to the ordering described herein.

This defined format is specified for reference purposes and may be used by some decoders as an interface format for the output of decoded pictures. However, the use of this defined format is not a requirement for conformance to this Specification.

Interpretation of alpha channel information (when present) is considered pre-multiplied or not pre-multiplied as follows:

- If PIXEL_FORMAT is equal to 32bppPBGRA, 64bppPRGBA, or 128bppPRGBAFloat, the channels other than the alpha channel are considered to be in pre-multiplied form in relation to the alpha channel.
- Otherwise, the channels other than the alpha channel are considered not to be in pre-multiplied form in relation to the alpha channel. In these cases, the value of PREMULTIPLIED_ALPHA_FLAG in the associated IMAGE_HEADER( ) of the coded image that contains the alpha channel shall be equal to 0.

NOTE 4~~3~~ – The designation of an alpha channel as pre-multiplied indicates that the decoded sample values do not require multiplication by the alpha channel values when performing compositing (as any necessary such multiplication process was performed as a pre-processing step prior to encoding).

JPEG XR supports three types of numerical encoding: unsigned integer, fixed point, and floating point, each at a variety of bit depths.

PIXEL_FORMAT values having the "Num" column of Table A.6 indicating "UINT" are unsigned integer formats as follows.

- If BPC is equal to BD8, the minimum value is 0 and the maximum value is 255, providing 256 unique values.
- Otherwise (BPC is equal to BD16), the minimum value is 0 and the maximum value is 65 535, providing 65 536 unique values.

In ~~all~~ unsigned integer cases, a value of zero ordinarily represents the minimum level or the encoding black for the specific channel and the maximum possible value represents the maximum value for that channel. When all viewable channels for a pixel format are at their maximum numerical value, this corresponds to the brightest representable color, or the encoding white. Exceptions to this general rule include the following unsigned integer cases:

- ~~The only exception is when the PIXEL_FORMAT is BlackWhite and~~When OUTPUT_BITDEPTH is BD1BLACK1, a value of zero represents the maximum level or white and a value of one represents the minimum level or black.
- When the OUTPUT_CLR_FMT is YUV444, YUV422, or YUV420, the U and V components are interpreted as color difference representations that are offset by a constant value, such that the middle value of the range of possible integer values is used for the representation of both the encoding black and the encoding white.
- When the OUTPUT_CLR_FMT is CMYK or CMYKDIRECT, the K component is interpreted as a degree of proximity toward black, such that the maximum value for the K component is used for the encoding black and the minimum value for the K component is used for the encoding white.
- When the OUTPUT_CLR_FMT is NCOMPONENT and the number of encoded components other than the Alpha channel (when present) is greater than 3, in the absence of other information to assist in the color channel interpretation, the fourth channel should be interpreted as a K channel indicating a degree of proximity toward black as in the cases when the OUTPUT_CLR_FMT is CMYK or CMYKDIRECT (see also Annex C).

PIXEL_FORMAT values having mnemonic names that end in "FixedPoint" specify fixed point representations.

NOTE 5~~4~~ – A fixed point numerical representation is not commonly supported in ~~today's~~ prior image file formats. It is ~~being introduced~~supported in JPEG XR as ~~an optimal format~~a way to encode an extended range of numerical values more directly while ~~still~~ retaining ~~all~~ the performance advantages of integer processing.

JPEG XR supports fixed point numerical encoding for 16-bit and 32-bit signed values. In this Specification,~~This document will also use~~ the abbreviation SINT is used to refer to signed integer~~,~~ or fixed point values.

Fixed point values having BPC equal to 16 or 32 are interpreted as follows.

– 16-bit Fixed Point, a format referred to as s2.13: The 16 bits that make up an individual value are interpreted as a sign bit, two integer bits and thirteen fractional bits. Using this interpretation, a numerical range of −4.0 to +3.999… can be represented, with the value of 1.0 represented by the signed integer value 8 192 (0x2000).

– 32-bit Fixed Point, a format referred to as s7.24: The 32 bits that make up an individual value are interpreted as a sign bit, seven integer bits and twenty-four fractional bits. Using this interpretation, a numerical range of −128.0 to +127.999… can be represented, with the value of 1.0 represented by the signed integer value 16 777 216 (0x01000000).

NOTE 6~~5~~ – JPEG XR does not enable fully lossless compression for 32-bit data in general. The encoding and decoding algorithms use 32-bit computations, and some dynamic range is lost to necessary headroom for signal processing calculations such as overlap and core transform computations~~so some precision is lost during these calculations~~. A minimum of 22 bits and typically 24 bits or more precision is retained through the end-to-end encoding and decoding process.

PIXEL_FORMAT values having the "Num" column of Table A.6 indicating "Float" (floating point values) are floating point formats.

JPEG XR supports floating point numerical encoding for 16-bit and 32-bit depths. A packed bit RGB float format is also supported in the form of the RGBE format.

The format of the floating point values is based on the "BPC" column of Table A.6 as follows:

– BPC equal to BD16F indicates the "s5e10" format. The 16 bits are formatted in accordance with the HALF floating point format, with 1 sign bit, 5 exponent bits and 10 normalized mantissa bits.

– BPC equal to BD8 with PIXEL_FORMAT equal to 32bppRGBE indicates a packed bit representation such that three 16-bit floating point values are represented using four bytes. The bytes include unsigned 8-bit mantissas for the red, green and blue channels, plus a shared 8-bit exponent.

NOTE 7~~6~~ – When the exponent for each channel is the same, this representation is a more compact method to encode image data as compared to the representation where BPC is equal to 16 and PIXEL_FORMAT is not equal to 32bppRGBE.

– BPC equal to 32F indicates the "s8e23" format. The numerical value is encoded as a 4-byte IEC 60559 floating point number in little-endian form. This encoding uses 1 sign bit, 8 exponent bits and 23 normalized mantissa bits.

NOTE 8~~7~~ – JPEG XR does not enable fully lossless compression for 32-bit data in general. The encoding and decoding algorithms use 32-bit computations, so some precision is lost during these calculations. A minimum of 22 bits and typically 24 bits or more precision is retained through the end-to-end encoding and decoding process.

NOTE 9~~8~~ – For some applications, the bounded range provided by an integer or fixed point representation may not be sufficient. Therefore, JPEG XR also supports a floating point numerical representation. Floating point formatting of image data will typically not compress as efficiently, but floating point capability provides a dramatically larger numerical range while maintaining high precision for small absolute values.

• • •

A.7.20 IMAGE_TYPE

• • •

All images in a sequence of pages shall have the same dimensions, as follows:

– For all images with IMAGE_TYPE & 1 equal to 0 and (IMAGE_TYPE >> 1) & 1 ~~is~~ equal to 1, the value of IMAGE_HEIGHT and IMAGE_WIDTH shall be equal to the value of IMAGE_HEIGHT and IMAGE_WIDTH for any other image in the file with IMAGE_TYPE & 1 equal to 0 and (IMAGE_TYPE >> 1) & 1 ~~is~~ equal to 1.

– For all images with IMAGE_TYPE & 1 equal to 1 and (IMAGE_TYPE >> 1) & 1 ~~is~~ equal to 1, the value of IMAGE_HEIGHT and IMAGE_WIDTH shall be equal to the value of IMAGE_HEIGHT and IMAGE_WIDTH for any other image in the file with IMAGE_TYPE & 1 equal to 1~~0~~ and (IMAGE_TYPE >> 1) & 1 ~~is~~ equal to 1.

When a file contains only a single image, IMAGE_TYPE should not be present.

• • •

A.7.29 ALPHA_OFFSET

ALPHA_OFFSET (when present) specifies the byte position, relative to the beginning of the file, of the start of a CODED_IMAGE( ) syntax structure for the alpha plane of the image in the IMAGE_FILE_DIRECTORY( ). When PIXEL_FORMAT does not indicate the presence of an alpha channel, ALPHA_OFFSET shall not be present.

When PIXEL_FORMAT indicates the presence of an alpha channel, the form of the encoding of the alpha channel is specified as follows.

– If ALPHA_OFFSET is present, the alpha channel is present in a separate CODED_IMAGE( ) syntax structure of the same IMAGE_FILE_DIRECTORY( ) syntax structure from the CODED_IMAGE( ) for the primary image plane. In this case, the syntax element ALPHA_IMAGE_PLANE_FLAG for the CODED_IMAGE( ) syntax structure at the position specified by IMAGE_OFFSET (subclause A.7.27) shall be equal to zero and the syntax element ALPHA_IMAGE_PLANE_FLAG for the CODED_IMAGE( ) syntax structure at the position specified by ALPHA_OFFSET shall be equal to 1. In this case, the alpha image plane is a separate alpha image plane.

– Otherwise, the alpha channel is present within the same CODED_IMAGE( ) syntax structure as the primary image plane and the syntax element ALPHA_IMAGE_PLANE_FLAG shall be equal to one. In this case, the alpha image plane is an interleaved alpha image plane.

When ALPHA_BYTE_COUNT is present, ALPHA_OFFSET shall also be present.

• • •

Annex B

Profiles and Levels

(This annex forms an integral part of this Recommendation | International Standard.)

• • •

B.2.1 Sub-Baseline profile

The Sub-Baseline profile is defined as the set of coded image syntax that has LONG_WORD_FLAG set equal to FALSE, has OVERLAP_MODE set equal to either 0 or 1, and that is consistent with the use of the following PIXEL_FORMAT mnemonic values (specified in Table A.6): 24bppRGB, 24bppBGR, 32bppBGR, 8bppGray, BlackWhite, 16bppBGR555, 16bppBGR565, or 32bppBGR101010.

NOTE 1 – Although this constraint is specified in terms of concepts defined in Annex A, it also applies when the file format specified in Annex A is not in use. Concepts specified in Annex A are used in this annex only as a reference to establish the set of features supported by the profiles specified in this annex.

NOTE 2 – Encoders may need to set SCALED_FLAG equal to 0 for 32bppBGR~~RGB~~101010 encoding to avoid exceeding the dynamic range constraints imposed when LONG_WORD_FLAG is equal to FALSE.

• • •

B.3 Levels

• • •

Table B.2 – Pseudocode for function ImageBufferBytes(valNC)

ImageBufferBytes(valNC) {	**Reference**
numBytes = 0	
if (OUTPUT_BITDEPTH = = BD8)	

numBytes ~~returnVal~~ = valNC * ExtendedWidth[0] * ExtendedHeight[0]	
else if ((OUTPUT_BITDEPTH = = BD16) | | (OUTPUT_BITDEPTH = = BD16S) | | (OUTPUT_BITDEPTH = = BD16F))	
numBytes = 2 * valNC * ExtendedWidth[0] * ExtendedHeight[0]	
else if ((OUTPUT_BITDEPTH = = BD32S) | | (OUTPUT_BITDEPTH = = BD32F))	
numBytes = 4 * valNC * ExtendedWidth[0] * ExtendedHeight[0]	
else if ((OUTPUT_BITDEPTH = = BD1WHITE1) | | (OUTPUT_BITDEPTH = = BD1BLACK1))	
numBytes = ExtendedWidth[0] * ExtendedHeight[0] / 8	
else if ((OUTPUT_BITDEPTH = = BD5) | | (OUTPUT_BITDEPTH = = BD565))	
numBytes = 2 * ExtendedWidth[0] * ExtendedHeight[0]	
else /* In the remaining case, OUTPUT_BITDEPTH is equal to BD10 */	
if (OUTPUT_CLR_FMT = = RGB)	
numBytes = 4 * ExtendedWidth[0] * ExtendedHeight[0]	
else	
numBytes = 2 * valNC * ExtendedWidth[0] * ExtendedHeight[0]	
return numBytes	
}	

• • •

Annex C

Color imagery representation and color management

(This annex does not form an integral part of this Recommendation | International Standard.)

C.1 Background information

While it might be theoretically possible to agree on one method for assigning specific numerical values to real world colors, doing so is not practical. Since any specific device has its own limited range for color reproduction, the device's range may be a small portion of the agreed-upon universal color range. As a result, such an approach is an extremely inefficient use of the available numerical values, especially when using only 8 bits (or 256 unique values) per channel.

To represent pixel values as efficiently as possible, devices use a numeric encoding optimized for their own range of possible colors or gamut. In addition, a color profile is often provided that describes the numeric encoding for the specific device relative to some pre-defined reference standard. For example, a color profile may include a specification of a non-linear transformation from the range of integer values to the luminance (or radiance) of the color values as reproduced. This non-linear transformation is called the color component transfer function (CCTF). In some cases, it may be possible to specify the CCTF using a power function with a single numerical value specifying the exponent or "gamma". The term "gamma" is also sometimes used more informally to refer to any CCTF, although this terminology is not strictly correct. Color profiles deal with all aspects of the color interpretation of digital values, and in addition to specifying the CCTF, specify the relations between device values and the profile connection space (PCS) coordinates, which in the case of ICC profiles are based on the CIE 1931 2° standard observer specified in ISO 11664-1~~/CIE 10527~~. Color profiles may provide additional information about the encoded image, such as the viewing conditions or image state. ICC profiles also provide a transform to a standard reference medium for interoperability. Color profiles make it possible to convert image data between different color encodings, thereby controlling the capture and production of colors using a variety of devices. This science is known as "color management".

• • •

C.2 Color interpretation in the JPEG XR context

• • •

ICC profile data as specified by ICC.1:2001-04 (ICC version 2.4.0.0) or ISO 15076-1 (ICC version 4.2.0.0) can provide at least one unambiguous interpretation of the colors associated with encoded image data. Multiple interpretations may be provided using different rendering intents. The file format specified in Annex A supports the inclusion of ICC profile data for imagery encoded according to this Specification by allowing the use of the field tag value 0x8773, which is specified in ICC.1:2001-04 and ISO 150~~05~~76-1 for use in tag-based file formats for embedding of ICC profile data.

Unless otherwise selected by the user, device, or software receiving the image data, the perceptual rendering intent (0xA2B0 in ICC.1:2001-04 or ISO 15076-1) transform should be used for interpretation of the color imagery.

• • •

Fixed point or floating point RGB data, as may be specified using the PIXEL_FORMAT tag defined in subclause A.7.18, should use the scRGB color space defined in IEC 61966-2-2, but without the offset and scaling that are applied to produce scRGB unsigned integer values (i.e the floating point scRGB is only a matrix transform from the CIE 1931 XYZ color space specified in ISO 11664-1~~/CIE 10527~~). scRGB is a linear CCTF (gamma = 1.0) color encoding that uses the same color primaries and white point chromaticity as sRGB, but has a different CCTF and image state. The scRGB black point is the numerical value 0.0 and corresponds to zero photons in the scene. The scRGB perfect diffuse white point is specified by all three color channels set to a value of 1.0.

• • •

Annex D

Encoder processing

(This annex does not form an integral part of this Recommendation | International Standard.)

• • •

D.3 Color conversion

D.3.1 General

The encoder uses a reversible color format conversion to convert between the OUTPUT_C~~OLOR_FORMAT~~LR_FMT and INTERNAL_C~~OLOR_FORMAT~~LR_FMT.

RGB to YUV444 color format conversion is performed using FwdColorFmtConvert1( ) that is specified in subclause D.3.2. In order to convert from RGB to YUV422 or YUV420, downsampling must be performed after color format conversion using FwdColorFmtConvert1( ). RGB to YONLY conversions may be performed using FwdColorFmtConvert1( ) by discarding the U and V components on the encoder side. CMYK to YUVK color format conversion is performed using FwdColorFmtConvert2( ) that is specified in subclause D.3.3. CMYKDIRECT to YUVK color format conversion is performed using FwdColorFmtConvert3( ) that is specified in subclause D.3.4.

Prior to color conversion, a bias is subtracted from all values, to zero center their range. The amount of the bias is determined by the source bit depth, and is exactly as specified in subclause 9.10.5. When the scaling mode is used, the color values are shifted left prior to encoder color conversion.

D.3.2 FwdColorFmtConvert1( )

The function FwdColorFmtConvert1( ) implements the following operations to convert from RGB to YUV444 in Table D.6.

Table D.6 – Pseudocode for function FwdColorFmtConvert1( )

FwdColorFmtConvert1(arrayIn[ ], arrayOut[ ]) {	**Reference**
/* arrayIn[ ] = {R, G, B} */	
/* arrayOut[ ] = {Y, U, V} */	
arrayOut[2] = arrayIn[2] − arrayIn[0] /* V = B − R */	
tempT = arrayIn[0] − arrayIn[1] + Ceiling(arrayOut[2] ~~/~~÷ 2) ______-/* t = R − G + Ceiling(V ~~/~~÷ 2) */	
arrayOut[0] = arrayIn[1] + Floor(tempT ~~/~~÷ 2) /* Y = G + Floor(t ~~/~~÷ 2) */	

Table D.6 – Pseudocode for function FwdColorFmtConvert1( )

arrayOut[1] = −tempT /*U = −t */	
}	

D.3.3 FwdColorFmtConvert2( )

The function FwdColorConvert2( ) implements the following operations to convert from CMYK to YUVK in Table D.7.

Table D.7 – Pseudocode for function FwdColorFmtConvert2( )

FwdColorFmtConvert2(arrayIn[ ], arrayOut[ ]) {	**Reference**
/* arrayIn[ ] = {c, m, y, k} */	
/* arrayOut[ ] = {Y, U, V, K} */	
arrayOut[2] = arrayIn[0] – arrayIn[2] /* c − y */	
arrayOut[1] = arrayIn[0] − arrayIn[1] + Floor(arrayOut[2] ~~/~~÷ 2) /* c − m − Floor(V ~~/~~÷ 2) */	
arrayOut[0] = arrayIn[3] − arrayIn[1] + Floor(arrayOut[1] ~~/~~÷ 2) /* k − m − Floor(U ~~/~~÷ 2) */	
arrayOut[3] = arrayIn[3] − Floor(arrayOut[0] ~~/~~÷ 2) /* K = k − Floor(Y ~~/~~÷ 2) */	
}	

• • •

Bibliography

• • •

[7] Adobe TIFF Revision 6.0 (1992), Adobe Systems Incorporated (available at http://partners.adobe.com/public/developer/tiff/index.html~~http://www.adobe.com/Support/TechNotes.html and ftp://ftp.adobe.com/pub/adobe/DeveloperSupport/TechNotes/PDFfiles~~)

• • •

SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series D	General tariff principles
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Cable networks and transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Construction, installation and protection of cables and other elements of outside plant
Series M	Telecommunication management, including TMN and network maintenance
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Terminals and subjective and objective assessment methods
Series Q	Switching and signalling
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	**Terminals for telematic services**
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	Data networks, open system communications and security
Series Y	Global information infrastructure, Internet protocol aspects and next-generation networks
Series Z	Languages and general software aspects for telecommunication systems

Printed in Switzerland
Geneva, 2012