|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Fond-Rec_e | | **International Telecommunication Union** | | |
|  | |  | | |
| **ITU-T** | **T.832** | |
| TELECOMMUNICATION STANDARDIZATION SECTOR OF ITU | | (08/2016) |
|  | SERIES T: TERMINALS FOR TELEMATIC SERVICES  Still-image compression | JPEG XR | | | |
|  | **Information technology – JPEG XR image coding system – Image coding specification** | | | |
|  | Recommendation ITU‑T T.832 | | | |



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 |

|  |
| --- |
| Summary  Recommendation ITU-T T.832 specifies a coded image format, referred to as JPEG XR, which is designed primarily for the 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 the storage and interchange of such coded images. Annex B specifies profiles and levels that determine conformance requirements for classes of encoders and decoders. Aspects of colour imagery representations and colour management are discussed in Annex C. The typical expected encoding process is described in Annex D.  The first edition of Recommendation ITU-T T.832 was approved in 03/2009.  Corrigendum 1 (approved 12/2009) contained a set of various minor corrections.  Corrigendum 2 (approved 05/2011) contained additional minor corrections.  The second edition (approved 01/2012) included additional minor corrections.  This third edition (approved 08/2016) adds a media type specification for use in various Internet protocols to identify images encoded according to the tag-based format specified in Annex A of this Recommendation.  This 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 | Unique ID[[1]](#footnote-1)\* | | 1.0 | ITU-T T.832 | 2009-03-16 | 16 | [11.1002/1000/9721](http://handle.itu.int/11.1002/1000/9721) | | 1.1 | ITU-T T.832 (2009) Cor.1 | 2009-12-14 | 16 | [11.1002/1000/10647](http://handle.itu.int/11.1002/1000/10647) | | 1.2 | ITU-T T.832 (2009) Cor. 2 | 2011-05-14 | 16 | [11.1002/1000/11317](http://handle.itu.int/11.1002/1000/11317) | | 2.0 | ITU-T T.832 | 2012-01-13 | 16 | [11.1002/1000/11471](http://handle.itu.int/11.1002/1000/11471) | | 3.0 | ITU-T T.832 | 2016-08-13 | 16 | [11.1002/1000/12959](http://handle.itu.int/11.1002/1000/12959) | |

FOREWORD

The International Telecommunication Union (ITU) is the United Nations specialized agency in the field of tele­com­mu­ni­ca­tions, information and communication technologies (ICTs). The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of ITU. ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Assembly (WTSA), which meets every four years, establishes the topics for study by the ITU‑T study groups which, in turn, produce Recommendations on these topics.

The approval of ITU-T Recommendations is covered by the procedure laid down in WTSA Resolution 1.

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

NOTE

In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized operating agency.

Compliance with this Recommendation is voluntary. However, the Recommendation may contain certain mandatory provisions (to ensure, e.g., interoperability or applicability) and compliance with the Recommendation is achieved when all of these mandatory provisions are met. The words "shall" or some other obligatory language such as "must" and the negative equivalents are used to express requirements. The use of such words does not suggest that compliance with the Recommendation is required of any party.

INTELLECTUAL PROPERTY RIGHTS

ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.

As of the date of approval of this Recommendation, ITU had received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementers are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database at <http://www.itu.int/ITU-T/ipr/>.

© ITU 2017

All rights reserved. No part of this publication may be reproduced, by any means whatsoever, without the prior written permission of ITU.

**CONTENTS**

Page

[1 Scope 1](#_Toc462298538)

[2 Normative references 1](#_Toc462298539)

[3 Definitions 1](#_Toc462298540)

[4 Abbreviations 4](#_Toc462298541)

[5 Conventions 5](#_Toc462298542)

[5.1 Conformance language 5](#_Toc462298543)

[5.2 Operators 5](#_Toc462298544)

[5.2.1 Arithmetic operators 5](#_Toc462298545)

[5.2.2 Logical operators 6](#_Toc462298546)

[5.2.3 Relational operators 6](#_Toc462298547)

[5.2.4 Bit-wise operators 6](#_Toc462298548)

[5.2.5 Assignment operators 7](#_Toc462298549)

[5.2.6 Precedence order of operators 7](#_Toc462298550)

[5.2.7 Pseudocode operations 8](#_Toc462298551)

[5.2.8 Mathematical functions 9](#_Toc462298552)

[5.3 Syntax and semantics notation 12](#_Toc462298553)

[5.3.1 Method of specifying syntax in tabular form 12](#_Toc462298554)

[5.3.2 SYNTAX\_ELEMENT\_EXAMPLE semantics 13](#_Toc462298555)

[5.3.3 Syntax functions 13](#_Toc462298556)

[5.4 Formatting conventions 13](#_Toc462298557)

[5.4.1 Variable and array naming conventions 13](#_Toc462298558)

[5.4.2 Data structure naming conventions 14](#_Toc462298559)

[5.4.3 Syntax element naming conventions 14](#_Toc462298560)

[5.4.4 Syntax structure naming conventions 14](#_Toc462298561)

[5.4.5 Naming conventions for mnemonic constants 14](#_Toc462298562)

[5.4.6 Naming conventions for numerical values 14](#_Toc462298563)

[5.4.7 Array dimensions convention 14](#_Toc462298564)

[5.5 Global variables 15](#_Toc462298565)

[5.5.1 Image variables 15](#_Toc462298566)

[5.5.2 Image plane variables 15](#_Toc462298567)

[5.5.3 Tile variables 16](#_Toc462298568)

[5.5.4 Macroblock variables 16](#_Toc462298569)

[5.5.5 Data structures for adaptive VLC table selection 17](#_Toc462298570)

[5.5.6 Adaptive coefficient normalization data structure instances 19](#_Toc462298571)

[5.5.7 Adaptive CBPHP prediction data structure instance 19](#_Toc462298572)

[5.5.8 Adaptive count CBPLP variables 20](#_Toc462298573)

[5.6 Adaptive VLC deltaDisc tables 20](#_Toc462298574)

[5.7 Adaptive inverse scanning tables 20](#_Toc462298575)

[6 Image and codestream structures 20](#_Toc462298576)

[6.1 General 20](#_Toc462298577)

[6.2 Image planes and component arrays 21](#_Toc462298578)

[6.3 Image windowing 21](#_Toc462298579)

[6.4 Image partitioning 21](#_Toc462298580)

[6.5 Transform coefficients and frequency bands 22](#_Toc462298581)

[6.6 Codestream structure 23](#_Toc462298582)

[6.7 Precision and word length 24](#_Toc462298583)

[7 Overview of decoder 24](#_Toc462298584)

[7.1 General 24](#_Toc462298585)

[7.2 Overview of parsing process 25](#_Toc462298586)

[7.2.1 Overview of image layer codestream parsing 25](#_Toc462298587)

[7.2.2 Overview of tile layer codestream parsing 25](#_Toc462298588)

[7.2.3 Overview of macroblock layer codestream parsing 25](#_Toc462298589)

[7.3 Overview of the decoding process 25](#_Toc462298590)

[7.3.1 Overview of coefficient mapping 25](#_Toc462298591)

[7.3.2 Overview of coefficient prediction 26](#_Toc462298592)

[7.3.3 Overview of dequantization 26](#_Toc462298593)

[7.3.4 Overview of sample reconstruction 26](#_Toc462298594)

[7.3.5 Overview of output formatting 27](#_Toc462298595)

[8 Syntax, semantics, and parsing process 27](#_Toc462298596)

[8.1 General 27](#_Toc462298597)

[8.2 CODED\_IMAGE( ) 28](#_Toc462298598)

[8.2.1 Syntax structure 28](#_Toc462298599)

[8.2.2 SubsequentBytes 28](#_Toc462298600)

[8.2.3 RESERVED\_A\_BYTE 28](#_Toc462298601)

[8.2.4 VLW\_ESC( ) 29](#_Toc462298602)

[8.3 IMAGE\_HEADER( ) 30](#_Toc462298603)

[8.3.1 Syntax structure 30](#_Toc462298604)

[8.3.2 GDI\_SIGNATURE 30](#_Toc462298605)

[8.3.3 RESERVED\_B 31](#_Toc462298606)

[8.3.4 HARD\_TILING\_FLAG 31](#_Toc462298607)

[8.3.5 RESERVED\_C 31](#_Toc462298608)

[8.3.6 TILING\_FLAG 31](#_Toc462298609)

[8.3.7 FREQUENCY\_MODE\_CODESTREAM\_FLAG 31](#_Toc462298610)

[8.3.8 SPATIAL\_XFRM\_SUBORDINATE 31](#_Toc462298611)

[8.3.9 INDEX\_TABLE\_PRESENT\_FLAG 32](#_Toc462298612)

[8.3.10 OVERLAP\_MODE 32](#_Toc462298613)

[8.3.11 SHORT\_HEADER\_FLAG 32](#_Toc462298614)

[8.3.12 LONG\_WORD\_FLAG 32](#_Toc462298615)

[8.3.13 WINDOWING\_FLAG 33](#_Toc462298616)

[8.3.14 TRIM\_FLEXBITS\_FLAG 33](#_Toc462298617)

[8.3.15 RESERVED\_D 33](#_Toc462298618)

[8.3.16 RED\_BLUE\_NOT\_SWAPPED\_FLAG 33](#_Toc462298619)

[8.3.17 PREMULTIPLIED\_ALPHA\_FLAG 33](#_Toc462298620)

[8.3.18 ALPHA\_IMAGE\_PLANE\_FLAG 34](#_Toc462298621)

[8.3.19 OUTPUT\_CLR\_FMT 34](#_Toc462298622)

[8.3.20 OUTPUT\_BITDEPTH 35](#_Toc462298623)

[8.3.21 WIDTH\_MINUS1 35](#_Toc462298624)

[8.3.22 HEIGHT\_MINUS1 35](#_Toc462298625)

[8.3.23 NUM\_VER\_TILES\_MINUS1 36](#_Toc462298626)

[8.3.24 NUM\_HOR\_TILES\_MINUS1 36](#_Toc462298627)

[8.3.25 TILE\_WIDTH\_IN\_MB[n] 36](#_Toc462298628)

[8.3.26 TILE\_HEIGHT\_IN\_MB[n] 36](#_Toc462298629)

[8.3.27 TOP\_MARGIN 37](#_Toc462298630)

[8.3.28 LEFT\_MARGIN 37](#_Toc462298631)

[8.3.29 BOTTOM\_MARGIN 37](#_Toc462298632)

[8.3.30 RIGHT\_MARGIN 37](#_Toc462298633)

[8.4 IMAGE\_PLANE\_HEADER( ) 37](#_Toc462298634)

[8.4.1 Syntax structure 37](#_Toc462298635)

[8.4.2 INTERNAL\_CLR\_FMT 38](#_Toc462298636)

[8.4.3 SCALED\_FLAG 39](#_Toc462298637)

[8.4.4 BANDS\_PRESENT 39](#_Toc462298638)

[8.4.5 RESERVED\_E\_BIT 39](#_Toc462298639)

[8.4.6 CHROMA\_CENTERING\_X 40](#_Toc462298640)

[8.4.7 RESERVED\_F 40](#_Toc462298641)

[8.4.8 RESERVED\_G\_BIT 40](#_Toc462298642)

[8.4.9 CHROMA\_CENTERING\_Y 40](#_Toc462298643)

[8.4.10 RESERVED\_H 41](#_Toc462298644)

[8.4.11 NUM\_COMPONENTS\_MINUS1 41](#_Toc462298645)

[8.4.12 NUM\_COMPONENTS\_EXTENDED\_MINUS16 41](#_Toc462298646)

[8.4.13 SHIFT\_BITS 41](#_Toc462298647)

[8.4.14 LEN\_MANTISSA 41](#_Toc462298648)

[8.4.15 EXP\_BIAS 41](#_Toc462298649)

[8.4.16 DC\_IMAGE\_PLANE\_UNIFORM\_FLAG 41](#_Toc462298650)

[8.4.17 RESERVED\_I\_BIT 42](#_Toc462298651)

[8.4.18 LP\_IMAGE\_PLANE\_UNIFORM\_FLAG 42](#_Toc462298652)

[8.4.19 RESERVED\_J\_BIT 42](#_Toc462298653)

[8.4.20 HP\_IMAGE\_PLANE\_UNIFORM\_FLAG 42](#_Toc462298654)

[8.4.21 BYTE\_ALIGNMENT\_BIT 42](#_Toc462298655)

[8.4.22 DC\_QP( ) 42](#_Toc462298656)

[8.4.23 LP\_QP( ) 43](#_Toc462298657)

[8.4.24 HP\_QP( ) 44](#_Toc462298658)

[8.5 INDEX\_TABLE\_TILES( ) 45](#_Toc462298659)

[8.5.1 Syntax structure 45](#_Toc462298660)

[8.5.2 INDEX\_TABLE\_STARTCODE 45](#_Toc462298661)

[8.5.3 IndexOffsetTile[n] 45](#_Toc462298662)

[8.6 PROFILE\_LEVEL\_INFO( ) 45](#_Toc462298663)

[8.6.1 Syntax structure 45](#_Toc462298664)

[8.6.2 PROFILE\_IDC 46](#_Toc462298665)

[8.6.3 LEVEL\_IDC 46](#_Toc462298666)

[8.6.4 RESERVED\_L 46](#_Toc462298667)

[8.6.5 LAST\_FLAG 46](#_Toc462298668)

[8.7 CODED\_TILES( ) 46](#_Toc462298669)

[8.7.1 Syntax structure 46](#_Toc462298670)

[8.7.2 TILE\_SPATIAL( ) 47](#_Toc462298671)

[8.7.3 TILE\_DC( ) 48](#_Toc462298672)

[8.7.4 TILE\_HEADER\_DC( ) 48](#_Toc462298673)

[8.7.5 TILE\_LOWPASS( ) 49](#_Toc462298674)

[8.7.6 TILE\_HEADER\_LOWPASS( ) 49](#_Toc462298675)

[8.7.7 TILE\_HIGHPASS( ) 50](#_Toc462298676)

[8.7.8 TILE\_HEADER\_HIGHPASS( ) 51](#_Toc462298677)

[8.7.9 TILE\_FLEXBITS( ) 51](#_Toc462298678)

[8.7.10 Tile-level semantics 51](#_Toc462298679)

[8.7.11 MB\_DC( ) 54](#_Toc462298680)

[8.7.12 DECODE\_DC( ) 55](#_Toc462298681)

[8.7.13 DECODE\_ABS\_LEVEL( ) 55](#_Toc462298682)

[8.7.14 Macroblock DC( ) semantics 56](#_Toc462298683)

[8.7.15 Macroblock low-pass 57](#_Toc462298684)

[8.7.16 General 57](#_Toc462298685)

[8.7.17 Coded block pattern high-pass (CBPHP) 61](#_Toc462298686)

[8.7.18 Macroblock high-pass 67](#_Toc462298687)

[8.7.19 Macroblock FLEXBITS 75](#_Toc462298688)

[8.8 Adaptive VLC code table selection 76](#_Toc462298689)

[8.8.1 General 76](#_Toc462298690)

[8.8.2 Adaptive VLC deltaDisc tables 77](#_Toc462298691)

[8.8.3 Initialization 79](#_Toc462298692)

[8.8.4 Update of adaptive VLC code table selection 81](#_Toc462298693)

[8.9 Adaptation of CBPLP state variables 84](#_Toc462298694)

[8.9.1 General 84](#_Toc462298695)

[8.9.2 InitializeCountCBPLP( ) 84](#_Toc462298696)

[8.9.3 UpdateCountCBPLP( ) 84](#_Toc462298697)

[8.10 Adaptive CBPHP prediction 84](#_Toc462298698)

[8.10.1 InitializeCBPHPModel( ) 84](#_Toc462298699)

[8.10.2 UpdateCBPHPModel( ) 84](#_Toc462298700)

[8.11 Adaptive inverse scanning 85](#_Toc462298701)

[8.11.1 Adaptive inverse scanning tables 85](#_Toc462298702)

[8.11.2 InitializeAdaptiveScanLP( ) 86](#_Toc462298703)

[8.11.3 InitializeAdaptiveScanHP( ) 86](#_Toc462298704)

[8.11.4 ResetTotalsAdaptiveScanLP( ) 86](#_Toc462298705)

[8.11.5 ResetTotalsAdaptiveScanHP( ) 86](#_Toc462298706)

[8.11.6 AdaptiveLPScan( ) 87](#_Toc462298707)

[8.11.7 AdaptiveHPScan( ) 87](#_Toc462298708)

[8.12 Adaptive coefficient normalization 88](#_Toc462298709)

[8.12.1 InitializeModelMB( ) 88](#_Toc462298710)

[8.12.2 UpdateModelMB( ) 88](#_Toc462298711)

[9 Decoding process 89](#_Toc462298712)

[9.1 General 89](#_Toc462298713)

[9.2 Image decoding 89](#_Toc462298714)

[9.3 Image plane decoding 90](#_Toc462298715)

[9.4 Tile transform coefficient processing 90](#_Toc462298716)

[9.4.1 Overview 90](#_Toc462298717)

[9.4.2 DC transform coefficient decoding 91](#_Toc462298718)

[9.4.3 Low-pass transform coefficient decoding 91](#_Toc462298719)

[9.4.4 High-pass transform coefficient decoding 91](#_Toc462298720)

[9.5 Coefficient remapping 92](#_Toc462298721)

[9.5.1 DC coefficient remapping 92](#_Toc462298722)

[9.5.2 Low-pass coefficient remapping 92](#_Toc462298723)

[9.5.3 High-pass macroblock coefficient remapping 93](#_Toc462298724)

[9.5.4 High-pass block coefficient remapping 93](#_Toc462298725)

[9.6 Transform coefficient prediction 93](#_Toc462298726)

[9.6.1 DC coefficient prediction 93](#_Toc462298727)

[9.6.2 Low-pass prediction 96](#_Toc462298728)

[9.6.3 High-pass prediction 98](#_Toc462298729)

[9.7 Derivation of quantization parameters 102](#_Toc462298730)

[9.7.1 Derivation of DC quantization parameters 102](#_Toc462298731)

[9.7.2 Derivation of low-pass quantization parameters 103](#_Toc462298732)

[9.7.3 Derivation of high-pass quantization parameters 104](#_Toc462298733)

[9.8 Dequantization 105](#_Toc462298734)

[9.8.1 Dequantization of DC coefficients 105](#_Toc462298735)

[9.8.2 Dequantization of low-pass coefficients 106](#_Toc462298736)

[9.8.3 Dequantization of high-pass coefficients 106](#_Toc462298737)

[9.8.4 QuantMap( ) 107](#_Toc462298738)

[9.9 Sample reconstruction 108](#_Toc462298739)

[9.9.1 Overview 108](#_Toc462298740)

[9.9.2 First level inverse transform 108](#_Toc462298741)

[9.9.3 First level overlap filtering 109](#_Toc462298742)

[9.9.4 Second level coefficient combination 121](#_Toc462298743)

[9.9.5 Second level inverse transform 122](#_Toc462298744)

[9.9.6 Second level overlap filtering 123](#_Toc462298745)

[9.9.7 Inverse transform basic operations 129](#_Toc462298746)

[9.9.8 Overlap filtering functions 132](#_Toc462298747)

[9.10 Output formatting 136](#_Toc462298748)

[9.10.1 Overview 136](#_Toc462298749)

[9.10.2 Output formatting stage 136](#_Toc462298750)

[9.10.3 Sampling conversion 137](#_Toc462298751)

[9.10.4 Conversion from INTERNAL\_CLR\_FMT to OUTPUT\_CLR\_FMT 139](#_Toc462298752)

[9.10.5 AddBias( ) 142](#_Toc462298753)

[9.10.6 ComputeScaling( ) 143](#_Toc462298754)

[9.10.7 Postscaling process 144](#_Toc462298755)

[9.10.8 Clipping and packing stage 146](#_Toc462298756)

[Annex A Tag-based file format 150](#_Toc462298757)

[A.1 General 150](#_Toc462298758)

[A.2 Normative references 150](#_Toc462298759)

[A.2.1 Identical Recommendations | International Standards 150](#_Toc462298760)

[A.2.2 Paired Recommendations | International Standards equivalent in technical content 150](#_Toc462298761)

[A.2.3 Additional references 150](#_Toc462298762)

[A.3 Definitions 150](#_Toc462298763)

[A.4 Abbreviations 151](#_Toc462298764)

[A.5 FILE\_HEADER( ) 151](#_Toc462298765)

[A.5.1 Syntax structure 151](#_Toc462298766)

[A.5.2 FIXED\_FILE\_HEADER\_II\_2BYTES 151](#_Toc462298767)

[A.5.3 FIXED\_FILE\_HEADER\_0XBC\_BYTE 151](#_Toc462298768)

[A.5.4 FILE\_VERSION\_ID 151](#_Toc462298769)

[A.5.5 FIRST\_IFD\_OFFSET 151](#_Toc462298770)

[A.6 IMAGE\_FILE\_DIRECTORY( ) 152](#_Toc462298771)

[A.6.1 Syntax structure 152](#_Toc462298772)

[A.6.2 NUM\_ENTRIES 152](#_Toc462298773)

[A.6.3 ZERO\_OR\_NEXT\_IFD\_OFFSET 152](#_Toc462298774)

[A.7 IFD\_ENTRY( ) 152](#_Toc462298775)

[A.7.1 Syntax structure 152](#_Toc462298776)

[A.7.2 FIELD\_TAG 154](#_Toc462298777)

[A.7.3 ELEMENT\_TYPE 154](#_Toc462298778)

[A.7.4 NUM\_ELEMENTS 155](#_Toc462298779)

[A.7.5 VALUES\_OR\_OFFSET 155](#_Toc462298780)

[A.7.6 DOCUMENT\_NAME 155](#_Toc462298781)

[A.7.7 IMAGE\_DESCRIPTION 155](#_Toc462298782)

[A.7.8 EQUIPMENT\_MAKE 155](#_Toc462298783)

[A.7.9 EQUIPMENT\_MODEL 155](#_Toc462298784)

[A.7.10 PAGE\_NAME 156](#_Toc462298785)

[A.7.11 PAGE\_NUMBER 156](#_Toc462298786)

[A.7.12 SOFTWARE\_NAME\_VERSION 156](#_Toc462298787)

[A.7.13 DATE\_TIME 156](#_Toc462298788)

[A.7.14 ARTIST\_NAME 157](#_Toc462298789)

[A.7.15 HOST\_COMPUTER 157](#_Toc462298790)

[A.7.16 COPYRIGHT\_NOTICE 157](#_Toc462298791)

[A.7.17 COLOR\_SPACE 158](#_Toc462298792)

[A.7.18 PIXEL\_FORMAT 158](#_Toc462298793)

[A.7.19 SPATIAL\_XFRM\_PRIMARY 166](#_Toc462298794)

[A.7.20 IMAGE\_TYPE 166](#_Toc462298795)

[A.7.21 PTM\_COLOR\_INFO( ) 167](#_Toc462298796)

[A.7.22 PROFILE\_LEVEL\_CONTAINER( ) 172](#_Toc462298797)

[A.7.23 IMAGE\_WIDTH 172](#_Toc462298798)

[A.7.24 IMAGE\_HEIGHT 172](#_Toc462298799)

[A.7.25 WIDTH\_RESOLUTION 172](#_Toc462298800)

[A.7.26 HEIGHT\_RESOLUTION 172](#_Toc462298801)

[A.7.27 IMAGE\_OFFSET 172](#_Toc462298802)

[A.7.28 IMAGE\_BYTE\_COUNT 172](#_Toc462298803)

[A.7.29 ALPHA\_OFFSET 173](#_Toc462298804)

[A.7.30 ALPHA\_BYTE\_COUNT 173](#_Toc462298805)

[A.7.31 IMAGE\_BAND\_PRESENCE 173](#_Toc462298806)

[A.7.32 ALPHA\_BAND\_PRESENCE 173](#_Toc462298807)

[A.7.33 PADDING\_DATA 173](#_Toc462298808)

[Annex B Profiles and levels 174](#_Toc462298809)

[B.1 General 174](#_Toc462298810)

[B.2 Profiles 174](#_Toc462298811)

[B.2.1 Sub-Baseline profile 174](#_Toc462298812)

[B.2.2 Baseline profile 174](#_Toc462298813)

[B.2.3 Main profile 175](#_Toc462298814)

[B.2.4 Advanced profile 175](#_Toc462298815)

[B.2.5 Reserved values of PROFILE\_IDC 175](#_Toc462298816)

[B.3 Levels 175](#_Toc462298817)

[Annex C Color imagery representation and color management 177](#_Toc462298818)

[C.1 Background information 177](#_Toc462298819)

[C.2 Color interpretation in the JPEG XR context 177](#_Toc462298820)

[Annex D Encoder processing 180](#_Toc462298821)

[D.1 Overview 180](#_Toc462298822)

[D.2 Pre-scaling 180](#_Toc462298823)

[D.3 Color conversion 182](#_Toc462298824)

[D.3.1 General 182](#_Toc462298825)

[D.3.2 FwdColorFmtConvert1( ) 182](#_Toc462298826)

[D.3.3 FwdColorFmtConvert2( ) 182](#_Toc462298827)

[D.3.4 FwdColorFmtConvert3( ) 183](#_Toc462298828)

[D.3.5 Macroblock alignment and padding 183](#_Toc462298829)

[D.4 Transform 183](#_Toc462298830)

[D.4.1 T2x2h( ) 185](#_Toc462298831)

[D.4.2 TOdd( ) 185](#_Toc462298832)

[D.4.3 TOddOdd( ) 185](#_Toc462298833)

[D.4.4 FwdPermute( ) 185](#_Toc462298834)

[D.4.5 FCT operations 186](#_Toc462298835)

[D.5 Overlap pre-filtering 187](#_Toc462298836)

[D.5.1 Overview 187](#_Toc462298837)

[D.5.2 OverlapPreFilter4( ) 190](#_Toc462298838)

[D.5.3 OverlapPrefilter2x2 191](#_Toc462298839)

[D.5.4 OverlapPreFilter2 191](#_Toc462298840)

[D.6 Coefficient prediction 191](#_Toc462298841)

[D.7 Coefficient scanning 191](#_Toc462298842)

[D.8 Quantization 191](#_Toc462298843)

[D.9 Adaptive coefficient normalization 192](#_Toc462298844)

[Annex E Media type specification 193](#_Toc462298845)

[E.1 General 193](#_Toc462298846)

[E.2 Media type specification template information 193](#_Toc462298847)

[Bibliography 195](#_Toc462298848)

**LIST OF FIGURES**

*Page*

[Figure 1 – Informative overview of image partitions and internal windowing 22](#_Toc462298849)

[Figure 2 – Informative overview of codestream structure. Image header is followed by a sequence of tiles which are in spatial or frequency mode 23](#_Toc462298850)

[Figure 3 – Informative decoding process block diagram 24](#_Toc462298851)

[Figure 4 – Informative overview of sample reconstruction process 27](#_Toc462298852)

[Figure 5 – DC and LP coefficients in (a) 4×4, (b) 422 chroma, and (c) 420 chroma block 96](#_Toc462298853)

[Figure 6 – HP prediction from left 99](#_Toc462298854)

[Figure D.1 – Encoder transform 184](#_Toc462298855)

[Figure D.2 – Representation of the adaptive normalization process and partitioning of coefficients into VLC coded and FLC coded parts 192](#_Toc462298856)

**LIST OF TABLES**

*Page*

[Table 1 – Precedence order of operators 8](#_Toc462298857)

[Table 2 – Example of a pseudocode table 8](#_Toc462298858)

[Table 3 – Example of a pseudocode comment 9](#_Toc462298859)

[Table 4 – Example of a pseudocode while statement 9](#_Toc462298860)

[Table 5 – Example of a pseudocode for statement 9](#_Toc462298861)

[Table 6 – Example of a pseudocode conditional statement 9](#_Toc462298862)

[Table 7 – Example of the initialization of values in an array in pseudocode 9](#_Toc462298863)

[Table 8 – Definition of mathematical function Max( ) 10](#_Toc462298864)

[Table 9 – Definition of mathematical function Min( ) 10](#_Toc462298865)

[Table 10 – Definition of mathematical function Abs( ) 10](#_Toc462298866)

[Table 11 – Definition of mathematical function Sign( ) 11](#_Toc462298867)

[Table 12 – Definition of mathematical function Round( ) 11](#_Toc462298868)

[Table 13 – Definition of mathematical function Clip( ) 11](#_Toc462298869)

[Table 14 – Definition of mathematical function Numones( ) 12](#_Toc462298870)

[Table 15 – Syntax table example 13](#_Toc462298871)

[Table 16 – Defined mnemonic values 14](#_Toc462298872)

[Table 17 – Pseudocode to calculate chroma component array sizes 21](#_Toc462298873)

[Table 18 – CODED\_IMAGE( ) syntax structure 28](#_Toc462298874)

[Table 19 – VLW\_ESC( ) syntax structure 29](#_Toc462298875)

[Table 20 – IMAGE\_HEADER( ) syntax structure 30](#_Toc462298876)

[Table 21 – Interpretation of SPATIAL\_XFRM\_SUBORDINATE 32](#_Toc462298877)

[Table 22 – OUTPUT\_CLR\_FMT 34](#_Toc462298878)

[Table 23 – OUTPUT\_BITDEPTH 35](#_Toc462298879)

[Table 24 – Pseudocode to determine the position of the left boundaries of the tiles 36](#_Toc462298880)

[Table 25 – Pseudocode to determine the position of the top boundaries of the tiles 36](#_Toc462298881)

[Table 26 – Pseudocode to determine the number of macroblocks in each tile 37](#_Toc462298882)

[Table 27 – IMAGE\_PLANE\_HEADER( ) syntax structure 38](#_Toc462298883)

[Table 28 – INTERNAL\_CLR\_FMT 39](#_Toc462298884)

[Table 29 – BANDS\_PRESENT 39](#_Toc462298885)

[Table 30 – Pseudocode to determine the number of bands present in the codestream, NumBands 39](#_Toc462298886)

[Table 31 – Pseudocode to determine the number of components present in the codestream 41](#_Toc462298887)

[Table 32 – DC\_QP( ) syntax structure 42](#_Toc462298888)

[Table 33 – COMPONENT\_MODE 43](#_Toc462298889)

[Table 34 – LP\_QP( ) syntax structure 43](#_Toc462298890)

[Table 35 – HP\_QP( ) syntax structure 44](#_Toc462298891)

[Table 36 – INDEX\_TABLE\_TILES( ) syntax structure 45](#_Toc462298892)

[Table 37 – PROFILE\_LEVEL\_INFO( ) syntax structure 45](#_Toc462298893)

[Table 38 – CODED\_TILES( ) syntax structure 46](#_Toc462298894)

[Table 39 – TILE\_SPATIAL( ) syntax structure 47](#_Toc462298895)

[Table 40 – TILE\_DC( ) syntax structure 48](#_Toc462298896)

[Table 41 – TILE\_HEADER\_DC( ) syntax structure 48](#_Toc462298897)

[Table 42 – TILE\_LOWPASS( ) syntax structure 49](#_Toc462298898)

[Table 43 – TILE\_HEADER\_LOWPASS( ) syntax structure 49](#_Toc462298899)

[Table 44 – TILE\_HIGHPASS( ) syntax structure 50](#_Toc462298900)

[Table 45 – TILE\_HEADER\_HIGHPASS( ) syntax structure 51](#_Toc462298901)

[Table 46 – TILE\_FLEXBITS( ) syntax structure 51](#_Toc462298902)

[Table 47 – DECODE\_QP\_INDEX( ) syntax structure 53](#_Toc462298903)

[Table 48 – MB\_DC( ) syntax structure 54](#_Toc462298904)

[Table 49 – DECODE\_DC( ) syntax structure 55](#_Toc462298905)

[Table 50 – DECODE\_ABS\_LEVEL( ) syntax structure 55](#_Toc462298906)

[Table 51 – Code table for VAL\_DC\_YUV 56](#_Toc462298907)

[Table 52 – Code table for ABS\_LEVEL\_INDEX 57](#_Toc462298908)

[Table 53 – MB\_LP( ) syntax structure 58](#_Toc462298909)

[Table 54 – REFINE\_LP( ) syntax structure 60](#_Toc462298910)

[Table 55 – Code table for CBPLP\_YUV1 when INTERNAL\_CLR\_FMT is equal toYUV444 60](#_Toc462298911)

[Table 56 – Code table for CBPLP\_YUV1 when INTERNAL\_CLR\_FMT is equal to YUV420 or YUV422 60](#_Toc462298912)

[Table 57 – MB\_CBPHP( ) syntax structure 62](#_Toc462298913)

[Table 58 – REFINE\_CBPHP( ) syntax structure 63](#_Toc462298914)

[Table 59 – Code table for NUM\_CBPHP( ) 64](#_Toc462298915)

[Table 60 – Code table for NUM\_BLKCBPHP if INTERNAL\_CLR\_FMT is one of {YUVK, NCOMPONENT, YONLY} 64](#_Toc462298916)

[Table 61 – Code table for NUM\_BLKCBPHP if INTERNAL\_CLR\_FMT is not one of {YUVK, NCOMPONENT, YONLY} 64](#_Toc462298917)

[Table 62 – Code table for CHR\_CBPHP, VAL\_INC, and CBPHP\_CH\_BLK 64](#_Toc462298918)

[Table 63 – Code table for NUM\_CH\_BLK 65](#_Toc462298919)

[Table 64 – Code table for REF\_CBPHP1 65](#_Toc462298920)

[Table 65 – Pseudocode for function PredCBPHP( ) 65](#_Toc462298921)

[Table 66 – Pseudocode for function PredCBPHP444( ) 66](#_Toc462298922)

[Table 67 – Pseudocode for function PredCBPHP422( ) 66](#_Toc462298923)

[Table 68 – Pseudocode for function PredCBPHP420( ) 67](#_Toc462298924)

[Table 69 – MB\_HP( ) syntax structure 67](#_Toc462298925)

[Table 70 – MB\_HP\_FLEX( ) syntax structure 68](#_Toc462298926)

[Table 71 – DECODE\_BLOCK\_ADAPTIVE( ) syntax structure 69](#_Toc462298927)

[Table 72 – DECODE\_BLOCK( ) syntax structure 70](#_Toc462298928)

[Table 73 – DECODE\_RUN( ) syntax structure 71](#_Toc462298929)

[Table 74 – DECODE\_INDEX( ) syntax structure 71](#_Toc462298930)

[Table 75 – DECODE\_FIRST\_INDEX( ) syntax structure 72](#_Toc462298931)

[Table 76 – Code table of RUN\_VALUE if iMaxRun = = 2 72](#_Toc462298932)

[Table 77 – Code table of RUN\_VALUE if iMaxRun = = 3 73](#_Toc462298933)

[Table 78 – Code table of RUN\_VALUE if iMaxRun = = 4 73](#_Toc462298934)

[Table 79 – Code table of RUN\_INDEX 73](#_Toc462298935)

[Table 80 – Code table for INDEX\_A 73](#_Toc462298936)

[Table 81 – Code table for INDEX\_B 74](#_Toc462298937)

[Table 82 – Code table for FIRST\_INDEX 74](#_Toc462298938)

[Table 83 – MB\_FLEXBITS( ) syntax structure 75](#_Toc462298939)

[Table 84 – BLOCK\_FLEXBITS( ) syntax structure 76](#_Toc462298940)

[Table 85 – DECODE\_FLEX( ) syntax structure 76](#_Toc462298941)

[Table 86 – Constant table AbslevelIndexDelta[m][n] 77](#_Toc462298942)

[Table 87 – Constant table FirstIndexDelta[m][n] 78](#_Toc462298943)

[Table 88 – Constant table Index1Delta[m][n] 78](#_Toc462298944)

[Table 89 – Constant table NumCBPHPDelta[m][n] 78](#_Toc462298945)

[Table 90 – Constant table NumBlkCBPHPDelta[m][n] when INTERNAL\_CLR\_FMT is YONLY, NCOMPONENT, or YUVK 79](#_Toc462298946)

[Table 91 – Constant table NumBlkCBPHPDelta for INTERNAL\_CLR\_FMT other than YONLY, NCOMPONENT, and YUVK 79](#_Toc462298947)

[Table 92 – Pseudocode for function InitializeDCVLC( ) 80](#_Toc462298948)

[Table 93 – Pseudocode for function InitializeLPVLC( ) 80](#_Toc462298949)

[Table 94 – Pseudocode for function InitializeHPVLC( ) 80](#_Toc462298950)

[Table 95 – Pseudocode for function InitializeCBPHPVLC( ) 80](#_Toc462298951)

[Table 96 – Pseudocode for function InitializeVLCTable1( ) 81](#_Toc462298952)

[Table 97 – Pseudocode for function InitializeVLCTable2( ) 81](#_Toc462298953)

[Table 98 – Pseudocode for function AdaptDC( ) 81](#_Toc462298954)

[Table 99 – Pseudocode for function AdaptLP( ) 82](#_Toc462298955)

[Table 100 – Pseudocode for function AdaptHP( ) 82](#_Toc462298956)

[Table 101 – Pseudocode for function AdaptVLCTable1( ) 82](#_Toc462298957)

[Table 102 – Pseudocode for function AdaptVLCTable2( ) 83](#_Toc462298958)

[Table 103 – Pseudocode for function InitializeCountCBPLP( ) 84](#_Toc462298959)

[Table 104 – Pseudocode for function UpdateCountCBPLP( ) 84](#_Toc462298960)

[Table 105 – Pseudocode for function InitializeCBPHPModel( ) 84](#_Toc462298961)

[Table 106 – Pseudocode for function UpdateCBPHPModel( ) 85](#_Toc462298962)

[Table 107 – Definitions of ScanOrder0 and ScanOrder1 85](#_Toc462298963)

[Table 108 – Definition of ScanTotals 85](#_Toc462298964)

[Table 109 – Pseudocode for function InitializeAdaptiveScanLP( ) 86](#_Toc462298965)

[Table 110 – Pseudocode for function InitializeAdaptiveScanHP( ) 86](#_Toc462298966)

[Table 111 – Pseudocode for function ResetTotalsAdaptiveScanLP( ) 86](#_Toc462298967)

[Table 112 – Pseudocode for function ResetTotalsAdaptiveScanHP( ) 86](#_Toc462298968)

[Table 113 – Pseudocode for function AdaptiveLPScan( ) 87](#_Toc462298969)

[Table 114 – Pseudocode for function AdaptiveHPScan( ) 87](#_Toc462298970)

[Table 115 – Pseudocode for function InitializeModelMB( ) 88](#_Toc462298971)

[Table 116 – Pseudocode for function UpdateModelMB( ) 88](#_Toc462298972)

[Table 117 – Pseudocode for function ImageDecoding( ) 89](#_Toc462298973)

[Table 118 – Pseudocode for function ImagePlaneDecoding( ) 90](#_Toc462298974)

[Table 119 – Pseudocode for function TileTransformCoefficientProcessing( ) 91](#_Toc462298975)

[Table 120 – Pseudocode for function DCTransformCoefficientDecoding( ) 91](#_Toc462298976)

[Table 121 – Pseudocode for function LPTransformCoefficientDecoding( ) 91](#_Toc462298977)

[Table 122 – Pseudocode for function HPTransformCoefficientDecoding( ) 92](#_Toc462298978)

[Table 123 – Pseudocode for function DCMBCoefficientRemap( ) 92](#_Toc462298979)

[Table 124 – Pseudocode for function LPMBCoefficientRemap( ) 92](#_Toc462298980)

[Table 125 – Pseudocode for function HPMBCoefficientRemap( ) 93](#_Toc462298981)

[Table 126 – Pseudocode for function HPBlockCoefficientRemap( ) 93](#_Toc462298982)

[Table 127 – Pseudocode for function DCPredictionGeneral( ) 94](#_Toc462298983)

[Table 128 – Pseudocode for function CalcDCPredMode( ) 94](#_Toc462298984)

[Table 129 – Pseudocode for function DCCoefficientPrediction( ) 95](#_Toc462298985)

[Table 130 – Pseudocode for function UpdateDCPredictionVariables( ) 96](#_Toc462298986)

[Table 131 – Pseudocode for function LPPredictionGeneral( ) 97](#_Toc462298987)

[Table 132 – Pseudocode for function CalcLPPredMode( ) 97](#_Toc462298988)

[Table 133 – Pseudocode for function LPCoefficientPrediction( ) 97](#_Toc462298989)

[Table 134 – Pseudocode for function UpdateLPPredictionVariables( ) 98](#_Toc462298990)

[Table 135 – Pseudocode for function CalcHPPredMode( ) 100](#_Toc462298991)

[Table 136 – Pseudocode for function HPCoefficientPrediction( ) 100](#_Toc462298992)

[Table 137 – Pseudocode for function ImagePlaneDCQP( ) 102](#_Toc462298993)

[Table 138 – Pseudocode for function TileLevelDCQP( ) 102](#_Toc462298994)

[Table 139 – Pseudocode for function AssignDCQuantizationParameters( ) 102](#_Toc462298995)

[Table 140 – Pseudocode for function ImagePlaneLPQP( ) 103](#_Toc462298996)

[Table 141 – Pseudocode for function TileLevelLPQP( ) 103](#_Toc462298997)

[Table 142 – Pseudocode for function AssignLPQuantizationParameters( ) 104](#_Toc462298998)

[Table 143 – Pseudocode for function ImagePlaneHPQP( ) 104](#_Toc462298999)

[Table 144 – Pseudocode for function TileLevelHPQP( ) 105](#_Toc462299000)

[Table 145 – Pseudocode for function AssignHPQuantizationParameters( ) 105](#_Toc462299001)

[Table 146 – Pseudocode for function DequantizeDCCoefficients( ) 106](#_Toc462299002)

[Table 147 – Pseudocode for function DequantizeLPCoefficients( ) 106](#_Toc462299003)

[Table 148 – Pseudocode for function DequantizeHPCoefficients( ) 107](#_Toc462299004)

[Table 149 – Pseudocode for function QuantMap( ) 107](#_Toc462299005)

[Table 150 – Pseudocode for function SampleReconstruction( ) 108](#_Toc462299006)

[Table 151 – Pseudocode for function FirstLevelInverseTransform( ) 109](#_Toc462299007)

[Table 152 – Pseudocode for function FirstLevelOverlapFiltering( ) 110](#_Toc462299008)

[Table 153 – Pseudocode for function FirstLevelOverlapFilteringPrimary( ) 110](#_Toc462299009)

[Table 154 – Pseudocode for function FirstLevelOverlapFiltering422( ) 114](#_Toc462299010)

[Table 155 – Pseudocode for function FirstLevelOverlapFiltering420( ) 118](#_Toc462299011)

[Table 156 – Pseudocode for function FirstLevelCallOverlapPostFilter4x4( ) 121](#_Toc462299012)

[Table 157 – Pseudocode for function SecondLevelCoefficientCombination( ) 122](#_Toc462299013)

[Table 158 – Pseudocode for function SecondLevelInverseTransform( ) 123](#_Toc462299014)

[Table 159 – Pseudocode for function SecondLevelOverlapFiltering( ) 123](#_Toc462299015)

[Table 160 – Pseudocode for function ICT4x4( ) 129](#_Toc462299016)

[Table 161 – Pseudocode for function T2x2h( ) 130](#_Toc462299017)

[Table 162 – Pseudocode for function InvTodd( ) 131](#_Toc462299018)

[Table 163 – Pseudocode for function InvToddodd 131](#_Toc462299019)

[Table 164 – Inverse Permutation 131](#_Toc462299020)

[Table 165 – Pseudocode for function InvPermute( ) 132](#_Toc462299021)

[Table 166 – Pseudocode for function InvPermute2pt( ) 132](#_Toc462299022)

[Table 167 – Pseudocode for function T2pt( ) 132](#_Toc462299023)

[Table 168 – Pseudocode for function OverlapPostFilter4x4( ) 133](#_Toc462299024)

[Table 169 – Pseudocode for function OverlapPostFilter4( ) 134](#_Toc462299025)

[Table 170 – Pseudocode for function OverlapPostFilter2x2( ) 135](#_Toc462299026)

[Table 171 – Pseudocode for function OverlapPostFilter2( ) 135](#_Toc462299027)

[Table 172 – Pseudocode for function InvRotate( ) 135](#_Toc462299028)

[Table 173 – Pseudocode for function InvScale( ) 135](#_Toc462299029)

[Table 174 – Pseudocode for function T2x2hPOST( ) 136](#_Toc462299030)

[Table 175 – Pseudocode for function InvToddoddPOST( ) 136](#_Toc462299031)

[Table 176 – Conformance-specified output formatting combinations of OUTPUT\_BITDEPTH and OUTPUT\_CLR\_FMT 137](#_Toc462299032)

[Table 177 – Pseudocode for function OutputFormatting( ) 137](#_Toc462299033)

[Table 178 – Pseudocode for function SamplingConversion( ) 138](#_Toc462299034)

[Table 179 – Conformance-specified sampling conversions 138](#_Toc462299035)

[Table 180 – Pseudocode for function Upsample( ) 138](#_Toc462299036)

[Table 181 – Upsampling filter coefficient for different chroma positions 139](#_Toc462299037)

[Table 182 – Pseudocode for function ConvertInternalToOutputClrFmt( ) 139](#_Toc462299038)

[Table 183 – Conformance-specified color format conversions 140](#_Toc462299039)

[Table 184 – Pseudocode for function InvColorFmtConvert1( ) 140](#_Toc462299040)

[Table 185 – Pseudocode for function InvColorFmtConvert2( ) 141](#_Toc462299041)

[Table 186 – Pseudocode for function InvColorFmtConvert3( ) 141](#_Toc462299042)

[Table 187 – Pseudocode for function InvColorFmtConvert4( ) 141](#_Toc462299043)

[Table 188 – Pseudocode for function AddBias( ) 142](#_Toc462299044)

[Table 189 – Pseudocode for function ComputeScaling( ) 143](#_Toc462299045)

[Table 190 – Pseudocode for function PostscalingProcess( ) 144](#_Toc462299046)

[Table 191 – Pseudocode for function PostScalingInt( ) 144](#_Toc462299047)

[Table 192 – Pseudocode for function PostScalingFl( ) 145](#_Toc462299048)

[Table 193 – Pseudocode for function PostScalingF2( ) 145](#_Toc462299049)

[Table 194 – Pseudocode for function ClippingAndPackingStage( ) 146](#_Toc462299050)

[Table 195 – Pseudocode for function ClippingBasic( ) 147](#_Toc462299051)

[Table 196 – Pseudocode for function ClipAndPackBD565( ) 148](#_Toc462299052)

[Table 197 – Pseudocode for function ClipAndPackBD5( ) 148](#_Toc462299053)

[Table 198 – Pseudocode for function ClipAndPackBD10( ) 148](#_Toc462299054)

[Table 199 – Pseudocode for function ClipAndPackBD1BorW( ) 149](#_Toc462299055)

[Table A.1 – FILE\_HEADER( ) syntax structure 151](#_Toc462299056)

[Table A.2 – IMAGE\_FILE\_DIRECTORY( ) syntax structure 152](#_Toc462299057)

[Table A.3 – IFD\_ENTRY( ) syntax structure 152](#_Toc462299058)

[Table A.4 – Interpretation, allowed combinations, and presence of syntax elements of the IFD\_ENTRY( ) syntax structure 153](#_Toc462299059)

[Table A.5 – ELEMENT\_TYPE 154](#_Toc462299060)

[Table A.6 – Interpretation of PIXEL\_FORMAT 160](#_Toc462299061)

[Table A.7 – PTM\_COLOR\_INFO( ) syntax structure 167](#_Toc462299062)

[Table A.8 – Interpretation of COLOR\_PRIMARIES syntax element 167](#_Toc462299063)

[Table A.9 – Interpretation of TRANSFER\_CHARACTERISTICS syntax element 168](#_Toc462299064)

[Table A.10 – Interpretation of MATRIX\_COEFFICIENTS syntax element 169](#_Toc462299065)

[Table A.11 – Interpretation of IMAGE\_BAND\_PRESENCE and ALPHA\_BAND\_PRESENCE 173](#_Toc462299066)

[Table B.1 – Parameters of levels of conformance to this Specification 176](#_Toc462299067)

[Table B.2 – Pseudocode for function ImageBufferBytes(valNC) 176](#_Toc462299068)

[Table D.1 – Pseudocode for function PreScaling1( ) 180](#_Toc462299069)

[Table D.2 – Pseudocode for function PreScalingBD16F( ) 180](#_Toc462299070)

[Table D.3 – Pseudocode for function PreScalingBD32F( ) 181](#_Toc462299071)

[Table D.4 – Pseudocode for function PreScalingRGBE( ) 181](#_Toc462299072)

[Table D.5 – Pseudocode for function RGBEForwardConversion( ) 181](#_Toc462299073)

[Table D.6 – Pseudocode for function FwdColorFmtConvert1( ) 182](#_Toc462299074)

[Table D.7 – Pseudocode for function FwdColorFmtConvert2( ) 183](#_Toc462299075)

[Table D.8 – Pseudocode for function FwdColorFmtConvert3( ) 183](#_Toc462299076)

[Table D.9 – Pseudocode for function TOdd( ) 185](#_Toc462299077)

[Table D.10 – Pseudocode for function TOddOdd( ) 185](#_Toc462299078)

[Table D.11 – Forward Permutation 186](#_Toc462299079)

[Table D.12 – Pseudocode for function FwdPermute( ) 186](#_Toc462299080)

[Table D.13 – Pseudocode for function FCT4x4( ) 186](#_Toc462299081)

[Table D.14 – Pseudocode for function T2x2hEnc( ) 188](#_Toc462299082)

[Table D.15 – Pseudocode for function FwdRotate( ) 188](#_Toc462299083)

[Table D.16 – Pseudocode for function FwdScale( ) 188](#_Toc462299084)

[Table D.17 – Pseudocode for function FwdTOddOdd( ) 188](#_Toc462299085)

[Table D.18 – Pseudocode for function OverlapPreFilter4x4( ) 189](#_Toc462299086)

[Table D.19 – Pseudocode for function OverlapPreFilter4( ) 190](#_Toc462299087)

[Table D.20 – Pseudocode for function OverlapPreFilter2x2( ) 191](#_Toc462299088)

[Table D.21 – Pseudocode for function OverlapPreFilter2( ) 191](#_Toc462299089)

Introduction

This Recommendation | International Standard 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, i.e., they are applicable to a broad range of applications using compressed colour and grayscale images in communications and computer systems, and within embedded applications, including mobile devices.

As of 2008, the most widely used digital photography format is a nominal implementation of the first JPEG coding format as specified in 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 colour values).

More demanding applications may require a bit depth of 16, providing 65 536 representable values for each channel, and resulting in over 2.8 \* 1014 colour values. Additional scenarios may necessitate even greater bit depths and sample representation formats. When memory or processing power is at a premium, as few as 5 or 6 bit per channel may be used.

The JPEG XR specification enables greater effective use of compressed imagery with this broadened diversity of application requirements. JPEG XR supports a wide range of colour encoding formats including monochrome, red–green–blue (RGB), cyan–magenta–yellow–key (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 HDR 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 10bit per sample RGB,
* Radiance red–green–blue–exponent (RGBE).

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

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

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

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

The main body of this 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 colour imagery representations and colour management are discussed in Annex C. The typical expected encoding process is described in Annex D.

Corrigendum 1 (2009) and Corrigendum 2 (2011) introduced a set of various minor corrections to Rec. ITU-T T.832 (2009). Further corrections were introduced in a second edition of Rec. ITU-T T.832 (2012).

Recommendation ITU-T T.832

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

# Scope

This Recommendation | International Standard specifies a coding format, referred to as JPEG XR, which is designed primarily for continuous-tone photographic content. The main body of this Recommendation | International Standard 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 the storage and interchange of such coded images. Annex B specifies profiles and levels that determine conformance requirements for classes of encoders and decoders. Aspects of colour imagery representations and colour management are discussed in Annex C. The typical expected encoding process is described in Annex D.

# Normative references

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

# Definitions

For the purposes of this Recommendation | International Standard, the following definitions apply. Definitions of terms that have a scope that is limited to the use of the *file format* specified in Annex A are listed in A.3.

In this clause, *italic* is used to identify all occurrences of terms that are defined in this clause.

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 *lowpass coefficients* is called *FLC refinement* and the *fixed-length* coded part of *highpass coefficients* is called *flexbits*.
2. **adaptive inverse scanning**:  A parsing sub-process where the *zigzag scan order* associated with a set of *transform coefficients* is dynamically modified, based on the statistics of previously parsed *transform coefficients*.
3. **adaptive VLC**:  A parsing sub-process where the code table associated with *VLC* parsing of a particular *syntax element* is switched, among a finite set of fixed tables, based on the statistics of previously parsed instances of this syntax element.
4. **alpha image plane**:  An optional secondary *image plane* associated with an *image*, of the same dimensions as the *luma* *component* of the *primary image plane*. The *alpha image plane* has one *component*, a *luma* *component*.
5. **block**:  An m×n array of *samples* or an m×n array of *transform coefficients*.
6. **block index**:  An integer in the range 0 to 15, identifying, by its position in *raster scan order*, a particular 4×4 *block*, within a partition of a 16×16 *block* into 16 4×4 *blocks*.
7. **byte**:  A sequence of 8 bits.
8. **byte-aligned**:  A bit in a *codestream* is *byte-aligned* if its position is an integer multiple of 8 bits from the beginning of the *codestream*, where the first bit in the *codestream* is at position 0.
9. **chroma**:  A *component* of the *primary image plane* with non-zero index or the *transform coefficients* and sample values associated with this *component*.
10. **coded block pattern highpass**:  The *coded block pattern highpass* is a *syntax element* indicating the *coded block status*, i.e., the presence or absence of non-zero *highpass transform coefficients*, for each of the *blocks* in the *macroblock*.
11. **coded block pattern lowpass**:  The *coded block pattern lowpass* is a *syntax element* indicating the presence or absence of non-zero *lowpass transform coefficients* in the *macroblock*.
12. **coded block status**:  The *coded block status* is an indication of the presence or absence of non-zero *transform coefficients* in that *block.*
13. **codestream**:  A sequence of bits contained in a sequence of *bytes* from which syntax elements are parsed, such that the most significant bit of the first *byte* is the first bit of the *codestream*, the next most significant bit of the first *byte* is the second bit of the *codestream*, and so on, to the least significant bit of the first *byte* (which is the eighth bit of the *codestream*), followed by the most significant bit of the second *byte* (which is the ninth bit of the *codestream*), and so on, up to and including the least significant bit of the last *byte* of the sequence of *bytes* (which is the last bit of the *codestream*).
14. **component**:  One of the arrays of samples associated with an *image plane*.
15. **context**:  A possible value of a specific instance of a *context* *variable*.
16. **context variable**:  A variable used in the *parsing process* to select which data structure is to be used for the *adaptive VLC* parsing of a given syntax element.
17. **DC coefficient**:  The first subset when the *transform coefficients*, that are contained in a specific *macroblock* and a specific *component,* are partitioned into 3 subsets.
18. **DC-LP array**:  The array of all DC and lowpass (LP) *transform coefficients*, for all *macroblocks* associated with a specific *component*.
19. **decoder**:  An embodiment of a *parsing process* and *decoding process*.
20. **decoding process**:  The process of computing output sample values from the parsed syntax elements of the *codestream*.
21. **dequantization**:  The process of rescaling the quantized *transform coefficients* after their value has been parsed from the *codestream* and before they are presented to the *inverse transform process*.
22. **discriminant**:  A collective term for one of DiscrimVal1 or DiscrimVal2, which are the two member variables of an instance of the *adaptive VLC* data structure specified in 5.5.5.
23. **encoder**:  An embodiment of an *encoding process*.
24. **encoding process**:  The process of converting source sample values into a *codestream* conforming to this Specification.
25. **extended image**:  The *image* produced by the *decoding process* prior to *windowing*. The *extended image* has a *luma* array that is an integer multiple of 16 in width and height.
26. **file**:  A finite-length sequence of *bytes* that is accessible to a *decoder* in a manner such that the *decoder* can obtain access to the data at specified positions within the sequence of *bytes* (e.g. by storing the entire sequence of *bytes* in random access memory or by performing "position seek" operations to specified positions within the sequence of *bytes*).
27. **file format**:  A specified structure for the content of a *file*.
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.
29. **FLC refinement**:  The *fixed-length coded* part of a *DC coefficient* or *lowpass coefficient* that is parsed using adaptive fixed-length codes.
30. **flexbits**:  The *fixed-length coded* part of the *highpass coefficient* information which is parsed using adaptive fixed-length codes.
31. **frequency band**:  A collective term for one of the following three subsets of the *transform coefficients* for an *image*, which are separately parsed: *DC coefficients*, *lowpass coefficients*, and *highpass coefficients*.
32. **frequency mode**:  A *codestream* structure mode where the DC, lowpass, highpass and *flexbits* *frequency* *bands* for each *tile* are grouped separately.
33. **hard tiles**:  A *codestream* structure mode where the overlap operators are not applied across tile boundaries. Instead, boundary overlap operators are applied at tile boundaries.
34. **highpass coefficients**:  The third subset, when the *transform coefficients* that are contained in a specific *macroblock* and a specific *component* are partitioned into three subsets.
35. **image**:  The result of the *decoding process*, consisting of a *primary image plane* and an optional *alpha image plane*.
36. **image plane**:  A collective term for a grouping of the *components* of the *image*.
37. **initial level value**:  One of two values used to compute the *VLC*-coded part of a *transform coefficient*.
38. **internal colour format**:  The colour format associated with the spatial-domain samples obtained through the *inverse transform process* and the *sample reconstruction process*, and distinguished from the *output colour format* associated with the *output formatting process*.
39. **inverse core transform (ICT)**:  The two steps of the *inverse transform process* that involve processing of *transform coefficients* associated with each *macroblock* independently, with no *overlap filtering*.
40. **inverse transform process**:  The part of the *decoding process* by which a set of *dequantized* *transform coefficients* are converted into spatial-domain values.
41. **inverse scanning**:  The process of reordering an ordered set of parsed *syntax elements* from the *codestream* to form an array of *transform coefficients* associated with a specific *component* and *macroblock*.
42. **little-endian form**:  An ordering of the *bytes* that represent a numerical value as an integer number of *bytes* in which the *bytes* representing the number are in ascending order of significance, i.e., with the least significant *byte* first, followed by the next least significant *byte*, etc.
43. **lowpass coefficients**:  The second subset, when the *transform coefficients* that are contained in a specific *macroblock* and a specific *component* are partitioned into three subsets.
44. **luma**:  The *component* of an *image plane* with index zero, and the *transform coefficients* and sample values associated with this *component*. Although this term is commonly associated with a signal that conveys perceptual brightness information, as used in this Specification the term is primarily an identifier of a particular array of samples or *transform coefficients* for an *image*.
45. **macroblock**:  The collection of *transform coefficients* or samples, across all *components*, that have the same indices i and j with respect to a *macroblock partition*.
46. **macroblock partition**:  The partitioning of each *component*, into 16×16, 8×8 or 16×8 *blocks*, depending on the *internal colour format*.
47. **output bit depth**:  The representation, including the number of bits and the interpretation of the bit pattern, used for the sample values of the output *image* that are the result of the *decoding process*.
48. **output colour format**:  The colour format associated with the output *image* that is the result of the *decoding process*.
49. **output formatting process**:  The process of converting the arrays of samples – that are the result of the *sample reconstruction process* – into the output samples that constitute the output of the *decoding process*. This specifies a conversion (if necessary) into the appropriate *output colour format* and *output bit depth*.
50. **output image height**:  The height of the sub-array of the *luma* *component* of the *primary image plane* that is output by the *decoding* *process*.
51. **output image width**:  The width of the sub-array, of the *luma* *component* of the *primary image plane* that is output by the *decoding* *process*.
52. **overlap filtering**:  The steps of the *inverse transform process* that involve processing of *transform coefficients* across adjacent *blocks* and *macroblocks*.

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

1. **parsing process**:  The process of extracting bit sequences from the *codestream*, converting these bit sequences to syntax element values, and setting the values of global variables for use in the *decoding process*.
2. **prediction**:  The process of computing an estimate of the sample value or data element that is currently being decoded.
3. **prediction residual**:  The difference between the result of a *prediction* process invoked for a sample or data element and its intended value.
4. **primary image plane**:  The *image plane* that consists of all *image* *components* that are not a part of the *alpha image plane*.
5. **QP index**:  An integer, which for a particular *frequency band* and *macroblock* specifies the index into the table of *quantization parameters* available for this *frequency band* and *tile*. This *QP index* thereby selects, for this *macroblock*, the *quantization parameter* used for the *dequantization* of the *transform coefficients* in the specific *frequency band*.
6. **QP set**:  The set of *quantization parameters* associated with a particular *frequency* *band*, corresponding to the *luma* and *chroma* *components*.
7. **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.
8. **raster scan order**:  The scan order in which a two-dimensional array of values is scanned row-wise from left to right, and the rows are scanned from the top row to the bottom.
9. **refinement**:  The process of modifying a predicted or partially computed *transform coefficient*.
10. **run**:  The number of zero valued coefficient levels that precede a non-zero valued coefficient level in the *zigzag scan order* during the *inverse scanning* process.
11. **sample reconstruction process**:  The process of converting dequantized *transform coefficients* into samples of the *image*.
12. **soft tiles**:  A *codestream* structure mode where the overlap operators are applied across tile boundaries.
13. **spatial co-location**:  Sub-arrays of samples are *spatially co-located* across *components* when they correspond to the same spatial region of the decoded *image*. The *macroblock partition* of the *image* ensures that the i-th *macroblock* horizontally and j-th *macroblock* vertically across all *components* are *spatially co-located*.
14. **spatial mode**:  A *codestream* structure mode where the *DC*, *lowpass*, *highpass* and *flexbits* *frequency bands* for each specific *macroblock* are grouped together.
15. **spatial transformation**:  An element in the *codestream* indicating the preferred final displayed orientation of the decoded *image*, as specified in 8.3.8. The *spatial transformation* is only a suggestion, and *decoder* conformance is checked only for the decoded *image* prior to the application of this transformation (i.e., for orientation 0).
16. **start code**:  A bit pattern that specifies the beginning of a *tile packet* or other distinguished, contiguous set of syntax elements in the *codestream*.
17. **tile**:  The collection of *macroblocks* that have the same indices i and j with respect to a *tile partition*. Each *tile* corresponds to the *macroblocks* for a rectangular region of the *image*.
18. **tile packet**:  A contiguous subset of the *codestream*, which contains the coded *syntax elements* associated with a specific *tile*.
19. **tile partition**:  A partition of the *image* into rectangular arrays of *macroblocks*, as specified in 6.4.
20. **transform coefficients**:  The values, associated with each specific *macroblock* and specific *component*, that − after *dequantization* −form the input arrays into the *inverse transform process*.
21. **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.
22. **VLC refinement**:  One of two values used to compute the *VLC*-coded part of a *transform coefficient*. The number of bits required to specify the *VLC-refinement* is dependent on the value of the *initial level value*. The *VLC refinement* is added to the *initial level value* to produce the *VLC*-coded part of the transform coefficient.
23. **windowing**:  A selection of *spatially co-located* sub-arrays of the *components* of all present *image planes* associated with an *image* that are output by the *decoding process*.
24. **zigzag scan order**:  An adaptive ordering for the *inverse scanning* process, which assigns array indices to each subsequent *transform coefficient* parsed from the *codestream*.

# Abbreviations

For the purposes of this Recommendation | International Standard, the following abbreviations apply. Abbreviations which have a scope that is limited to the use of the file format specified in Annex A are listed in A.4.

CBPHP Coded Block Pattern Highpass

CBPLP Coded Block Pattern Lowpass

CMYK Cyan–Magenta–Yellow–Key

DC Direct Current

FCT Forward Core Transform

FLC Fixed-Length Code

HDR High Dynamic Range

HP Highpass

ICT Inverse Core Transform

JPEG Joint Photographic Experts Group

LP Lowpass

LSB Least Significant Bit

MSB Most Significant Bit

QP Quantization Parameter

RGB Red–Green–Blue

RGBE Red–Green–Blue–Exponent

VLC Variable Length Code

XR extended Range

# Conventions

## Conformance language

This Recommendation | International Standard consists of normative and informative text.

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

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

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

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

## Operators

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

### Arithmetic operators

|  |  |
| --- | --- |
| + | Addition |
| − | Subtraction (as a binary operator) or negation (as a unary prefix operator) |
| ++ | Increment by one as a unary postfix operator |
| − − | Decrement by one as a unary postfix operator |
| \* | Multiplication |
| / | Integer division, where the result is truncated towards zero |
| ÷ | Division in mathematical equations where no truncation or rounding is intended |
|  | Division in mathematical equations where no truncation or rounding is intended |
| % | x % a is defined as the modulus operator for x >= 0 and a > 0  x % a is defined as −(((−x) % a)) for x < 0 and a > 0 |

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

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

### Logical operators

|  |  |
| --- | --- |
| | | | Logical OR |
| && | Logical AND |
| ! | Logical NOT |

TRUE/FALSE convention:

* When a variable or arithmetic expression having a non-zero value is evaluated as a logical condition or as an element of a logical expression, it is evaluated as TRUE, and when a variable or expression having a zero value is evaluated as a logical condition or as an element of a logical expression, it is evaluated as FALSE.
* When the value of a variable or arithmetic expression is compared to the value TRUE (in text or using a relational operator), it is compared to the value 1, and when the value of a variable or arithmetic expression is compared to the value FALSE (in text or using a relational operator), it is compared to the value 0.
* When a variable is set to the value TRUE, it is set to the value 1; and when a variable is set to the value FALSE, it is set to the value 0.

### Relational operators

|  |  |
| --- | --- |
| > | Greater than |
| >= | Greater than or equal to |
| < | Less than |
| <= | Less than or equal to |
| = = | Equal to |
| != | Not equal to |

### Bit-wise operators

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

|  |  |
| --- | --- |
| & | AND. When operating on a bit pattern argument that contains fewer bits than the other argument, the shorter argument is extended by adding more significant bits equal to the MSB of the shorter argument, such that the number of bits representing the shorter argument is made the same as the number of bits for the longer argument. |
| | | OR. When operating on a bit pattern argument that contains fewer bits than the other argument, the shorter argument is extended by adding more significant bits equal to the MSB of the shorter argument, such that the number of bits representing the shorter argument is made the same as the number of bits for the longer argument. |
| ^ | XOR. When operating on a bit pattern argument that contains fewer bits than the other argument, the shorter argument is extended by adding more significant bits equal to the MSB of the shorter argument, such that the number of bits representing the shorter argument is made the same as the number of bits for the longer argument. |
| x >> b | Arithmetic right shift of a two's complement integer representation of x by b binary digits, where b is a non-negative integer. Bits shifted into the MSBs as a result of the right shift have a value equal to the MSB of x prior to the shift operation. |
| x << b | Arithmetic left shift of a two's complement integer representation of x by b binary digits, where b is a non-negative integer. Bits shifted into the least significant bits (LSBs) as a result of the left shift have a value equal to 0. |

### Assignment operators

|  |  |
| --- | --- |
| = | Assignment operator |
| += | x += a is defined as x = x + a |
| −= | x −= a is defined as x = x − a |
| ^= | x ^= a is defined as x = x ^ a |
| \*= | x \*= a is defined as x = x \* a |
| <<= | x <<= a is defined as x = (x << a) |
| >>= | x >>= a is defined as x = (x >> a) |

### Precedence order of operators

Operators are listed below in descending order of precedence. If several operators appear in the same line, they have equal precedence. When several operators of equal precedence appear at the same level in an expression, evaluation proceeds according to the associativity of the operator either from right to left or from left to right. See Table 1.

Table 1 – Precedence order of operators

|  |  |  |
| --- | --- | --- |
| **Operators** | **Type of operation** | **Associativity** |
| ( ), [ ], . | Expression | Left to Right |
| ++, − − | Postfix operators | Right to Left |
| −, ! | Unary |  |
| \*, /, %, | Multiplication and division | Left to Right |
| +, − | Addition and subtraction | Left to Right |
| <<, >> | Arithmetic shift | Left to Right |
| < , >, <=, >= | Relational | Left to Right |
| = =, != | Equality test | Left to Right |
| &, |, ^ | Bit-wise operator | Left to Right |
| &&, | | | Logical operators | Left to Right |
| ?, =, \*=, +=, −=, ^=, <<=, >>= | Assignment operators | Right to Left |

### Pseudocode operations

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

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

* Non-array variables, which are passed by value (e.g., valueArgument1 in Table 2).
* Arrays, which are passed by reference (e.g., arrayArgument2[ ] in Table 2).

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

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

Table 2 – Example of a pseudocode table

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

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

Table 3 – Example of a pseudocode comment

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

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

Table 4 – Example of a pseudocode while statement

|  |  |
| --- | --- |
| **ExamplePseudocodeWhileStatement( ) {** | **Reference** |
| while (condition) |  |
| statement |  |
| } |  |

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

Table 5 – Example of a pseudocode for statement

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

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

Table 6 – Example of a pseudocode conditional statement

|  |  |
| --- | --- |
| **ExamplePseudocodeConditionalStatement( ) {** | **Reference** |
| if (condition) |  |
| statement |  |
| else |  |
| alternative statement |  |
| } |  |

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

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

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

### Mathematical functions

|  |  |
| --- | --- |
| Ceiling(x) | Ceiling function. Returns the smallest integer that is greater than or equal to the real-valued argument x. |
| Floor(x) | Floor function. Returns the largest integer that is less than or equal to the real-valued argument x. |
| Max(a, b) | Maximum of two arguments as specified in Table 8.  Table 8 – Definition of mathematical function Max( )   |  |  | | --- | --- | | **Max(a, b) {** | **Reference** | | if (a >= b) |  | | valueReturn = a |  | | else |  | | valueReturn = b |  | | return valueReturn |  | | } |  | |
| Min(a, b) | Minimum of two arguments as specified in Table 9.  Table 9 – Definition of mathematical function Min( )   |  |  | | --- | --- | | **Min(a, b) {** | **Reference** | | if (a <= b) |  | | valueReturn = a |  | | else |  | | valueReturn = b |  | | return valueReturn |  | | } |  | |
| Abs(x) | Absolute value of an argument as specified in Table 10.  Table 10 – Definition of mathematical function Abs( )   |  |  | | --- | --- | | **Abs(x) {** | **Reference** | | if (x >= 0) |  | | valueReturn = x |  | | else |  | | valueReturn = −x |  | | return valueReturn |  | | } |  | |
| Sign(x) | Sign of an argument as specified in Table 11.  Table 11 – Definition of mathematical function Sign( )   |  |  | | --- | --- | | **Sign(x) {** | **Reference** | | if (x >= 0) |  | | valueReturn = 1 |  | | else |  | | valueReturn = −1 |  | | return valueReturn |  | | } |  | |
| Round(x) | Rounding to integer value as specified in Table 12.  Table 12 – Definition of mathematical function Round( )   |  |  | | --- | --- | | **Round(x) {** | **Reference** | | valueReturn = Sign(x) \* Floor(Abs(x) + 0.5) |  | | return valueReturn |  | | } |  | |
| Clip(x, iLow, iHigh) | Clip integer value to a range between integers iLow and iHigh is specified in Table 13.  Table 13 – Definition of mathematical function Clip( )   |  |  | | --- | --- | | **Clip(x, iLow, iHigh) {** | **Reference** | | valueReturn = Max(x, iLow) |  | | valueReturn = Min(valueReturn, iHigh) |  | | return valueReturn |  | | } |  | |
| Sqrt(x) | Square root of x |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numones(x) | Returns the number of bits in an argument that are set, for a positive integer argument x that is represented in two's complement arithmetic as specified in Table 14.  Table 14 – Definition of mathematical function Numones( )   |  |  | | --- | --- | | **Numones(x) {** | **Reference** | | valueReturn = 0 |  | | while (x != 0) { |  | | valueReturn += (x & 1) |  | | x >>= 1 |  | | } |  | | return valueReturn |  | | } |  | |

## Syntax and semantics notation

### 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.

See 5.3.2 for 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 MSB is the leftmost 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 MSB is the leftmost 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 MSB of the code is the leftmost 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 MSB of the code is the left most bit. This indicates a variable-length coded syntax element, where the VLC table used to parse the syntax element is selected adaptively based on the values of other previously parsed syntax elements.

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

Table 15 – Syntax table example

|  |  |  |
| --- | --- | --- |
| **HEADER\_EXAMPLE( ) {** | **Descriptor** | **Reference** |
| /\* A statement can be a syntax element or a conditional statement that specifies the presence and type of syntax element \*/ |  |  |
| /\* The conditional statements are expressed in terms of the pseudocode operations defined in 5.2.7 \*/ |  |  |
| if (condition) |  |  |
| SYNTAX\_ELEMENT\_EXAMPLE | u(8) | 5.3.2 |
| } |  |  |

### SYNTAX\_ELEMENT\_EXAMPLE semantics

SYNTAX\_ELEMENT\_EXAMPLE is an example 8-bit syntax element having semantics and constraints specified in this subclause, as identified in the "Reference" column of Table 15.

### Syntax functions

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

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

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

IS\_BYTE\_ALIGNED( ) is a syntax function specified as follows.

* If the current position in the codestream is on a byte boundary, i.e., the next bit in the codestream is the first bit in a byte, the return value of IS\_BYTE\_ALIGNED( ) is equal to TRUE.
* Otherwise, the return value of IS\_BYTE\_ALIGNED( ) is equal to FALSE.

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

## Formatting conventions

### Variable and array naming conventions

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

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

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

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

### Data structure naming conventions

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

### Syntax element naming conventions

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

### Syntax structure naming conventions

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

### Naming conventions for mnemonic constants

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

Table 16 – Defined mnemonic values

|  |  |
| --- | --- |
| **Mnemonics** | **Syntax element and Reference** |
| YUV444, YUV422, YUV420, YONLY, CMYK, CMYKDIRECT, RGB, RGBE, NCOMPONENT | Table 22 for syntax element OUTPUT\_CLR\_FMT |
| YUV444, YUV422, YUV420, YONLY, YUVK, NCOMPONENT | Table 28 for syntax element INTERNAL\_CLR\_FMT |
| BD1WHITE1, BD8, BD16, BD16S, BD16F, BD32S, BD5, BD10, BD565, BD1BLACK1 | Table 23 for syntax element OUTPUT\_BITDEPTH |
| ALL, NOFLEXBITS, NOHIGHPASS, DCONLY | Table 29 for syntax element BANDS\_PRESENT |

### Naming conventions for numerical values

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

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

### Array dimensions convention

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

## Global variables

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

### Image variables

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

### Image plane variables

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

### Tile variables

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

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

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

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

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

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

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

### Macroblock variables

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

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

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

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

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

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

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

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

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

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

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

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

### Data structures for adaptive VLC table selection

#### General

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

**AdaptiveVLC** data structure template member variables are as follows:

* DiscrimVal1: This variable accumulates statistics about the code table choice, used for adaptively switching to other code tables.
* DiscrimVal2: This is a second variable used to accumulate statistics about the code table choice, used for adaptively switching to other code tables. When there are more than two code tables for the parsing of a given syntax element, the associated AdaptiveVLC data structure instance requires two discriminants.
* TableIndex: The index selecting which code table is used for the current macroblock.
* DeltaTableIndex: The index selecting which Delta Table is used for modifying DiscrimVal1, for the current macroblock.
* Delta2TableIndex: The index selecting which Delta Table is used for modifying DiscrimVal2, for the current macroblock.

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

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

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

#### DC adaptiveVLC data structure instances

**AbslevelIndDCLum**: This data structure instance accumulates statistics for the ABS\_LEVEL\_INDEX syntax element, during parsing of the syntax structure DECODE\_ABS\_LEVEL( ) (Table 50) from within DECODE\_DC( ) (Table 49), referring to luma DC values. The corresponding TableIndex chooses between the code tables for the syntax element ABS\_LEVEL\_INDEX, which are specified in ‎8.7.14.5.

**AbslevelIndDCChr**: This data structure instance accumulates statistics for the ABS\_LEVEL\_INDEX syntax element, during parsing of the syntax structure DECODE\_ABS\_LEVEL( ) (Table 50) from within DECODE\_DC( ) (Table 49), referring to chroma DC values. The corresponding TableIndex chooses between the code tables for the syntax element ABS\_LEVEL\_INDEX, which are specified in ‎8.7.14.5.

#### Lowpass adaptiveVLC data structure instances

**DecFirstIndLPLum**: This data structure instance accumulates statistics for the FIRST\_INDEX syntax element, during parsing of the syntax structure DECODE\_FIRST\_INDEX( ) (Table 75) from within DECODE\_BLOCK( ) (Table 72), referring to luma LP values. The corresponding TableIndex chooses between the code tables for the syntax element FIRST\_INDEX, which are specified in ‎8.7.18.9.7.

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

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

**DecFirstIndLPChr**: This data structure instance accumulates statistics for the FIRST\_INDEX syntax element, during parsing of the syntax structure DECODE\_FIRST\_INDEX( ) (Table 75) from within DECODE\_BLOCK( ) (Table 72), referring to chroma LP values. The corresponding TableIndex chooses between the code tables for the syntax element FIRST\_INDEX, which are specified in ‎8.7.18.9.7.

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

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

**AbsLevelIndLP0**: This data structure instance accumulates statistics for the ABS\_LEVEL\_INDEX syntax element, during parsing of the syntax structure DECODE\_ABS\_LEVEL( ) (Table 50) from within DECODE\_BLOCK( ) (Table 72), referring to LP values with iContext (local to the pseudocode table for DECODE\_BLOCK( )) equal to 0. The corresponding TableIndex chooses between the code tables for the syntax element ABS\_LEVEL\_INDEX, which are specified in ‎8.7.14.5.

**AbsLevelIndLP1**: This data structure instance accumulates statistics for the ABS\_LEVEL\_INDEX syntax element, during parsing of the syntax structure DECODE\_ABS\_LEVEL( ) (Table 50) from within DECODE\_BLOCK( ) (Table 72), referring to LP values with iContext (local to the pseudocode table for DECODE\_BLOCK( )) equal to 1. The corresponding TableIndex chooses between the code tables for the syntax element ABS\_LEVEL\_INDEX, which are specified in ‎8.7.14.5.

#### Highpass adaptiveVLC data structure instances

**DecFirstIndHPLum**: This data structure instance accumulates statistics for the FIRST\_INDEX syntax element, during parsing of the syntax structure DECODE\_FIRST\_INDEX( ) (Table 75) from within DECODE\_BLOCK( ) (Table 72), referring to luma HP values. The TableIndex chooses between the code tables for the syntax element FIRST\_INDEX, which are specified in ‎8.7.18.9.7.

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

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

**DecFirstIndHPChr**: This data structure instance accumulates statistics for the FIRST\_INDEX syntax element, during parsing of the syntax structure DECODE\_FIRST\_INDEX( ) (Table 75) from within DECODE\_BLOCK( ) (Table 72), referring to chroma HP values. The TableIndex chooses between the code tables for the syntax element FIRST\_INDEX, which are specified in ‎8.7.18.9.7.

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

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

**AbsLevelIndHP0**: This data structure instance accumulates statistics for the ABS\_LEVEL\_INDEX syntax element, during parsing of the syntax structure DECODE\_ABS\_LEVEL( ) (Table 50) from within DECODE\_BLOCK( ) (Table 72), referring to HP values with iContext (local to the pseudocode table for DECODE\_BLOCK( )) equal to 0. The TableIndex chooses between the code tables for the syntax element ABS\_LEVEL\_INDEX, which are specified in ‎8.7.14.5.

**AbsLevelIndHP1**: This data structure instance accumulates statistics for the ABS\_LEVEL\_INDEX syntax element, during parsing of the syntax structure DECODE\_ABS\_LEVEL( ) (Table 50) from within DECODE\_BLOCK( ) (Table 72), referring to HP values with iContext (local to the pseudocode table for DECODE\_BLOCK( )) equal to 1. The TableIndex chooses between the code tables for the syntax element ABS\_LEVEL\_INDEX. This syntax element's code tables are specified in ‎8.7.14.5.

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

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

NOTE – The number of code tables for the NUM\_BLKCBPHP syntax element is either 5 or 9, depending on the INTERNAL\_CLR\_FMT of the image.

### Adaptive coefficient normalization data structure instances

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

**Model** data structure template member variables:

* MState[i]: This variable maintains the associated state, where MState[0] represents the information for the luma component, and MState[1] represents the information for the chroma components;
* MBits[i]: This variable represents the number of bits, where MBits[0] represents the information for the luma component, and MBits[1] represents the information for the chroma components.

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

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

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

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

### Adaptive CBPHP prediction data structure instance

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

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

* CBPHPState[i]: This variable maintains the state;
* CountOnes[i]: This variable represents the count of blocks, computed as specified in 8.10.2, with the value of coded status (derived from CBPHP) equal to 1;
* CountZeroes[i]: This variable represents the count of blocks, computed as specified in 8.10.2, with the value of coded status equal to 0.

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

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

### Adaptive count CBPLP variables

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

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

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

## Adaptive VLC deltaDisc tables

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

**AbslevelIndexDelta[i][j]**: This variable holds the value associated with the incremental discriminant value for the i-th syntax element and the j-th code table associated with syntax element ABS\_LEVEL\_INDEX (8.7.14.5).

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

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

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

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

## Adaptive inverse scanning tables

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

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

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

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

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

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

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

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

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

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

# Image and codestream structures

## General

In this clause, *italic* is used to identify all occurrences of terms defined in clause 3.

## Image planes and component arrays

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

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

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

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

ExtendedHeight[0] is referred to as the *extended image* height*.* It specifies the number of rows in the *luma* array that are processed within the *decoding process*. Its value is set equal to HEIGHT\_MINUS1 + 1 + TOP\_MARGIN + BOTTOM\_MARGIN.

ExtendedWidth[0] is referred to as the *extended image* width*.* It specifies the number of columns in the *luma* array that are processed within the *decoding process*. Its value is set equal to WIDTH\_MINUS1 + 1 + LEFT\_MARGIN + RIGHT\_MARGIN.

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

Table 17 – Pseudocode to calculate chroma component array sizes

|  |  |
| --- | --- |
| **CalculateChromaComponentArraySizes( ) {** | **Reference** |
| if (INTERNAL\_CLR\_FMT = = YUV420) { |  |
| ExtendedHeight[1] = ExtendedHeight[0] / 2 |  |
| ExtendedWidth[1] = ExtendedWidth[0] / 2 |  |
| } else if (INTERNAL\_CLR\_FMT = = YUV422) { |  |
| ExtendedHeight[1] = ExtendedHeight[0] |  |
| ExtendedWidth[1] = ExtendedWidth[0] / 2 |  |
| } else { |  |
| ExtendedHeight[1] = ExtendedHeight[0] |  |
| ExtendedWidth[1] = ExtendedWidth[0] |  |
| } |  |
| } |  |

## Image windowing

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

## Image partitioning

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

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

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

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

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

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

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



Figure 1 – Informative overview of image partitions and internal windowing

## Transform coefficients and frequency bands

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

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

* If the *component* is a *luma* *component* or a *chroma* *component* with no down sampling, the following is true. The *component* contains 256 *transform coefficients* for each *macroblock*. These 256 *transform coefficients* are partitioned into three subsets. One set is of size 1, and this coefficient constitutes the *DC coefficient* of this *component*. Another set is of size 15; this set constitutes the *lowpass coefficients*. The third set is of size 240, and constitutes the *highpass coefficients*.
* If the *component* is a *chroma* *component* with a sampling rate of ½ for both the horizontal and vertical directions, the following holds true. The *component* contains 64 *transform coefficients* for each *macroblock*. These 64 *transform coefficients* are partitioned into three subsets. One set is of size 1, and this coefficient constitutes the *DC coefficient* of this *component*. Another set is of size 3; this set constitutes the *lowpass coefficients*. The third set is of size 60, and constitutes the *highpass coefficients*.
* If the *component* is a *chroma* *component* with a sampling rate of ½ for the horizontal direction and a sampling rate of 1 for the vertical direction, the following holds true. The *component* contains 128 *transform coefficients* for each *macroblock*. These 128 *transform coefficients* are partitioned into three subsets. One set is of size 1, and this coefficient constitutes the *DC coefficient* of this *component*. Another set is of size 7; this set constitutes the *lowpass coefficients*. The third set is of size 120, and constitutes the *highpass coefficients*.

NOTE 1 – The partitioning of *transform coefficients* into three sets, and the use of the terms DC, lowpass, and highpass is based on the hierarchical nature of the transform.

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

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

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

## Codestream structure

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

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



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

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

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

## Precision and word length

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

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

NOTE – Encoding with SCALED\_FLAG equal to TRUE typically improves rate-distortion performance for lossy coding.

# Overview of decoder

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

## General

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



Figure 3 – Informative decoding process block diagram

The parsing process consists of the following stages:

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

The decoding process consists of the following stages:

1. coefficient remapping;
2. coefficient prediction;
3. dequantization;
4. sample reconstruction, which consists of the following stages:
   1. first level inverse transform,
   2. when indicated, a first level overlap filter,
   3. second level inverse transform,
   4. when indicated, a second level overlap filter:
5. output formatting.

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

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

## Overview of parsing process

### Overview of image layer codestream parsing

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

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

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

### Overview of tile layer codestream parsing

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

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

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

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

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

### Overview of macroblock layer codestream parsing

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

See 8.7.11 for the specification of the syntax structure MB\_DC( ), which parses the syntax elements related to the DC coefficient, for each component.

See 8.7.16.1 for the specification of the syntax structure MB\_LP( ), which parses the syntax elements related to the lowpass coefficients for each component and also performs inverse scanning of the coefficients.

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

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

## Overview of the decoding process

### Overview of coefficient mapping

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

### Overview of coefficient prediction

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

### Overview of dequantization

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

### Overview of sample reconstruction

See 9.9 for a specification of the sample reconstruction process.

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

* An inverse core transform (ICT) is applied to each 4×4 block corresponding to reconstructed DC and LP coefficients arranged in an array known as the DC-LP array. The first level inverse transform process is specified in 9.9.2.
* An overlap filter operation, when indicated, is applied to 4×4 areas evenly straddling blocks in the DC-LP array. For images with soft tiles, this filter is applied to all such blocks. For images with hard tiles, this filter is applied only to the interior of tiles. Furthermore, an overlap filter is applied to boundary 2×4 and 4×2 areas, as well as the four 2×2 corner areas. For images with hard tiles, these filters are additionally applied at tile boundaries. The first level overlap filtering process is specified in 9.9.3. For INTERNAL\_CLR\_FMT equal to YUV420 or YUV422, alternate filter operations are applied to the 2×2 interior blocks and 2×1 and 1×2 edge blocks of the chroma components. For these cases, a prediction process is used for the corner samples, denoted 'OverlapPostFilter1' in Figure 4 and detailed in Table 154 and Table 155.
* The resulting array contains coefficients of the 4×4 blocks corresponding to the first level transform. These coefficients are combined with the reconstructed HP coefficients into a larger array. This coefficient combination process is specified in 9.9.4.
* An ICT is applied to each 4×4 block. The second level inverse transform process is specified in 9.9.5.
* An overlap filter operation, when indicated, is applied to 4×4 areas evenly straddling blocks in the DC-LP array. For images with soft tiles, this filter is applied to all such blocks. For images with hard tiles, this filter is applied only to the interior of tiles. Furthermore, an overlap filter is applied to boundary 2×4 and 4×2 areas, as well as the four 2×2 corner areas. For images with hard tiles, these filters are additionally applied at tile boundaries. The second level overlap filtering process is specified in 9.9.6.

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



Figure 4 – Informative overview of sample reconstruction process

### Overview of output formatting

See 9.10 for a specification of the outputting process that accounts for the various transformations required to handle the different colour formats and bit depths.

# Syntax, semantics, and parsing process

## General

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

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

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

## CODED\_IMAGE( )

### Syntax structure

The CODED\_IMAGE( ) syntax structure is specified in Table 18.

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

Table 18 – CODED\_IMAGE( ) syntax structure

|  |  |  |
| --- | --- | --- |
| **CODED\_IMAGE( ) {** | **Descriptor** | **Reference** |
| IMAGE\_HEADER( ) |  | 8.3 |
| IsCurrPlaneAlphaFlag = FALSE |  |  |
| IMAGE\_PLANE\_HEADER( ) |  | 8.4 |
| if (ALPHA\_IMAGE\_PLANE\_FLAG) { |  |  |
| IsCurrPlaneAlphaFlag = TRUE |  |  |
| IMAGE\_PLANE\_HEADER( ) |  | 8.4 |
| } |  |  |
| if (INDEX\_TABLE\_PRESENT\_FLAG) |  |  |
| INDEX\_TABLE\_TILES( ) |  | 8.5 |
| SubsequentBytes = VLW\_ESC( ) |  | 8.2.4 |
| if (SubsequentBytes > 0) { |  |  |
| iBytes = PROFILE\_LEVEL\_INFO( ) |  | 8.6 |
| valueAdditionalBytes = SubsequentBytes − iBytes |  |  |
| for (iBytes = 0; iBytes < valueAdditionalBytes; iBytes++) |  |  |
| RESERVED\_A\_BYTE | u(8) | 8.2.3 |
| } |  |  |
| CODED\_TILES( ) |  | 8.7 |
| } |  |  |

### SubsequentBytes

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

The value of the variable valueAdditionalBytes is derived from the value of SubsequentBytes as shown in Table 18.

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

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

### RESERVED\_A\_BYTE

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

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

### VLW\_ESC( )

#### Syntax structure

The VLW\_ESC( ) syntax structure is specified in Table 19.

Table 19 – VLW\_ESC( ) syntax structure

|  |  |  |
| --- | --- | --- |
| **VLW\_ESC( ) {** | **Descriptor** | **Reference** |
| FIRST\_BYTE | u(8) | 8.2.4.2 |
| if (FIRST\_BYTE < 0xFB) { |  |  |
| SECOND\_BYTE | u(8) | 8.2.4.3 |
| iValue = FIRST\_BYTE \* 256 + SECOND\_BYTE |  |  |
| } else if (FIRST\_BYTE = = 0xFB) { |  |  |
| FOUR\_BYTES | u(32) | 8.2.4.4 |
| iValue = FOUR\_BYTES |  |  |
| } else if (FIRST\_BYTE = = 0xFC) { |  |  |
| EIGHT\_BYTES | u(64) | 8.2.4.5 |
| iValue = EIGHT\_BYTES |  |  |
| } else /\* FIRST\_BYTE is 0xFD, or 0xFE, or 0xFF \*/ |  |  |
| iValue = 0 /\* Escape Mode \*/ |  |  |
| return iValue |  |  |
| } |  |  |

#### FIRST\_BYTE

FIRST\_BYTE is an 8-bit syntax element that affects the computation of iValue as specified in 8.2.4.1.

#### SECOND\_BYTE

SECOND\_BYTE is an 8-bit syntax element that affects the computation of iValue as specified in 8.2.4.1.

#### FOUR\_BYTES

FOUR\_BYTES is a 32-bit syntax element that affects the computation of iValue as specified in 8.2.4.1.

#### EIGHT\_BYTES

EIGHT\_BYTES is a 64-bit syntax element that affects the computation of iValue as specified in 8.2.4.1.

## IMAGE\_HEADER( )

### Syntax structure

The IMAGE\_HEADER( ) syntax structure is specified in Table 20.

Table 20 – IMAGE\_HEADER( ) syntax structure

|  |  |  |
| --- | --- | --- |
| **IMAGE\_HEADER( ) {** | **Descriptor** | **Reference** |
| GDI\_SIGNATURE | u(64) | 8.3.2 |
| RESERVED\_B | u(4) | 8.3.3 |
| HARD\_TILING\_FLAG | u(1) | 8.3.4 |
| RESERVED\_C | u(3) | 8.3.5 |
| TILING\_FLAG | u(1) | 8.3.6 |
| FREQUENCY\_MODE\_CODESTREAM\_FLAG | u(1) | 8.3.7 |
| SPATIAL\_XFRM\_SUBORDINATE | u(3) | 8.3.8 |
| INDEX\_TABLE\_PRESENT\_FLAG | u(1) | 8.3.9 |
| OVERLAP\_MODE | u(2) | 8.3.10 |
| SHORT\_HEADER\_FLAG | u(1) | 8.3.11 |
| LONG\_WORD\_FLAG | u(1) | 8.3.12 |
| WINDOWING\_FLAG | u(1) | 8.3.13 |
| TRIM\_FLEXBITS\_FLAG | u(1) | 8.3.14 |
| RESERVED\_D | u(1) | 8.3.15 |
| RED\_BLUE\_NOT\_SWAPPED\_FLAG | u(1) | 8.3.16 |
| PREMULTIPLIED\_ALPHA\_FLAG | u(1) | 8.3.17 |
| ALPHA\_IMAGE\_PLANE\_FLAG | u(1) | 8.3.18 |
| OUTPUT\_CLR\_FMT | u(4) | 8.3.19 |
| OUTPUT\_BITDEPTH | u(4) | 8.3.20 |
| if (SHORT\_HEADER\_FLAG) { |  |  |
| WIDTH\_MINUS1 | u(16) | 8.3.21 |
| HEIGHT\_MINUS1 | u(16) | 8.3.22 |
| } else { |  |  |
| WIDTH\_MINUS1 | u(32) | 8.3.21 |
| HEIGHT\_MINUS1 | u(32) | 8.3.22 |
| } |  |  |
| if (TILING\_FLAG) { |  |  |
| NUM\_VER\_TILES\_MINUS1 | u(12) | 8.3.23 |
| NUM\_HOR\_TILES\_MINUS1 | u(12) | 8.3.24 |
| } |  |  |
| for (n = 0; n < NUM\_VER\_TILES\_MINUS1; n++) |  |  |
| if (SHORT\_HEADER\_FLAG) |  |  |
| TILE\_WIDTH\_IN\_MB[n] | u(8) | 8.3.25 |
| else |  |  |
| TILE\_WIDTH\_IN\_MB[n] | u(16) | 8.3.25 |
| for (n = 0; n < NUM\_HOR\_TILES\_MINUS1; n++) |  |  |
| if (SHORT\_HEADER\_FLAG) |  |  |
| TILE\_HEIGHT\_IN\_MB[n] | u(8) | 8.3.26 |
| else |  |  |
| TILE\_HEIGHT\_IN\_MB[n] | u(16) | 8.3.26 |
| if (WINDOWING\_FLAG) { |  |  |
| TOP\_MARGIN | u(6) | 8.3.27 |
| LEFT\_MARGIN | u(6) | 8.3.28 |
| BOTTOM\_MARGIN | u(6) | 8.3.29 |
| RIGHT\_MARGIN | u(6) | 8.3.30 |
| } |  |  |
| } |  |  |

### GDI\_SIGNATURE

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

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

### RESERVED\_B

RESERVED\_B is a 4-bit syntax element that shall be equal to 1 in all codestreams conforming to this version of this Specification. All other values are reserved.

NOTE – Alternative values for RESERVED\_B may be specified in the future as an indication of a codestream that is not compatible with prior decoder versions.

### HARD\_TILING\_FLAG

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

### RESERVED\_C

RESERVED\_C is a 3-bit syntax element that shall be equal to 1 in all codestreams conforming to this version of this Specification. All other values are reserved. Decoders conforming to this version of this Specification shall ignore the value of RESERVED\_C.

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

### TILING\_FLAG

TILING\_FLAG is a 1-bit syntax element. If TILING\_FLAG is equal to TRUE, both the syntax elements NUM\_VER\_TILES\_MINUS1 and NUM\_HOR\_TILES\_MINUS1 are present in the codestream. Otherwise, these syntax elements are not present, and the number of tiles is equal to 1.

### FREQUENCY\_MODE\_CODESTREAM\_FLAG

FREQUENCY\_MODE\_CODESTREAM\_FLAG is a 1-bit syntax element.

If FREQUENCY\_MODE\_CODESTREAM\_FLAG is equal to FALSE, the codestream is laid out in the spatial mode. If FREQUENCY\_MODE\_CODESTREAM\_FLAG is equal to TRUE, the codestream is laid out in the frequency mode.

### SPATIAL\_XFRM\_SUBORDINATE

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

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

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

Table 21 – Interpretation of SPATIAL\_XFRM\_SUBORDINATE

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **SPATIAL\_XFRM\_SUBORDINATE** | **RCW** | **FlipH** | **FlipV** | **Example** | **Fill** |
| **0** | 0 | 0 | 0 | A1 | TL |
| **1** | 0 | 0 | 1 | B3 | BL |
| **2** | 0 | 1 | 0 | B1 | TR |
| **3** | 0 | 1 | 1 | A3 | BR |
| **4** | 1 | 0 | 0 | A2 | RT |
| **5** | 1 | 0 | 1 | B2 | RB |
| **6** | 1 | 1 | 0 | B4 | LT |
| **7** | 1 | 1 | 1 | A4 | LB |

### INDEX\_TABLE\_PRESENT\_FLAG

INDEX\_TABLE\_PRESENT\_FLAG is a 1-bit syntax element that specifies whether the index table is present in the codestream. If FREQUENCY\_MODE\_CODESTREAM\_FLAG is equal to TRUE or NUM\_VER\_TILES\_MINUS1 is greater than 0 or NUM\_HOR\_TILES\_MINUS1 is greater than 0, it is a requirement of codestream conformance that INDEX\_TABLE\_PRESENT\_FLAG shall be equal to TRUE. If INDEX\_TABLE\_PRESENT\_FLAG is equal to TRUE, the index table is present in the codestream. Otherwise, the index table is not present in the codestream. See 8.5.

### OVERLAP\_MODE

OVERLAP\_MODE is a 2-bit syntax element that specifies the overlap processing mode.

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

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

### 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.

### LONG\_WORD\_FLAG

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

* All values of the global array variable MbDCLP[ ][ ][ ][ ]
* All values of the global array variable MBBuffer[ ][ ][ ][ ]
* All values of the global array variable PredDCLP[ ][ ][ ][ ]
* Values of global array variable ImagePlane[ ][ ][ ] in the sample reconstruction process specified in ‎9.9.

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

* All values of local variables used in the sample reconstruction process specified in clause‎9.9 except the index variables used to address the elements of arrays.

NOTE 2 – Local variables used in the output formatting process specified in ‎9.10 are not included in the scope of the constraints imposed by LONG\_WORD\_FLAG equal to FALSE.

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

* If LONG\_WORD\_FLAG is equal to TRUE, it is a requirement for codestream conformance to this Specification that the range of values of all variables associated with the decoding process (clause ‎9) shall not exceed the signed 32-bit range from −231 range to 231−1, inclusive, although the range of values of these variables may exceed the signed 16-bit range from −215 range to 215−1, inclusive. Thus all variables of the decoding process can be stored by decoders as 32 bit two's complement integers while producing output conforming to this Specification (regardless of the value of LONG\_WORD\_FLAG).
* Otherwise (LONG\_WORD\_FLAG is equal to FALSE), it is a requirement for codestream conformance to this Specification that the range of values of the specific affected variables listed above shall not exceed the signed 16-bit range from −215 range to 215−1, inclusive, and that the range of values of all intermediate expressions and other variables associated with the decoding process for the codestream shall not exceed the signed 32-bit range from −231 range to 231−1, inclusive. In this case, it is sufficient for decoders to store the affected variables of the decoding process as 16 bit two's complement integers in order to produce output images conforming to this Specification.

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

### 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 8.3.27 to 8.3.30) are present in the codestream. If WINDOWING\_FLAG is equal to TRUE, these syntax elements are present in the codestream. If WINDOWING\_FLAG is equal to FALSE, these syntax elements are not present in the codestream.

### TRIM\_FLEXBITS\_FLAG

TRIM\_FLEXBITS\_FLAG is a 1-bit syntax element that specifies whether TRIM\_FLEXBITS syntax element is present in the TILE\_SPATIAL( ) syntax structure and TILE\_FLEXBITS( ) syntax structure. If TRIM\_FLEXBITS\_FLAG is equal to TRUE, TRIM\_FLEXBITS is present. Otherwise, TRIM\_FLEXBITS is not present.

### RESERVED\_D

RESERVED\_D is a 1-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.

### RED\_BLUE\_NOT\_SWAPPED\_FLAG

RED\_BLUE\_NOT\_SWAPPED\_FLAG is a 1-bit syntax element. Its interpretation is specified as follows.

* If OUTPUT\_CLR\_FMT is equal to RGB and OUTPUT\_BITDEPTH is equal to BD5, BD565 or BD10, the value of RED\_BLUE\_NOT\_SWAPPED\_FLAG affects the operation of the output formatting process.
* Otherwise, the value of RED\_BLUE\_NOT\_SWAPPED\_FLAG shall be equal to 0 in all codestreams conforming to this version of this Specification. The value 1 is reserved. Decoders conforming to this version of this Specification shall ignore the value of RED\_BLUE\_NOT\_SWAPPED\_FLAG in this case.

NOTE – The specification of semantics for RED\_BLUE\_NOT\_SWAPPED\_FLAG was not included in the original edition of this Specification. The specification of RED\_BLUE\_NOT\_SWAPPED\_FLAG was added later to address a problem with respect to the observed behaviour of existing implementations. In principle, when OUTPUT\_CLR\_FMT is equal to RGB and OUTPUT\_BITDEPTH is equal to BD5, BD565, or BD10, the value 1 for RED\_BLUE\_NOT\_SWAPPED\_FLAG should ordinarily provide better compression capability than the value 0. However, early product implementations of this Specification have operated in a manner corresponding to the value 0.

### 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 such necessary 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.

### ALPHA\_IMAGE\_PLANE\_FLAG

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

NOTE – If ALPHA\_IMAGE\_PLANE\_FLAG is equal to FALSE, an alpha image plane can be carried as a separate image within a system or file.

### OUTPUT\_CLR\_FMT

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

Table 22 – OUTPUT\_CLR\_FMT

|  |  |
| --- | --- |
| **OUTPUT\_CLR\_FMT** | **Mnemonic** |
| 0 | YONLY |
| 1 | YUV420 |
| 2 | YUV422 |
| 3 | YUV444 |
| 4 | CMYK |
| 5 | CMYKDIRECT |
| 6 | NCOMPONENT |
| 7 | RGB |
| 8 | RGBE |
| 9-15 | RESERVED |

If IsCurrPlaneAlphaFlag is equal to TRUE, the value of OUTPUT\_CLR\_FMT shall be equal to 0.

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

### OUTPUT\_BITDEPTH

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

The values of OUTPUT\_BITDEPTH and OUTPUT\_CLR\_FMT shall be constrained as specified in 9.10.

NOTE – See 9.10.7 for more details on the representation of these formats.

Table 23 – OUTPUT\_BITDEPTH

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

### WIDTH\_MINUS1

WIDTH\_MINUS1 plus 1 is the output image width. If SHORT\_HEADER\_FLAG is equal to TRUE, WIDTH\_MINUS1 is a 16-bit syntax element. Otherwise, WIDTH\_MINUS1 is a 32-bit syntax element. When OUTPUT\_CLR\_FMT is equal to YUV420 or YUV422, it is a requirement of codestream conformance to this Specification that the value of WIDTH\_MINUS1 + 1 shall be an integer multiple of 2.

It is a requirement of codestream conformance to this Specification that the value of WIDTH\_MINUS1 + 1 + LEFT\_MARGIN + RIGHT\_MARGIN shall be an integer multiple of 16. When INTERNAL\_CLR\_FMT is equal to YUV420 or YUV422 and OVERLAP\_MODE is equal to 2, it is a requirement of codestream conformance to this Specification that the value of WIDTH\_MINUS1 + 1 + LEFT\_MARGIN + RIGHT\_MARGIN shall be greater than or equal to 32.

NOTE – Images with INTERNAL\_CLR\_FMT equal to YUV420 or YUV422 and OVERLAP\_MODE equal to 2 must have a width of at least 2 macroblocks due to the adjacent coefficient residual process described in 9.9.3.1.

### HEIGHT\_MINUS1

HEIGHT\_MINUS1 plus 1 is the output image height. If SHORT\_HEADER\_FLAG is equal to TRUE, HEIGHT\_MINUS1 is a 16-bit syntax element. Otherwise, HEIGHT\_MINUS1 is a 32-bit syntax element. When OUTPUT\_CLR\_FMT is equal to YUV420, it is a requirement of codestream conformance to this Specification that the value of HEIGHT\_MINUS1 + 1 shall be an integer multiple of 2.

It is a requirement of codestream conformance to this Specification that the value of HEIGHT\_MINUS1 + 1 + TOP\_MARGIN + BOTTOM\_MARGIN shall be an integer multiple of 16.

### NUM\_VER\_TILES\_MINUS1

NUM\_VER\_TILES\_MINUS1 is a 12-bit syntax element that is present when TILING\_FLAG is equal to TRUE, and specifies the number of tiles in a row minus 1. When NUM\_VER\_TILES\_MINUS1 is not present, its value shall be inferred to be equal to 0.

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

### NUM\_HOR\_TILES\_MINUS1

NUM\_HOR\_TILES\_MINUS1 is a 12-bit syntax element that is present when TILING\_FLAG is equal to TRUE, and specifies the number of tiles in a column minus 1. When NUM\_HOR\_TILES\_MINUS1 is not present, its value shall be inferred to be equal to 0.

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

### TILE\_WIDTH\_IN\_MB[n]

TILE\_WIDTH\_IN\_MB[n] is a syntax element that specifies the width (in macroblock units) of the n-th tile column, where the 0-th tile column is the leftmost tile column in the image, and subsequent tile columns are numbered consecutively, left to right. If SHORT\_HEADER\_FLAG is equal to TRUE, TILE\_WIDTH\_IN\_MB[n] is an 8-bit syntax element. Otherwise, it is a 16-bit syntax element.

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

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

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

|  |  |
| --- | --- |
| **DetermineLeftBoundaryofTile( ) {** | **Reference** |
| LeftMBIndexOfTile[0] = 0 |  |
| for (n = 0; n < NUM\_VER\_TILES\_MINUS1; n++) |  |
| LeftMBIndexOfTile[n+1] = LeftMBIndexOfTile[n] + TILE\_WIDTH\_IN\_MB[n] |  |
| LeftMBIndexOfTile[NUM\_VER\_TILES\_MINUS1 + 1] = MBWidth |  |
| } |  |

When INTERNAL\_CLR\_FMT is equal to YUV420 or YUV422, OVERLAP\_MODE is equal to 2, and HARD\_TILING\_FLAG is equal to TRUE, TILE\_WIDTH\_IN\_MB[n] shall be greater than or equal to 2 for all tiles.

When INTERNAL\_CLR\_FMT is equal to YUV420 or YUV422, OVERLAP\_MODE is equal to 2, and HARD\_TILING\_FLAG is equal to TRUE, MBWidth − LeftMBIndexOfTile[NUM\_VER\_TILES\_MINUS1] shall be greater than or equal to 2.

### TILE\_HEIGHT\_IN\_MB[n]

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

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

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

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

|  |  |
| --- | --- |
| **DetermineTopBoundaryofTile( ) {** | **Reference** |
| TopMBIndexOfTile[0] = 0 |  |
| for (n = 0; n < NUM\_HOR\_TILES\_MINUS1; n++) |  |
| TopMBIndexOfTile[n+1] = TopMBIndexOfTile[n] + TILE\_HEIGHT\_IN\_MB[n] |  |
| TopMBIndexOfTile[NUM\_HOR\_TILES\_MINUS1 + 1] = MBHeight |  |
| } |  |

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

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

|  |  |
| --- | --- |
| **DetermineNumMBInTile( ) {** | **Reference** |
| n = 0 |  |
| for (i = 0; i < NUM\_HOR\_TILES\_MINUS1 + 1; i++) |  |
| for (j = 0; j < NUM\_VER\_TILES\_MINUS1 + 1; j++) { |  |
| NumMBInTile[n] =   TILE\_HEIGHT\_IN\_MB[i] \* TILE\_WIDTH\_IN\_MB[j] |  |
| n++ |  |
| } |  |
| } |  |

### TOP\_MARGIN

TOP\_MARGIN is a 6-bit syntax element that is present when WINDOWING\_FLAG is equal to TRUE, and specifies the vertical offset of the top boundary of the output image relative to the top edge of the extended image. When TOP\_MARGIN is not present, its value shall be inferred to be equal to 0. When OUTPUT\_CLR\_FMT is equal to YUV420, it is a requirement of codestream conformance to this Specification that the value of TOP\_MARGIN shall be an integer multiple of 2.

### LEFT\_MARGIN

LEFT\_MARGIN is a 6-bit syntax element that is present when WINDOWING\_FLAG is equal to TRUE, and specifies the horizontal offset of the left boundary of the output image relative to the left edge of the extended image. When LEFT\_MARGIN is not present, its value shall be inferred to be equal to 0. When OUTPUT\_CLR\_FMT is equal to YUV420 or YUV422, it is a requirement of codestream conformance to this Specification that the value of LEFT\_MARGIN shall be an integer multiple of 2.

### BOTTOM\_MARGIN

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

* If HEIGHT\_MINUS1 + 1 is an integer multiple of 16, BOTTOM\_MARGIN shall be inferred to be equal to 0.
* Otherwise, BOTTOM\_MARGIN shall be inferred to be equal to 16 − ((HEIGHT\_MINUS1 + 1) % 16).

When OUTPUT\_CLR\_FMT is equal to YUV420, it is a requirement of codestream conformance to this Specification that the value of BOTTOM\_MARGIN shall be an integer multiple of 2.

### RIGHT\_MARGIN

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

* If WIDTH\_MINUS1 + 1 is an integer multiple of 16, RIGHT\_MARGIN shall be inferred to be equal to 0.
* Otherwise, RIGHT\_MARGIN shall be inferred to be equal to 16 − ((WIDTH\_MINUS1 + 1) % 16).

When OUTPUT\_CLR\_FMT is equal to YUV420 or YUV422, it is a requirement of codestream conformance to this Specification that the value of RIGHT\_MARGIN shall be an integer multiple of 2.

## IMAGE\_PLANE\_HEADER( )

### Syntax structure

The IMAGE\_PLANE\_HEADER( ) syntax structure is specified in Table 27.

|  |  |  |
| --- | --- | --- |
| Table 27 – IMAGE\_PLANE\_HEADER( ) syntax structure | | |
| **IMAGE\_PLANE\_HEADER( ) {** | **Descriptor** | **Reference** |
| INTERNAL\_CLR\_FMT | u(3) | 8.4.2 |
| SCALED\_FLAG | u(1) | 8.4.3 |
| BANDS\_PRESENT | u(4) | 8.4.4 |
| if (INTERNAL\_CLR\_FMT = = YUV444 | | INTERNAL\_CLR\_FMT = = YUV420 | |  INTERNAL\_CLR\_FMT = = YUV422) { |  |  |
| if (INTERNAL\_CLR\_FMT = = YUV420 | | INTERNAL\_CLR\_FMT = = YUV422) { |  |  |
| RESERVED\_E\_BIT | u(1) | 8.4.5 |
| CHROMA\_CENTERING\_X | u(3) | 8.4.6 |
| } else /\* INTERNAL\_CLR\_FMT = = YUV444 \*/ |  |  |
| RESERVED\_F | u(4) | 8.4.7 |
| if (INTERNAL\_CLR\_FMT = = YUV420) { |  |  |
| RESERVED\_G\_BIT | u(1) | 8.4.8 |
| CHROMA\_CENTERING\_Y | u(3) | 8.4.9 |
| } else |  |  |
| RESERVED\_H | u(4) | 8.4.10 |
| } else if (INTERNAL\_CLR\_FMT = = NCOMPONENT) { |  |  |
| NUM\_COMPONENTS\_MINUS1 | u(4) | 8.4.11 |
| if (NUM\_COMPONENTS\_MINUS1 = = 0xF) |  |  |
| NUM\_COMPONENTS\_EXTENDED\_MINUS16 | u(12) | 8.4.12 |
| else |  |  |
| RESERVED\_H | u(4) | 8.4.10 |
| } |  |  |
| if (OUTPUT\_BITDEPTH = = BD16 | | OUTPUT\_BITDEPTH = = BD16S | |   OUTPUT\_BITDEPTH = = BD32S) |  |  |
| SHIFT\_BITS | u(8) | 8.4.13 |
| if (OUTPUT\_BITEPTH = = BD32F) { |  |  |
| LEN\_MANTISSA | u(8) | 8.4.14 |
| EXP\_BIAS | i(8) | 8.4.15 |
| } |  |  |
| DC\_IMAGE\_PLANE\_UNIFORM\_FLAG | u(1) | 8.4.16 |
| if (DC\_IMAGE\_PLANE\_UNIFORM\_FLAG) |  |  |
| DC\_QP( ) |  | 8.4.22 |
| if (BANDS\_PRESENT != DCONLY) { |  |  |
| RESERVED\_I\_BIT | u(1) | 8.4.17 |
| LP\_IMAGE\_PLANE\_UNIFORM\_FLAG | u(1) | 8.4.18 |
| if (LP\_IMAGE\_PLANE\_UNIFORM\_FLAG) { |  |  |
| NumLPQPs = 1 |  |  |
| LP\_QP( ) |  | 8.4.23 |
| } |  |  |
| if (BANDS\_PRESENT != NOHIGHPASS) { |  |  |
| RESERVED\_J\_BIT | u(1) | 8.4.19 |
| HP\_IMAGE\_PLANE\_UNIFORM\_FLAG | u(1) | 8.4.20 |
| if (HP\_IMAGE\_PLANE\_UNIFORM\_FLAG) { |  |  |
| NumHPQPs = 1 |  |  |
| HP\_QP( ) |  | 8.4.24 |
| } |  |  |
| } |  |  |
| } |  |  |
| while (!IS\_BYTE\_ALIGNED( )) |  |  |
| BYTE\_ALIGNMENT\_BIT | u(1) | 8.4.21 |
| } |  |  |

### INTERNAL\_CLR\_FMT

INTERNAL\_CLR\_FMT is a 3-bit syntax element that specifies the internal colour format of the coded image as specified in Table 28. For OUTPUT\_BITDEPTH of BD16F and BD32F and OUTPUT\_CLR\_FMT of RGBE, only YUV444 shall be used. The values of INTERNAL\_CLR\_FMT and OUTPUT\_CLR\_FMT shall be constrained as specified in 9.10.

Table 28 – INTERNAL\_CLR\_FMT

|  |  |
| --- | --- |
| **INTERNAL\_CLR\_FMT** | **Mnemonic** |
| 0 | YONLY |
| 1 | YUV420 |
| 2 | YUV422 |
| 3 | YUV444 |
| 4 | YUVK |
| 5 | RESERVED |
| 6 | NCOMPONENT |
| 7 | RESERVED |

When IsCurrPlaneAlphaFlag is equal to TRUE, the value of INTERNAL\_CLR\_FMT shall be equal to 0.

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

### SCALED\_FLAG

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

### BANDS\_PRESENT

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

Table 29 – BANDS\_PRESENT

|  |  |  |
| --- | --- | --- |
| **BANDS\_PRESENT** | **Mnemonic** | **Interpretation** |
| 0 | ALL | All sub-bands are present |
| 1 | NOFLEXBITS | Flexbits is not present |
| 2 | NOHIGHPASS | Flexbits and HP are not present |
| 3 | DCONLY | Only DC is present |
| 4-15 | RESERVED |  |

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

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

|  |  |
| --- | --- |
| **DetermineNumBands( ) {** | **Reference** |
| if (BANDS\_PRESENT = = ALL) |  |
| NumBands = 4 |  |
| else if (BANDS\_PRESENT = = NOFLEXBITS) |  |
| NumBands = 3 |  |
| else if (BANDS\_PRESENT = = NOHIGHPASS) |  |
| NumBands = 2 |  |
| else /\* (BANDS\_PRESENT = = DCONLY) \*/ |  |
| NumBands = 1 |  |
| if (IsCurrPlaneAlphaFlag = = FALSE) |  |
| NumBandsOfPrimary = NumBands |  |
| } |  |

### RESERVED\_E\_BIT

RESERVED\_E\_BIT is a 1-bit syntax element and is present when INTERNAL\_CLR\_FMT is equal to YUV422 or INTERNAL\_CLR\_FMT is equal to YUV420.

When RESERVED\_E\_BIT is present, its value shall be equal to 0. The value 1 for RESERVED\_E\_BIT is reserved.

The value of RESERVED\_E\_BIT shall be ignored by decoders.

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

### CHROMA\_CENTERING\_X

CHROMA\_CENTERING\_X is a 3-bit syntax element. It is present when INTERNAL\_CLR\_FMT is equal to YUV422 or YUV420. When CHROMA\_CENTERING\_X is not present, its value should be inferred to be equal to 0.

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

For example, when CHROMA\_CENTERING\_X is equal to 0 and INTERNAL\_CLR\_FMT is equal to YUV422 or YUV420, each chroma sample is considered to be horizontally located at the same position as the left-most sample of a pair of luma samples.

The value of CHROMA\_CENTERING\_X shall be equal to 0, 1, 2, 3, 4, or 7. The values 5 and 6 are reserved. Decoders conforming to this version of this Specification should treat the values 5 and 6 as equivalent to the value 7.

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

### RESERVED\_F

RESERVED\_F is a 4-bit syntax element that is present when INTERNAL\_CLR\_FMT is equal to YUV444.

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

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

### RESERVED\_G\_BIT

RESERVED\_G\_BIT is a 1-bit syntax element that is present when INTERNAL\_CLR\_FMT is equal to YUV420.

When RESERVED\_G\_BIT is present, its value shall be equal to 0. The value 1 for RESERVED\_G\_BIT is reserved.

Decoders shall ignore the value of RESERVED\_G\_BIT.

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

### CHROMA\_CENTERING\_Y

CHROMA\_CENTERING\_Y is a 3-bit syntax element. It is present when INTERNAL\_CLR\_FMT is equal to YUV420. When CHROMA\_CENTERING\_Y is not present, its value should be inferred to be equal to 0.

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

For example, when CHROMA\_CENTERING\_Y is equal to 0 and INTERNAL\_CLR\_FMT is equal to YUV420, each chroma sample is considered to be vertically located at the same position as the upper sample of a pair of luma samples.

The value of CHROMA\_CENTERING\_Y shall be equal to 0, 1, 2, 3, 4 or 7. The values 5 and 6 are reserved. Decoders conforming to this version of this Specification should treat the values 5 and 6 as equivalent to the value 7.

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

### RESERVED\_H

RESERVED\_H is a 4-bit syntax element that is present when INTERNAL\_CLR\_FMT is equal to one of YUV444, YUV422, or NCOMPONENT.

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

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

### NUM\_COMPONENTS\_MINUS1

NUM\_COMPONENTS\_MINUS1 is a 4-bit syntax element that is present when INTERNAL\_CLR\_FMT is equal to NCOMPONENT.

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

### NUM\_COMPONENTS\_EXTENDED\_MINUS16

NUM\_COMPONENTS\_EXTENDED\_MINUS16 is a 12-bit syntax element that is present when NUM\_COMPONENTS\_MINUS1 is equal to 0xF.

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

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

|  |  |
| --- | --- |
| **DetermineNumComponents( ) {** | **Reference** |
| if (INTERNAL\_CLR\_FMT = = NCOMPONENT) |  |
| if (NUM\_COMPONENTS\_MINUS1 = = 0xF) |  |
| NumComponents =  NUM\_COMPONENTS\_EXTENDED\_MINUS16 + 16 |  |
| else |  |
| NumComponents = NUM\_COMPONENTS\_MINUS1 + 1 |  |
| else if (INTERNAL\_CLR\_FMT= = YONLY) |  |
| NumComponents = 1 |  |
| else if (INTERNAL\_CLR\_FMT = = YUV420 | |   INTERNAL\_CLR\_FMT = = YUV422 | |   INTERNAL\_CLR\_FMT = = YUV444) |  |
| NumComponents = 3 |  |
| else if (INTERNAL\_CLR\_FMT = = YUVK) |  |
| NumComponents = 4 |  |
| } |  |

### SHIFT\_BITS

SHIFT\_BITS is an 8-bit syntax element that is present when OUTPUT\_BITDEPTH is equal to BD16, BD16S or BD32S. SHIFT\_BITS is used to left-shift the sample values in the output formatting stage as specified in 9.10.7.

### LEN\_MANTISSA

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

### EXP\_BIAS

EXP\_BIAS is an 8-bit syntax element that is present when OUTPUT\_BITDEPTH is equal to BD32F. This element specifies the bias of the exponent in the representation of floating point data as specified in 9.10.7.

### DC\_IMAGE\_PLANE\_UNIFORM\_FLAG

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

### RESERVED\_I\_BIT

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

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

### LP\_IMAGE\_PLANE\_UNIFORM\_FLAG

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

### RESERVED\_J\_BIT

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

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

### HP\_IMAGE\_PLANE\_UNIFORM\_FLAG

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

### BYTE\_ALIGNMENT\_BIT

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

### DC\_QP( )

#### Syntax structure

The DC\_QP( ) syntax structure is specified in Table 32.

Table 32 – DC\_QP( ) syntax structure

|  |  |  |
| --- | --- | --- |
| **DC\_QP( ) {** | **Descriptor** | **Reference** |
| if (NumComponents != 1) |  |  |
| COMPONENT\_MODE | u(2) | 8.4.22.2 |
| if (COMPONENT\_MODE = = UNIFORM) |  |  |
| DC\_QUANT | u(8) | 8.4.22.3 |
| else if (COMPONENT\_MODE = = SEPARATE) { |  |  |
| DC\_QUANT\_LUMA | u(8) | 8.4.22.4 |
| DC\_QUANT\_CHROMA | u(8) | 8.4.22.5 |
| } else if (COMPONENT\_MODE = =   INDEPENDENT) |  |  |
| for (i = 0; i < NumComponents; i++) |  |  |
| DC\_QUANT\_CH[i] | u(8) | 8.4.22.6 |
| } |  |  |

NOTE – This function, DC\_QP( ), is called from two locations: IMAGE\_PLANE\_HEADER or TILE\_HEADER\_DC. Care should be taken to use the correct value of DC\_QUANT\_CH[ ] when quantization parameters vary on a per tile basis.

#### COMPONENT\_MODE

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

Table 33 – COMPONENT\_MODE

|  |  |
| --- | --- |
| **Value** | **COMPONENT\_MODE** |
| 0 | UNIFORM |
| 1 | SEPARATE |
| 2 | INDEPENDENT |
| 3 | RESERVED |

#### DC\_QUANT

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

#### DC\_QUANT\_LUMA

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

#### DC\_QUANT\_CHROMA

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

#### DC\_QUANT\_CH[i]

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

### LP\_QP( )

#### Syntax structure

The LP\_QP( ) syntax structure is specified in Table 34.

Table 34 – LP\_QP( ) syntax structure

|  |  |  |
| --- | --- | --- |
| **LP\_QP( ) {** | **Descriptor** | **Reference** |
| for (q = 0; q < NumLPQPs; q++) { |  |  |
| if (NumComponents != 1) |  |  |
| COMPONENT\_MODE | u(2) | 8.4.22.2 |
| if (COMPONENT\_MODE = = UNIFORM) |  |  |
| LP\_QUANT[q] | u(8) | 8.4.23.2 |
| else if (COMPONENT\_MODE = = SEPARATE) { |  |  |
| LP\_QUANT\_LUMA[q] | u(8) | 8.4.23.3 |
| LP\_QUANT\_CHROMA[q] | u(8) | 8.4.23.4 |
| } else if (COMPONENT\_MODE = = INDEPENDENT) |  |  |
| for (i = 0; i < NumComponents; i++) |  |  |
| LP\_QUANT\_CH[i][q] | u(8) | 8.4.23.5 |
| } |  |  |
| } |  |  |

NOTE – This function, LP\_QP( ), is called from two locations: IMAGE\_PLANE\_HEADER or TILE\_HEADER\_LOWPASS. Care should be taken to use the correct value of LP\_QUANT\_CH[ ][ ] when quantization parameters vary on a per tile basis.

#### LP\_QUANT[q]

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

#### LP\_QUANT\_LUMA[q]

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

#### LP\_QUANT\_CHROMA[q]

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

#### LP\_QUANT\_CH[i][q]

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

### HP\_QP( )

#### Syntax structure

The HP\_QP( ) syntax structure is specified in Table 35.

Table 35 – HP\_QP( ) syntax structure

|  |  |  |
| --- | --- | --- |
| **HP\_QP( ) {** | Descriptor | Reference |
| for (q = 0; q < NumHPQPs; q++) { |  |  |
| if (NumComponents != 1) |  |  |
| COMPONENT\_MODE | u(2) | 8.4.22.2 |
| if (COMPONENT\_MODE = = UNIFORM) |  |  |
| HP\_QUANT[q] | u(8) | 8.4.24.2 |
| else if (COMPONENT\_MODE = = SEPARATE) { |  |  |
| HP\_QUANT\_LUMA[q] | u(8) | 8.4.24.3 |
| HP\_QUANT\_CHROMA[q] | u(8) | 8.4.24.4 |
| } else if (COMPONENT\_MODE = = INDEPENDENT) |  |  |
| for (i = 0; i < NumComponents; i++) |  |  |
| HP\_QUANT\_CH[i][q] | u(8) | 8.4.24.5 |
| } |  |  |
| } |  |  |

NOTE – This function, HP\_QP( ), is called from two locations: IMAGE\_PLANE\_HEADER or TILE\_HEADER\_HIGHPASS. Care should be taken to use the correct value of HP\_QUANT\_CH[ ][ ] when quantization parameters vary on a per tile basis.

#### HP\_QUANT[q]

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

#### HP\_QUANT\_LUMA[q]

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

#### HP\_QUANT\_CHROMA[q]

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

#### HP\_QUANT\_CH[i][q]

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

## INDEX\_TABLE\_TILES( )

### Syntax structure

The INDEX\_TABLE\_TILES( ) syntax structure is specified in Table 36.

Table 36 – INDEX\_TABLE\_TILES( ) syntax structure

|  |  |  |
| --- | --- | --- |
| **INDEX\_TABLE\_TILES( ) {** | Descriptor | Reference |
| if (FREQUENCY\_MODE\_CODESTREAM\_FLAG = = FALSE) |  |  |
| valueNumIndexTableEntries = (NUM\_HOR\_TILES\_MINUS1 + 1) \*   (NUM\_VER\_TILES\_MINUS1 + 1) |  |  |
| else /\* FREQUENCY\_MODE\_CODESTREAM\_FLAG = = TRUE \*/ |  |  |
| valueNumIndexTableEntries = (NUM\_HOR\_TILES\_MINUS1 + 1) \*   (NUM\_VER\_TILES\_MINUS1 + 1) \* NumBandsOfPrimary |  |  |
| INDEX\_TABLE\_STARTCODE | u(16) | 8.5.2 |
| for (n = 0; n < valueNumIndexTableEntries; n++) |  |  |
| IndexOffsetTile[n] = VLW\_ESC( ) |  | ‎8.5.3 |
| } |  |  |

### INDEX\_TABLE\_STARTCODE

INDEX\_TABLE\_STARTCODE is a 16-bit syntax element which indicates the start of the INDEX\_TABLE\_TILES( ). This element shall have the value 0x0001. Other values of INDEX\_TABLE\_STARTCODE are reserved.

### IndexOffsetTile[n]

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

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

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

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

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

## PROFILE\_LEVEL\_INFO( )

### Syntax structure

The PROFILE\_LEVEL\_INFO( ) syntax structure is specified in 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 |  |  |
| } |  |  |

### PROFILE\_IDC

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

### LEVEL\_IDC

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

### RESERVED\_L

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

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

No particular combination of PROFILE\_IDC, LEVEL\_IDC, and RESERVED\_L shall appear more than once in the PROFILE\_LEVEL\_INFO( ) syntax structure.

### LAST\_FLAG

LAST\_FLAG (when present) is a 1-bit syntax element. It indicates whether the preceding PROFILE\_IDC, LEVEL\_IDC, and RESERVED\_L syntax elements are the last such syntax elements in the PROFILE\_LEVEL\_INFO( ) syntax structure.

## CODED\_TILES( )

### Syntax structure

The CODED\_TILES( ) syntax structure is specified in Table 38.

Table 38 – CODED\_TILES( ) syntax structure

|  |  |  |
| --- | --- | --- |
| **CODED\_TILES( ) {** | **Descriptor** | **Reference** |
| if (FREQUENCY\_MODE\_CODESTREAM\_FLAG = = FALSE) |  |  |
| for (n = 0; n < (NUM\_HOR\_TILES\_MINUS1 + 1) \*  (NUM\_VER\_TILES\_MINUS1 + 1); n++) { |  |  |
| NumMBInCurrentTile = NumMBInTile[n] |  |  |
| POS\_SEEK(IndexOffsetTile[n]) |  |  |
| TILE\_SPATIAL( ) |  | 8.7.2 |
| } |  |  |
| else { /\* FREQUENCY\_MODE\_CODESTREAM\_FLAG = = TRUE \*/ |  |  |
| for (n = 0; n < (NUM\_HOR\_TILES\_MINUS1 + 1) \*   (NUM\_VER\_TILES\_MINUS1 + 1); n++) { |  |  |
| NumMBInCurrentTile = NumMBInTile[n] |  |  |
| POS\_SEEK(IndexOffsetTile[n \* NumBandsOfPrimary]) |  |  |
| TILE\_DC( ) |  | 8.7.3 |
| } |  |  |
| if (NumBandsOfPrimary > 1) |  |  |
| for (n = 0; n < (NUM\_HOR\_TILES\_MINUS1 + 1) \*   (NUM\_VER\_TILES\_MINUS1 + 1); n++) { |  |  |
| NumMBInCurrentTile = NumMBInTile[n] |  |  |
| POS\_SEEK(IndexOffsetTile[n \* NumBandsOfPrimary + 1]) |  |  |
| TILE\_LOWPASS( ) |  | 8.7.5 |
| } |  |  |
| if (NumBandsOfPrimary > 2) |  |  |
| for (n = 0; n < (NUM\_HOR\_TILES\_MINUS1 + 1) \*   (NUM\_VER\_TILES\_MINUS1 + 1); n++) { |  |  |
| NumMBInCurrentTile = NumMBInTile[n] |  |  |
| POS\_SEEK(IndexOffsetTile[n \* NumBandsOfPrimary + 2]) |  |  |
| TILE\_HIGHPASS( ) |  | 8.7.7 |
| } |  |  |
| if (NumBandsOfPrimary > 3) |  |  |
| for (n = 0; n < (NUM\_HOR\_TILES\_MINUS1 + 1) \*   (NUM\_VER\_TILES\_MINUS1 + 1); n++) { |  |  |
| NumMBInCurrentTile = NumMBInTile[n] |  |  |
| POS\_SEEK(IndexOffsetTile[n \* NumBandsOfPrimary + 3]) |  |  |
| TILE\_FLEXBITS( ) |  | 8.7.9 |
| } |  |  |
| } |  |  |
| } |  |  |

### TILE\_SPATIAL( )

The TILE\_SPATIAL( ) syntax structure is specified in Table 39.

| Table 39 – TILE\_SPATIAL( ) syntax structure | | |
| --- | --- | --- |
| **TILE\_SPATIAL( ) {** | **Descriptor** | **Reference** |
| TILE\_STARTCODE | u(24) | 8.7.10.1 |
| ARBITRARY\_BYTE | u(8) | 8.7.10.2 |
| if (TRIM\_FLEXBITS\_FLAG) |  |  |
| TRIM\_FLEXBITS | u(4) | 8.7.10.3 |
| IsCurrPlaneAlphaFlag = FALSE |  |  |
| TILE\_HEADER\_DC( ) |  | 8.7.4 |
| if (BANDS\_PRESENT != DCONLY) { /\* BANDS\_PRESENT of Primary Plane \*/ |  |  |
| TILE\_HEADER\_LOWPASS( ) |  | 8.7.6 |
| if (BANDS\_PRESENT != NOHIGHPASS) |  |  |
| TILE\_HEADER\_HIGHPASS( ) |  | 8.7.8 |
| } |  |  |
| if (ALPHA\_IMAGE\_PLANE\_FLAG) { |  |  |
| IsCurrPlaneAlphaFlag = TRUE |  |  |
| TILE\_HEADER\_DC( ) |  | 8.7.4 |
| if (BANDS\_PRESENT != DCONLY) {   /\* BANDS\_PRESENT of Alpha plane \*/ |  |  |
| TILE\_HEADER\_LOWPASS( ) |  | 8.7.6 |
| if (BANDS\_PRESENT != NOHIGHPASS) |  |  |
| TILE\_HEADER\_HIGHPASS( ) |  | 8.7.8 |
| } |  |  |
| } |  |  |
| for (n = 0; n < NumMBInCurrentTile; n++) { |  |  |
| IsCurrPlaneAlphaFlag = FALSE |  |  |
| if (BANDS\_PRESENT != DCONLY) { |  |  |
| if (NumLPQPs > 1 && USE\_DC\_QP\_FLAG = = FALSE) |  |  |
| LP\_QP\_INDEX[n] = DECODE\_QP\_INDEX(NumLPQPs) |  | 8.7.10.10 |
| if (BANDS\_PRESENT != NOHIGHPASS && NumHPQPs > 1 &&   USE\_LP\_QP\_FLAG = = FALSE) |  |  |
| HP\_QP\_INDEX[n] = DECODE\_QP\_INDEX(NumHPQPs) |  | 8.7.10.10 |
| } |  |  |
| MB\_DC( ) |  | 8.7.11 |
| if (BANDS\_PRESENT != DCONLY) { |  |  |
| MB\_LP( ) |  | 8.7.16.1 |
| if (BANDS\_PRESENT != NOHIGHPASS) { |  |  |
| MB\_CBPHP( ) |  | 8.7.17.2 |
| MB\_HP\_FLEX( ) |  | 8.7.18.3 |
| } |  |  |
| } |  |  |
| if (ALPHA\_IMAGE\_PLANE\_FLAG) { |  |  |
| IsCurrPlaneAlphaFlag = TRUE |  |  |
| if (BANDS\_PRESENT != DCONLY) {  /\* BANDS\_PRESENT of Alpha plane\*/ |  |  |
| if (NumLPQPs > 1 && USE\_DC\_QP\_FLAG = = FALSE) |  |  |
| LP\_QP\_INDEX[n] = DECODE\_QP\_INDEX(NumLPQPs) |  | 8.7.10.10 |
| if (BANDS\_PRESENT != NOHIGHPASS && NumHPQPs > 1 &&   USE\_LP\_QP\_FLAG = = FALSE) |  |  |
| HP\_QP\_INDEX[n] = DECODE\_QP\_INDEX(NumHPQPs) |  | 8.7.10.10 |
| } |  |  |
| MB\_DC( ) |  | 8.7.11 |
| if (BANDS\_PRESENT != DCONLY) { |  |  |
| MB\_LP( ) |  | 8.7.16.1 |
| if (BANDS\_PRESENT != NOHIGHPASS) { |  |  |
| MB\_CBPHP( ) |  | 8.7.17.2 |
| MB\_HP\_FLEX( ) |  | 8.7.18.3 |
| } |  |  |
| } |  |  |
| } /\* for if (ALPHA\_IMAGE\_PLANE\_FLAG) \*/ |  |  |
| } /\* for (n = 0; n < NumMBInCurrentTile; n++) \*/ |  |  |
| while (!IS\_BYTE\_ALIGNED( )) |  |  |
| BYTE\_ALIGNMENT\_BIT | u(1) | 8.4.21 |
| } |  |  |

### TILE\_DC( )

The TILE\_DC( ) syntax structure is specified in Table 40.

Table 40 – TILE\_DC( ) syntax structure

|  |  |  |
| --- | --- | --- |
| **TILE\_DC( ) {** | **Descriptor** | **Reference** |
| TILE\_STARTCODE | u(24) | 8.7.10.1 |
| ARBITRARY\_BYTE | u(8) | 8.7.10.2 |
| IsCurrPlaneAlphaFlag = FALSE |  |  |
| TILE\_HEADER\_DC( ) |  | 8.7.4 |
| if (ALPHA\_IMAGE\_PLANE\_FLAG) { |  |  |
| IsCurrPlaneAlphaFlag = TRUE |  |  |
| TILE\_HEADER\_DC( ) |  | 8.7.4 |
| } |  |  |
| for (n = 0; n < NumMBInCurrentTile; n++) { |  |  |
| IsCurrPlaneAlphaFlag = FALSE |  |  |
| MB\_DC( ) |  | 8.7.11 |
| if (ALPHA\_IMAGE\_PLANE\_FLAG) { |  |  |
| IsCurrPlaneAlphaFlag = TRUE |  |  |
| MB\_DC( ) |  | 8.7.11 |
| } |  |  |
| } |  |  |
| while (!IS\_BYTE\_ALIGNED( )) |  |  |
| BYTE\_ALIGNMENT\_BIT | u(1) | 8.4.21 |
| } |  |  |

### TILE\_HEADER\_DC( )

The TILE\_HEADER\_DC( ) syntax structure is specified in Table 41.

Table 41 – TILE\_HEADER\_DC( ) syntax structure

|  |  |  |
| --- | --- | --- |
| **TILE\_HEADER\_DC( ) {** | **Descriptor** | **Reference** |
| if (DC\_IMAGE\_PLANE\_UNIFORM\_FLAG = = FALSE) |  |  |
| DC\_QP( ) |  | 8.4.22 |
| } |  |  |

### TILE\_LOWPASS( )

The TILE\_LOWPASS( ) syntax structure is specified in 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 |
| } |  |  |

### TILE\_HEADER\_LOWPASS( )

The TILE\_HEADER\_LOWPASS( ) syntax structure is specified in Table 43.

Table 43 – TILE\_HEADER\_LOWPASS( ) syntax structure

|  |  |  |
| --- | --- | --- |
| **TILE\_HEADER\_LOWPASS( ) {** | **Descriptor** | **Reference** |
| if (LP\_IMAGE\_PLANE\_UNIFORM\_FLAG = = FALSE) { |  |  |
| USE\_DC\_QP\_FLAG | u(1) | 8.7.10.4 |
| if (USE\_DC\_QP\_FLAG) |  |  |
| NumLPQPs = 1 |  |  |
| else { |  |  |
| NUM\_LP\_QPS\_MINUS1 | u(4) | 8.7.10.5 |
| NumLPQPs = NUM\_LP\_QPS\_MINUS1 + 1 |  |  |
| LP\_QP( ) |  | 8.4.23 |
| } |  |  |
| } |  |  |
| } |  |  |

### TILE\_HIGHPASS( )

The TILE\_HIGHPASS( ) syntax structure is specified in 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 |
| } |  |  |

### TILE\_HEADER\_HIGHPASS( )

The TILE\_HEADER\_HIGHPASS( ) syntax structure is specified in Table 45.

Table 45 – TILE\_HEADER\_HIGHPASS( ) syntax structure

|  |  |  |
| --- | --- | --- |
| **TILE\_HEADER\_HIGHPASS( ) {** | **Descriptor** | **Reference** |
| if (HP\_IMAGE\_PLANE\_UNIFORM\_FLAG = = FALSE) { |  |  |
| USE\_LP\_QP\_FLAG | u(1) | 8.7.10.6 |
| if (USE\_LP\_QP\_FLAG) |  |  |
| NumHPQPs = NumLPQPs |  |  |
| else { |  |  |
| NUM\_HP\_QPS\_MINUS1 | u(4) | 8.7.10.7 |
| NumHPQPs = NUM\_HP\_QPS\_MINUS1 + 1 |  |  |
| HP\_QP( ) |  | 8.4.24 |
| } |  |  |
| } |  |  |
| } |  |  |

### TILE\_FLEXBITS( )

The TILE\_FLEXBITS( ) syntax structure is specified in Table 46.

Table 46 – TILE\_FLEXBITS( ) syntax structure

|  |  |  |
| --- | --- | --- |
| **TILE\_FLEXBITS( ) {** | **Descriptor** | **Reference** |
| TILE\_STARTCODE | u(24) | 8.7.10.1 |
| ARBITRARY\_BYTE | u(8) | 8.7.10.2 |
| if (TRIM\_FLEXBITS\_FLAG) |  |  |
| TRIM\_FLEXBITS | u(4) | 8.7.10.3 |
| for (n = 0; n < NumMBInCurrentTile; n++) { |  |  |
| IsCurrPlaneAlphaFlag = FALSE |  |  |
| if (BANDS\_PRESENT = = ALL)  /\* BANDS\_PRESENT of primary image plane \*/ |  |  |
| MB\_FLEXBITS( ) |  | 8.7.19.1 |
| if (ALPHA\_IMAGE\_PLANE\_FLAG) { |  |  |
| IsCurrPlaneAlphaFlag = TRUE |  |  |
| if (BANDS\_PRESENT = = ALL)  /\* BANDS\_PRESENT of alpha image plane \*/ |  |  |
| MB\_FLEXBITS( ) |  | 8.7.19.1 |
| } |  |  |
| } |  |  |
| while (!IS\_BYTE\_ALIGNED( )) |  |  |
| BYTE\_ALIGNMENT\_BIT | u(1) | 8.4.21 |
| } |  |  |

### Tile-level semantics

#### TILE\_STARTCODE

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

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

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

#### ARBITRARY\_BYTE

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

#### TRIM\_FLEXBITS

TRIM\_FLEXBITS is a 4-bit syntax element that is present if TRIM\_FLEXBITS\_FLAG is equal to TRUE. Otherwise, TRIM\_FLEXBITS shall be inferred to be equal to 0.

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

#### USE\_DC\_QP\_FLAG

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

#### NUM\_LP\_QPS\_MINUS1

NUM\_LP\_QPS\_MINUS1 is a 4-bit syntax element that is present if LP\_IMAGE\_PLANE\_UNIFORM\_FLAG is equal to FALSE. This syntax element specifies the number of LP band QPs, per colour component in each tile, minus 1.

#### USE\_LP\_QP\_FLAG

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

#### NUM\_HP\_QPS\_MINUS1

NUM\_HP\_QPS\_MINUS1 is a 4-bit syntax element that is present if HP\_IMAGE\_PLANE\_UNIFORM\_FLAG is equal to FALSE. This syntax element specifies the number of HP band QPs, per colour component in each tile, minus 1.

#### LP\_QP\_INDEX[n]

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

#### HP\_QP\_INDEX[n]

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

* If USE\_LP\_QP\_FLAG is equal to TRUE, HP\_QP\_INDEX[n] shall be inferred to be equal to LP\_QP\_INDEX[n].
* Otherwise, HP\_QP\_INDEX[n] shall be inferred to be equal to 0.

#### DECODE\_QP\_INDEX( )

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

The syntax structure DECODE\_QP\_INDEX( ) is specified in Table 47.

Table 47 – DECODE\_QP\_INDEX( ) syntax structure

|  |  |  |
| --- | --- | --- |
| **DECODE\_QP\_INDEX(iNumQP) {** | Descriptor | Reference |
| iBitsQPIndex[ ] = {0, 0, 1, 1, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4} |  |  |
| iBits = iBitsQPIndex[iNumQP] |  |  |
| IS\_QPINDEX\_NONZERO\_FLAG | u(1) | 8.7.10.11 |
| if (IS\_QPINDEX\_NONZERO\_FLAG = = FALSE) |  |  |
| iQPIndex = 0 |  |  |
| else { /\* iBits > 0 as iNumQP > 1 \*/ |  |  |
| QPINDEX\_REF | u(iBits) | 8.7.10.12 |
| iQPIndex = QPINDEX\_REF + 1 |  |  |
| } |  |  |
| return iQPIndex |  |  |
| } |  |  |

#### IS\_QPINDEX\_NONZERO\_FLAG

IS\_QPINDEX\_NONZERO\_FLAG is a 1-bit syntax element. If IS\_QPINDEX\_NONZERO\_FLAG is equal to TRUE, the QP index is derived from the syntax element QPINDEX\_REF. Otherwise, the QP index is set to 0.

#### QPINDEX\_REF

QPINDEX\_REF is a syntax element that specifies the QP index when IS\_QPINDEX\_NONZERO\_FLAG is equal to TRUE. The value of QPINDEX\_REF shall be in the range of 0 to iNumQP−2. All other values are reserved.

### MB\_DC( )

The MB\_DC( ) syntax structure is specified in Table 48.

Table 48 – MB\_DC( ) syntax structure

|  |  |  |
| --- | --- | --- |
| **MB\_DC( ) {** | **Descriptor** | **Reference** |
| /\* IsCurrPlaneAlphaFlag is equal to TRUE for parsing alpha image plane,   and IsCurrPlaneAlphaFlag is equal to FALSE for parsing primary   image plane \*/ |  |  |
| iBand = 0 /\* 0 = DC band, 1 = LP band, 2 = HP band \*/ |  |  |
| bInitializeContext = (IsMBLeftEdgeOfTileFlag && IsMBTopEdgeOfTileFlag) |  |  |
| if (bInitializeContext) { |  |  |
| InitializeDCVLC( ) |  | 8.8.3.1 |
| InitializeModelMB(ModelDC, iBand) |  | 8.12.1 |
| } |  |  |
| iLapMean[ ] = {0, 0} |  |  |
| if (INTERNAL\_CLR\_FMT = = YONLY | |   INTERNAL\_CLR\_FMT = = YUVK | |   INTERNAL\_CLR\_FMT = = NCOMPONENT) |  |  |
| for (n=0; n < NumComponents; n++) { |  |  |
| IS\_DC\_CH\_FLAG | u(1) | 8.7.14.1 |
| bAbsLevel = IS\_DC\_CH\_FLAG |  |  |
| m = 0 |  |  |
| if (n != 0) |  |  |
| m = 1 |  |  |
| if (bAbsLevel) |  |  |
| iLapMean[m] = iLapMean[m] + 1 |  |  |
| bChroma = FALSE /\* Luma \*/ |  |  |
| DCInput[n] =   DECODE\_DC(ModelDC.MBits[m], iBand, bChroma, bAbsLevel) |  | 8.7.12 |
| } |  |  |
| else { /\* INTERNAL\_CLR\_FMT is not YONLY, YUVK,   or NCOMPONENT \*/ |  |  |
| VAL\_DC\_YUV /\* Parse with VAL\_DC\_YUV Code table \*/ | e(v) | 8.7.14.2 |
| /\* Luma (Y) DC Parsing \*/ |  |  |
| bAbsLevel = ((VAL\_DC\_YUV & 4) != 0) |  |  |
| if (bAbsLevel) |  |  |
| iLapMean[0] = iLapMean[0] + 1 |  |  |
| bChroma = FALSE /\* i.e., Luma \*/ |  |  |
| DCInput[0] =   DECODE\_DC(ModelDC.MBits[0], iBand, bChroma, bAbsLevel) |  | 8.7.12 |
| /\* First chroma (U) DC Parsing \*/ |  |  |
| bAbsLevel = ((VAL\_DC\_YUV & 2) != 0) |  |  |
| if (bAbsLevel) |  |  |
| iLapMean[1] = iLapMean[1] + 1 |  |  |
| bChroma = TRUE /\* i.e., Chroma \*/ |  |  |
| DCInput[1] =   DECODE\_DC(ModelDC.MBits[1], iBand, bChroma, bAbsLevel) |  | 8.7.12 |
| /\* Second chroma (V) DC Parsing \*/ |  |  |
| bAbsLevel = ((VAL\_DC\_YUV & 1) != 0) |  |  |
| if (bAbsLevel) |  |  |
| iLapMean[1] = iLapMean[1] + 1 /\* Same index for U and V \*/ |  |  |
| bChroma = TRUE /\* i.e., Chroma \*/ |  |  |
| DCInput[2] =   DECODE\_DC(ModelDC.MBits[1], iBand, bChroma, bAbsLevel) |  | 8.7.12 |
| } |  |  |
| UpdateModelMB(iLapMean[ ], ModelDC, iBand) |  | 8.12.2 |
| bResetContext = (MBx = = (LeftMBIndexOfTile[TileIndexx + 1] − 1) | |   (MBx − LeftMBIndexOfTile[TileIndexx]) % 16 = = 0) |  |  |
| if (bResetContext) |  |  |
| AdaptDC( ) |  | 8.8.4.1 |
| } |  |  |

### DECODE\_DC( )

Table 49 – DECODE\_DC( ) syntax structure

|  |  |  |
| --- | --- | --- |
| **DECODE\_DC(iModelBits, iBand, bChroma, bAbsLevel) {** | **Descriptor** | **Reference** |
| iDC = 0 |  |  |
| iContext = 0 |  |  |
| if (bAbsLevel) |  |  |
| iDC = DECODE\_ABS\_LEVEL(iBand, bChroma, iContext) − 1 |  | 8.7.13 |
| if (iModelBits) { |  |  |
| DC\_REF | u(iModelBits) | 8.7.14.3 |
| iDCRef = DC\_REF |  |  |
| iDC = (iDC<< iModelBits) | iDCRef |  |  |
| } |  |  |
| if (iDC != 0) { |  |  |
| SIGN\_FLAG | u(1) | 8.7.14.4 |
| if (SIGN\_FLAG) |  |  |
| iDC = −iDC |  |  |
| } |  |  |
| return iDC |  |  |
| } |  |  |

### DECODE\_ABS\_LEVEL( )

Table 50 – DECODE\_ABS\_LEVEL( ) syntax structure

| **DECODE\_ABS\_LEVEL(iBand, bChroma, iContext) {** | **Descriptor** | **Reference** |
| --- | --- | --- |
| /\* **sAdaptVLC** is local instance of AdaptiveVLC data structure \*/ |  |  |
| if (iBand = = 0) /\* DC \*/ |  |  |
| if (bChroma) |  |  |
| **sAdaptVLC** = **AbsLevelIndDCChr** |  |  |
| else |  |  |
| **sAdaptVLC** = **AbsLevelIndDCLum** |  |  |
| else if (iBand = = 1) /\* LP \*/ |  |  |
| if (iContext) |  |  |
| **sAdaptVLC** = **AbsLevelIndLP1** |  |  |
| else |  |  |
| **sAdaptVLC** = **AbsLevelIndLP0** |  |  |
| else if (iBand = = 2) /\* HP \*/ |  |  |
| if (iContext) |  |  |
| **sAdaptVLC** = **AbsLevelIndHP1** |  |  |
| else |  |  |
| **sAdaptVLC** = **AbsLevelIndHP0** |  |  |
| iRemap[ ] = {2, 3, 4, 6, 10, 14} |  |  |
| iFixedLen[ ] = {0, 0, 1, 2, 2, 2} |  |  |
| ABS\_LEVEL\_INDEX /\* Parse with table indexed by **sAdaptVLC**.TableIndex \*/ | ae(v) | ‎8.7.14.5 |
| **sAdaptVLC**.DiscrimVal1 += AbslevelIndexDelta[0][ABS\_LEVEL\_INDEX] |  | Table 86 |
| if (ABS\_LEVEL\_INDEX < 6) { |  |  |
| iFixed = iFixedLen[ABS\_LEVEL\_INDEX] |  |  |
| iLevel = iRemap[ABS\_LEVEL\_INDEX] |  |  |
| if (iFixed > 0) { |  |  |
| LEVEL\_REF | u(iFixed) | 8.7.14.6 |
| iLevel += LEVEL\_REF |  |  |
| } |  |  |
| } else { /\* Escape mode \*/ |  |  |
| FIXED\_NUM | u(4) | ‎8.7.14.7 |
| iFixed = FIXED\_NUM + 4 |  |  |
| if (iFixed = = 19) { |  |  |
| FIXED\_NUM\_EXT | u(2) | 8.7.14.8 |
| iFixed += FIXED\_NUM\_EXT |  |  |
| if (iFixed = = 22) { |  |  |
| FIXED\_NUM\_EXT2 | u(3) | 8.7.14.9 |
| iFixed += FIXED\_NUM\_EXT2 |  |  |
| } |  |  |
| } |  |  |
| LEVEL\_REF | u(iFixed) | 8.7.14.6 |
| iLevel = 2 + (1 << iFixed) + LEVEL\_REF |  |  |
| } |  |  |
| return iLevel |  |  |
| } |  |  |

### Macroblock DC( ) semantics

#### IS\_DC\_CH\_FLAG

IS\_DC\_CH\_FLAG is a 1-bit syntax element that is present if INTERNAL\_CLR\_FMT is one of YONLY, YUVK or NCOMPONENT. If IS\_DC\_CH\_FLAG is equal to TRUE, the variable-length coded part of the DC coefficient of the corresponding colour component is specified in the codestream. If IS\_DC\_CH\_FLAG is equal to FALSE, the variable-length coded part of the DC coefficient of the corresponding colour component is equal to 0.

#### VAL\_DC\_YUV

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

Table 51 – Code table for VAL\_DC\_YUV

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

#### DC\_REF

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

#### SIGN\_FLAG

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

#### ABS\_LEVEL\_INDEX

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

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

Table 52 – Code table for ABS\_LEVEL\_INDEX

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

#### LEVEL\_REF

LEVEL\_REF is a syntax element which specifies the VLC refinement. The number of bits, iFixed, needed to specify this syntax element is computed as specified in 8.7.13 from ABS\_LEVEL\_INDEX if ABS\_LEVEL\_INDEX is less than 6 or from FIXED\_NUM, FIXED\_NUM\_EXT and FIXED\_NUM\_EXT2 if ABS\_LEVEL\_INDEX is greater than or equal to 6.

#### FIXED\_NUM

FIXED\_NUM is a 4-bit syntax element that is present if ABS\_LEVEL\_INDEX is equal to 6. It specifies the number of bits needed to specify the initial level value.

#### FIXED\_NUM\_EXT

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

#### FIXED\_NUM\_EXT2

FIXED\_NUM\_EXT2 is a 3-bit syntax element that is present if FIXED\_NUM is equal to 15 and FIXED\_NUM\_EXT is equal to 3. It specifies the number of additional extension bits needed to specify the initial level value.

### Macroblock lowpass

### General

This subclause specifies the derivation of the LP coefficients of the blocks in a macroblock. The presence of non-zero entropy coded LP coefficients, i.e. coded block pattern lowpass, in a macroblock is represented by the variable iCBPLP as computed, as specified in Table 53, from the syntax elements CBPLP\_YUV1 or CBPLP\_YUV2 or CBPLP\_CH\_BIT.

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

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

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

#### MB\_LP( )

The MB\_LP( ) syntax structure is specified in Table 53.

Table 53 – MB\_LP( ) syntax structure

| **MB\_LP( ) {** | **Descriptor** | **Reference** |
| --- | --- | --- |
| /\* IsCurrPlaneAlphaFlag is equal to TRUE for parsing alpha image plane, and   IsCurrPlaneAlphaFlag is equal to FALSE for parsing primary image plane \*/ |  |  |
| iBand = 1 /\* 0 = DC 1 = LP, 2 = HP \*/ |  |  |
| iTranspose444[ ] = {0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15} |  |  |
| iTranspose422[ ] = {0, 2, 1, 3, 4, 6, 5, 7} |  |  |
| iTranspose420[ ] = {0, 2, 1, 3} |  |  |
| bInitializeContext = (IsMBLeftEdgeOfTileFlag && IsMBTopEdgeOfTileFlag) |  |  |
| if (bInitializeContext) { |  |  |
| InitializeCountCBPLP( ) |  | 8.9.2 |
| InitializeLPVLC( ) |  | 8.8.3.2 |
| InitializeAdaptiveScanLP( ) |  | ‎8.11.2 |
| InitializeModelMB(ModelLP, iBand) |  | 8.12.1 |
| } |  |  |
| bResetTotals = ((MBx − LeftMBIndexOfTile[TileIndexx]) % 16) = = 0) |  |  |
| if (bResetTotals) |  |  |
| ResetTotalsAdaptiveScanLP( ) |  | 8.11.4 |
| iLapMean[ ] = {0, 0} |  |  |
| if (INTERNAL\_CLR\_FMT = = YUV422 | |   INTERNAL\_CLR\_FMT = = YUV420) |  |  |
| iFullPlanes = 2 |  |  |
| else |  |  |
| iFullPlanes = NumComponents |  |  |
| if (INTERNAL\_CLR\_FMT = = YUV420 | |   INTERNAL\_CLR\_FMT = = YUV422 | |   INTERNAL\_CLR\_FMT = = YUV444) { |  |  |
| iMax = iFullPlanes \* 4 − 5 /\* Max value of CBPLP \*/ |  |  |
| if (CountZeroCBPLP <= 0 | | CountMaxCBPLP < 0) { |  |  |
| CBPLP\_YUV1 | e(v) | 8.7.16.3.1 |
| if (CountMaxCBPLP < CountZeroCBPLP) |  |  |
| iCBPLP = iMax − CBPLP\_YUV1 |  |  |
| else |  |  |
| iCBPLP = CBPLP\_YUV1 |  |  |
| } else { |  |  |
| CBPLP\_YUV2 | u(iFullPlanes) | 8.7.16.3.2 |
| iCBPLP = CBPLP\_YUV2 |  |  |
| } |  |  |
| UpdateCountCBPLP(iCBPLP, iMax) |  | 8.9.3 |
| } else { |  |  |
| iCBPLP = 0 |  |  |
| for (n=0; n < NumComponents; n++) { |  |  |
| CBPLP\_CH\_BIT | u(1) | 8.7.16.3.3 |
| iCBPLP |= (CBPLP\_CH\_BIT << n) |  |  |
| } |  |  |
| } |  |  |
| for (n = 0; n < NumComponents; n++) { |  |  |
| if (INTERNAL\_CLR\_FMT = = YUV420) |  |  |
| jMax = 3 |  |  |
| else if (INTERNAL\_CLR\_FMT = = YUV422) |  |  |
| jMax = 7 |  |  |
| else |  |  |
| jMax = 15 |  |  |
| for (j = 0; j <= jMax; j++) |  |  |
| LPInput[k][j] = 0 |  |  |
| } |  |  |
| for (n = 0; n < iFullPlanes; n++) { |  |  |
| if (n = = 0) |  |  |
| iIndex = 0 |  |  |
| else |  |  |
| iIndex = 1 |  |  |
| iNumNonZero = 0 |  |  |
| if ((iCBPLP >> n) & 1) { |  |  |
| for (i = 0; i < 32; i++) |  |  |
| iRLCoeffs[i] = 0 |  |  |
| iLocation = 1 |  |  |
| if ((INTERNAL\_CLR\_FMT = = YUV420) && n) |  |  |
| iLocation = 10 |  |  |
| if ((INTERNAL\_CLR\_FMT = = YUV422) && n) |  |  |
| iLocation = 2 |  |  |
| iNumNonZero =   DECODE\_BLOCK(iIndex, iRLCoeffs[ ], iBand, iLocation) |  | 8.7.18.5 |
| if ((INTERNAL\_CLR\_FMT = = YUV420 | |   INTERNAL\_CLR\_FMT = = YUV422) && n) { |  |  |
| iTemp[14] = 0 /\* Initializing the array iTemp to zero. \*/ |  |  |
| iRemapArr[ ] = {4, 1, 2, 3, 5, 6, 7} |  |  |
| iRemapOffset = 0 |  |  |
| if (INTERNAL\_CLR\_FMT = = YUV420) |  |  |
| iRemapOffset = 1 |  |  |
| if (INTERNAL\_CLR\_FMT = = YUV422) |  |  |
| iCountChr = 14 |  |  |
| else |  |  |
| iCountChr = 6 |  |  |
| i = 0 |  |  |
| for (k = 0; k< iNumNonZero; k++) { |  |  |
| i += iRLCoeffs[k \* 2] |  |  |
| iTemp[i] = iRLCoeffs[k \* 2 + 1] |  |  |
| i++ |  |  |
| } |  |  |
| for (k = 0; k < iCountChr; k++) { |  |  |
| iRemap = iRemapArr[(k >> 1) + iRemapOffset] |  |  |
| if (INTERNAL\_CLR\_FMT = = YUV420) |  |  |
| LPInput[(k & 1) + 1][iTranspose420[iRemap]] = iTemp[k] |  |  |
| else |  |  |
| LPInput[(k & 1) + 1][iTranspose422[iRemap]] = iTemp[k] |  |  |
| } |  |  |
| } else { |  |  |
| i = 1 |  |  |
| for (k = 0; k< iNumNonZero; k++) { |  |  |
| i += iRLCoeffs[k\*2] |  |  |
| AdaptiveLPScan(n, i, iRLCoeffs[k \* 2 + 1]) /\* Updates LPInput \*/ |  | 8.11.6 |
| i++ |  |  |
| } |  |  |
| } |  |  |
| } /\* if ((iCBPLP>>n) & 1) \*/ |  |  |
| iModelBits = ModelLP.MBits[iIndex] |  |  |
| iLapMean[iIndex] += iNumNonZero |  |  |
| if (iModelBits) |  |  |
| if ((INTERNAL\_CLR\_FMT= =YUV420) && n) |  |  |
| for (k = 1; k < 4; k++) { |  |  |
| LPInput[1][iTranspose420[k]]=   REFINE\_LP(LPInput[1][iTranspose420[k]], iModelBits) |  | 8.7.16.2 |
| LPInput[2][iTranspose420[k]] =   REFINE\_LP(LPInput[2][iTranspose420[k]], iModelBits) |  | 8.7.16.2 |
| } |  |  |
| else if ((INTERNAL\_CLR\_FMT= =YUV422) && n) |  |  |
| for (k = 1; k < 8; k++) { |  |  |
| LPInput[1][iTranspose422[k]]=   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 |
| } |  |  |

#### REFINE\_LP( )

The REFINE\_LP( ) syntax structure is specified in Table 54.

Table 54 – REFINE\_LP( ) syntax structure

|  |  |  |
| --- | --- | --- |
| **REFINE\_LP(iCoeff, iModelBits) {** | **Descriptor** | **Reference** |
| COEFF\_REF | u(iModelBits) | 8.7.16.3.4 |
| if (iCoeff > 0) { |  |  |
| iCoeff <<= iModelBits |  |  |
| iCoeff += COEFF\_REF |  |  |
| } else if (iCoeff < 0) { |  |  |
| iCoeff <<= iModelBits |  |  |
| iCoeff −= COEFF\_REF |  |  |
| } else { |  |  |
| iCoeff = COEFF\_REF |  |  |
| if (iCoeff) { |  |  |
| SIGN\_FLAG | u(1) | 8.7.14.4 |
| if (SIGN\_FLAG) |  |  |
| iCoeff = −iCoeff |  |  |
| } |  |  |
| } |  |  |
| return iCoeff |  |  |
| } |  |  |

#### MB\_LP( ) and REFINE\_LP( ) semantics

##### CBPLP\_YUV1

CBPLP\_YUV1 is a syntax element that is present if INTERNAL\_CLR\_FMT is one of YUV420, YUV422 or YUV444, and also CountZeroCBPLP is less than or equal to 0 or CountMaxCBPLP is less than 0; it jointly specifies the coded block pattern lowpass of the Y, U and V colour components as follows:

* if INTERNAL\_CLR\_FMT is YUV444, the parsing of CBPLP\_YUV1 is specified in Table 55:
* if INTERNAL\_CLR\_FMT is YUV420 or YUV422, the parsing of CBPLP\_YUV1 is specified in Table 56.

NOTE – If (CountZeroCBPLP > 0 && CountMaxCBPLP >= 0), the coded block pattern lowpass band is computed as specified in 8.7.16.1.

Table 55 – Code table for CBPLP\_YUV1 when INTERNAL\_CLR\_FMT is equal toYUV444

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

Table 56 – Code table for CBPLP\_YUV1 when INTERNAL\_CLR\_FMT is equal to YUV420 or YUV422

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

##### CBPLP\_YUV2

CBPLP\_YUV2 is a syntax element that is present when INTERNAL\_CLR\_FMT is equal to YUV420, YUV422 or YUV444, and CountZeroCBPLP is greater than 0 and CountMaxCBPLP is greater than or equal to 0. The number of bits required to specify this syntax element is specified by iFullPlanes.

##### CBPLP\_CH\_BIT

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

##### COEFF\_REF

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

### Coded block pattern highpass (CBPHP)

#### General

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

First step: the syntax element NUM\_CBPHP and REFINE\_CBPHP( ) process specify the residual CBPHP status of block groups where each block group consists of multiple blocks as specified in the following.

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

If INTERNAL\_CLR\_FMT is equal to YONLY, YUVK or NCOMPONENT, there is a NUM\_CBPHP syntax for each colour component in that macroblock, and each block group consists of 2×2 groups of blocks in that colour component. Thus, there are 4 block groups for each colour component. NUM\_CBPHP takes a value between 0 and 4 and specifies the number of block groups (in that colour component) where CBPHP residual values are non-zero, and the REFINE\_CBPHP( ) process is invoked to determine which of the block groups has non-zero residual CBPHP values.

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

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

If INTERNAL\_CLR\_FMT is equal to YUV444 or YUV422, the chroma component with non-zero residual CBPHP has multiple blocks. If INTERNAL\_CLR\_FMT is equal to YUV444, the syntax element NUM\_CH\_BLK and the process REFINE\_CBPHP( ) specify the chroma blocks that have non-zero residual CBPHP. If INTERNAL\_CLR\_FMT is equal to YUV422, the syntax element CBPHP\_CH\_BLK specifies the chroma blocks that have non-zero residual CBPHP.

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

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

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

#### MB\_CBPHP( )

The MB\_CBPHP( ) syntax structure is specified in Table 57.

Table 57 – MB\_CBPHP( ) syntax structure

| **MB\_CBPHP( ) {** | **Descriptor** | **Reference** |
| --- | --- | --- |
| /\* IsCurrPlaneAlphaFlag is equal to TRUE for parsing alpha image plane, and   IsCurrPlaneAlphaFlag is equal to FALSE for parsing primary image plane \*/ |  |  |
| /\* **sAdaptVLC** is local instance of AdaptiveVLC data structure \*/ |  |  |
| iFLC[ ] = {0, 2, 1, 2, 2, 0} |  |  |
| iOff[ ] = {0, 4, 2, 8, 12, 1} |  |  |
| iOut[ ] ={0, 15, 3, 12, 1, 2, 4, 8, 5, 6, 9, 10, 7, 11, 13, 14} |  |  |
| iDiffCBPHP[NumComponents] =0 /\* Initializing the array to zero \*/ |  |  |
| bInitializeContext =   (IsMBLeftEdgeOfTileFlag && IsMBTopEdgeOfTileFlag) |  |  |
| if (bInitializeContext) |  |  |
| InitializeCBPHPVLC( ) |  | 8.8.3.4 |
| if (INTERNAL\_CLR\_FMT = = YUVK | |   INTERNAL\_CLR\_FMT = = NCOMPONENT) |  |  |
| iComponent = NumComponents |  |  |
| else |  |  |
| iComponent = 1 |  |  |
| for (i = 0; i < iComponent; i++) { |  |  |
| **sAdaptVLC** = **DecNumCBPHP**   /\* reference AdaptiveVLC struct for NUM\_CBPHP \*/ |  |  |
| NUM\_CBPHP /\* using **sAdaptVLC** \*/ | ae(v) | 8.7.17.4.1 |
| **sAdaptVLC**.DiscrimVal1 +=   NumCBPHPDelta[**sAdaptVLC**.DeltaTableIndex][NUM\_CBPHP]  /\* **sAdaptVLC**.DeltaTableIndex is equal to 0 \*/ |  | Table 89 |
| iCBPHP = REFINE\_CBPHP(NUM\_CBPHP) |  | 8.7.17.3 |
| for (iBlock = 0; iBlock < 4; iBlock++) |  |  |
| if (iCBPHP & (1 << iBlock)) { |  |  |
| **sAdaptVLC** = **DecNumBlkCBPHP**   /\* reference AdaptiveVLC struct for NUM\_BLKCBPHP \*/ |  |  |
| NUM\_BLKCBPHP /\* using **sAdaptVLC** \*/ | ae(v) | 8.7.17.4.2 |
| **sAdaptVLC**.DiscrimVal1 +=   NumBlkCBPHPDelta[**sAdaptVLC**.DeltaTableIndex][NUM\_BLKCBPHP] |  | Table 90, Table 91 |
| iVal = NUM\_BLKCBPHP+1 |  |  |
| iBlkCBPHP = 0 |  |  |
| if (iVal >= 6) { /\* Is chroma \*/ |  |  |
| CHR\_CBPHP | e(v) | 8.7.17.4.3 |
| iBlkCBPHP = 0x10 \* (CHR\_CBPHP + 1) |  |  |
| if (iVal >= 9) { |  |  |
| VAL\_INC | e(v) | 8.7.17.4.4 |
| iVal += VAL\_INC |  |  |
| } |  |  |
| iVal −= 6 |  |  |
| } |  |  |
| iCode = iOff[iVal] |  |  |
| if (iFLC[iVal]) { |  |  |
| CODE\_INC | u(iFLC[iVal]) | ‎8.7.17.4.5 |
| iCode += CODE\_INC |  |  |
| } |  |  |
| iBlkCBPHP += iOut[iCode] |  |  |
| if (INTERNAL\_CLR\_FMT = = YUV444) { |  |  |
| iDiffCBPHP[0] |= ((iBlkCBPHP & 0x0F) << (iBlock \* 4)) |  |  |
| for (k = 0; k < 2; k++) |  |  |
| if ((iBlkCBPHP >> (k + 4)) & 0x01) { |  |  |
| NUM\_CH\_BLK | e(v) | 8.7.17.4.6 |
| iCBPHPChr = REFINE\_CBPHP(NUM\_CH\_BLK + 1) |  | 8.7.17.3 |
| iDiffCBPHP[k + 1] |= (iCBPHPChr << (iBlock \* 4)) |  |  |
| } |  |  |
| /\* INTERNAL\_CLR\_FMT = = YUV444 \*/ |  |  |
| } else if (INTERNAL\_CLR\_FMT = = YUV422) { |  |  |
| iDiffCBPHP[0] |= ((iBlkCBPHP & 0x0F) << (iBlock \* 4)) |  |  |
| for (k = 0; k < 2; k++) |  |  |
| if ((iBlkCBPHP >> (k + 4)) & 0x01) { |  |  |
| iShift[4] = {0, 1, 4, 5} |  |  |
| CBPHP\_CH\_BLK | e(v) | 8.7.17.4.7 |
| iCBPHPChr = iShift[CBPHP\_CH\_BLK + 1] |  |  |
| iDiffCBPHP[k + 1] |= (iCBPHPChr << iShift[iBlock]) |  |  |
| } |  |  |
| } else if (INTERNAL\_CLR\_FMT = = YUV420) { |  |  |
| iDiffCBPHP[0] |= ((iBlkCBPHP & 0x0F) << (iBlock \* 4)) |  |  |
| iDiffCBPHP[1] |= (((iBlkCBPHP >> 4) & 0x01) << iBlock) |  |  |
| iDiffCBPHP[2] |= (((iBlkCBPHP >> 5) & 0x01) << iBlock) |  |  |
| } else /\* Default \*/ |  |  |
| iDiffCBPHP[i] |= ((iBlkCBPHP ) << (iBlock \* 4)) |  |  |
| } /\* if (iCBPHP…) \*/ |  |  |
| } /\* i \*/ |  |  |
| PredCBPHP(iDiffCBPHP) |  | 8.7.17.5.1 |
| } |  |  |

#### REFINE\_CBPHP( )

The REFINE\_CBPHP( ) syntax structure is specified in Table 58.

Table 58 – REFINE\_CBPHP( ) syntax structure

|  |  |  |
| --- | --- | --- |
| **REFINE\_CBPHP(iNum) {** | **Descriptor** | **Reference** |
| if (iNum = = 2) { |  |  |
| REF\_CBPHP1 | e(v) | 8.7.17.4.8 |
| iRef = REF\_CBPHP1 |  |  |
| } else if (iNum = = 1) { |  |  |
| REF\_CBPHP | u(2) | 8.7.17.4.9 |
| iRef = (1<<REF\_CBPHP) |  |  |
| } else if (iNum = = 3) { |  |  |
| REF\_CBPHP | u(2) | 8.7.17.4.9 |
| iRef = (0x0F ^ (1<<REF\_CBPHP)) |  |  |
| } else if (iNum = = 4) |  |  |
| iRef = 0x0F |  |  |
| else |  |  |
| iRef = 0 |  |  |
| return iRef |  |  |
| } |  |  |

#### MB\_CBPHP( ) and REFINE\_CBPHP( ) semantics

##### NUM\_CBPHP

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

Table 59 – Code table for NUM\_CBPHP( )

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

##### NUM\_BLKCBPHP

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

Table 60 – Code table for NUM\_BLKCBPHP   
if INTERNAL\_CLR\_FMT is one of {YUVK, NCOMPONENT, YONLY}

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

Table 61 – Code table for NUM\_BLKCBPHP   
if INTERNAL\_CLR\_FMT is not one of {YUVK, NCOMPONENT, YONLY}

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

##### CHR\_CBPHP

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

Table 62 – Code table for CHR\_CBPHP, VAL\_INC, and CBPHP\_CH\_BLK

|  |  |
| --- | --- |
| **Code** | **Value** |
| 1 | 0 |
| 01 | 1 |
| 00 | 2 |

NOTE – Non-zero residual CBPHP in: a) U component blocks are indicated by CHR\_CBPHP = 0; b) V component blocks are indicated by CHR\_CBPHP = 1; and c) both U and V component blocks are indicated by CHR\_CBPHP = 2.

##### VAL\_INC

VAL\_INC is a syntax element that refines the CBPHP of a block group as specified in 8.7.17.1 when NUM\_BLKCBPHP plus 1 is greater than or equal to 9. The VLC that specifies the parsing of VAL\_INC is specified in Table 62.

##### CODE\_INC

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

##### NUM\_CH\_BLK

When INTERNAL\_CLR\_FMT is equal to YUV444, NUM\_CH\_BLK is a syntax element that specifies the number of coded chroma blocks in a 2×2 block group as specified in 8.7.17.1. The VLC that specifies the parsing of NUM\_CH\_BLK is specified in Table 63.

Table 63 – Code table for NUM\_CH\_BLK

|  |  |
| --- | --- |
| **Code** | **Value** |
| 1 | 0 |
| 01 | 1 |
| 000 | 2 |
| 001 | 3 |

##### CBPHP\_CH\_BLK

When INTERNAL\_CLR\_FMT is equal to YUV422, CBPHP\_CH\_BLK is a syntax element that refines the chroma CBPHP for a block group as specified in 8.7.17.1. The VLC that specifies the parsing of CBPHP\_CH\_BLK is specified in Table 62.

##### REF\_CBPHP1

REF\_CBPHP1 is a variable size syntax element that refines the CBPHP of a block group as specified in 8.7.17.1. The VLC that specifies the parsing of REF\_CBPHP1 is specified in Table 64.

Table 64 – Code table for REF\_CBPHP1

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

##### REF\_CBPHP

REF\_CBPHP is a 2-bit syntax element that refines the CBPHP of a block group as specified in 8.7.17.1.

#### CBPHP prediction

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

##### PredCBPHP( )

The pseudocode for the function PredCBPHP( ) is specified in Table 65.

Table 65 – Pseudocode for the function PredCBPHP( )

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

##### PredCBPHP444( )

The pseudocode for the function PredCBPHP444( ) is specified in Table 66.

Table 66 – Pseudocode for the function PredCBPHP444( )

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

##### PredCBPHP422( )

The pseudocode for the function PredCBPHP422( ) is specified in Table 67.

Table 67 – Pseudocode for the function PredCBPHP422( )

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

##### PredCBPHP420( )

The pseudocode for the function PredCBPHP420( ) is specified in Table 68.

Table 68 – Pseudocode for the function PredCBPHP420( )

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

### Macroblock highpass

#### General

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

#### MB\_HP( )

The MB\_HP( ) syntax structure is specified in Table 69.

Table 69 – MB\_HP( ) syntax structure

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

#### MB\_HP\_FLEX( )

The MB\_HP\_FLEX ( ) syntax structure is specified in Table 70.

Table 70 – MB\_HP\_FLEX( ) syntax structure

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

#### DECODE\_BLOCK\_ADAPTIVE( )

The DECODE\_BLOCK\_ADAPTIVE( ) syntax structure is specified in Table 71.

Table 71 – DECODE\_BLOCK\_ADAPTIVE( ) syntax structure

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

#### DECODE\_BLOCK( )

The DECODE\_BLOCK( ) syntax structure is specified in Table 72.

Table 72 – DECODE\_BLOCK( ) syntax structure

|  |  |  |
| --- | --- | --- |
| **DECODE\_BLOCK(bChroma, iCoeff[ ], iBand, iLocation) {** | **Descriptor** | **Reference** |
| iNumNZ = 1 |  |  |
| iFirstIndex = DECODE\_FIRST\_INDEX(bChroma, iBand) |  | 8.7.18.8 |
| SIGN\_FLAG | u(1) | 8.7.14.4 |
| iSR = (iFirstIndex & 1) |  |  |
| iSRn = (iFirstIndex >> 2) |  |  |
| iContext = (iSR & iSRn) |  |  |
| if (iFirstIndex & 2) |  |  |
| iCoeff[1] = DECODE\_ABS\_LEVEL(iBand, bChroma, iContext) |  | 8.7.13 |
| else |  |  |
| iCoeff[1] = 1 |  |  |
| if (SIGN\_FLAG) |  |  |
| iCoeff[1] = −iCoeff[1] |  |  |
| iCoeff[0] = 0 |  |  |
| if (iSR = = 0) |  |  |
| iCoeff[0] = DECODE\_RUN(15 − iLocation) |  | 8.7.18.6 |
| iLocation += iCoeff[0] + 1 |  |  |
| while (iSRn != 0) { |  |  |
| iSR = (iSRn & 1) |  |  |
| iCoeff[iNumNZ \* 2] = 0 |  |  |
| if (iSR = = 0) |  |  |
| iCoeff[iNumNZ \* 2] = DECODE\_RUN(15 − iLocation) |  | 8.7.18.6 |
| iLocation += (iCoeff[iNumNZ \* 2] + 1) |  |  |
| iIndex = DECODE\_INDEX(iLocation, bChroma, iBand, iContext) |  | 8.7.18.7 |
| iSRn = (iIndex >> 1) |  |  |
| iContext &= iSRn |  |  |
| SIGN\_FLAG | u(1) | 8.7.14.4 |
| if (iIndex & 1) |  |  |
| iCoeff[(iNumNZ \* 2) + 1] =   DECODE\_ABS\_LEVEL(iBand, bChroma, iContext) |  | 8.7.13 |
| else |  |  |
| iCoeff[(iNumNZ \* 2) + 1] = 1 |  |  |
| if (SIGN\_FLAG) |  |  |
| iCoeff[(iNumNZ \* 2) + 1] = −iCoeff[(iNumNZ \* 2) + 1] |  |  |
| iNumNZ++ |  |  |
| } |  |  |
| return iNumNZ |  |  |
| } |  |  |

#### DECODE\_RUN( )

The DECODE\_RUN( ) syntax structure is specified in Table 73.

Table 73 – DECODE\_RUN( ) syntax structure

|  |  |  |
| --- | --- | --- |
| **DECODE\_RUN(iMaxRun) {** | **Descriptor** | **Reference** |
| iRemap[ ] = {1, 2, 3, 5, 7, 1, 2, 3, 5, 7, 1, 2, 3, 4, 5} |  |  |
| iRunBin[ ] = {−1, −1, −1, −1, 2, 2, 2, 1, 1, 1, 1, 0, 0, 0, 0} |  |  |
| iRunFixedLength[ ] = {0, 0, 1, 1, 3, 0, 0, 1, 1, 2, 0, 0, 0, 0, 1} |  |  |
| if (iMaxRun < 5) { |  |  |
| iRun = 1; |  |  |
| if (iMaxRun != 1) { |  |  |
| RUN\_VALUE | e(v) | 8.7.18.9.1 |
| iRun = RUN\_VALUE |  |  |
| } |  |  |
| } else { |  |  |
| RUN\_INDEX | e(v) | 8.7.18.9.2 |
| iIndex = RUN\_INDEX + 5\* iRunBin[iMaxRun] |  |  |
| iFixed = iRunFixedLength[iIndex] |  |  |
| iRun = iRemap[iIndex] |  |  |
| if (iFixed) { |  |  |
| RUN\_REF | u(iFixed) | 8.7.18.9.3 |
| iRun += RUN\_REF |  |  |
| } |  |  |
| } |  |  |
| return iRun |  |  |
| } |  |  |

#### DECODE\_INDEX( )

The DECODE\_INDEX( ) syntax structure is specified in Table 74.

Table 74 – DECODE\_INDEX( ) syntax structure

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

#### DECODE\_FIRST\_INDEX( )

The DECODE\_FIRST\_INDEX( ) syntax structure is specified in Table 75.

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 |  |  |
| } |  |  |

#### Block-level semantics

##### RUN\_VALUE

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

Table 76 – Code table of RUN\_VALUE if iMaxRun = = 2

|  |  |
| --- | --- |
| **Code** | **Value** |
| 1 | 1 |
| 0 | 2 |

Table 77 – Code table of RUN\_VALUE if iMaxRun = = 3

|  |  |
| --- | --- |
| **Code** | **Value** |
| 1 | 1 |
| 01 | 2 |
| 00 | 3 |

Table 78 – Code table of RUN\_VALUE if iMaxRun = = 4

|  |  |
| --- | --- |
| **Code** | **Value** |
| 1 | 1 |
| 01 | 2 |
| 001 | 3 |
| 000 | 4 |

##### RUN\_INDEX

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

Table 79 – Code table of RUN\_INDEX

|  |  |
| --- | --- |
| **Code** | **Value** |
| 1 | 0 |
| 01 | 1 |
| 001 | 2 |
| 0000 | 3 |
| 0001 | 4 |

##### RUN\_REF

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

##### INDEX\_A

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

Table 80 – Code table for INDEX\_A

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

INDEX\_A jointly codes the following two events.

* The binary event of whether the magnitude of the next non-zero coefficient is equal to 1 or greater than 1 as follows.
* If (INDEX\_A & 1) is equal to 0, this magnitude is equal to 1.
* Otherwise, this magnitude is greater than 1.
* The ternary of event whether this coefficient is the last coefficient in the block, and if there are more non-zero coefficients, whether the run before the next non-zero coefficient is zero or non-zero, as follows.
* If (INDEX\_A >> 1) is equal to 0, this coefficient is the last coefficient in the block.
* Otherwise, if (INDEX\_A >> 1) is equal to 1, the run before the next non-zero coefficient is zero.
* Otherwise (i.e. when (INDEX\_A >> 1) is equal to 2), the run before the next non-zero coefficient is non-zero.

NOTE – Thus INDEX\_A has an alphabet size of 2\*3= 6.

##### INDEX\_B

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

Table 81 – Code table for INDEX\_B

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

INDEX\_B jointly codes the following two events.

* The binary event of whether the magnitude of the next non-zero coefficient is equal to 1 or greater than 1 as follows.
* If (INDEX\_B & 1) is equal to 0, this magnitude is equal to 1.
* Otherwise, this magnitude is greater than 1.
* The binary event of whether this coefficient is the last coefficient in the block or if there are more non-zero coefficients, as follows.
* If (INDEX\_B >> 1) is equal to 0, this coefficient is the last coefficient in the block.
* Otherwise, the run before the next non-zero coefficient is zero.

NOTE – Thus INDEX\_B has an alphabet size of 2\*2= 4.

##### INDEX\_C\_FLAG

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

##### FIRST\_INDEX

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

Table 82 – Code table for FIRST\_INDEX

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

FIRST\_INDEX jointly codes the following three events.

* The binary event of whether the run before the first non-zero coefficient is non-zero or zero as follows.
* If (FIRST\_INDEX & 1) is equal to 0, this run is non-zero.
* Otherwise this run is zero.
* The binary event of whether the magnitude of the first non-zero coefficient is equal to 1 or greater than 1 as follows.
* If (FIRST\_INDEX & 2) is equal to 0, this magnitude is equal to 1.
* Otherwise, this magnitude is greater than 1.
* The ternary event of whether the first coefficient is the last coefficient in the block, and if there are more non-zero coefficients whether the run before the next non-zero coefficient is zero or non-zero, as follows.
* If (FIRST\_INDEX >> 2) is equal to 0, the first coefficient is the last coefficient in the block.
* Otherwise, if (FIRST\_INDEX >> 2) is equal to 1, the run before the next non-zero coefficient is zero.
* Otherwise ((FIRST\_INDEX >> 2) is equal to 2), the run before the next non-zero coefficient is non-zero.

NOTE – Thus FIRST\_INDEX has an alphabet size of 2\*3\*3 = 12.

### Macroblock FLEXBITS

#### MB\_FLEXBITS( )

The MB\_FLEXBITS( ) syntax structure is specified in Table 83.

Table 83 – MB\_FLEXBITS( ) syntax structure

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

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

#### BLOCK\_FLEXBITS( )

The BLOCK\_FLEXBITS( ) syntax structure is specified in Table 84.

Table 84 – BLOCK\_FLEXBITS( ) syntax structure

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

#### DECODE\_FLEX( )

The DECODE\_FLEX( ) syntax structure is specified in Table 85.

Table 85 – DECODE\_FLEX( ) syntax structure

|  |  |  |
| --- | --- | --- |
| **DECODE\_FLEX(iVLCCoeff, iFlexBitsLeft) {** | **Descriptor** | **Reference** |
| FLEX\_REF | u(iFlexBitsLeft) | 8.7.19.4.1 |
| if (iVLCCoeff > 0) |  |  |
| iFlexCoeff = FLEX\_REF |  |  |
| else if (iVLCCoeff < 0) |  |  |
| iFlexCoeff = −FLEX\_REF |  |  |
| else { |  |  |
| iFlexCoeff = FLEX\_REF |  |  |
| if (iFlexCoeff) { |  |  |
| SIGN\_FLAG | u(1) | 8.7.14.4 |
| if (SIGN\_FLAG) |  |  |
| iFlexCoeff = −iFlexCoeff |  |  |
| } |  |  |
| } |  |  |
| return iFlexCoeff |  |  |
| } |  |  |

#### FLEXBITS semantics

##### FLEX\_REF

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

## Adaptive VLC code table selection

### General

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

Let the set of VLC tables be: **vT1**, **vT2**, ..., **vTn** where the ordering of the tables is predefined based on their relative similarity. That is **vT1** and **vT2** exhibit greater similarity to each other than **vT1** and **vT3** do to each other. The measure of similarity between tables can be qualified by using relative entropy. Let table **vTi** be a VLC table used for entropy coding of symbol iX, and let **vTj** and **vTk** be the two nearest tables (in terms of similarity). For each value of symbol iX, a metric deltaDisc estimating the relative advantage of coding this value using table **vTj** or table **vTk** instead of table **vTi** is precomputed and stored in tables **vTi**.DeltaTableIndex and **vTi**.Delta2TableIndex. This deltaDisc is positive if this symbol is more efficiently coded by the new table instead of the current table. This deltaDisc is negative if this symbol is less efficiently coded by the new table instead of the current table. For example, if a given value of the symbol requires 3 bits in the new table, while it requires 5 bits in the current table, this value is more efficiently coded in the new table and the corresponding deltaDisc is (5 − 3 = 2).

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

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

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

### Adaptive VLC deltaDisc tables

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

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

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

* **sAdaptVLC**.DiscrimVal1 is incremented by deltaDisc[**sAdaptVLC**.DeltaTableIndex][iVal].
* **sAdaptVLC**.DiscrimVal2 is incremented by deltaDisc[**sAdaptVLC**.Delta2TableIndex][iVal].

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

For the syntax element ABS\_LEVEL\_INDEX (‎8.7.14.5), there is one deltaDisc table for switching between Code tables 0 and 1, as specified in Table 86.

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

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

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

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

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

For the syntax element INDEX\_A (8.7.18.9.4), there are three Delta tables, as specified in Table 88.

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

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

For the syntax element NUM\_CBPHP (8.7.17.4.1), there is one deltaDisc table for switching between Code tables 0 and 1 as specified in Table 89.

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

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

For the syntax element NUM\_BLKCBPHP (8.7.17.4.2), there is one deltaDisc table for switching between Code tables 0 and 1. In the case where the INTERNAL\_CLR\_FMT is YONLY, NCOMPONENT or YUVK, the code tables have five symbols and the deltaDisc table is specified in Table 90.

Table 90 – Constant table NumBlkCBPHPDelta[m][n]  
when INTERNAL\_CLR\_FMT is  
YONLY, NCOMPONENT or YUVK

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

For all other values of INTERNAL\_CLR\_FMT, the code table for NUM\_BLKCBPHP has nine symbols, and the deltaDisc table is specified in Table 91.

Table 91 – Constant table NumBlkCBPHPDelta  
for INTERNAL\_CLR\_FMT other than  
YONLY, NCOMPONENT, and YUVK

|  |  |
| --- | --- |
| **Index value n** | **Value for m = 0** |
| **0** | 2 |
| **1** | 2 |
| **2** | 1 |
| **3** | 1 |
| **4** | −1 |
| **5** | −2 |
| **6** | −2 |
| **7** | −2 |
| **8** | −3 |

### Initialization

The relevant adaptive VLC data structures that are associated with each of the three bands DC, LP and HP are initialized by the functions InitializeDCVLC( ), InitializeLPVLC( ) and InitializeHPVLC( ). These functions are specified in 8.8.3.1, 8.8.3.2, and 8.8.3.3, respectively. The adaptive VLC structure associated with CBPHP is initialized by the function InitializeCBPHPVLC( ) that is specified in 8.8.3.4.

These functions in turn make use of the functions InitializeVLCTable1( ) and InitializeVLCTable2( ) which are specified in 8.8.3.5 and 8.8.3.6, respectively.

NOTE – InitializeVLCTable1( ) is used for initializing VLC code tables when there are exactly two code tables. If there are more than two code table tables, InitializeVLCTable2( ) is used.

#### InitializeDCVLC( )

The pseudocode for the function InitializeDCVLC( ) is specified in Table 92.

Table 92 – Pseudocode for the function InitializeDCVLC( )

|  |  |
| --- | --- |
| **InitializeDCVLC( ) {** | **Reference** |
| **AbsLevelIndDCLum** = InitializeVLCTable1(**AbsLevelIndDCLum**) | 8.8.3.5 |
| **AbsLevelIndDCChr** = InitializeVLCTable1(**AbsLevelIndDCChr**) | 8.8.3.5 |
| } |  |

#### InitializeLPVLC( )

The pseudocode for the function InitializeLPVLC( ) is specified in Table 93.

Table 93 – Pseudocode for the function InitializeLPVLC( )

|  |  |
| --- | --- |
| **InitializeLPVLC( ) {** | **Reference** |
| **DecFirstIndLPLum** = InitializeVLCTable2(**DecFirstIndLPLum**) | 8.8.3.6 |
| **DecIndLPLum0 =** InitializeVLCTable2(**DecIndLPLum0**) | 8.8.3.6 |
| **DecIndLPLum1 =** InitializeVLCTable2(**DecIndLPLum1**) | 8.8.3.6 |
| **DecFirstIndLPChr** = InitializeVLCTable2(**DecFirstIndLPChr**) | 8.8.3.6 |
| **DecIndLPChr0 =** InitializeVLCTable2(**DecIndLPChr0**) | 8.8.3.6 |
| **DecIndLPChr1 =** InitializeVLCTable2(**DecIndLPChr1**) | 8.8.3.6 |
| **AbsLevelIndLP0 =** InitializeVLCTable1(**AbsLevelIndLP0**) | 8.8.3.5 |
| **AbsLevelIndLP1 =** InitializeVLCTable1(**AbsLevelIndLP1**) | 8.8.3.5 |
| } |  |

#### InitializeHPVLC( )

The pseudocode for the function InitializeHPVLC( ) is specified in Table 94.

Table 94 – Pseudocode for the function InitializeHPVLC( )

|  |  |
| --- | --- |
| **InitializeHPVLC**( ) **{** | **Reference** |
| **DecFirstIndHPLum** = InitializeVLCTable2(**DecFirstIndHPLum**) | 8.8.3.6 |
| **DecIndHPLum0 =** InitializeVLCTable2(**DecIndHPLum0**) | 8.8.3.6 |
| **DecIndHPLum1 =** InitializeVLCTable2(**DecIndHPLum1**) | 8.8.3.6 |
| **DecFirstIndHPChr** = InitializeVLCTable2(**DecFirstIndHPChr**) | 8.8.3.6 |
| **DecIndHPChr0 =** InitializeVLCTable2(**DecIndHPChr0**) | 8.8.3.6 |
| **DecIndHPChr1 =** InitializeVLCTable2(**DecIndHPChr1**) | 8.8.3.6 |
| **AbsLevelIndHP0 =** InitializeVLCTable1(**AbsLevelIndHP0**) | 8.8.3.5 |
| **AbsLevelIndHP1 =** InitializeVLCTable1(**AbsLevelIndHP1**) | 8.8.3.5 |
| } |  |

#### InitializeCBPHPVLC( )

The pseudocode for the function InitializeCBPHPVLC( ) is specified in Table 95.

Table 95 – Pseudocode for the function InitializeCBPHPVLC( )

|  |  |
| --- | --- |
| **InitializeCBPHPVLC**( ) **{** | **Reference** |
| **DecNumCBPHP** = InitializeVLCTable1(**DecNumCBPHP**) | 8.8.3.5 |
| **DecNumBlkCBPHP** = InitializeVLCTable1(**DecNumBlkCBPHP**) | 8.8.3.5 |
| } |  |

#### InitializeVLCTable1( )

The pseudocode for the function InitializeVLCTable1( ) is specified in Table 96.

Table 96 – Pseudocode for the function InitializeVLCTable1( )

|  |  |
| --- | --- |
| **InitializeVLCTable1(sAdaptVLC) {** | **Reference** |
| /\* **sAdaptVLC** is an instance of the AdaptiveVLC data structure \*/ |  |
| **sAdaptVLC**.TableIndex = 0 |  |
| **sAdaptVLC**.DeltaTableIndex = 0 |  |
| **sAdaptVLC**.DiscrimVal1 = 0 |  |
| return **sAdaptVLC** |  |
| } |  |

#### InitializeVLCTable2( )

The pseudocode for the function InitializeVLCTable2( ) is specified in Table 97.

Table 97 – Pseudocode for the function InitializeVLCTable2( )

|  |  |
| --- | --- |
| **InitializeVLCTable2(sAdaptVLC) {** | **Reference** |
| /\* **sAdaptVLC** is an instance of the AdaptiveVLC data structure \*/ |  |
| **sAdaptVLC**.TableIndex = 1 |  |
| **sAdaptVLC**.DeltaTableIndex = 0 |  |
| **sAdaptVLC**.Delta2TableIndex = 1 |  |
| **sAdaptVLC**.DiscrimVal1 = 0 |  |
| **sAdaptVLC**.DiscrimVal2 = 0 |  |
| return **sAdaptVLC** |  |
| } |  |

### Update of adaptive VLC code table selection

The relevant adaptive VLC data structures that are associated with each of the three bands DC, LP and HP are updated by the functions AdaptDC( ), AdaptLP( ) and AdaptHP( ), respectively. The pseudocode for the functions AdaptDC( ), AdaptLP( ) and AdaptHP( ) are specified in 8.8.4.1, 8.8.4.2 and 8.8.4.3, respectively:

The functions AdaptLP( ) and AdaptHP( ) perform the updates by using the functions AdaptVLCTable1( ) and AdaptVLCTable2( ) which are specified in 8.8.4.4 and 8.8.4.5.

#### AdaptDC( )

The pseudocode for the function AdaptDC( ) is specified in Table 98.

Table 98 – Pseudocode for the function AdaptDC( )

|  |  |
| --- | --- |
| **AdaptDC( ) {** | **Reference** |
| **AbsLevelIndDCLum** = AdaptVLCTable1(**AbsLevelIndDCLum**) | 8.8.4.4 |
| **AbsLevelIndDCChr** = AdaptVLCTable1(**AbsLevelIndDCChr**) | 8.8.4.4 |
| } |  |

#### AdaptLP( )

The pseudocode for the function AdaptLP( ) is specified in Table 99.

Table 99 – Pseudocode for the function AdaptLP( )

|  |  |
| --- | --- |
| **AdaptLP( ) {** | **Reference** |
| **DecFirstIndLPLum** = AdaptVLCTable2(**DecFirstIndLPLum**, 4) | 8.8.4.5 |
| **DecIndLPLum0 =** AdaptVLCTable2(**DecIndLPLum0,** 3) | 8.8.4.5 |
| **DecIndLPLum1 =** AdaptVLCTable2(**DecIndLPLum1**, 3) | 8.8.4.5 |
| **DecFirstIndLPChr** = AdaptVLCTable2(**DecFirstIndLPChr**, 4) | 8.8.4.5 |
| **DecIndLPChr0 =** AdaptVLCTable2(**DecIndLPChr0**, 3) | 8.8.4.5 |
| **DecIndLPChr1 =** AdaptVLCTable2(**DecIndLPChr1**, 3) | 8.8.4.5 |
| **AbsLevelIndLP0** = AdaptVLCTable1(**AbsLevelIndLP0**) | 8.8.4.4 |
| **AbsLevelIndLP1 =** AdaptVLCTable1(**AbsLevelIndLP1**) | 8.8.4.4 |
| } |  |

#### AdaptHP( )

The pseudocode for the function AdaptHP( ) is specified in Table 100.

Table 100 – Pseudocode for the function AdaptHP( )

|  |  |
| --- | --- |
| **AdaptHP( ) {** | **Reference** |
| **DecFirstIndHPLum** = AdaptVLCTable2(**DecFirstIndHPLum**, 4) | 8.8.4.5 |
| **DecIndHPLum0 =** AdaptVLCTable2(**DecIndHPLum0**, 3) | 8.8.4.5 |
| **DecIndHPLum1 =** AdaptVLCTable2(**DecIndHPLum1**, 3) | 8.8.4.5 |
| **DecFirstIndHPChr** = AdaptVLCTable2(**DecFirstIndHPChr**, 4) | 8.8.4.5 |
| **DecIndHPChr0 =** AdaptVLCTable2(**DecIndHPChr0**, 3) | 8.8.4.5 |
| **DecIndHPChr1 =** AdaptVLCTable2(**DecIndHPChr1**, 3) | 8.8.4.5 |
| **AbsLevelIndHP0 =** AdaptVLCTable1(**AbsLevelIndHP0**) | 8.8.4.4 |
| **AbsLevelIndHP**1 = AdaptVLCTable1(**AbsLevelIndHP**1) | 8.8.4.4 |
| **DecNumCBPHP** = AdaptVLCTable1(**DecNumCBPHP**) | 8.8.4.4 |
| **DecNumBlkCBPHP** = AdaptVLCTable1(**DecNumBlkCBPHP**) | 8.8.4.4 |
| } |  |

#### AdaptVLCTable1( )

AdaptVLCTable1( )is used for choosing VLC code tables when there are exactly two code tables. In this case, the index TableIndex takes only the values 0 or 1, and there is only one parameter (DiscrimVal1) which determines the selection of VLC code tables. DeltaTableIndex takes only the value 0, and there is only the one deltaDisc table.

The pseudocode for the function Adapt VLCTable1( ) is specified in Table 101.

Table 101 – Pseudocode for the function AdaptVLCTable1( )

|  |  |
| --- | --- |
| **AdaptVLCTable1(sAdaptVLC) {** | **Reference** |
| /\* **sAdaptVLC** is an instance of the AdaptiveVLC data structure \*/ |  |
| iMaxTableIndex = 1 /\* Only two code tables \*/ |  |
| cLowerBound = −8 |  |
| cUpperBound = 8 |  |
| /\* **sAdaptVLC**.DeltaTableIndex = 0, since only 2 code tables \*/ |  |
| if (**sAdaptVLC**.DiscrimVal1 < cLowerBound &&  **sAdaptVLC**.TableIndex != 0) { |  |
| **sAdaptVLC**.TableIndex− − |  |
| **sAdaptVLC**.DiscrimVal1 = 0 |  |
| } else if (**sAdaptVLC**.DiscrimVal1 > cUpperBound &&   **sAdaptVLC**.TableIndex != iMaxTableIndex ) { |  |
| **sAdaptVLC**.TableIndex++ |  |
| **sAdaptVLC**.DiscrimVal1 = 0 |  |
| } else { |  |
| /\* no change to table, but clip the discriminant \*/ |  |
| if (**sAdaptVLC**.DiscrimVal1 < −64) |  |
| **sAdaptVLC**.DiscrimVal1 = −64 |  |
| if (**sAdaptVLC**.DiscrimVal1 > 64) |  |
| **sAdaptVLC**.DiscrimVal1 = 64 |  |
| } |  |
| return **sAdaptVLC** |  |
| } |  |

#### 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 determine 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.

The pseudocode for the function Adapt VLCTable2( ) is specified in Table 102.

Table 102 – Pseudocode for the 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** |  |
| } |  |

## Adaptation of CBPLP state variables

### General

The parsing of CBPLP depends on the value of the variables CountZeroCBPLP and CountMaxCBPLP. The functions specified in this subclause specify the initialization and updating of these constants.

### InitializeCountCBPLP( )

The variable CountZeroCBPLP and CountMaxCBPLP for LP coefficients are initialized by the function InitializeCountCBPLP( ) specified in Table 103.

Table 103 – Pseudocode for function InitializeCountCBPLP( )

|  |  |
| --- | --- |
| **InitializeCountCBPLP( ) {** | **Reference** |
| CountZeroCBPLP = 1 |  |
| CountMaxCBPLP = 1 |  |
| } |  |

### UpdateCountCBPLP( )

The function UpdateCountCBPLP( ) updates the variables CountZeroCBPLP and CountMaxCBPLP. The pseudocode for this function is specified in Table 104.

Table 104 – Pseudocode for function UpdateCountCBPLP( )

|  |  |
| --- | --- |
| **UpdateCountCBPLP(iCBPLP, iMax) {** | **Reference** |
| CountZeroCBPLP += 1 − (4 \* (iCBPLP = = 0)) |  |
| CountZeroCBPLP = Max(−8, Min(7,CountZeroCBPLP)) |  |
| CountMaxCBPLP += 1 − (4 \* (iCBPLP = = iMax)) |  |
| CountMaxCBPLP = Max(−8, Min(7,CountMaxCBPLP)) |  |
| } |  |

## Adaptive CBPHP prediction

CBPHP prediction depends on the value of member variables of the data structure instance **CBPHPModelHP**. The functions specified in this subclause specify the initialization and updating of this data structure.

### InitializeCBPHPModel( )

The data structure instance **CBPHPModelHP** is initialized in the function InitializeCBPHPModel( ) specified in Table 105.

Table 105 – Pseudocode for function InitializeCBPHPModel( )

|  |  |
| --- | --- |
| **InitializeCBPHPModel( ) {** | **Reference** |
| **CBPHPModelHP**.CBPHPState[0] = **CBPHPModelHP**.CBPHPState[1] = 0 |  |
| **CBPHPModelHP**.CountOnes[0] = **CBPHPModelHP**.CountOnes[1] = −4 |  |
| **CBPHPModelHP**.CountZeroes[0] = **CBPHPModelHP**.CountZeroes[1] = 4 |  |
| } |  |

### UpdateCBPHPModel( )

The variables associated with the data structure instance **CBPHPModelHP** are updated by the function UpdateCBPHPModel( ) as specified in Table 106.

Table 106 – Pseudocode for function UpdateCBPHPModel( )

|  |  |
| --- | --- |
| **UpdateCBPHPModel(i, iNOrig) {** | **Reference** |
| iNDiff = 3 |  |
| **CBPHPModelHP**.CountOnes[i] += iNOrig − iNDiff |  |
| **CBPHPModelHP**.CountOnes[i] = Clip(**CBPHPModelHP**.CountOnes[i], −16,15) |  |
| CBPHPModelHP.CountZeroes[i] += 16 − iNOrig − iNDiff |  |
| **CBPHPModelHP**.CountZeroes[i] = Clip(**CBPHPModelHP**.CountZeroes[i], −16, 15) |  |
| if (**CBPHPModelHP**.CountOnes[i] < 0) |  |
| if (**CBPHPModelHP**.CountOnes[i] < CBPHPModelHP.CountZeroes[i]) |  |
| **CBPHPModelHP**.CBPHPState[i] = 1 |  |
| else |  |
| **CBPHPModelHP**.CBPHPState[i] = 2 |  |
| else if (**CBPHPModelHP**.CountZeroes[i] < 0) |  |
| **CBPHPModelHP**.CBPHPState[i] = 2 |  |
| else |  |
| **CBPHPModelHP**.CBPHPState[i] = 0 |  |
| } |  |

## Adaptive inverse scanning

The parsing of syntax elements corresponding to LP and HP coefficients depends on the state of the inverse scanning tables LowpassScanOrder[i], HighpassHorScanOrder[i], and HighpassVerScanOrder[i]. The functions specified in this subclause define the initialization and updating of these tables.

### Adaptive inverse scanning tables

The inverse scanning order of transform coefficients is a permutation of the integers 1 to 15. Let the integer i represent the order in which a given transform coefficient is parsed from the codestream, and let the local example list listScanOrder[ ] specify an inverse scanning order as follows: the i-th transform coefficient is put into the block in the j‑th position in raster scan order, where j is equal to listScanOrder[i].

The three lists LowpassScanOrder[ ], HighpassHorScanOrder[ ] and HighpassVerScanOrder[ ] are used to specify the inverse scanning order of LP coefficients, HP coefficients in the case of prediction from the left (9.6) and HP coefficients in the case of prediction from the top, respectively. These lists are initialized to scan orders as specified in the following. However, the lists are adaptive, and thus may change over the course of parsing, based on the statistics of non-zero transform coefficients in the codestream.

The three lists LowpassScanOrder[ ], HighpassHorScanOrder[ ] and HighpassVerScanOrder[ ] are initialized as specified in ‎8.11.2 and 8.11.3. The initial orders are specified by the two lists ScanOrder0[ ] and ScanOrder1[ ], which are specified in Table 107.

Table 107 – Definitions of ScanOrder0 and ScanOrder1

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **i** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScanOrder0[i]** | 4 | 1 | 5 | 8 | 2 | 9 | 6 | 12 | 3 | 10 | 13 | 7 | 14 | 11 | 15 |
| **ScanOrder1[i]** | 1 | 2 | 5 | 4 | 3 | 6 | 9 | 8 | 7 | 12 | 15 | 13 | 10 | 11 | 14 |

Each of the lists LowpassScanOrder[ ], HighpassHorScanOrder[ ] and HighpassVerScanOrder[ ] also has an associated list that determines how the scan order is updated. These corresponding lists are LowpassTotals[ ], HighpassHorTotals[ ] and HighpassVerTotals[ ], respectively. These associated lists are initialized to be equal to the list ScanTotals[ ], which is specified in Table 108.

Table 108 – Definition of ScanTotals

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **i** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScanTotals[i]** | 32 | 30 | 28 | 26 | 24 | 22 | 20 | 18 | 16 | 14 | 12 | 10 | 8 | 6 | 4 |

The three lists LowpassScanOrder[ ], HighpassHorScanOrder[ ] and HighpassVerScanOrder[ ] are updated as specified in 8.11.

### InitializeAdaptiveScanLP( )

The adaptive inverse scanning tables for LP coefficients are initialized in the function InitializeAdaptiveScanLP( ) specified in Table 109.

Table 109 – Pseudocode for function InitializeAdaptiveScanLP( )

|  |  |
| --- | --- |
| **InitializeAdaptiveScanLP( ) {** | **Reference** |
| for (i = 1; i <= 15; i++) { |  |
| LowpassScanOrder[i] = ScanOrder0[i] | ScanOrder0[i] specified in 8.11.1 |
| LowpassTotals[i] = ScanTotals[i] | ScanTotals[i] specified in 8.11.1 |
| } |  |
| } |  |

### InitializeAdaptiveScanHP( )

The adaptive inverse scanning tables for HP coefficients are initialized in the function InitializeAdaptiveScanHP( ) specified in Table 110.

Table 110 – Pseudocode for function InitializeAdaptiveScanHP( )

|  |  |
| --- | --- |
| **InitializeAdaptiveScanHP( ) {** | **Reference** |
| for (i = 1; i <= 15; i++) { |  |
| HighpassHorScanOrder[i] = ScanOrder0[i] | ScanOrder0[i] specified in 8.11.1 |
| HighpassVerScanOrder[i] = ScanOrder1[i] | ScanOrder1[i] specified in 8.11.1 |
| HighpassHorTotals[i] = ScanTotals[i] | ScanTotals[i] specified in 8.11.1 |
| HighpassVerTotals[i] = ScanTotals[i] | ScanTotals[i] specified in 8.11.1 |
| } |  |
| } |  |

### ResetTotalsAdaptiveScanLP( )

The list LowpassTotals of the adaptive inverse scanning tables for LP coefficients is reset in the function ResetTotalsAdaptiveScanLP( ) specified in Table 111.

Table 111 – Pseudocode for function ResetTotalsAdaptiveScanLP( )

|  |  |
| --- | --- |
| **ResetTotalsAdaptiveScanLP( ) {** | **Reference** |
| for (i = 1; i <= 15; i++) |  |
| LowpassTotals[i] = ScanTotals[i] | ScanTotals[i] specified in 8.11.1 |
| } |  |

### ResetTotalsAdaptiveScanHP( )

The lists HighpassHorTotals and HighpassVerTotals of the adaptive inverse scanning tables for HP coefficients are reset in the function ResetTotalsAdaptiveScanHP( ) specified in Table 112.

Table 112 – Pseudocode for function ResetTotalsAdaptiveScanHP( )

|  |  |
| --- | --- |
| **ResetTotalsAdaptiveScanHP( ) {** | **Reference** |
| for (i = 1; i <= 15; i++) { |  |
| HighpassHorTotals[i] = ScanTotals[i] | ScanTotals[i] specified in 8.11.1 |
| HighpassVerTotals[i] = ScanTotals[i] | ScanTotals[i] specified in 8.11.1 |
| } |  |
| } |  |

### AdaptiveLPScan( )

The function AdaptiveLPScan( ) updates the list LPInput[k] and also updates the variables associated with tracking and modifying the LP scan order LowpassScanOrder[i] as specified in Table 113.

Table 113 – Pseudocode for function AdaptiveLPScan( )

|  |  |
| --- | --- |
| **AdaptiveLPScan(n, i, iValue) {** | **Reference** |
| k = LowpassScanOrder[i] |  |
| LPInput[n][k] = iValue |  |
| LowpassTotals[i]++ |  |
| if ((i > 1) && (LowpassTotals[i] > LowpassTotals[i−1])) { |  |
| valueTemp = LowpassTotals[i] |  |
| LowpassTotals[i] = LowpassTotals[i−1] |  |
| LowpassTotals[i−1] = valueTemp |  |
| valueTemp = LowpassScanOrder[i] |  |
| LowpassScanOrder[i] = LowpassScanOrder[i−1] |  |
| LowpassScanOrder[i−1] = valueTemp |  |
| } |  |
| } |  |

### AdaptiveHPScan( )

The function AdaptiveHPScan( ) updates the list HPInputVLC[iComponent][iBlock][k], and also updates the variables associated with tracking and modifying the HP scan orders HighpassHorScanOrder[i] and HighpassVerScanOrder[i] as specified in Table 114.

Table 114 – Pseudocode for function AdaptiveHPScan( )

|  |  |
| --- | --- |
| **AdaptiveHPScan(iComponent, iBlock, i, iValue) {** | **Reference** |
| if (MBHPMode = = 1) { /\* vertical scan order \*/ |  |
| k = HighpassVerScanOrder[i] |  |
| HighpassVerTotals[i]++ |  |
| HPInputVLC[iComponent][iBlock][k] = iValue |  |
| if ((i > 1) &&   (HighpassVerTotals[i] > HighpassVerTotals[i−1])) { |  |
| valueTemp = HighpassVerTotals[i] |  |
| HighpassVerTotals[i] = HighpassVerTotals[i−1] |  |
| HighpassVerTotals[i−1] = valueTemp |  |
| valueTemp = HighpassVerScanOrder[i] |  |
| HighpassVerScanOrder[i] = HighpassVerScanOrder[i−1] |  |
| HighpassVerScanOrder[i−1] = valueTemp |  |
| } |  |
| } else { /\* horizontal scan order \*/ |  |
| k = HighpassHorScanOrder[i] |  |
| HighpassHorTotals[i]++ |  |
| HPInputVLC[iComponent][iBlock][k] = iValue |  |
| if ((i > 1) &&   (HighpassHorTotals[i] > HighpassHorTotals[i−1])) { |  |
| valueTemp = HighpassHorTotals[i] |  |
| HighpassHorTotals[i] = HighpassHorTotals[i−1] |  |
| HighpassHorTotals[i−1] = valueTemp |  |
| valueTemp = HighpassHorScanOrder[i] |  |
| HighpassHorScanOrder[i] = HighpassHorScanOrder[i−1] |  |
| HighpassHorScanOrder[i−1] = valueTemp |  |
| } |  |
| } |  |
| } |  |

The variable MBHPMode is computed during the HP prediction direction computation process specified in 9.6.3.2. The scan order is selected based on the value of the variable MBHPMode. AdaptiveHPScan( ) shall only be invoked on a macroblock after the HP prediction direction computation process specified in 9.6.3.2 has been invoked and completed for this macroblock. The HP prediction direction process shall be invoked only after the completion of the LP transform coefficient parsing process.

NOTE – See 9.6 for more information.

## Adaptive coefficient normalization

### InitializeModelMB( )

The initialization of the **Model** data structure is specified by the function InitializeModelMB( ) as specified in Table 115.

Table 115 – Pseudocode for function InitializeModelMB( )

|  |  |
| --- | --- |
| **InitializeModelMB(Model, iBand) {** | **Reference** |
| /\* iBand is the frequency band (DC = 0, LP = 1, HP = 2) \*/ |  |
| **Model**.MState[0] = **Model**.MState[1] = 0 |  |
| **Model**.MBits[0] = **Model**.MBits[1] = (2 − iBand) \* 4 |  |
| } |  |

### UpdateModelMB( )

The adaptation of the **Model** data structure is specified by the function UpdateModelMB( ) as specified in Table 116.

Table 116 – Pseudocode for function UpdateModelMB( )

| **UpdateModelMB(iLapMean[ ], Model, iBand) {** | **Reference** |
| --- | --- |
| /\* INTERNAL\_CLR\_FMT is the colour format of the image \*/ |  |
| /\* iBand is the frequency band (DC = 0, LP = 1, HP = 2) \*/ |  |
| iModelWeight = 70 |  |
| iWeight0[3] = {240 /\*DC\*/, 12 /\*LP\*/, 1} |  |
| iWeight1[3][MAX\_COMPONENTS] = { |  |
| {0, 240, 120, 80, 60, 48, 40,34, 30, 27, 24, 22, 20, 18, 17, 16}, |  |
| {0, 12, 6, 4, 3, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1}, |  |
| {0, 16, 8, 5, 4, 3, 3, 2, 2, 2, 2, 1, 1, 1, 1, 1} |  |
| } |  |
| iWeight2[6] = {120, 37, 2 /\*YUV420\*/, 120, 18, 1 /\*YUV422\*/} |  |
| iLapMean[0] \*= iWeight0[iBand] |  |
| if (INTERNAL\_CLR\_FMT = = YUV420) |  |
| iLapMean[1] \*= iWeight2[iBand] |  |
| else if (INTERNAL\_CLR\_FMT = = YUV422) |  |
| iLapMean[1] \*= iWeight2[3 + iBand] |  |
| else { |  |
| iLapMean[1] \*= iWeight1[iBand][NumComponents − 1] |  |
| if (iBand = = 2) |  |
| iLapMean[1] >>= 4 |  |
| } |  |
| iNumModels = 2 |  |
| if (INTERNAL\_CLR\_FMT = = YONLY) |  |
| iNumModels = 1 |  |
| for (j = 0; j < iNumModels; j++) { |  |
| iMS = **Model**.MState[j] |  |
| iDelta = ((iLapMean[j] − iModelWeight) >> 2) |  |
| if (iDelta <= −8) { |  |
| iDelta += 4 |  |
| if (iDelta < −16) |  |
| iDelta = −16 |  |
| iMS += iDelta |  |
| if (iMS < −8) |  |
| if (**Model**.MBits[j] = = 0) |  |
| iMS = −8 |  |
| else { |  |
| iMS = 0 |  |
| **Model**.MBits[j]− − |  |
| } |  |
| } else if (iDelta >= 8) { |  |
| iDelta −= 4 |  |
| if (iDelta > 15) |  |
| iDelta = 15 |  |
| iMS += iDelta |  |
| if (iMS > 8) |  |
| if (**Model**.MBits[j] >= 15) { |  |
| **Model**.MBits[j] = 15 |  |
| iMS = 8 |  |
| } else { |  |
| iMS = 0 |  |
| **Model**.MBits[j]++ |  |
| } |  |
| } |  |
| **Model**.MState[j] = iMS |  |
| } |  |
| } |  |

# Decoding process

## 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 Specification.

The image decoding process proceeds as specified in clause9.2.

## Image decoding

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

The image decoding process proceeds as specified 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)) /\* Horizontally 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] |  |
| } |  |
| 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; x++) |  |
| ImageAlpha[0][x][y] = ImagePlane[0][x][y] |  |
| } |  |

NOTE – Throughout the parsing of syntax elements in clause 8, it is assumed that if ALPHA\_IMAGE\_PLANE\_FLAG is equal to TRUE, there are two sets of parsed syntax elements: one set corresponding to the primary image plane and another 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.

## Image plane decoding

This process is invoked for each image plane.

The outputs of this process are the decoded samples for this image plane, ImagePlane[i][x][y].

The image plane decoding process proceeds as specified in Table 118.

Table 118 – Pseudocode for function ImagePlaneDecoding( )

|  |  |
| --- | --- |
| **ImagePlaneDecoding( ) {** | **Reference** |
| ImagePlaneDCQP( ) | 9.7.1.1 |
| ImagePlaneLPQP( ) | 9.7.2.1 |
| ImagePlaneHPQP( ) | 9.7.3.1 |
| for (TileIndexy = 0; TileIndexy < NumTileRows; TileIndexy++) |  |
| for (TileIndexx = 0; TileIndexx < NumTileCols; TileIndexx++) |  |
| TileTransformCoefficientProcessing( )   /\* At this point, transform coefficients for the entire image plane have   been obtained \*/ | 9.4.1 |
| SampleReconstruction( )   /\* This process performs both levels of the inverse transform and overlap   operations on the entire image \*/ | 9.9.1 |
| OutputFormatting( ) | 9.10.2 |
| } |  |

## Tile transform coefficient processing

### Overview

This process is invoked for each tile. The inputs to this process are the horizontal and vertical indices of the current tile.

The outputs of this process are transform coefficients for each macroblock in the tile.

The transform coefficient processing proceeds as specified in Table 119.

Table 119 – Pseudocode for function TileTransformCoefficientProcessing( )

|  |  |
| --- | --- |
| **TileTransformCoefficientProcessing( ) {** | **Reference** |
| TileLevelDCQP( ) | 9.7.1.2 |
| TileLevelLPQP( ) | 9.7.2.2 |
| TileLevelHPQP( ) | 9.7.3.2 |
| n = 0 |  |
| for (MBy = TopMBIndexOfTile[TileIndexy]; MBy < TopMBIndexOfTile[TileIndexy + 1]; MBy++) |  |
| for (MBx = LeftMBIndexOfTile[TileIndexx]; MBx < LeftMBIndexOfTile[TileIndexx + 1]; MBx++) { |  |
| MBQPIndexLP[MBx][MBy] = LP\_QP\_INDEX[n] |  |
| MBQPIndexHP[MBx][MBy] = HP\_QP\_INDEX[n] |  |
| DCTransformCoefficientDecoding( ) | 9.4.2 |
| LPTransformCoefficientDecoding( ) | 9.4.3 |
| HPTransformCoefficientDecoding( ) | 9.4.4 |
| n += 1 |  |
| } |  |
| } |  |

NOTE – The computation of the global variable MBHPMode is dependent upon the completion of the LP transform coefficient decoding process. MBHPMode is computed at the beginning of the HP coefficient decoding process.

### DC transform coefficient decoding

The outputs of this process are DC transform coefficients for each colour component of the current macroblock.

The DC transform coefficient processing proceeds as specified in Table 120.

Table 120 – Pseudocode for function DCTransformCoefficientDecoding( )

|  |  |
| --- | --- |
| **DCTransformCoefficientDecoding( ) {** | **Reference** |
| DCMBCoefficientRemap( ) | 9.5.1 |
| DCPredictionGeneral( ) | 9.6.1.1 |
| DequantizeDCCoefficients( ) | 9.8.1 |
| } |  |

### Lowpass transform coefficient decoding

The outputs of this process are LP transform coefficients for each colour component of the current macroblock.

The LP transform coefficient processing proceeds as specified in Table 121.

Table 121 – Pseudocode for function LPTransformCoefficientDecoding( )

|  |  |
| --- | --- |
| **LPTransformCoefficientDecoding( ) {** | **Reference** |
| LPMBCoefficientRemap( ) | 9.5.2 |
| LPPredictionGeneral( ) | 9.6.2.1 |
| DequantizeLPCoefficients( ) | 9.8.2 |
| } |  |

### Highpass transform coefficient decoding

Input to this process is the value of MBHPMode for the current macroblock.

The outputs of this process are HP transform coefficients for each colour component of the current macroblock.

The HP transform coefficient processing proceeds as specified in Table 122.

Table 122 – Pseudocode for function HPTransformCoefficientDecoding( )

|  |  |
| --- | --- |
| **HPTransformCoefficientDecoding( ) {** | **Reference** |
| CalcHPPredMode( ) | 9.6.3.2 |
| HPMBCoefficientRemap( ) | 9.5.3 |
| DequantizeHPCoefficients( ) | 9.8.3 |
| HPCoefficientPrediction( ) | 9.6.3.3 |
| } |  |

## Coefficient remapping

This subclause specifies the coefficient remapping processes for DC, LP and HP coefficients.

### DC coefficient remapping

Input to this process is a list DCInput[i] of DC level values for each colour component i, and the variables MBx and MBy, which identify the current macroblock.

The output of this process is the list of values MbDCLP[MBx][MBy][i][0], of DC transform coefficients, for each colour component i of the current macroblock.

NOTE – The values DCInput[i] are the outputs of the MB\_DC( ) syntax structure of 8.7.11.

The values in the array MbDCLP[MBx][MBy][i][0] are set by Table 123.

Table 123 – Pseudocode for function DCMBCoefficientRemap( )

|  |  |
| --- | --- |
| **DCMBCoefficientRemap( ) {** | **Reference** |
| for (i = 0; i < NumComponents; i++) |  |
| MbDCLP[MBx][MBy][i][0] = DCInput[i] |  |
| } |  |

### Lowpass coefficient remapping

Inputs to this process are a list of variables LPInput[i][j] which hold the j-th LP transform coefficient value for each colour component i.

The output of this process is the array of variables MbDCLP[MBx][MBy][i][j] which hold the j-th LP transform coefficients, indexed in raster scan order, of colour component i.

The LP coefficient remapping process proceeds as specified in Table 124.

Table 124 – Pseudocode for function LPMBCoefficientRemap( )

| **LPMBCoefficientRemap( ) {** | **Reference** |
| --- | --- |
| for (i = 0; i < NumComponents; i++) |  |
| if (i != 0) && ((INTERNAL\_CLR\_FMT = = YUV422) | |   (INTERNAL\_CLR\_FMT = = YUV420)) |  |
| if (INTERNAL\_CLR\_FMT = = YUV422) |  |
| for (j = 1; j <= 7; j++) |  |
| if (BANDS\_PRESENT != DCONLY) |  |
| MbDCLP[MBx][MBy][i][j] = LPInput[i][j] |  |
| else |  |
| MbDCLP[MBx][MBy][i][j] = 0 |  |
| else /\* INTERNAL\_CLR\_FMT = = YUV420 \*/ |  |
| for (j = 1; j <= 3; j++) |  |
| if (BANDS\_PRESENT != DCONLY) |  |
| MbDCLP[MBx][MBy][i][j] = LPInput[i][j] |  |
| else |  |
| MbDCLP[MBx][MBy][i][j] = 0 |  |
| else |  |
| for (j = 1; j <= 15; j++) |  |
| if (BANDS\_PRESENT != DCONLY) |  |
| MbDCLP[MBx][MBy][i][j] = LPInput[i][j] |  |
| else |  |
| MbDCLP[MBx][MBy][i][j] = 0 |  |
| } |  |

### Highpass macroblock coefficient remapping

The HP coefficient remapping process proceeds as specified in Table 125.

Table 125 – Pseudocode for function HPMBCoefficientRemap( )

|  |  |
| --- | --- |
| **HPMBCoefficientRemap( ) {** | **Reference** |
| for (i = 0; i < NumComponents; i++) { |  |
| if (i != 0 && INTERNAL\_CLR\_FMT = = YUV420) |  |
| jMax = 3 |  |
| else if (i != 0 && INTERNAL\_CLR\_FMT = = YUV422) |  |
| jMax = 7 |  |
| else |  |
| jMax = 15 |  |
| for (j = 0; j <= jMax; j++) |  |
| HPBlockCoefficientRemap(i, j) | 9.5.4 |
| } |  |
| } |  |

### Highpass block coefficient remapping

Inputs to this process are: the array HPInputVLC[currentComponent][blkIndex][j] for the current colour component currentComponent, and the current block index blkIndex, with j ranging from 1 to 15, the array HPInputFlex[currentComponent][blkIndex][j] for the current colour component currentComponent, and the current block index blkIndex, with j ranging from 1 to 15, the variable ModelBits[MBx][MBy], representing the number of flexbits for the current macroblock, and the variables MBx and MBy, which identify the current macroblock.

Outputs of this process are the values MBBuffer[MBx][MBy][currentComponent][k] of HP transform coefficients, with k ranging from (16 \* blkIndex + 1) to (16 \* blkIndex + 15), corresponding to the current block. Pseudocode for this process is specified in Table 126.

Table 126 – Pseudocode for function HPBlockCoefficientRemap( )

|  |  |
| --- | --- |
| **HPBlockCoefficientRemap(currentComponent, blkIndex) {** | **Reference** |
| if (currentComponent = = 0) |  |
| iIndex = 0 |  |
| else |  |
| iIndex = 1 |  |
| for (j = 1; j <= 15; j++) { |  |
| k = 16 \* blkIndex + j |  |
| if (BANDS\_PRESENT = = ALL | |  BANDS\_PRESENT = = NOFLEXBITS) |  |
| MBBuffer[MBx][MBy][currentComponent][k] =   HPInputVLC[currentComponent][blkIndex][j] <<   ModelBitsMBHP[MBx][MBy][iIndex] |  |
| else |  |
| MBBuffer[MBx][MBy][currentComponent][k] = 0 |  |
| if (BANDS\_PRESENT = = ALL) |  |
| MBBuffer[MBx][MBy][currentComponent][k] +=   HPInputFlex[currentComponent][blkIndex][j] |  |
| } |  |
| } |  |

## Transform coefficient prediction

### DC coefficient prediction

#### Overview of DC prediction

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

Four modes are defined for the prediction of the DC coefficient of a macroblock. These modes are:

* predict from left;
* predict from top;
* predict from left and top;
* no prediction.

The prediction mode is determined from the position of the macroblock, as well as the DC values to the left, top and top-left of the macroblock. Furthermore, if the image has chroma components, the corresponding DC values of the chroma components are also used.

#### DCPredictionGeneral( )

Inputs to this process are the Boolean variables IsMBLeftEdgeofTileFlag and IsMBTopEdgeofTileFlag, as well as the values MbDCLP[MBx][MBy][i][0], for each colour component i of the current macroblock. The variable IsMBLeftEdgeofTileFlag is equal to TRUE when the current macroblock is at the left edge of the tile; IsMBTopEdgeofTileFlag is equal to TRUE when the current macroblock is at the top edge of the current tile.

NOTE – The values MbDCLP[MBx][MBy][i][0] come from the DC coefficient remapping process of clause 9.5.1.

Outputs of this process are the updated values MbDCLP[MBx][MBy][i][0], for each colour component i of the current macroblock.

The DC prediction process proceeds as specified in Table 127.

Table 127 – Pseudocode for function DCPredictionGeneral( )

|  |  |
| --- | --- |
| **DCPredictionGeneral( ) {** | **Reference** |
| CalcDCPredMode( ) | 9.6.1.3 |
| DCCoefficientPrediction( ) | 9.6.1.4 |
| UpdateDCPredictionVariables( ) | 9.6.1.5 |
| } |  |

#### DC prediction direction computation

Inputs to this process are the variables IsMBLeftEdgeofTileFlag and IsMBTopEdgeofTileFlag, and the values PredDCLP[MBx−1][MBy][i][0], PredDCLP[MBx][MBy−1][i][0] and PredDCLP[MBx−1][MBy−1][i][0], for each colour component i.

The output of this process is the value of MBDCMode. The possible values of MBDCMode are as follows: 0 specifies prediction from the left macroblock; 1 specifies prediction from the top macroblock;2 specifies prediction from both the top and left macroblocks; and 3 specifies no prediction.

The DC prediction direction process proceeds as specified in Table 128.

Table 128 – Pseudocode for function CalcDCPredMode( )

| **CalcDCPredMode( ) {** | **Reference** |
| --- | --- |
| if (IsMBLeftEdgeofTileFlag = = TRUE &&   IsMBTopEdgeofTileFlag = = TRUE) |  |
| MBDCMode = 3 /\* no prediction \*/ |  |
| else if (IsMBLeftEdgeofTileFlag = = TRUE &&   IsMBTopEdgeofTileFlag = = FALSE) |  |
| MBDCMode = 1 /\* prediction from top only \*/ |  |
| else if (IsMBLeftEdgeofTileFlag = = FALSE &&   IsMBTopEdgeofTileFlag = = TRUE) |  |
| MBDCMode = 0 /\* prediction from left only \*/ |  |
| else /\* if (IsMBLeftEdgeofTileFlag = = FALSE &&   IsMBTopEdgeofTileFlag = = FALSE) \*/ { |  |
| iLeft = PredDCLP[MBx−1][MBy][0][0] |  |
| iTop = PredDCLP[MBx][MBy−1][0][0] |  |
| iTopLeft = PredDCLP[MBx−1][MBy−1][0][0] |  |
| if (INTERNAL\_CLR\_FMT = = Y\_ONLY | |   INTERNAL\_CLR\_FMT = =NCOMPONENT) { |  |
| iStrHor = Abs(iTopLeft − iLeft) |  |
| iStrVer = Abs(iTopLeft − iTop) |  |
| } else { |  |
| iLeftU = PredDCLP[MBx−1][MBy][1][0] |  |
| iTopU = PredDCLP[MBx][MBy−1][1][0] |  |
| iTopLeftU = PredDCLP[MBx−1][MBy−1][1][0] |  |
| iLeftV = PredDCLP[MBx−1][MBy][2][0] |  |
| iTopV = PredDCLP[MBx][MBy−1][2][0] |  |
| iTopLeftV = PredDCLP[MBx−1][MBy−1][2][0] |  |
| iScale = 2 |  |
| if (INTERNAL\_CLR\_FMT = = YUV420) |  |
| iScale = 8 |  |
| if (INTERNAL\_CLR\_FMT = = YUV422) |  |
| iScale = 4 |  |
| iStrHor = Abs(iTopLeft − iLeft) \* iScale +   Abs(iTopLeftU − iLeftU) + Abs(iTopLeftV − iLeftV) |  |
| iStrVer = Abs(iTopLeft − iTop) \* iScale +   Abs(iTopLeftU − iTopU) + Abs(iTopLeftV − iTopV) |  |
| } |  |
| iOrWt = 4 |  |
| if ((iStrHor \* iOrWt) < iStrVer) |  |
| MBDCMode = 1 |  |
| else if ((iStrVer \* iOrWt) < iStrHor) |  |
| MBDCMode = 0 |  |
| else |  |
| MBDCMode = 2 |  |
| } |  |
| } |  |

The value MBDCMode is used in subsequent stages of the DC prediction process.

#### DC coefficient prediction

This process occurs when MBDCMode is not equal to 3.

Inputs to this process are the variable MBDCMode representing the DC prediction direction, and the array variable MbDCLP[MBx][MBy][i][0] for each colour component i.

Outputs to this process are the updated values MbDCLP[MBx][MBy][i][0], for each colour component i of the current macroblock.

The DC coefficient prediction process proceeds as specified in Table 129.

Table 129 – Pseudocode for function DCCoefficientPrediction( )

|  |  |
| --- | --- |
| **DCCoefficientPrediction( ) {** | **Reference** |
| for (i = 0; i < NumComponents; i++) { |  |
| iLeft = PredDCLP[MBx−1][MBy][i][0] |  |
| iTop = PredDCLP[MBx][MBy−1][i][0] |  |
| if (MBDCMode = = 0) |  |
| MbDCLP[MBx][MBy][i][0] += iLeft |  |
| else if (MBDCMode = = 1) |  |
| MbDCLP[MBx][MBy][i][0] += iTop |  |
| else if (MBDCMode = = 2) { |  |
| if (i = = 0) | | ((INTERNAL\_CLR\_FMT != YUV420) &&   (INTERNAL\_CLR\_FMT != YUV422)) |  |
| MbDCLP[MBx][MBy][i][0] += (iTop + iLeft) >> 1 |  |
| else /\* (INTERNAL\_CLR\_FMT = = YUV420 | |   INTERNAL\_CLR\_FMT = = YUV422)\*/ |  |
| MbDCLP[MBx][MBy][i][0] +=   (iTop + iLeft + 1) >> 1 |  |
| } |  |
| } |  |
| } |  |

#### Update of DC prediction variables

Outputs of this process are the updated variables PredDCLP[MBx][MBy][i], for each colour component i, where MBx and MBy are indexing the current macroblock.

The update of DC prediction variables process proceeds as specified in Table 130.

Table 130 – Pseudocode for function UpdateDCPredictionVariables( )

|  |  |
| --- | --- |
| **UpdateDCPredictionVariables( ) {** | **Reference** |
| for (i = 0; i < NumComponents; i++) |  |
| PredDCLP[MBx][MBy][i][0] = MbDCLP[MBx][MBy][i][0] |  |
| } |  |

### Lowpass prediction

#### Overview of lowpass prediction

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

Three modes are defined for the prediction of the LP coefficient of the inner transform of a macroblock. These modes are:

* prediction from left when MBLPMode is equal to 0;
* prediction from top when MBLPMode is equal to 1;
* no prediction when MBLPMode is equal to 2.

The LP coefficient prediction mode (MBLPMode) is determined by the DC coefficient prediction mode, together with the quantization parameters of both the current block and the block from which the DC values were predicted.

This rule ensures that prediction of LP coefficients does not take place across macroblocks with different quantization parameters.

Not all the LP coefficients associated with a macroblock are predicted. The definition and indices of DC and LP coefficients that are predicted are shown in Figure 5. The DC coefficient of the blocks shown in dark grey is at position 0 and the LP coefficients that can be predicted are shown in light grey. For the colour format YUV422, the LP coefficient associated with position 5 is predicted from position 1 in Figure 5, and if MBDCMode is equal to 1, irrespective of the value of MBLPMode, the LP coefficient associated with position 4 (indicated by crosshatching) can be predicted from both top and left.



Figure 5 – DC and LP coefficients in: (a) 4×4; (b) 422 chroma; and (c) 420 chroma block

#### LPPredictionGeneral( )

Inputs to this process are the variable MBDCMode for the current macroblock, as well as the values MbDCLP[MBx][MBy][i][j], for each colour component i of the current macroblock, and index j referencing the LP transform coefficients, indexed in raster scan order.

NOTE – The values MbDCLP[MBx][MBy][i][j] come from the LP coefficient remapping process of 9.5.2.

Outputs to this process are the values MbDCLP[MBx][MBy][i][j], for each colour component i of the current macroblock.

The LP prediction process proceeds as specified in Table 131.

Table 131 – Pseudocode for function LPPredictionGeneral( )

|  |  |
| --- | --- |
| **LPPredictionGeneral( ) {** | **Reference** |
| CalcLPPredMode( ) | 9.6.2.3 |
| LPCoefficientPrediction( ) | 9.6.2.4 |
| UpdateLPPredictionVariables( ) | 9.6.2.5 |
| } |  |

#### Lowpass prediction direction computation

Inputs to this process are the variables MBDCMode and MBQPIndexLP[MBx][MBy].

The output to this process is the value of MBLPMode. The possible values of MBLPMode are as follows: the value 0 represents prediction from the left macroblock; the value 1 represents prediction from the top macroblock; and the value 2 represents no prediction.

The LP prediction direction process proceeds as specified in Table 132.

Table 132 – Pseudocode for function CalcLPPredMode( )

|  |  |
| --- | --- |
| **CalcLPPredMode( ) {** | **Reference** |
| if (MBDCMode = = 0 &&  MBQPIndexLP[MBx][MBy] = = MBQPIndexLP[MBx−1][MBy]) |  |
| MBLPMode = 0 |  |
| else if (MBDCMode = = 1 &&   MBQPIndexLP[MBx][MBy] = = MBQPIndexLP[MBx][MBy−1]) |  |
| MBLPMode = 1 |  |
| else |  |
| MBLPMode = 2 |  |
| } |  |

#### Lowpass coefficient prediction

Inputs to this process are: the variable MBLPMode representing the LP prediction direction; the variables MBx and MBy, which index the current macroblock in the image; and the variables PredDCLP[MBx][MBy][i][j].

Outputs to this process are the updated values MbDCLP[MBx][MBy][i][j], for each colour component i of the current macroblock, and j an index referencing the LP transform coefficients, indexed in raster scan order.

The LP coefficient prediction process proceeds as specified in Table 133.

Table 133 – Pseudocode for function LPCoefficientPrediction( )

| **LPCoefficientPrediction( ) {** | **Reference** |
| --- | --- |
| for (i = 0; i < NumComponents; i++) { |  |
| if (i = = 0 | | ((INTERNAL\_CLR\_FMT != YUV420) &&   (INTERNAL\_CLR\_FMT != YUV422))) { |  |
| if (MBLPMode = = 0) { |  |
| MbDCLP[MBx][MBy][i][4] += PredDCLP[MBx−1][MBy][i][4] |  |
| MbDCLP[MBx][MBy][i][8] += PredDCLP[MBx−1][MBy][i][5] |  |
| MbDCLP[MBx][MBy][i][12] += PredDCLP[MBx−1][MBy][i][6] |  |
| } else if (MBLPMode = = 1) { |  |
| MbDCLP[MBx][MBy][i][1] += PredDCLP[MBx][MBy−1][i][1] |  |
| MbDCLP[MBx][MBy][i][2] += PredDCLP[MBx][MBy−1][i][2] |  |
| MbDCLP[MBx][MBy][i][3] += PredDCLP[MBx][MBy−1][i][3] |  |
| } |  |
| } else if (INTERNAL\_CLR\_FMT = = YUV420) { /\* i is equal to 1 or 2 here \*/ |  |
| if (MBLPMode = = 0) /\* Prediction from left \*/ |  |
| MbDCLP[MBx][MBy][i][2] += PredDCLP[MBx−1][MBy][i][2] |  |
| else if (MBLPMode = = 1) /\* Prediction from top \*/ |  |
| MbDCLP[MBx][MBy][i][1] += PredDCLP[MBx][MBy−1][i][1] |  |
| } else if (INTERNAL\_CLR\_FMT = = YUV422) /\* i is equal to 1 or 2 here \*/ |  |
| if (MBLPMode = = 0) { /\* Prediction from left \*/ |  |
| MbDCLP[MBx][MBy][i][4] += PredDCLP[MBx−1][MBy][i][4] |  |
| MbDCLP[MBx][MBy][i][2] += PredDCLP[MBx−1][MBy][i][2] |  |
| MbDCLP[MBx][MBy][i][6] += PredDCLP[MBx−1][MBy][i][6] |  |
| } else if (MBLPMode = = 1) { /\* Prediction from top \*/ |  |
| MbDCLP[MBx][MBy][i][4] += PredDCLP[MBx][MBy−1][i][4] |  |
| MbDCLP[MBx][MBy][i][1] += PredDCLP[MBx][MBy−1][i][5] |  |
| MbDCLP[MBx][MBy][i][5] += MbDCLP[MBx][MBy][i][1]  /\* In this line, prediction occurs using the *current* macroblock's data \*/ |  |
| } else if (MBDCMode = = 1) |  |
| MbDCLP[MBx][MBy][i][5] += MbDCLP[MBx][MBy][i][1]   /\* When the colour format is YUV422,  MBLPMode is equal to 2 (no prediction), and  MBDCMode is equal to 1 (prediction from the top), the LP  coefficient associated with j=5 is predicted from that for j=1 \*/ |  |
| } |  |
| } |  |

#### Update of lowpass prediction variables

Inputs to this process are the variables MBx and MBy, which index the current macroblock in the image.

Outputs of this process are the variables PredDCLP[MBx][MBy][i][j], for each colour component i, and selected LP indices j.

The update of LP prediction variables process proceeds as specified in Table 134.

Table 134 – Pseudocode for function UpdateLPPredictionVariables( )

|  |  |
| --- | --- |
| **UpdateLPPredictionVariables( ) {** | **Reference** |
| for (i = 0; i < NumComponents; i++) { |  |
| if (i = = 0 | | ((INTERNAL\_CLR\_FMT != YUV420) &&   (INTERNAL\_CLR\_FMT != YUV422))) { |  |
| PredDCLP[MBx][MBy][i][1] = MbDCLP[MBx][MBy][i][1] |  |
| PredDCLP[MBx][MBy][i][2] = MbDCLP[MBx][MBy][i][2] |  |
| PredDCLP[MBx][MBy][i][3] = MbDCLP[MBx][MBy][i][3] |  |
| PredDCLP[MBx][MBy][i][4] = MbDCLP[MBx][MBy][i][4] |  |
| PredDCLP[MBx][MBy][i][5] = MbDCLP[MBx][MBy][i][8] |  |
| PredDCLP[MBx][MBy][i][6] = MbDCLP[MBx][MBy][i][12] |  |
| } else if (INTERNAL\_CLR\_FMT = = YUV420) { |  |
| PredDCLP[MBx][MBy][i][1] = MbDCLP[MBx][MBy][i][1] |  |
| PredDCLP[MBx][MBy][i][2] = MbDCLP[MBx][MBy][i][2] |  |
| } else if (INTERNAL\_CLR\_FMT = = YUV422) { |  |
| PredDCLP[MBx][MBy][i][1] = MbDCLP[MBx][MBy][i][1] |  |
| PredDCLP[MBx][MBy][i][2] = MbDCLP[MBx][MBy][i][2] |  |
| PredDCLP[MBx][MBy][i][4] = MbDCLP[MBx][MBy][i][4] |  |
| PredDCLP[MBx][MBy][i][5] = MbDCLP[MBx][MBy][i][5] |  |
| PredDCLP[MBx][MBy][i][6] = MbDCLP[MBx][MBy][i][6] |  |
| } |  |
| } |  |
| } |  |

### Highpass prediction

#### Overview of highpass prediction

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

There are two prediction processes involving HP coefficients: the HP prediction direction process and the HP prediction process. The process that computes HP prediction direction also sets the HP direction variable MBHPMode, which determines the initial HP inverse scanning order. Therefore, the process that computes HP prediction direction shall be executed before the HP coefficient inverse scanning process. The HP prediction process is executed after the HP coefficient remapping process.

Information in the LP transform coefficients is used to compute a simple metric to determine the *orientation* of prediction of HP coefficients associated with each macroblock. Three modes are defined for the prediction of the HP coefficients of the outer transform. The same mode is used for all blocks within a macroblock for which in-macroblock prediction is possible. For blocks that have no valid reference within the macroblock, null prediction is used. The three modes are:

* prediction from left when MBHPMode is equal to 0;
* prediction from top when MBHPMode is equal to 1;
* no prediction when MBHPMode is equal to 2.

Prediction from left is shown in Figure 6. Prediction from top is similar, with the pattern of arrows transposed to point downwards.



Figure 6 – HP prediction from left

NOTE – In the implementation of a decoder, the only information that needs to be available for future use is 1 DC + 6 LP = 7 coefficients per macroblock component (fewer for YUV420 / YUV422 chrominance). Therefore, at most for YUV444, 21 coefficients need to be cached per macroblock. Furthermore, the coefficients used for prediction from left can be discarded after the next macroblock is predicted. For YUV444, therefore, it is necessary to only cache 12 coefficients per macroblock for use in the next row of macroblocks. More state is required on the encoder side: the HP coefficients must be maintained throughout this process, as the encoding of HP coefficients is dependent on the encoding of LP coefficients.

#### Highpass prediction direction computation

Inputs to this process are the variables MBx and MBy, indexing the location of the current macroblock in the image.

The output of this process is the variable MBHPMode for the current macroblock.

The HP prediction process proceeds as specified in Table 135.

Table 135 – Pseudocode for function CalcHPPredMode( )

|  |  |
| --- | --- |
| **CalcHPPredMode( ) {** | **Reference** |
| iStrHor = Abs(MbDCLP[MBx][MBy][0][1]) +   Abs(MbDCLP[MBx][MBy][0][2]) +   Abs(MbDCLP[MBx][MBy][0][3]) |  |
| iStrVer = Abs(MbDCLP[MBx][MBy][0][4]) +   Abs(MbDCLP[MBx][MBy][0][8]) +   Abs(MbDCLP[MBx][MBy][0][12]) |  |
| if ((INTERNAL\_CLR\_FMT != YONLY) &&   (INTERNAL\_CLR\_FMT != NCOMPONENT)) { |  |
| for (i = 1; i <= 2; i++) { |  |
| iStrHor += Abs(MbDCLP[MBx][MBy][i][1]) |  |
| if (INTERNAL\_CLR\_FMT = = YUV420) |  |
| iStrVer += Abs(MbDCLP[MBx][MBy][i][2]) |  |
| else if (INTERNAL\_CLR\_FMT = = YUV422) { |  |
| iStrVer += Abs(MbDCLP[MBx][MBy][i][2]) +   Abs(MbDCLP[MBx][MBy][i][6]) |  |
| iStrHor += Abs(MbDCLP[MBx][MBy][i][5]) |  |
| } else |  |
| iStrVer += Abs(MbDCLP[MBx][MBy][i][4]) |  |
| } |  |
| } |  |
| iOrWt = 4 |  |
| if (iStrHor \* iOrWt < iStrVer) |  |
| MBHPMode = 0 /\* predict from left \*/ |  |
| else if (iStrVer \* iOrWt < iStrHor) |  |
| MBHPMode = 1 /\* predict from top \*/ |  |
| else |  |
| MBHPMode = 2 /\* no prediction \*/ |  |
| } |  |

#### Highpass prediction

Inputs to this process are: the variable MBHPMode, which indicates the HP prediction direction; the values MBx and MBy, which index the current macroblock in the image; and the values MBBuffer[MBx][MBy][i][j], which hold the HP transform coefficients obtained from the HP coefficient remapping process of 9.5.3. The outputs of this process are the updated values MBBuffer[MBx][MBy][i][k] of HP transform coefficients.

The HP prediction process proceeds as specified in Table 136.

Table 136 – Pseudocode for function HPCoefficientPrediction( )

| **HPCoefficientPrediction( ) {** | **Reference** |
| --- | --- |
| if (INTERNAL\_CLR\_FMT = = YUV420) | | (INTERNAL\_CLR\_FMT = = YUV422) |  |
| iComponents = 1 |  |
| else |  |
| iComponents = NumComponents |  |
| for (i = 0; i < iComponents; i++) { |  |
| if (MBHPMode = = 0) { |  |
| blkId[ ] = {1,2,3,5,6,7,9,10,11,13,14,15} |  |
| for (j = 0; j < 12; j++) { |  |
| MBBuffer[MBx][MBy][i][16\*blkId[j] + 4] +=   MBBuffer[MBx][MBy][i][16\*(blkId[j] − 1) + 4] |  |
| MBBuffer[MBx][MBy][i][16\*blkId[j] + 8] +=   MBBuffer[MBx][MBy][i][16\*(blkId[j] − 1) + 8] |  |
| MBBuffer[MBx][MBy][i][16\*blkId[j] + 12] +=   MBBuffer[MBx][MBy][i][16\*(blkId[j] − 1) + 12] |  |
| } |  |
| } else if (MBHPMode = = 1) { |  |
| blkId[ ] = {4,5,6,7,8,9,10,11,12,13,14,15} |  |
| for (j = 0; j < 12; j++) { |  |
| MBBuffer[MBx][MBy][i][16\*blkId[j] + 1] +=   MBBuffer[MBx][MBy][i][16\*(blkId[j] − 4) + 1] |  |
| MBBuffer[MBx][MBy][i][16\*blkId[j] + 2] +=   MBBuffer[MBx][MBy][i][16\*(blkId[j] − 4) + 2] |  |
| MBBuffer[MBx][MBy][i][16\*blkId[j] + 3] +=   MBBuffer[MBx][MBy][i][16\*(blkId[j] − 4) + 3] |  |
| } |  |
| } |  |
| } |  |
| if (INTERNAL\_CLR\_FMT = = YUV420) { |  |
| for (i = 1; i <= 2; i++) { |  |
| if (MBHPMode = = 0) { |  |
| blkId[ ] = {1,3} |  |
| for (j = 0; j < 2; j++) { |  |
| MBBuffer[MBx][MBy][i][16\*blkId[j] + 4] +=   MBBuffer[MBx][MBy][i][16\*(blkId[j] − 1) + 4] |  |
| MBBuffer[MBx][MBy][i][16\*blkId[j] + 8] +=   MBBuffer[MBx][MBy][i][16\*(blkId[j] − 1) + 8] |  |
| MBBuffer[MBx][MBy][i][16\*blkId[j] + 12] +=   MBBuffer[MBx][MBy][i][16\*(blkId[j] − 1) + 12] |  |
| } |  |
| } else if (MBHPMode = = 1) { |  |
| blkId[ ] = {2,3} |  |
| for (j = 0; j < 2; j++) { |  |
| MBBuffer[MBx][MBy][i][16\*blkId[j] + 1] +=   MBBuffer[MBx][MBy][i][16\*(blkId[j] − 2) + 1] |  |
| MBBuffer[MBx][MBy][i][16\*blkId[j] + 2] +=   MBBuffer[MBx][MBy][i][16\*(blkId[j] − 2) + 2] |  |
| MBBuffer[MBx][MBy][i][16\*blkId[j] + 3] +=   MBBuffer[MBx][MBy][i][16\*(blkId[j] − 2) + 3] |  |
| } |  |
| } |  |
| } |  |
| } else if (INTERNAL\_CLR\_FMT = = YUV422) { |  |
| for (i = 1; i <= 2; i++) { |  |
| if (MBHPMode = = 0) { |  |
| blkId[ ] = {1,3,5,7} |  |
| for (j = 0; j < 4; j++) { |  |
| MBBuffer[MBx][MBy][i][16\*blkId[j] + 4] +=   MBBuffer[MBx][MBy][i][16\*(blkId[j] − 1) + 4] |  |
| MBBuffer[MBx][MBy][i][16\*blkId[j] + 8] +=   MBBuffer[MBx][MBy][i][16\*(blkId[j] − 1) + 8] |  |
| MBBuffer[MBx][MBy][i][16\*blkId[j] + 12] +=   MBBuffer[MBx][MBy][i][16\*(blkId[j] − 1) + 12] |  |
| } |  |
| } else if (MBHPMode = = 1) { |  |
| blkId[ ] = {2,4,6,3,5,7} |  |
| for (j = 0; j < 6; j++) { |  |
| MBBuffer[MBx][MBy][i][16\*blkId[j] + 1] +=   MBBuffer[MBx][MBy][i][16\*(blkId[j] − 2) + 1] |  |
| MBBuffer[MBx][MBy][i][16\*blkId[j] + 2] +=   MBBuffer[MBx][MBy][i][16\*(blkId[j] − 2) + 2] |  |
| MBBuffer[MBx][MBy][i][16\*blkId[j] + 3] +=   MBBuffer[MBx][MBy][i][16\*(blkId[j] − 2) + 3] |  |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  |

## Derivation of quantization parameters

### Derivation of DC quantization parameters

#### Image plane level derivation of DC quantization parameters

This process derives the array DCQuantParam[i] of image plane level DC quantization parameters for each colour component i, if these parameters are specified at the image plane level; otherwise, it does nothing, and it is expected that the tile-level derivation process will derive the array DCQuantParam[i], for each tile in the image plane.

The image plane level derivation process of DC quantization parameters proceeds as specified in Table 137.

Table 137 – Pseudocode for function ImagePlaneDCQP( )

|  |  |
| --- | --- |
| **ImagePlaneDCQP( ) {** | **Reference** |
| if (DC\_IMAGE\_PLANE\_UNIFORM\_FLAG = = TRUE) |  |
| AssignDCQuantizationParameters( ) | 9.7.1.3 |
| } |  |

NOTE – If DC\_IMAGE\_PLANE\_UNIFORM\_FLAG is equal to FALSE, the quantization parameters are specified at tile level.

#### Tile level derivation of DC quantization parameters

If these parameters are specified at the tile level, this process derives the array DCQuantParam[i] of tile-level DC quantization parameters for each colour component i. Otherwise, it does nothing and it is assumed that the image plane level derivation process has already derived the array DCQuantParam[i].

The tile-level derivation process of DC quantization parameters proceeds as specified in Table 138.

Table 138 – Pseudocode for function TileLevelDCQP( )

|  |  |
| --- | --- |
| **TileLevelDCQP( ) {** | **Reference** |
| if (DC\_IMAGE\_PLANE\_UNIFORM\_FLAG = = FALSE) |  |
| AssignDCQuantizationParameters( ) | 9.7.1.3 |
| } |  |

NOTE – If DC\_IMAGE\_PLANE\_UNIFORM\_FLAG is equal to TRUE, quantization parameters were set at the image plane level.

#### Assignment of DC quantization parameters

The assignment process of DC quantization parameters proceeds as specified in Table 139.

Table 139 – Pseudocode for function AssignDCQuantizationParameters( )

|  |  |
| --- | --- |
| **AssignDCQuantizationParameters( ) {** | **Reference** |
| if (COMPONENT\_MODE = = UNIFORM) |  |
| for (i = 0; i <=NumComponents−1; i++) |  |
| DCQuantParam[i] = DC\_QUANT |  |
| else if (COMPONENT\_MODE = = SEPARATE) { |  |
| DCQuantParam[0] = DC\_QUANT\_LUMA |  |
| for (i = 1; i <=NumComponents−1; i++) |  |
| DCQuantParam[i] = DC\_QUANT\_CHROMA |  |
| } else if (COMPONENT\_MODE = = INDEPENDENT) |  |
| for (i = 0; i <=NumComponents−1; i++) |  |
| DCQuantParam[i] = DC\_QUANT\_CH[i] |  |
| } |  |

### Derivation of lowpass quantization parameters

#### Image plane level derivation of lowpass quantization parameters

This process derives the values LPQuantParam[i][j] of image plane level LP quantization parameters, for each colour component i, and each allowable index j (for image plane level LP quantization parameters, j can only take the value 0). These values are derived if LP\_IMAGE\_PLANE\_UNIFORM\_FLAG is equal to TRUE, which indicates that these quantization parameters are specified at the image plane level; otherwise, it does nothing, and it is expected that the tile-level derivation process will derive the array LPQuantParam[i][j], for each tile in the image plane.

The image plane level derivation process of LP quantization parameters proceeds as specified in Table 140.

Table 140 – Pseudocode for function ImagePlaneLPQP( )

|  |  |
| --- | --- |
| **ImagePlaneLPQP( ) {** | **Reference** |
| if (LP\_IMAGE\_PLANE\_UNIFORM\_FLAG) |  |
| AssignLPQuantizationParameters( ) | 9.7.2.3 |
| } |  |

NOTE 1 – When LP\_IMAGE\_PLANE\_UNIFORM\_FLAG is equal to TRUE, NumLPQPs is always equal to 1.

NOTE 2 – If LP\_IMAGE\_PLANE\_UNIFORM\_FLAG is equal to FALSE, quantization parameters are set at the tile level.

#### Tile level derivation of lowpass quantization parameters

This process derives the values LPQuantParam[i][j] of tile-level LP quantization parameters, for each colour component i, and each allowable index j (ranging from 0 to NumLPQPs−1). These values are derived if these quantization parameters are specified at the tile level; otherwise, it does nothing, and the image plane level derivation process derived the array LPQuantParam[i][j], used for each tile in the current image plane.

The tile-level derivation process of LP quantization parameters proceeds as specified in Table 141.

Table 141 – Pseudocode for function TileLevelLPQP( )

|  |  |
| --- | --- |
| **TileLevelLPQP( ) {** | **Reference** |
| if ((LP\_IMAGE\_PLANE\_UNIFORM\_FLAG != TRUE) &&   (USE\_DC\_QP\_FLAG = = TRUE)) { |  |
| NumLPQPs = 1 |  |
| for (i = 0; i <=NumComponents−1; i++) |  |
| LPQuantParam[i][0] = DCQuantParam[i] |  |
| } else if (LP\_IMAGE\_PLANE\_UNIFORM\_FLAG != TRUE) |  |
| AssignLPQuantizationParameters( ) | 9.7.2.3 |
| } |  |

NOTE – When LP\_IMAGE\_PLANE\_UNIFORM\_FLAG is equal to TRUE, nothing is done in this function, because quantization parameters were set at image plane level.

#### Assignment of lowpass quantization parameters

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

Table 142 – Pseudocode for function AssignLPQuantizationParameters( )

|  |  |
| --- | --- |
| **AssignLPQuantizationParameters( ) {** | **Reference** |
| for (j = 0; j < NumLPQPs; j++) { |  |
| if (COMPONENT\_MODE = = UNIFORM) |  |
| for (i = 0; i <=NumComponents−1; i++) |  |
| LPQuantParam[i][j] = LP\_QUANT[j] |  |
| else if (COMPONENT\_MODE = = SEPARATE) { |  |
| LPQuantParam[0][j] = LP\_QUANT\_LUMA[j] |  |
| for (i = 1; i <=NumComponents−1; i++) |  |
| LPQuantParam[i][j] = LP\_QUANT\_CHROMA[j] |  |
| } else if (COMPONENT\_MODE = = INDEPENDENT) |  |
| for (i = 0; i <=NumComponents−1; i++) |  |
| LPQuantParam[i][j] = LP\_QUANT\_CH[i][j] |  |
| } |  |
| } |  |

### Derivation of highpass quantization parameters

#### Image plane level derivation of highpass quantization parameters

This process derives the values HPQuantParam[i][j] of image plane level HP quantization parameters, for each colour component i, and each allowable index j. These values are derived if these quantization parameters are specified at the image plane level; otherwise, it does nothing, and it is expected that the tile-level derivation process will derive the array HPQuantParam[i][j], for each tile in the image plane.

The image plane level derivation process of HP quantization parameters proceeds as specified in Table 143.

Table 143 – Pseudocode for function ImagePlaneHPQP( )

|  |  |
| --- | --- |
| **ImagePlaneHPQP( ) {** | **Reference** |
| if (HP\_IMAGE\_PLANE\_UNIFORM\_FLAG) |  |
| AssignHPQuantizationParameters( ) | 9.7.3.3 |
| } |  |

NOTE 1 – When HP\_IMAGE\_PLANE\_UNIFORM\_FLAG is equal to TRUE, NumHPQPs is always equal to 1.

NOTE 2 – When HP\_IMAGE\_PLANE\_UNIFORM\_FLAG is equal to FALSE, quantization parameters are set at the tile level.

#### Tile level derivation of highpass quantization parameters

This process derives the values HPQuantParam[i][j] of tile-level HP quantization parameters, for each colour component i, and each allowable index j (ranging from 0 to NumHPQPs−1). These values are derived if these quantization parameters are specified at the tile level; otherwise, this process has no effect, and it is expected that the image plane level derivation process derived the array HPQuantParam[i][j], used for each tile in the current image plane.

The tile-level derivation process of HP quantization parameters proceeds as specified in Table 144.

Table 144 – Pseudocode for function TileLevelHPQP( )

|  |  |
| --- | --- |
| **TileLevelHPQP( ) {** | **Reference** |
| if ((HP\_IMAGE\_PLANE\_UNIFORM\_FLAG != TRUE) &&   (USE\_LP\_QP\_FLAG = = TRUE)) { |  |
| NumHPQPs = NumLPQPs |  |
| for (i = 0; i <=NumComponents−1; i++) |  |
| for (j = 0; j <=NumLPQPs; j++) |  |
| HPQuantParam[i][j] = LPQuantParam[i][j] |  |
| } else if (HP\_IMAGE\_PLANE\_UNIFORM\_FLAG != TRUE) |  |
| AssignHPQuantizationParameters( ) | 9.7.3.3 |
| } |  |

NOTE – When HP\_IMAGE\_PLANE\_UNIFORM\_FLAG is equal to TRUE, nothing is done in this function, because quantization parameters were set at image plane level.

#### Assignment of highpass quantization parameters

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

Table 145 – Pseudocode for function AssignHPQuantizationParameters( )

|  |  |
| --- | --- |
| **AssignHPQuantizationParameters( ) {** | **Reference** |
| for (j = 0; j < NumHPQPs; j++) { |  |
| if (COMPONENT\_MODE = = UNIFORM) |  |
| for (i = 0; i <=NumComponents−1; i++) |  |
| HPQuantParam[i][j] = HP\_QUANT[j] |  |
| else if (COMPONENT\_MODE = = SEPARATE) { |  |
| HPQuantParam[0][j] = HP\_QUANT\_LUMA[j] |  |
| for (i = 1; i <=NumComponents−1; i++) |  |
| HPQuantParam[i][j] = HP\_QUANT\_CHROMA[j] |  |
| } else if (COMPONENT\_MODE = = INDEPENDENT) |  |
| for (i = 0; i <=NumComponents−1; i++) |  |
| HPQuantParam[i][j] = HP\_QUANT\_CH[i][j] |  |
| } |  |
| } |  |

## Dequantization

### Dequantization of DC coefficients

This process is applied for the DC coefficients of a macroblock, for all colour components.

The input to this process is the array DCQuantParam[i] of DC quantization parameters for each colour component i; the array MbDCLP[MBx][MBy][i][0] of DC transform coefficients for each colour component i of the current macroblock; and the variables MBx and MBy which identify the current macroblock in the image.

This process uses the local variable array iQuantScalingFactor[i], holding the scaling factor, for each colour component i.

The output of this process is an array of scaled DC transform coefficients MbDCLP[MBx][MBy][i][0], for each colour component i.

The dequantization process for DC coefficients proceeds as specified in Table 146.

Table 146 – Pseudocode for function DequantizeDCCoefficients( )

|  |  |
| --- | --- |
| **DequantizeDCCoefficients( ) {** | **Reference** |
| for (i = 0; i < NumComponents; i++) { |  |
| if (i = = 0) |  |
| iQuantScalingFactor[i] = QuantMap(DCQuantParam[i], 1) | 9.8.4 |
| else |  |
| iQuantScalingFactor[i] = QuantMap(DCQuantParam[i], 0) | 9.8.4 |
| MbDCLP[MBx][MBy][i][0] =   MbDCLP[MBx][MBy][i][0] \* iQuantScalingFactor[i] |  |
| } |  |
| } |  |

### Dequantization of lowpass coefficients

This process is applied for the LP coefficients of an entire macroblock, for all colour components.

Inputs to this process are the values LPQuantParam[i][j] of LP quantization parameters, for each colour component i and index j; the quantization parameter index MBQPIndexLP[MBx][MBy]; the array MbDCLP[MBx][MBy][i][j] of LP transform coefficients, with i representing the colour component, and j referencing the LP transform coefficients, indexed in raster scan order.

The output of this process is an array of scaled LP transform coefficients MbDCLP[MBx][MBy][i][j], for each colour component i and index j ranging from 1 to 15, referencing the respective LP transform coefficient.

The dequantization process for LP coefficients proceeds as specified in Table 147.

Table 147 – Pseudocode for function DequantizeLPCoefficients( )

|  |  |
| --- | --- |
| **DequantizeLPCoefficients( ) {** | **Reference** |
| for (i = 0; i < NumComponents; i++) { |  |
| k = MBQPIndexLP[MBx][MBy] |  |
| valueQP[i] = LPQuantParam[i][k] |  |
| if (i = = 0) |  |
| iQuantScalingFactor[i] = QuantMap(valueQP[i], 1) | 9.8.4 |
| else |  |
| iQuantScalingFactor[i] = QuantMap(valueQP[i], 0) | 9.8.4 |
| if (i = = 0) /\* Luma Component \*/ |  |
| for (j = 1; j <= 15; j++) |  |
| MbDCLP[MBx][MBy][i][j] =   MbDCLP[MBx][MBy][i][j] \* iQuantScalingFactor[i] |  |
| else if ((INTERNAL\_CLR\_FMT != YUV422) &&   (INTERNAL\_CLR\_FMT != YUV420)) |  |
| for (j = 1; j <= 15; j++) |  |
| MbDCLP[MBx][MBy][i][j] =   MbDCLP[MBx][MBy][i][j] \* iQuantScalingFactor[i] |  |
| else if (INTERNAL\_CLR\_FMT = = YUV422) |  |
| for (j = 1; j <= 7; j++) |  |
| MbDCLP[MBx][MBy][i][j] =   MbDCLP[MBx][MBy][i][j] \* iQuantScalingFactor[i] |  |
| else /\* if (INTERNAL\_CLR\_FMT = = YUV420) \*/ |  |
| for (j = 1; j <= 3; j++) |  |
| MbDCLP[MBx][MBy][i][j] =   MbDCLP[MBx][MBy][i][j] \* iQuantScalingFactor[i] |  |
| } |  |
| } |  |

### Dequantization of highpass coefficients

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

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

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

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

Table 148 – Pseudocode for function DequantizeHPCoefficients( )

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

### QuantMap( )

The function QuantMap is used above to compute the scaling parameters based on the parsed syntax elements QP. The pseudocode for this function is specified in Table 149.

Table 149 – Pseudocode for function QuantMap( )

| **QuantMap(iQP, iScaledShift) {** | **Reference** |
| --- | --- |
| if (0 = = iQP) |  |
| iQuantScalingFactorResult = 1 |  |
| else if (!SCALED\_FLAG) { |  |
| iNotScaledShift = −2 |  |
| if (iQP < 32) { |  |
| iMan = (iQP + 3) >> 2 |  |
| iExp = 0 |  |
| } else if (iQP < 48) { |  |
| iMan = (16 + (iQP % 16) + 1) >> 1 |  |
| iExp = (iQP>>4) + iNotScaledShift |  |
| } else { |  |
| iMan = 16 + (iQP % 16) |  |
| iExp = (iQP>>4) −1 + iNotScaledShift |  |
| } |  |
| iQuantScalingFactorResult = iMan << iExp |  |
| } else { /\* SCALED\_FLAG is TRUE, but not (0 = = iQP) \*/ |  |
| if (iQP < 16) { |  |
| iMan = iQP |  |
| iExp = iScaledShift |  |
| } else { |  |
| iMan = 16 + (iQP % 16) |  |
| iExp = ((iQP >> 4) − 1) + iScaledShift |  |
| } |  |
| iQuantScalingFactorResult = iMan << iExp |  |
| } |  |
| return iQuantScalingFactorResult |  |
| } |  |

NOTE – The input parameter iScaledShift takes the value of either 0 or 1, dependent on the component and band. When SCALED\_FLAG is equal to TRUE, the quantization scaling factor value can be modified by a power of 2. See the Note to 9.9.2.

## Sample reconstruction

### Overview

Inputs to this process are the values MbDCLP[MBx][MBy][i][j] and MBBuffer[MBx][MBy][i][j] for the entire image plane, and the syntax element OVERLAP\_MODE.

Outputs of this process are the decoded samples for the image plane.

The sample reconstruction process proceeds as specified in Table 150.

Table 150 – Pseudocode for function SampleReconstruction( )

|  |  |
| --- | --- |
| **SampleReconstruction( ) {** | **Reference** |
| FirstLevelInverseTransform( ) | 9.9.2 |
| if (OVERLAP\_MODE = = 2) |  |
| FirstLevelOverlapFiltering( ) | 9.9.3 |
| SecondLevelCoefficientCombination( ) | 9.9.4 |
| SecondLevelInverseTransform( ) | 9.9.5 |
| if (OVERLAP\_MODE != 0) |  |
| SecondLevelOverlapFiltering( ) | 9.9.6 |
| } |  |

NOTE – Because the first level overlap filtering process in general involves interaction with adjacent macroblocks, the first level transform process must be complete for these adjacent macroblocks, prior to the overlap filtering process being invoked. The precedence of this relationship also holds between the second level processes: the second level's transform process must be complete for the adjacent macroblocks prior to the second level overlap filtering process.

### First level inverse transform

Inputs to this process are the values MbDCLP[MBx][MBy][i][j] for the entire colour component.

Outputs to this process are the modified values MbDCLP[MBx][MBy][i][j] for the current macroblock.

The first level inverse transform process is specified as specified in Table 151.

Table 151 – Pseudocode for function FirstLevelInverseTransform( )

|  |  |
| --- | --- |
| **FirstLevelInverseTransform( ) {** | **Reference** |
| for (i = 0; i < NumComponents; i++) |  |
| for (MBy = 0; MBy < MBHeight; MBy++) |  |
| for (MBx = 0; MBx < MBWidth; MBx++) |  |
| if (i = = 0) | | ((INTERNAL\_CLR\_FMT != YUV420) &&   (INTERNAL\_CLR\_FMT != YUV422)) { |  |
| ICT4x4(MbDCLP[MBx][MBy][i][ ]) | 9.9.7.1 |
| if ((i > 0) && SCALED\_FLAG) |  |
| for (j = 0; j <= 15; j++) |  |
| MbDCLP[MBx][MBy][i][j] = 2 \* MbDCLP[MBx][MBy][i][j] |  |
| } else if (INTERNAL\_CLR\_FMT = = YUV420) { |  |
| T2x2h(MbDCLP[MBx][MBy][i][ ], 0) | 9.9.7.2 |
| arrayLocal[ ] = {MbDCLP[MBx][MBy][i][1], MbDCLP[MBx][MBy][i][2]} |  |
| InvPermute2pt(arrayLocal[ ]) | 9.9.7.6 |
| MbDCLP[MBx][MBy][i][1] = arrayLocal[0] |  |
| MbDCLP[MBx][MBy][i][2] = arrayLocal[1] |  |
| if ((i > 0) && SCALED\_FLAG) |  |
| for (j = 0; j <= 3; j++) |  |
| MbDCLP[MBx][MBy][i][j] = 2 \* MbDCLP[MBx][MBy][i][j] |  |
| } else if (INTERNAL\_CLR\_FMT = = YUV422) { |  |
| arrayLocal[ ] = {MbDCLP[MBx][MBy][i][0], MbDCLP[MBx][MBy][i][4]} |  |
| T2pt(arrayLocal[ ]) | 9.9.7.7 |
| MbDCLP[MBx][MBy][i][0] = arrayLocal[0] |  |
| MbDCLP[MBx][MBy][i][4] = arrayLocal[1] |  |
| T2x2h(MbDCLP[MBx][MBy][i][ ], 0) | 9.9.7.2 |
| arrayLocal[ ] = {MbDCLP[MBx][MBy][i][1], MbDCLP[MBx][MBy][i][2]} |  |
| InvPermute2pt(arrayLocal[ ]) | 9.9.7.6 |
| MbDCLP[MBx][MBy][i][1] = arrayLocal[0] |  |
| MbDCLP[MBx][MBy][i][2] = arrayLocal[1] |  |
| arrayLocal[ ] = {MbDCLP[MBx][MBy][i][4], MbDCLP[MBx][MBy][i][6],   MbDCLP[MBx][MBy][i][5], MbDCLP[MBx][MBy][i][7]} |  |
| T2x2h(arrayLocal[ ], 0) | 9.9.7.2 |
| MbDCLP[MBx][MBy][i][4] = arrayLocal[0] |  |
| MbDCLP[MBx][MBy][i][6] = arrayLocal[1] |  |
| MbDCLP[MBx][MBy][i][5] = arrayLocal[2] |  |
| MbDCLP[MBx][MBy][i][7] = arrayLocal[3] |  |
| arrayLocal[ ] = {MbDCLP[MBx][MBy][i][5], MbDCLP[MBx][MBy][i][6]} |  |
| InvPermute2pt(arrayLocal[ ]) | 9.9.7.6 |
| MbDCLP[MBx][MBy][i][5] = arrayLocal[0] |  |
| MbDCLP[MBx][MBy][i][6] = arrayLocal[1] |  |
| if ((i > 0) && SCALED\_FLAG) |  |
| for (j = 0; j <= 7; j++) |  |
| MbDCLP[MBx][MBy][i][j] = 2 \* MbDCLP[MBx][MBy][i][j] |  |
| } |  |
| } |  |

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

### First level overlap filtering

#### Overview

NOTE – The process specification in this subclause formalizes the geometric nature of the overlap filtering process. The various cases are described in the following.

* + interior: At every point where four 4×4 blocks meet in a corner, the 4×4 overlap filter process is applied to the 4×4 block straddling these four blocks evenly (i.e., overlapping with a 2×2 corner of each block). When HARD\_TILING\_FLAG is equal to FALSE, the 4×4 overlap filter process is applied across tile boundaries as well.
  + top and bottom two rows: Along both the top two sample rows and the bottom two sample rows, a 4-point overlap filter process is applied evenly across adjacent block boundaries (overlapping with a 1×2 strip of each block). When HARD\_TILING\_FLAG is equal to TRUE, the 4-point overlap filter process is applied across the top 2 rows and bottom 2 rows of tiles as well.
  + rightmost and leftmost two columns: Along both the leftmost two sample columns and the rightmost two sample columns, a 4-point overlap filter process is applied evenly across adjacent block boundaries (overlapping with a 2×1 strip of each block). When HARD\_TILING\_FLAG is equal to TRUE, the 4-point overlap filter process is applied across the top 2 columns and bottom 2 columns of tiles as well.
  + four corners: Over the corner 2×2 blocks in the top-left, top-right, bottom-left and bottom-right, a 4-point overlap filter process is applied in a raster scan order (top-left, top-right, bottom-left, then bottom-right). When HARD\_TILING\_FLAG is equal to TRUE, the 4-point overlap filter process is applied to the four corners of each tile as well.

Additionally, when INTERNAL\_CLR\_FMT is equal to either YUV422 or YUV420, alternate processes are considered for the chroma components for each of the above cases. These cases are described in the following.

* + chroma interior: The 2×2 overlap filter process is applied to the 2×2 block straddling interior block boundaries. When HARD\_TILING\_FLAG is equal to FALSE, the 2×2 overlap filter process is applied across tile boundaries as well.
  + chroma top and bottom rows: Along both the top sample row and the bottom sample row, a 2-point overlap filter process is applied evenly across adjacent block boundaries. When HARD\_TILING\_FLAG is equal to TRUE, the 2-point overlap filter process is applied across the top row and bottom row of tiles as well.
  + chroma rightmost and leftmost columns: Along both the leftmost sample column and the rightmost sample column, a 2-point overlap filter process is applied evenly across adjacent block boundaries. When HARD\_TILING\_FLAG is equal to TRUE, the 2-point overlap filter process is applied across the top column and bottom column of tiles as well.
  + chroma four corners: Over the corner 1×1 blocks in the top-left, top-right, bottom-left and bottom-right, an adjacent coefficient residual process (specified in 9.9.3.3 and 9.9.3.4) is applied. When HARD\_TILING\_FLAG is equal to TRUE, the adjacent coefficient residual process is applied to the four corners of each tile as well.

Inputs to this process are the values MbDCLP[MBx][MBy][i][j] for the entire image plane, and the values MBWidth and MBHeight.

Outputs to this process are the modified values MbDCLP[MBx][MBy][i][j] for the current macroblock.

The first level overlap filtering process is specified in Table 152.

Table 152 – Pseudocode for function FirstLevelOverlapFiltering( )

|  |  |
| --- | --- |
| **FirstLevelOverlapFiltering( ) {** | **Reference** |
| for (i = 0; i < NumComponents; i++) |  |
| if ((i = =0) | | ((INTERNAL\_CLR\_FMT != YUV420) &&(INTERNAL\_CLR\_FMT != YUV422))) |  |
| FirstLevelOverlapFilteringPrimary(i) | 9.9.3.2 |
| else if (INTERNAL\_CLR\_FMT = = YUV422) |  |
| FirstLevelOverlapFiltering422(i) | 9.9.3.3 |
| else if (INTERNAL\_CLR\_FMT = = YUV420) |  |
| FirstLevelOverlapFiltering420(i) | 9.9.3.4 |
| } |  |

#### 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++) { |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][10], MbDCLP[x][y][i][14],   MbDCLP[x][y+1][i][2], MbDCLP[x][y+1][i][6]} |  |
| OverlapPostFilter4(arrayLocal[ ]) | 9.9.8.2 |
| MbDCLP[x][y][i][10] = arrayLocal[0] |  |
| MbDCLP[x][y][i][14] = arrayLocal[1] |  |
| MbDCLP[x][y+1][i][2] = arrayLocal[2] |  |
| MbDCLP[x][y+1][i][6] = arrayLocal[3] |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][11], MbDCLP[x][y][i][15],   MbDCLP[x][y+1][i][3], MbDCLP[x][y+1][i][7]} |  |
| OverlapPostFilter4(arrayLocal[ ]) | 9.9.8.2 |
| MbDCLP[x][y][i][11] = arrayLocal[0] |  |
| MbDCLP[x][y][i][15] = arrayLocal[1] |  |
| MbDCLP[x][y+1][i][3] = arrayLocal[2] |  |
| MbDCLP[x][y+1][i][7] = arrayLocal[3] |  |
| } |  |
| } |  |
| if ((Ty = = NumTileRows − 1) | | (HARD\_TILING\_FLAG = = TRUE)) { /\* Bottom edge \*/ |  |
| y = TopMBIndexOfTile[Ty + 1] − 1 |  |
| for (x = LeftMBIndexOfTile[Tx]; x <= LeftMBIndexOfTile[Tx + 1] − 2; x++) { |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][10], MbDCLP[x][y][i][11],   MbDCLP[x+1][y][i][8], MbDCLP[x+1][y][i][9]} |  |
| OverlapPostFilter4(arrayLocal[ ]) | 9.9.8.2 |
| MbDCLP[x][y][i][10] = arrayLocal[0] |  |
| MbDCLP[x][y][i][11] = arrayLocal[1] |  |
| MbDCLP[x+1][y][i][8] = arrayLocal[2] |  |
| MbDCLP[x+1][y][i][9] = arrayLocal[3] |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][14], MbDCLP[x][y][i][15],   MbDCLP[x+1][y][i][12], MbDCLP[x+1][y][i][13]} |  |
| OverlapPostFilter4(arrayLocal[ ]) | 9.9.8.2 |
| MbDCLP[x][y][i][14] = arrayLocal[0] |  |
| MbDCLP[x][y][i][15] = arrayLocal[1] |  |
| MbDCLP[x+1][y][i][12] = arrayLocal[2] |  |
| MbDCLP[x+1][y][i][13] = arrayLocal[3] |  |
| } |  |
| } |  |
| if (((Tx = = 0) && (Ty = = 0)) | | (HARD\_TILING\_FLAG = = TRUE)) { /\* Top left corner \*/ |  |
| x = LeftMBIndexOfTile[Tx] |  |
| y = TopMBIndexOfTile[Ty] |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][0], MbDCLP[x][y][i][1],   MbDCLP[x][y][i][4], MbDCLP[x][y][i][5]} |  |
| OverlapPostFilter4(arrayLocal[ ]) | 9.9.8.2 |
| MbDCLP[x][y][i][0] = arrayLocal[0] |  |
| MbDCLP[x][y][i][1] = arrayLocal[1] |  |
| MbDCLP[x][y][i][4] = arrayLocal[2] |  |
| MbDCLP[x][y][i][5] = arrayLocal[3] |  |
| } |  |
| if (((Tx = = NumTileCols − 1) && (Ty = = 0)) | | (HARD\_TILING\_FLAG = = TRUE)) {   /\* Top right corner \*/ |  |
| x = LeftMBIndexOfTile[Tx + 1] − 1 |  |
| y = TopMBIndexOfTile[Ty] |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][2], MbDCLP[x][y][i][3],   MbDCLP[x][y][i][6], MbDCLP[x][y][i][7]} |  |
| OverlapPostFilter4(arrayLocal[ ]) | 9.9.8.2 |
| MbDCLP[x][y][i][2] = arrayLocal[0] |  |
| MbDCLP[x][y][i][3] = arrayLocal[1] |  |
| MbDCLP[x][y][i][6] = arrayLocal[2] |  |
| MbDCLP[x][y][i][7] = arrayLocal[3] |  |
| } |  |
| if (((Tx = = 0) && (Ty = = NumTileRows − 1)) | | (HARD\_TILING\_FLAG = = TRUE)) {   /\* Bottom left corner \*/ |  |
| x = LeftMBIndexOfTile[Tx] |  |
| y = TopMBIndexOfTile[Ty + 1] − 1 |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][8], MbDCLP[x][y][i][9],   MbDCLP[x][y][i][12], MbDCLP[x][y][i][13]} |  |
| OverlapPostFilter4(arrayLocal[ ]) | 9.9.8.2 |
| MbDCLP[x][y][i][8] = arrayLocal[0] |  |
| MbDCLP[x][y][i][9] = arrayLocal[1] |  |
| MbDCLP[x][y][i][12] = arrayLocal[2] |  |
| MbDCLP[x][y][i][13] = arrayLocal[3] |  |
| } |  |
| if (((Tx = = NumTileCols − 1) && (Ty = = NumTileRows − 1)) | |   (HARD\_TILING\_FLAG = = TRUE)) { /\* Bottom right corner \*/ |  |
| x = LeftMBIndexOfTile[Tx + 1] − 1 |  |
| y = TopMBIndexOfTile[Ty + 1] − 1 |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][10], MbDCLP[x][y][i][11],   MbDCLP[x][y][i][14], MbDCLP[x][y][i][15]} |  |
| OverlapPostFilter4(arrayLocal[ ]) | 9.9.8.2 |
| MbDCLP[x][y][i][10] = arrayLocal[0] |  |
| MbDCLP[x][y][i][11] = arrayLocal[1] |  |
| MbDCLP[x][y][i][14] = arrayLocal[2] |  |
| MbDCLP[x][y][i][15] = arrayLocal[3] |  |
| } |  |
| if ((HARD\_TILING\_FLAG = = FALSE) && (Tx != NumTileCols − 1)) {   /\* Right across for soft tiles \*/ |  |
| x = LeftMBIndexOfTile[Tx + 1] − 1 |  |
| for (y = TopMBIndexOfTile[Ty]; y <= (TopMBIndexOfTile[Ty + 1] − 2); y++) |  |
| FirstLevelCallOverlapPostFilter4x4(i, x, y) | 9.9.3.5 |
| } |  |
| if ((HARD\_TILING\_FLAG = = FALSE) && (Ty != NumTileRows − 1)) {   /\* Bottom across for soft tiles \*/ |  |
| y = TopMBIndexOfTile[Ty + 1] − 1 |  |
| for (x = LeftMBIndexOfTile[Tx]; x <= (LeftMBIndexOfTile[Tx + 1] − 2); x++) |  |
| FirstLevelCallOverlapPostFilter4x4(i, x, y) | 9.9.3.5 |
| } |  |
| if ((HARD\_TILING\_FLAG = = FALSE) && (Tx != NumTileCols − 1) &&   (Ty != NumTileRows − 1)) { /\* Bottom across for soft tiles \*/ |  |
| x = LeftMBIndexOfTile[Tx + 1] − 1 |  |
| y = TopMBIndexOfTile[Ty + 1] − 1 |  |
| FirstLevelCallOverlapPostFilter4x4(i, x, y) | 9.9.3.5 |
| } |  |
| if ((HARD\_TILING\_FLAG = = FALSE) && (Tx = = 0) && (Ty != NumTileRows − 1)) {   /\* Left edge for soft tiles \*/ |  |
| x = LeftMBIndexOfTile[Tx] |  |
| y = TopMBIndexOfTile[Ty + 1] − 1 |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][8], MbDCLP[x][y][i][12],   MbDCLP[x][y+1][i][0], MbDCLP[x][y+1][i][4]} |  |
| OverlapPostFilter4(arrayLocal[ ]) | 9.9.8.2 |
| MbDCLP[x][y][i][8] = arrayLocal[0] |  |
| MbDCLP[x][y][i][12] = arrayLocal[1] |  |
| MbDCLP[x][y+1][i][0] = arrayLocal[2] |  |
| MbDCLP[x][y+1][i][4] = arrayLocal[3] |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][9], MbDCLP[x][y][i][13],   MbDCLP[x][y+1][i][1], MbDCLP[x][y+1][i][5]} |  |
| OverlapPostFilter4(arrayLocal[ ]) | 9.9.8.2 |
| MbDCLP[x][y][i][9] = arrayLocal[0] |  |
| MbDCLP[x][y][i][13] = arrayLocal[1] |  |
| MbDCLP[x][y+1][i][1] = arrayLocal[2] |  |
| MbDCLP[x][y+1][i][5] = arrayLocal[3] |  |
| } |  |
| if ((HARD\_TILING\_FLAG = = FALSE) && (Tx != NumTileCols − 1) && (Ty = = 0)) {   /\* Top edge for soft tiles \*/ |  |
| x = LeftMBIndexOfTile[Tx + 1] − 1 |  |
| y = TopMBIndexOfTile[Ty] |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][2], MbDCLP[x][y][i][3],   MbDCLP[x+1][y][i][0], MbDCLP[x+1][y][i][1]} |  |
| OverlapPostFilter4(arrayLocal[ ]) | 9.9.8.2 |
| MbDCLP[x][y][i][2] = arrayLocal[0] |  |
| MbDCLP[x][y][i][3] = arrayLocal[1] |  |
| MbDCLP[x+1][y][i][0] = arrayLocal[2] |  |
| MbDCLP[x+1][y][i][1] = arrayLocal[3] |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][6], MbDCLP[x][y][i][7],   MbDCLP[x+1][y][i][4], MbDCLP[x+1][y][i][5]} |  |
| OverlapPostFilter4(arrayLocal[ ]) | 9.9.8.2 |
| MbDCLP[x][y][i][6] = arrayLocal[0] |  |
| MbDCLP[x][y][i][7] = arrayLocal[1] |  |
| MbDCLP[x+1][y][i][4] = arrayLocal[2] |  |
| MbDCLP[x+1][y][i][5] = arrayLocal[3] |  |
| } |  |
| if ((HARD\_TILING\_FLAG = = FALSE) && (Tx = = NumTileCols − 1) &&   (Ty != NumTileRows − 1)) { /\* Right edge for soft tiles \*/ |  |
| x = LeftMBIndexOfTile[Tx + 1] − 1 |  |
| y = TopMBIndexOfTile[Ty + 1] − 1 |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][10], MbDCLP[x][y][i][14],   MbDCLP[x][y+1][i][2], MbDCLP[x][y+1][i][6]} |  |
| OverlapPostFilter4(arrayLocal[ ]) | 9.9.8.2 |
| MbDCLP[x][y][i][10] = arrayLocal[0] |  |
| MbDCLP[x][y][i][14] = arrayLocal[1] |  |
| MbDCLP[x][y+1][i][2] = arrayLocal[2] |  |
| MbDCLP[x][y+1][i][6] = arrayLocal[3] |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][11], MbDCLP[x][y][i][15],   MbDCLP[x][y+1][i][4], MbDCLP[x][y+1][i][7]} |  |
| OverlapPostFilter4(arrayLocal[ ]) | 9.9.8.2 |
| MbDCLP[x][y][i][11] = arrayLocal[0] |  |
| MbDCLP[x][y][i][15] = arrayLocal[1] |  |
| MbDCLP[x][y+1][i][4] = arrayLocal[2] |  |
| MbDCLP[x][y+1][i][7] = arrayLocal[3] |  |
| } |  |
| if ((HARD\_TILING\_FLAG = = FALSE) && (Tx != NumTileCols − 1) &&   (Ty = = NumTileRows − 1)) { /\* Bottom edge for soft tiles \*/ |  |
| x = LeftMBIndexOfTile[Tx + 1] − 1 |  |
| y = TopMBIndexOfTile[Ty + 1] − 1 |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][10], MbDCLP[x][y][i][11],   MbDCLP[x+1][y][i][8], MbDCLP[x+1][y][i][9]} |  |
| OverlapPostFilter4(arrayLocal[ ]) | 9.9.8.2 |
| MbDCLP[x][y][i][10] = arrayLocal[0] |  |
| MbDCLP[x][y][i][11] = arrayLocal[1] |  |
| MbDCLP[x+1][y][i][8] = arrayLocal[2] |  |
| MbDCLP[x+1][y][i][9] = arrayLocal[3] |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][14], MbDCLP[x][y][i][15],   MbDCLP[x+1][y][i][12], MbDCLP[x+1][y][i][13]} |  |
| OverlapPostFilter4(arrayLocal[ ]) | 9.9.8.2 |
| MbDCLP[x][y][i][14] = arrayLocal[0] |  |
| MbDCLP[x][y][i][15] = arrayLocal[1] |  |
| MbDCLP[x+1][y][i][12] = arrayLocal[2] |  |
| MbDCLP[x+1][y][i][13] = arrayLocal[3] |  |
| } |  |
| } |  |
| } |  |
| } |  |

#### FirstLevelOverlapFiltering422( )

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

Table 154 – Pseudocode for function FirstLevelOverlapFiltering422( )

| **FirstLevelOverlapFiltering422(i) {** | **Reference** |
| --- | --- |
| for (Ty = 0; Ty <= (NumTileRows − 1); Ty ++) { |  |
| if ((Ty = = 0) | | (HARD\_TILING\_FLAG = = TRUE)) { /\* Top edge \*/ |  |
| /\* OverlapPostFilter1 \*/ |  |
| y = TopMBIndexOfTile[Ty] |  |
| MbDCLP[LeftMBIndexOfTile[0]][y][i][0] −= MbDCLP[LeftMBIndexOfTile[0]][y][i][1]   /\* Upper left corner difference \*/ |  |
| MbDCLP[LeftMBIndexOfTile[NumTileCols] − 1][y][i][1] −=   MbDCLP[LeftMBIndexOfTile[NumTileCols] − 1][y][i][0] /\* Upper right corner difference \*/ |  |
| if (HARD\_TILING\_FLAG = = TRUE) |  |
| for (Tx = 1; Tx < (NumTileCols − 1); Tx++) { |  |
| MbDCLP[LeftMBIndexOfTile[Tx]][y][i][0] −= MbDCLP[LeftMBIndexOfTile[Tx]][y][i][1] |  |
| MbDCLP[LeftMBIndexOfTile[Tx] − 1][y][i][1] −=   MbDCLP[LeftMBIndexOfTile[Tx] − 1][y][i][0] |  |
| } |  |
| } |  |
| if ((Ty = = NumTileRows − 1) | | (HARD\_TILING\_FLAG = = TRUE)) { /\* Bottom edge \*/ |  |
| /\* OverlapPostFilter1 \*/ |  |
| y = TopMBIndexOfTile[Ty + 1] − 1 |  |
| MbDCLP[LeftMBIndexOfTile[0]][y][i][6] −= MbDCLP[LeftMBIndexOfTile[0]][y][i][7]   /\* Bottom left corner difference \*/ |  |
| MbDCLP[LeftMBIndexOfTile[NumTileCols] − 1][y][i][7] −=   MbDCLP[LeftMBIndexOfTile[NumTileCols] − 1][y][i][6] /\* Bottom right corner difference \*/ |  |
| if (HARD\_TILING\_FLAG = = TRUE) |  |
| for (Tx = 1; Tx < (NumTileCols − 1); Tx++) { |  |
| MbDCLP[LeftMBIndexOfTile[Tx]][y][i][6] −= MbDCLP[LeftMBIndexOfTile[Tx]][y][i][7] |  |
| MbDCLP[LeftMBIndexOfTile[Tx] − 1][y][i][7] −=   MbDCLP[LeftMBIndexOfTile[Tx] − 1][y][i][6] |  |
| } |  |
| } |  |
| for (Tx = 0; Tx <= (NumTileCols − 1); Tx++) { |  |
| for (y = TopMBIndexOfTile[Ty]; y <= (TopMBIndexOfTile[Ty + 1] − 1); y++) |  |
| for (x = LeftMBIndexOfTile[Tx]; x <= (LeftMBIndexOfTile[Tx + 1] − 2); x++) { |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x+1][y][i][2],   MbDCLP[x][y][i][5], MbDCLP[x+1][y][i][4]} |  |
| OverlapPostFilter2x2(arrayLocal[ ]) | 9.9.8.3 |
| MbDCLP[x][y][i][3] = arrayLocal[0] |  |
| MbDCLP[x+1][y][i][2] = arrayLocal[1] |  |
| MbDCLP[x][y][i][5] = arrayLocal[2] |  |
| 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); 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); 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]} |  |
| OverlapPostFilter2(arrayLocal[ ]) | 9.9.8.4 |
| MbDCLP[x][y][i][7] = arrayLocal[0] |  |
| MbDCLP[x][y+1][i][1] = arrayLocal[1] |  |
| } |  |
| } |  |
| } |  |
| if ((Ty = = 0) | | (HARD\_TILING\_FLAG = = TRUE)) { /\* Top edge \*/ |  |
| y = TopMBIndexOfTile[Ty] |  |
| for (x = LeftMBIndexOfTile[Tx]; x <= (LeftMBIndexOfTile[Tx + 1] − 2); x++) { |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][1], MbDCLP[x+1][y][i][0]} |  |
| OverlapPostFilter2(arrayLocal[ ]) | 9.9.8.4 |
| MbDCLP[x][y][i][1] = arrayLocal[0] |  |
| MbDCLP[x+1][y][i][0] = arrayLocal[1] |  |
| } |  |
| } |  |
| if ((Ty = = NumTileRows − 1) | | (HARD\_TILING\_FLAG = = TRUE)) { /\* Bottom edge \*/ |  |
| y = TopMBIndexOfTile[Ty + 1] − 1 |  |
| for (x = LeftMBIndexOfTile[Tx]; x <= (LeftMBIndexOfTile[Tx + 1] − 2); x++) { |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][7], MbDCLP[x+1][y][i][6]} |  |
| OverlapPostFilter2(arrayLocal[ ]) | 9.9.8.4 |
| MbDCLP[x][y][i][7] = arrayLocal[0] |  |
| MbDCLP[x+1][y][i][6] = arrayLocal[1] |  |
| } |  |
| } |  |
| if ((HARD\_TILING\_FLAG = = FALSE) && (Tx != NumTileCols − 1)) {   /\* Right across for soft tiles \*/ |  |
| x = LeftMBIndexOfTile[Tx + 1] − 1 |  |
| for (y = TopMBIndexOfTile[Ty]; y <= (TopMBIndexOfTile[Ty + 1] − 2); y++) { |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x+1][y][i][2],   MbDCLP[x][y][i][5], MbDCLP[x+1][y][i][4]} |  |
| OverlapPostFilter2x2(arrayLocal[ ]) | 9.9.8.3 |
| MbDCLP[x][y][i][3] = arrayLocal[0] |  |
| MbDCLP[x+1][y][i][2] = arrayLocal[1] |  |
| MbDCLP[x][y][i][5] = arrayLocal[2] |  |
| MbDCLP[x+1][y][i][4] = arrayLocal[3] |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][7], MbDCLP[x+1][y][i][6],   MbDCLP[x][y+1][i][1], MbDCLP[x+1][y+1][i][0]} |  |
| OverlapPostFilter2x2(arrayLocal[ ]) | 9.9.8.3 |
| MbDCLP[x][y][i][7] = arrayLocal[0] |  |
| MbDCLP[x+1][y][i][6] = arrayLocal[1] |  |
| MbDCLP[x][y+1][i][1] = arrayLocal[2] |  |
| MbDCLP[x+1][y+1][i][0] = arrayLocal[3] |  |
| } |  |
| } |  |
| if ((HARD\_TILING\_FLAG = = FALSE) && (Ty != NumTileRows − 1)) {   /\* Bottom across for soft tiles \*/ |  |
| y = TopMBIndexOfTile[Ty + 1] − 1 |  |
| for (x = LeftMBIndexOfTile[Tx]; x <= (LeftMBIndexOfTile[Tx + 1] − 2); x++) { |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x+1][y][i][2],   MbDCLP[x][y][i][5], MbDCLP[x+1][y][i][4]} |  |
| OverlapPostFilter2x2(arrayLocal[ ]) | 9.9.8.3 |
| MbDCLP[x][y][i][3] = arrayLocal[0] |  |
| MbDCLP[x+1][y][i][2] = arrayLocal[1] |  |
| MbDCLP[x][y][i][5] = arrayLocal[2] |  |
| MbDCLP[x+1][y][i][4] = arrayLocal[3] |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][7], MbDCLP[x+1][y][i][6],   MbDCLP[x][y+1][i][1], MbDCLP[x+1][y+1][i][0]} |  |
| OverlapPostFilter2x2(arrayLocal[ ]) | 9.9.8.3 |
| MbDCLP[x][y][i][7] = arrayLocal[0] |  |
| MbDCLP[x+1][y][i][6] = arrayLocal[1] |  |
| MbDCLP[x][y+1][i][1] = arrayLocal[2] |  |
| MbDCLP[x+1][y+1][i][0] = arrayLocal[3] |  |
| } |  |
| } |  |
| if ((HARD\_TILING\_FLAG = = FALSE) && (Tx != NumTileCols − 1) &&   (Ty != NumTileRows − 1)) { |  |
| x = LeftMBIndexOfTile[Tx + 1] − 1 |  |
| y = TopMBIndexOfTile[Ty + 1] − 1 |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x+1][y][i][2],   MbDCLP[x][y][i][5], MbDCLP[x+1][y][i][4]} |  |
| OverlapPostFilter2x2(arrayLocal[ ]) | 9.9.8.3 |
| MbDCLP[x][y][i][3] = arrayLocal[0] |  |
| MbDCLP[x+1][y][i][2] = arrayLocal[1] |  |
| MbDCLP[x][y][i][5] = arrayLocal[2] |  |
| MbDCLP[x+1][y][i][4] = arrayLocal[3] |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][7], MbDCLP[x+1][y][i][6],   MbDCLP[x][y+1][i][1], MbDCLP[x+1][y+1][i][0]} |  |
| OverlapPostFilter2x2(arrayLocal[ ]) | 9.9.8.3 |
| MbDCLP[x][y][i][7] = arrayLocal[0] |  |
| MbDCLP[x+1][y][i][6] = arrayLocal[1] |  |
| MbDCLP[x][y+1][i][1] = arrayLocal[2] |  |
| MbDCLP[x+1][y+1][i][0] = arrayLocal[3] |  |
| } |  |
| if ((HARD\_TILING\_FLAG = = FALSE) && (Tx = = 0) && (Ty != NumTileRows − 1)) { |  |
| x = LeftMBIndexOfTile[Tx] |  |
| y = TopMBIndexOfTile[Ty + 1] − 1 |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][2], MbDCLP[x][y][i][4]} |  |
| OverlapPostFilter2(arrayLocal[ ]) | 9.9.8.4 |
| MbDCLP[x][y][i][2] = arrayLocal[0] |  |
| MbDCLP[x][y][i][4] = arrayLocal[1] |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][6], MbDCLP[x][y+1][i][0]} |  |
| OverlapPostFilter2(arrayLocal[ ]) | 9.9.8.4 |
| MbDCLP[x][y][i][6] = arrayLocal[0] |  |
| MbDCLP[x][y+1][i][0] = arrayLocal[1] |  |
| } |  |
| if ((HARD\_TILING\_FLAG = = FALSE) && (Tx = = NumTileCols − 1) &&   (Ty != NumTileRows − 1)) { |  |
| x = LeftMBIndexOfTile[Tx + 1] − 1 |  |
| y = TopMBIndexOfTile[Ty + 1] − 1 |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x][y][i][5]} |  |
| OverlapPostFilter2(arrayLocal[ ]) | 9.9.8.4 |
| MbDCLP[x][y][i][3] = arrayLocal[0] |  |
| MbDCLP[x][y][i][5] = arrayLocal[1] |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][7], MbDCLP[x][y+1][i][1]} |  |
| OverlapPostFilter2(arrayLocal[ ]) | 9.9.8.4 |
| MbDCLP[x][y][i][7] = arrayLocal[0] |  |
| MbDCLP[x][y+1][i][1] = arrayLocal[1] |  |
| } |  |
| if ((HARD\_TILING\_FLAG = = FALSE) && (Tx != NumTileCols − 1) && (Ty = = 0)) { |  |
| x = LeftMBIndexOfTile[Tx + 1] − 1 |  |
| y = TopMBIndexOfTile[Ty] |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][1], MbDCLP[x+1][y][i][0]} |  |
| OverlapPostFilter2(arrayLocal[ ]) | 9.9.8.4 |
| MbDCLP[x][y][i][1] = arrayLocal[0] |  |
| MbDCLP[x+1][y][i][0] = arrayLocal[1] |  |
| } |  |
| if ((HARD\_TILING\_FLAG = = FALSE) && (Tx != NumTileCols − 1) &&   (Ty = = NumTileRows − 1)) { |  |
| x = LeftMBIndexOfTile[Tx + 1] − 1 |  |
| y = TopMBIndexOfTile[Ty + 1] − 1 |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][7], MbDCLP[x+1][y][i][6]} |  |
| OverlapPostFilter2(arrayLocal[ ]) | 9.9.8.4 |
| MbDCLP[x][y][i][7] = arrayLocal[0] |  |
| MbDCLP[x+1][y][i][6] = arrayLocal[1] |  |
| } |  |
| } |  |
| if ((Ty = = 0) | | (HARD\_TILING\_FLAG = = TRUE)) { /\*Top edge \*/ |  |
| /\* OverlapPostFilter1 \*/ |  |
| y = TopMBIndexOfTile[Ty] |  |
| MbDCLP[LeftMBIndexOfTile[0]][y][i][0] += MbDCLP[LeftMBIndexOfTile[0]][y][i][1]   /\* Upper left corner addition \*/ |  |
| MbDCLP[LeftMBIndexOfTile[NumTileCols] − 1][y][i][1] +=   MbDCLP[LeftMBIndexOfTile[NumTileCols] − 1][y][i][0] /\* Upper right corner addition \*/ |  |
| if (HARD\_TILING\_FLAG = = TRUE) |  |
| for (Tx = 1; Tx < (NumTileCols − 1); Tx++) { |  |
| MbDCLP[LeftMBIndexOfTile[Tx]][y][i][0] += MbDCLP[LeftMBIndexOfTile[Tx]][y][i][1] |  |
| MbDCLP[LeftMBIndexOfTile[Tx] − 1][y][i][1] +=   MbDCLP[LeftMBIndexOfTile[Tx] − 1][y][i][0] |  |
| } |  |
| } |  |
| if ((Ty = = NumTileRows − 1) | | (HARD\_TILING\_FLAG = = TRUE)) { /\* Bottom edge \*/ |  |
| /\* OverlapPostFilter1 \*/ |  |
| y = TopMBIndexOfTile[Ty + 1] − 1 |  |
| MbDCLP[LeftMBIndexOfTile[0]][y][i][6] += MbDCLP[LeftMBIndexOfTile[0]][y][i][7]   /\* Bottom left corner addition \*/ |  |
| MbDCLP[LeftMBIndexOfTile[NumTileCols] − 1][y][i][7] +=   MbDCLP[LeftMBIndexOfTile[NumTileCols] − 1][y][i][6] /\* Bottom right corner addition \*/ |  |
| if (HARD\_TILING\_FLAG = = TRUE) |  |
| for (Tx = 1; Tx < (NumTileCols − 1); Tx++) { |  |
| MbDCLP[LeftMBIndexOfTile[Tx]][y][i][6] += MbDCLP[LeftMBIndexOfTile[Tx]][y][i][7] |  |
| MbDCLP[LeftMBIndexOfTile[Tx] − 1][y][i][7] +=   MbDCLP[LeftMBIndexOfTile[Tx] − 1][y][i][6] |  |
| } |  |
| } |  |
| } |  |
| } |  |

#### FirstLevelOverlapFiltering420( )

Pseudocode for the function FirstLevelOverlapFiltering420( ) is specified in Table 155.

Table 155 – Pseudocode for function FirstLevelOverlapFiltering420( )

| **FirstLevelOverlapFiltering420(i) {** | **Reference** |
| --- | --- |
| for (Ty = 0; Ty <= (NumTileRows − 1); Ty ++) { |  |
| if ((Ty = = 0) | | (HARD\_TILING\_FLAG = = TRUE)) { /\* Top edge \*/ |  |
| /\* OverlapPostFilter1 \*/ |  |
| y = TopMBIndexOfTile[Ty] |  |
| MbDCLP[LeftMBIndexOfTile[0]][y][i][0] −= MbDCLP[LeftMBIndexOfTile[0]][y][i][1]   /\* Upper left corner difference \*/ |  |
| MbDCLP[LeftMBIndexOfTile[NumTileCols] − 1][y][i][1] −=   MbDCLP[LeftMBIndexOfTile[NumTileCols] − 1][y][i][0] /\* Upper right corner difference \*/ |  |
| if (HARD\_TILING\_FLAG = = TRUE) |  |
| for (Tx = 1; Tx < (NumTileCols − 1); Tx++) { |  |
| MbDCLP[LeftMBIndexOfTile[Tx]][y][i][0] −= MbDCLP[LeftMBIndexOfTile[Tx]][y][i][1] |  |
| MbDCLP[LeftMBIndexOfTile[Tx] − 1][y][i][1] −=   MbDCLP[LeftMBIndexOfTile[Tx] − 1][y][i][0] |  |
| } |  |
| } |  |
| if ((Ty = = NumTileRows − 1) | | (HARD\_TILING\_FLAG = = TRUE)) { /\* Bottom edge \*/ |  |
| /\* OverlapPostFilter1 \*/ |  |
| y = TopMBIndexOfTile[Ty + 1] − 1 |  |
| MbDCLP[LeftMBIndexOfTile[0]][y][i][2] −= MbDCLP[LeftMBIndexOfTile[0]][y][i][3]   /\* Bottom left corner difference \*/ |  |
| MbDCLP[LeftMBIndexOfTile[NumTileCols] − 1][y][i][3] −=   MbDCLP[LeftMBIndexOfTile[NumTileCols] − 1][y][i][2] /\* Bottom right corner difference \*/ |  |
| if (HARD\_TILING\_FLAG = = TRUE) |  |
| for (Tx = 1; Tx < (NumTileCols − 1); Tx++) { |  |
| MbDCLP[LeftMBIndexOfTile[Tx]][y][i][2] −= MbDCLP[LeftMBIndexOfTile[Tx]][y][i][3] |  |
| MbDCLP[LeftMBIndexOfTile[Tx] − 1][y][i][3] −=   MbDCLP[LeftMBIndexOfTile[Tx] − 1][y][i][2] |  |
| } |  |
| } |  |
| for (Tx = 0; Tx <= (NumTileCols − 1); Tx++) { |  |
| for (y = TopMBIndexOfTile[Ty]; y <= (TopMBIndexOfTile[Ty + 1] − 2); y++) |  |
| for (x = LeftMBIndexOfTile[Tx]; x <= (LeftMBIndexOfTile[Tx + 1] − 2); x++) { |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x+1][y][i][2],   MbDCLP[x][y+1][i][1], MbDCLP[x+1][y+1][i][0]} |  |
| OverlapPostFilter2x2(arrayLocal[ ]) | 9.9.8.3 |
| MbDCLP[x][y][i][3] = arrayLocal[0] |  |
| MbDCLP[x+1][y][i][2] = arrayLocal[1] |  |
| MbDCLP[x][y+1][i][1] = arrayLocal[2] |  |
| MbDCLP[x+1][y+1][i][0] = arrayLocal[3] |  |
| } |  |
| if ((Tx = = 0) | | (HARD\_TILING\_FLAG = = TRUE)) { /\* Left edge \*/ |  |
| x = LeftMBIndexOfTile[Tx] |  |
| for (y = TopMBIndexOfTile[Ty]; y <= (TopMBIndexOfTile[Ty + 1] − 2); y++) { |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][2], MbDCLP[x][y+1][i][0]} |  |
| OverlapPostFilter2(arrayLocal[ ]) | 9.9.8.4 |
| MbDCLP[x][y][i][2] = arrayLocal[0] |  |
| MbDCLP[x][y+1][i][0] = arrayLocal[1] |  |
| } |  |
| } |  |
| if ((Tx = = NumTileCols − 1) | | (HARD\_TILING\_FLAG = = TRUE)) { /\* Right edge \*/ |  |
| x = LeftMBIndexOfTile[Tx + 1] − 1 |  |
| for (y = TopMBIndexOfTile[Ty]; y <= (TopMBIndexOfTile[Ty + 1] − 2); y++) { |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x][y+1][i][1]} |  |
| OverlapPostFilter2(arrayLocal[ ]) | 9.9.8.4 |
| MbDCLP[x][y][i][3] = arrayLocal[0] |  |
| MbDCLP[x][y+1][i][1] = arrayLocal[1] |  |
| } |  |
| } |  |
| if ((Ty = = 0) | | (HARD\_TILING\_FLAG = = TRUE)) { /\* Top edge \*/ |  |
| y = TopMBIndexOfTile[Ty] |  |
| for (x = LeftMBIndexOfTile[Tx]; x <= (LeftMBIndexOfTile[Tx + 1] − 2); x++) { |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][1], MbDCLP[x+1][y][i][0]} |  |
| OverlapPostFilter2(arrayLocal[ ]) | 9.9.8.4 |
| MbDCLP[x][y][i][1] = arrayLocal[0] |  |
| MbDCLP[x+1][y][i][0] = arrayLocal[1] |  |
| } |  |
| } |  |
| if ((Ty = = NumTileRows − 1) | | (HARD\_TILING\_FLAG = = TRUE)) { /\* Bottom edge \*/ |  |
| y = TopMBIndexOfTile[Ty + 1] − 1 |  |
| for (x = LeftMBIndexOfTile[Tx]; x <= (LeftMBIndexOfTile[Tx + 1] − 2); x++) { |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x+1][y][i][2]} |  |
| OverlapPostFilter2(arrayLocal[ ]) | 9.9.8.4 |
| MbDCLP[x][y][i][3] = arrayLocal[0] |  |
| MbDCLP[x+1][y][i][2] = arrayLocal[1] |  |
| } |  |
| } |  |
| if ((HARD\_TILING\_FLAG = = FALSE) && (Tx != NumTileCols − 1)) {   /\* Right across for soft tiles \*/ |  |
| x = LeftMBIndexOfTile[Tx + 1] − 1 |  |
| for (y = TopMBIndexOfTile[Ty]; y <= (TopMBIndexOfTile[Ty + 1] − 2); y++) { |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x+1][y][i][2],   MbDCLP[x][y+1][i][1], MbDCLP[x+1][y+1][i][0]} |  |
| OverlapPostFilter2x2(arrayLocal[ ]) | 9.9.8.3 |
| MbDCLP[x][y][i][3] = arrayLocal[0] |  |
| MbDCLP[x+1][y][i][2] = arrayLocal[1] |  |
| MbDCLP[x][y+1][i][1] = arrayLocal[2] |  |
| MbDCLP[x+1][y+1][i][0] = arrayLocal[3] |  |
| } |  |
| } |  |
| if ((HARD\_TILING\_FLAG = = FALSE) && (Ty != NumTileRows − 1)) {   /\* Bottom across for soft tiles \*/ |  |
| y = TopMBIndexOfTile[Ty + 1] − 1 |  |
| for (x = LeftMBIndexOfTile[Tx]; x <= (LeftMBIndexOfTile[Tx + 1] − 2); x++) { |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x+1][y][i][2],   MbDCLP[x][y+1][i][1], MbDCLP[x+1][y+1][i][0]} |  |
| OverlapPostFilter2x2(arrayLocal[ ]) | 9.9.8.3 |
| MbDCLP[x][y][i][3] = arrayLocal[0] |  |
| MbDCLP[x+1][y][i][2] = arrayLocal[1] |  |
| MbDCLP[x][y+1][i][1] = arrayLocal[2] |  |
| MbDCLP[x+1][y+1][i][0] = arrayLocal[3] |  |
| } |  |
| } |  |
| if ((HARD\_TILING\_FLAG = = FALSE) && (Tx != NumTileCols − 1) &&   (Ty != NumTileRows − 1)) { |  |
| x = LeftMBIndexOfTile[Tx + 1] − 1 |  |
| y = TopMBIndexOfTile[Ty + 1] − 1 |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x+1][y][i][2],   MbDCLP[x][y+1][i][1], MbDCLP[x+1][y+1][i][0]} |  |
| OverlapPostFilter2x2(arrayLocal[ ]) | 9.9.8.3 |
| MbDCLP[x][y][i][3] = arrayLocal[0] |  |
| MbDCLP[x+1][y][i][2] = arrayLocal[1] |  |
| MbDCLP[x][y+1][i][1] = arrayLocal[2] |  |
| MbDCLP[x+1][y+1][i][0] = arrayLocal[3] |  |
| } |  |
| if ((HARD\_TILING\_FLAG = = FALSE) && (Tx = = 0) && (Ty != NumTileRows − 1)) { |  |
| x = LeftMBIndexOfTile[Tx] |  |
| y = TopMBIndexOfTile[Ty + 1] − 1 |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][2], MbDCLP[x][y+1][i][0]} |  |
| OverlapPostFilter2(arrayLocal[ ]) | 9.9.8.4 |
| MbDCLP[x][y][i][2] = arrayLocal[0] |  |
| MbDCLP[x][y+1][i][0] = arrayLocal[1] |  |
| } |  |
| if ((HARD\_TILING\_FLAG = = FALSE) && (Tx = = NumTileCols − 1) &&   (Ty != NumTileRows − 1)) { |  |
| x = LeftMBIndexOfTile[Tx + 1] − 1 |  |
| y = TopMBIndexOfTile[Ty + 1] − 1 |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x][y+1][i][1]} |  |
| OverlapPostFilter2(arrayLocal[ ]) | 9.9.8.4 |
| MbDCLP[x][y][i][3] = arrayLocal[0] |  |
| MbDCLP[x][y+1][i][1] = arrayLocal[1] |  |
| } |  |
| if ((HARD\_TILING\_FLAG = = FALSE) && (Tx != NumTileCols − 1) && (Ty = = 0)) { |  |
| x = LeftMBIndexOfTile[Tx + 1] − 1 |  |
| y = TopMBIndexOfTile[Ty] |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][1], MbDCLP[x+1][y][i][0]} |  |
| OverlapPostFilter2(arrayLocal[ ]) | 9.9.8.4 |
| MbDCLP[x][y][i][1] = arrayLocal[0] |  |
| MbDCLP[x+1][y][i][0] = arrayLocal[1] |  |
| } |  |
| if ((HARD\_TILING\_FLAG = = FALSE) && (Tx != NumTileCols − 1) &&   (Ty = = NumTileRows − 1)) { |  |
| x = LeftMBIndexOfTile[Tx + 1] − 1 |  |
| y = TopMBIndexOfTile[Ty + 1] − 1 |  |
| arrayLocal[ ] = {MbDCLP[x][y][i][3], MbDCLP[x+1][y][i][2]} |  |
| OverlapPostFilter2(arrayLocal[ ]) | 9.9.8.4 |
| MbDCLP[x][y][i][3] = arrayLocal[0] |  |
| MbDCLP[x+1][y][i][2] = arrayLocal[1] |  |
| } |  |
| } |  |
| if ((Ty = = 0) | | (HARD\_TILING\_FLAG = = TRUE)) { /\*Top edge \*/ |  |
| /\* OverlapPostFilter1 \*/ |  |
| y = TopMBIndexOfTile[Ty] |  |
| MbDCLP[LeftMBIndexOfTile[0]][y][i][0] += MbDCLP[LeftMBIndexOfTile[0]][y][i][1]   /\* Upper left corner addition \*/ |  |
| MbDCLP[LeftMBIndexOfTile[NumTileCols] − 1][y][i][1] +=   MbDCLP[LeftMBIndexOfTile[NumTileCols] − 1][y][i][0] /\* Upper right corner addition \*/ |  |
| if (HARD\_TILING\_FLAG = = TRUE) |  |
| for (Tx = 1; Tx < (NumTileCols − 1); Tx++) { |  |
| MbDCLP[LeftMBIndexOfTile[Tx]][y][i][0] += MbDCLP[LeftMBIndexOfTile[Tx]][y][i][1] |  |
| MbDCLP[LeftMBIndexOfTile[Tx] − 1][y][i][1] +=   MbDCLP[LeftMBIndexOfTile[Tx] − 1][y][i][0] |  |
| } |  |
| } |  |
| if ((Ty = = NumTileRows − 1) | | (HARD\_TILING\_FLAG = = TRUE)) { /\* Bottom edge \*/ |  |
| /\* OverlapPostFilter1 \*/ |  |
| y = TopMBIndexOfTile[Ty + 1] − 1 |  |
| MbDCLP[LeftMBIndexOfTile[0]][y][i][2] += MbDCLP[LeftMBIndexOfTile[0]][y][i][3]   /\* Bottom left corner addition \*/ |  |
| MbDCLP[LeftMBIndexOfTile[NumTileCols] − 1][y][i][3] +=   MbDCLP[LeftMBIndexOfTile[NumTileCols] − 1][y][i][2] /\* Bottom right corner addition \*/ |  |
| if (HARD\_TILING\_FLAG = = TRUE) |  |
| for (Tx = 1; Tx < (NumTileCols − 1); Tx++) { |  |
| MbDCLP[LeftMBIndexOfTile[Tx]][y][i][2] += MbDCLP[LeftMBIndexOfTile[Tx]][y][i][3] |  |
| MbDCLP[LeftMBIndexOfTile[Tx] − 1][y][i][3] +=   MbDCLP[LeftMBIndexOfTile[Tx] − 1][y][i][2] |  |
| } |  |
| } |  |
| } |  |
| } |  |

#### FirstLevelCallOverlapPostFilterx4x4( )

Pseudocode for the function FirstLevelCallOverlapPostFilter4x4( ) is specified in Table 156.

Table 156 – Pseudocode for function FirstLevelCallOverlapPostFilter4x4( )

|  |  |
| --- | --- |
| **FirstLevelCallOverlapPostFilter4x4(i, x, y) {** | **Reference** |
| arrayLocal[ ] = {MbDCLP[x][y][i][10], MbDCLP[x][y][i][11],   MbDCLP[x+1][y][i][8], MbDCLP[x+1][y][i][9],   MbDCLP[x][y][i][14], MbDCLP[x][y][i][15],   MbDCLP[x+1][y][i][12], MbDCLP[x+1][y][i][13],   MbDCLP[x][y+1][i][2], MbDCLP[x][y+1][i][3],   MbDCLP[x+1][y+1][i][0], MbDCLP[x+1][y+1][i][1],   MbDCLP[x][y+1][i][6], MbDCLP[x][y+1][i][7],   MbDCLP[x+1][y+1][i][4], MbDCLP[x+1][y+1][i][5]} |  |
| OverlapPostFilter4x4(arrayLocal[ ]) | 9.9.8.1 |
| MbDCLP[x][y][i][10] = arrayLocal[0] |  |
| MbDCLP[x][y][i][11] = arrayLocal[1] |  |
| MbDCLP[x+1][y][i][8] = arrayLocal[2] |  |
| MbDCLP[x+1][y][i][9] = arrayLocal[3] |  |
| MbDCLP[x][y][i][14] = arrayLocal[4] |  |
| MbDCLP[x][y][i][15] = arrayLocal[5] |  |
| MbDCLP[x+1][y][i][12] = arrayLocal[6] |  |
| MbDCLP[x+1][y][i][13] = arrayLocal[7] |  |
| MbDCLP[x][y+1][i][2] = arrayLocal[8] |  |
| MbDCLP[x][y+1][i][3] = arrayLocal[9] |  |
| MbDCLP[x+1][y+1][i][0] = arrayLocal[10] |  |
| MbDCLP[x+1][y+1][i][1] = arrayLocal[11] |  |
| MbDCLP[x][y+1][i][6] = arrayLocal[12] |  |
| MbDCLP[x][y+1][i][7] = arrayLocal[13] |  |
| MbDCLP[x+1][y+1][i][4] = arrayLocal[14] |  |
| MbDCLP[x+1][y+1][i][5] = arrayLocal[15] |  |
| } |  |

### Second level coefficient combination

NOTE – At this point in the process, the DC-LP array coefficients have gone through the first level transform and overlap filtering. The DC-LP array coefficients and the HP coefficients are then combined in an image plane represented by the values ImagePlane[i][x][y], where the colour component is specified by i, and x and y mark the location of the sample in the image plane.

Inputs to this process are the values MbDCLP[MBx][MBy][i][j] and MBBuffer[MBx][MBy][i][j], for the current macroblock.

Outputs to this process are the values of ImagePlane[i][x][y], for the current macroblock.

The second level coefficient combination process proceeds as specified in Table 157.

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[MBx][MBy][i][j] |  |
| } |  |
| } else if (INTERNAL\_CLR\_FMT = = YUV422) { |  |
| for (j = 0; j <= 7; j++) { |  |
| x = 8 \* 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 \* 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[MBx][MBy][i][j] |  |
| } |  |
| } else if (INTERNAL\_CLR\_FMT = = YUV420) { |  |
| for (j = 0; j <= 3; j++) { |  |
| x = 8 \* MBx + 4 \* (j % 2) |  |
| y = 8 \* MBy + 4 \* (j / 2) |  |
| ImagePlane[i][x][y] = MbDCLP[MBx][MBy][i][j] |  |
| } |  |
| for (j = 0; j <= 63; j++) { |  |
| x = 8 \* MBx + 4 \* ((j % 32) / 16) + ((j % 32) % 4) |  |
| y = 8 \* MBy + 4 \* (j / 32) + ((j / 4) % 4) |  |
| k = j % 16 |  |
| if (k != 0) /\* only the HP coefficients are copied \*/ |  |
| ImagePlane[i][x][y] = MBBuffer[MBx][MBy][i][j] |  |
| } |  |
| } |  |
| } |  |

### Second level inverse transform

Inputs to this process are the values ImagePlane[i][x][y] for the entire image plane.

Outputs to this process are the modified values ImagePlane[i][x][y] for the current macroblock.

The second level inverse transform process is specified in Table 158.

Table 158 – Pseudocode for function SecondLevelInverseTransform( )

|  |  |
| --- | --- |
| **SecondLevelInverseTransform( ) {** | **Reference** |
| for (i = 0; i < NumComponents; i++) |  |
| for (x = 0; x < ExtendedWidth[i]; x += 4) |  |
| for (y = 0; y < ExtendedHeight[i]; y += 4) { |  |
| arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y],   ImagePlane[i][x+2][y], ImagePlane[i][x+3][y],   ImagePlane[i][x][y+1], ImagePlane[i][x+1][y+1],   ImagePlane[i][x+2][y+1], ImagePlane[i][x+3][y+1],   ImagePlane[i][x][y+2], ImagePlane[i][x+1][y+2],   ImagePlane[i][x+2][y+2], ImagePlane[i][x+3][y+2],   ImagePlane[i][x][y+3], ImagePlane[i][x+1][y+3],   ImagePlane[i][x+2][y+3], ImagePlane[i][x+3][y+3]} |  |
| ICT4x4(arrayLocal[ ]) | 9.9.7.1 |
| ImagePlane[i][x][y] = arrayLocal[0] |  |
| ImagePlane[i][x+1][y] = arrayLocal[1] |  |
| ImagePlane[i][x+2][y] = arrayLocal[2] |  |
| ImagePlane[i][x+3][y] = arrayLocal[3] |  |
| ImagePlane[i][x][y+1] = arrayLocal[4] |  |
| ImagePlane[i][x+1][y+1] = arrayLocal[5] |  |
| ImagePlane[i][x+2][y+1] = arrayLocal[6] |  |
| ImagePlane[i][x+3][y+1] = arrayLocal[7] |  |
| ImagePlane[i][x][y+2] = arrayLocal[8] |  |
| ImagePlane[i][x+1][y+2] = arrayLocal[9] |  |
| ImagePlane[i][x+2][y+2] = arrayLocal[10] |  |
| ImagePlane[i][x+3][y+2] = arrayLocal[11] |  |
| ImagePlane[i][x][y+3] = arrayLocal[12] |  |
| ImagePlane[i][x+1][y+3] = arrayLocal[13] |  |
| ImagePlane[i][x+2][y+3] = arrayLocal[14] |  |
| ImagePlane[i][x+3][y+3] = arrayLocal[15] |  |
| } |  |
| } |  |

### Second level overlap filtering

Inputs to this process are the values ImagePlane[i][x][y] for the entire image plane.

Outputs to this process are the modified values ImagePlane[i][x][y] for the entire image plane.

Outputs to this process are the modified values ImagePlane[i][x][y] for the current macroblock.

NOTE – The process specification below formalizes the geometric nature of the overlap filtering process. There are 4 cases.

(1) (interior): at every point in the image plane where 4 blocks meet in a corner, the 4×4 overlap filter is applied to the 4×4 block straddling these 4 blocks evenly (i.e., overlapping with a 2×2 corner of each block).

(2) (top and bottom 2 rows): along both the top two sample rows and the bottom two sample rows of the image plane, the 4-point overlap filter is applied evenly across adjacent block boundaries (overlapping with a 1×2 strip of each block).

(3) (rightmost and leftmost columns): along both the leftmost two sample columns and the rightmost two sample columns, the 4-point overlap filter is applied evenly across adjacent block boundaries (overlapping with a 2×1 strip of each block).

(4) (four corners of the image plane): over the corner 2×2 blocks in the top-left, top-right, bottom-left and bottom-right, the 4-point overlap filter process is applied in a raster scan order (top-left, top-right, bottom-left, then bottom-right).

The second level overlap filtering process is specified in Table 159.

Table 159 – Pseudocode for function SecondLevelOverlapFiltering( )

| **SecondLevelOverlapFiltering( ) {** | **Reference** |
| --- | --- |
| for (i = 0; i < NumComponents; i++) { |  |
| if ((i != 0) && ((INTERNAL\_CLR\_FMT = = YUV422) | | (INTERNAL\_CLR\_FMT = = YUV420))) |  |
| dx = 2 |  |
| else |  |
| dx = 1 |  |
| if ((i != 0) && (INTERNAL\_CLR\_FMT = = YUV420)) |  |
| dy = 2 |  |
| else |  |
| dy = 1 |  |
| for (Tx = 0; Tx <= (NumTileCols − 1); Tx++) { |  |
| for (Ty = 0; Ty <= (NumTileRows − 1); Ty++) { |  |
| for (x = 16 \* LeftMBIndexOfTile[Tx] / dx + 2; x < (16 \* LeftMBIndexOfTile[Tx + 1] / dx − 2);  x += 4) |  |
| for (y = 16 \* TopMBIndexOfTile[Ty] / dy + 2; y < (16 \* TopMBIndexOfTile[Ty + 1] / dy − 2);  y += 4) { |  |
| arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y],   ImagePlane[i][x+2][y], ImagePlane[i][x+3][y],   ImagePlane[i][x][y+1], ImagePlane[i][x+1][y+1],   ImagePlane[i][x+2][y+1], ImagePlane[i][x+3][y+1],   ImagePlane[i][x][y+2], ImagePlane[i][x+1][y+2],   ImagePlane[i][x+2][y+2], ImagePlane[i][x+3][y+2],   ImagePlane[i][x][y+3], ImagePlane[i][x+1][y+3],   ImagePlane[i][x+2][y+3], ImagePlane[i][x+3][y+3]} |  |
| OverlapPostFilter4x4(arrayLocal[ ]) | 9.9.8.1 |
| ImagePlane[i][x][y] = arrayLocal[0] |  |
| ImagePlane[i][x+1][y] = arrayLocal[1] |  |
| ImagePlane[i][x+2][y] = arrayLocal[2] |  |
| ImagePlane[i][x+3][y] = arrayLocal[3] |  |
| ImagePlane[i][x][y+1] = arrayLocal[4] |  |
| ImagePlane[i][x+1][y+1] = arrayLocal[5] |  |
| ImagePlane[i][x+2][y+1] = arrayLocal[6] |  |
| ImagePlane[i][x+3][y+1] = arrayLocal[7] |  |
| ImagePlane[i][x][y+2] = arrayLocal[8] |  |
| ImagePlane[i][x+1][y+2] = arrayLocal[9] |  |
| ImagePlane[i][x+2][y+2] = arrayLocal[10] |  |
| ImagePlane[i][x+3][y+2] = arrayLocal[11] |  |
| ImagePlane[i][x][y+3] = arrayLocal[12] |  |
| ImagePlane[i][x+1][y+3] = arrayLocal[13] |  |
| ImagePlane[i][x+2][y+3] = arrayLocal[14] |  |
| ImagePlane[i][x+3][y+3] = arrayLocal[15] |  |
| } |  |
| if ((Tx = = 0) | | (HARD\_TILING\_FLAG = = TRUE)) { /\* Left edge \*/ |  |
| for (y = 16 \* TopMBIndexOfTile[Ty] / dy + 2; y < (16 \* TopMBIndexOfTile[Ty + 1] / dy − 2);  y += 4) { |  |
| x = 16 \* LeftMBIndexOfTile[Tx] / dx |  |
| arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x][y+1],   ImagePlane[i][x][y+2], ImagePlane[i][x][y+3]} |  |
| OverlapPostFilter4(arrayLocal[ ]) | 9.9.8.2 |
| ImagePlane[i][x][y] = arrayLocal[0] |  |
| ImagePlane[i][x][y+1] = arrayLocal[1] |  |
| ImagePlane[i][x][y+2] = arrayLocal[2] |  |
| ImagePlane[i][x][y+3] = arrayLocal[3] |  |
| x = 16 \* LeftMBIndexOfTile[Tx] / dx + 1 |  |
| arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x][y+1],   ImagePlane[i][x][y+2], ImagePlane[i][x][y+3]} |  |
| OverlapPostFilter4(arrayLocal[ ]) | 9.9.8.2 |
| ImagePlane[i][x][y] = arrayLocal[0] |  |
| ImagePlane[i][x][y+1] = arrayLocal[1] |  |
| ImagePlane[i][x][y+2] = arrayLocal[2] |  |
| ImagePlane[i][x][y+3] = arrayLocal[3] |  |
| } |  |
| } |  |
| if ((Ty = = 0) | | (HARD\_TILING\_FLAG = = TRUE)) { /\* Top edge \*/ |  |
| for (x = 16 \* LeftMBIndexOfTile[Tx] / dx + 2; x < (16 \* LeftMBIndexOfTile[Tx + 1] / dx − 2);  x += 4) { |  |
| y = 16 \* TopMBIndexOfTile[Ty] / dy |  |
| arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y],   ImagePlane[i][x+2][y], ImagePlane[i][x+3][y]} |  |
| OverlapPostFilter4(arrayLocal[ ]) | 9.9.8.2 |
| ImagePlane[i][x][y] = arrayLocal[0] |  |
| ImagePlane[i][x+1][y] = arrayLocal[1] |  |
| ImagePlane[i][x+2][y] = arrayLocal[2] |  |
| ImagePlane[i][x+3][y] = arrayLocal[3] |  |
| y = 16 \* TopMBIndexOfTile[Ty] / dy + 1 |  |
| arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y],   ImagePlane[i][x+2][y], ImagePlane[i][x+3][y]} |  |
| OverlapPostFilter4(arrayLocal[ ]) | 9.9.8.2 |
| ImagePlane[i][x][y] = arrayLocal[0] |  |
| ImagePlane[i][x+1][y] = arrayLocal[1] |  |
| ImagePlane[i][x+2][y] = arrayLocal[2] |  |
| ImagePlane[i][x+3][y] = arrayLocal[3] |  |
| } |  |
| } |  |
| if ((Tx = = NumTileCols − 1) | | (HARD\_TILING\_FLAG = = TRUE)) { /\* Right edge \*/ |  |
| for (y = 16 \* TopMBIndexOfTile[Ty] / dy + 2; y < (16 \* TopMBIndexOfTile[Ty + 1] / dy − 2);  y += 4) { |  |
| x = 16 \* LeftMBIndexOfTile[Tx + 1] / dx − 2 |  |
| arrayLocal[ ] = { ImagePlane[i][x][y], ImagePlane[i][x][y+1],   ImagePlane[i][x][y+2], ImagePlane[i][x][y+3]} |  |
| OverlapPostFilter4(arrayLocal[ ]) | 9.9.8.2 |
| ImagePlane[i][x][y] = arrayLocal[0] |  |
| ImagePlane[i][x][y+1] = arrayLocal[1] |  |
| ImagePlane[i][x][y+2] = arrayLocal[2] |  |
| ImagePlane[i][x][y+3] = arrayLocal[3] |  |
| x = 16 \* LeftMBIndexOfTile[Tx + 1] / dx − 1 |  |
| arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x][y+1],   ImagePlane[i][x][y+2], ImagePlane[i][x][y+3]} |  |
| OverlapPostFilter4(arrayLocal[ ]) | 9.9.8.2 |
| ImagePlane[i][x][y] = arrayLocal[0] |  |
| ImagePlane[i][x][y+1] = arrayLocal[1] |  |
| ImagePlane[i][x][y+2] = arrayLocal[2] |  |
| ImagePlane[i][x][y+3] = arrayLocal[3] |  |
| } |  |
| } |  |
| if ((Ty = = NumTileRows − 1) | | (HARD\_TILING\_FLAG = = TRUE)) { /\* Bottom edge \*/ |  |
| for (x = 16 \* LeftMBIndexOfTile[Tx] / dx + 2; x < (16 \* LeftMBIndexOfTile[Tx + 1] / dx − 2);  x += 4) { |  |
| y = 16 \* TopMBIndexOfTile[Ty + 1] / dy − 2 |  |
| arrayLocal[ ] = {ImagePlane[i][x][y], ImagePlane[i][x+1][y],   ImagePlane[i][x+2][y], ImagePlane[i][x+3][y]} |  |
| OverlapPostFilter4(arrayLocal[ ]) | 9.9.8.2 |
| ImagePlane[i][x][y] = arrayLocal[0] |  |
| ImagePlane[i][x+1][y] = arrayLocal[1] |  |
| ImagePlane[i][x+2][y] = arrayLocal[2] |  |
| ImagePlane[i][x+3][y] = arrayLocal[3] |  |
| y = 16 \* TopMBIndexOfTile[Ty + 1] / dy − 1 |  |
| 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 = = 0) && (Ty = = 0)) | | (HARD\_TILING\_FLAG = = TRUE)) { /\* Top left edge \*/ |  |
| x = 16 \* LeftMBIndexOfTile[Tx] / dx |  |
| y = 16 \* TopMBIndexOfTile[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 \* TopMBIndexOfTile[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 \* TopMBIndexOfTile[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 \* TopMBIndexOfTile[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 ((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] |  |
| 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 |
| 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] |  |
| 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] |  |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  |

### Inverse transform basic operations

#### ICT4x4( )

NOTE 1 – The 2D ICT4x4( ) is built using the three operators: T2x2h, InvTodd and InvToddodd, preceded by the permutation function InvPermute. After the initial permutation, the transform operation consists of two stages, where each stage operates on all 16 of the input values.

The function ICT4x4( ) is specified by the pseudocode in Table 160.

Table 160 – Pseudocode for function ICT4x4( )

| **ICT4x4(iCoeff[ ])** { | **Reference** |
| --- | --- |
| /\* Permute the coefficients \*/ |  |
| InvPermute(iCoeff[ ]) | 9.9.7.5 |
| /\* First stage \*/ |  |
| arrayLocal[ ] = {iCoeff[0], iCoeff[1], iCoeff[4], iCoeff[5]} |  |
| T2x2h(arrayLocal[ ], 1) | 9.9.7.2 |
| iCoeff[0] = arrayLocal[0] |  |
| iCoeff[1] = arrayLocal[1] |  |
| iCoeff[4] = arrayLocal[2] |  |
| iCoeff[5] = arrayLocal[3] |  |
| arrayLocal[ ] = {iCoeff[2], iCoeff[3], iCoeff[6], iCoeff[7]} |  |
| InvTodd(arrayLocal[ ]) | 9.9.7.3 |
| iCoeff[2] = arrayLocal[0] |  |
| iCoeff[3] = arrayLocal[1] |  |
| iCoeff[6] = arrayLocal[2] |  |
| iCoeff[7] = arrayLocal[3] |  |
| arrayLocal[ ] = {iCoeff[8], iCoeff[12], iCoeff[9], iCoeff[13]} |  |
| InvTodd(arrayLocal[ ]) | 9.9.7.3 |
| iCoeff[8] = arrayLocal[0] |  |
| iCoeff[12] = arrayLocal[1] |  |
| iCoeff[9] = arrayLocal[2] |  |
| iCoeff[13] = arrayLocal[3] |  |
| arrayLocal[ ] = {iCoeff[10], iCoeff[11], iCoeff[14], iCoeff[15]} |  |
| InvToddodd(arrayLocal[ ]) | 9.9.7.4 |
| iCoeff[10] = arrayLocal[0] |  |
| iCoeff[11] = arrayLocal[1] |  |
| iCoeff[14] = arrayLocal[2] |  |
| iCoeff[15] = arrayLocal[3] |  |
| /\* Second stage \*/ |  |
| arrayLocal[ ] = {iCoeff[0], iCoeff[3], iCoeff[12], iCoeff[15]} |  |
| T2x2h(arrayLocal[ ], 0) | 9.9.7.2 |
| iCoeff[0] = arrayLocal[0] |  |
| iCoeff[3] = arrayLocal[1] |  |
| iCoeff[12] = arrayLocal[2] |  |
| iCoeff[15] = arrayLocal[3] |  |
| arrayLocal[ ] = {iCoeff[5], iCoeff[6], iCoeff[9], iCoeff[10]} |  |
| T2x2h(arrayLocal[ ], 0) | 9.9.7.2 |
| iCoeff[5] = arrayLocal[0] |  |
| iCoeff[6] = arrayLocal[1] |  |
| iCoeff[9] = arrayLocal[2] |  |
| iCoeff[10] = arrayLocal[3] |  |
| arrayLocal[ ] = {iCoeff[1], iCoeff[2], iCoeff[13], iCoeff[14]} |  |
| T2x2h(arrayLocal[ ], 0) | 9.9.7.2 |
| iCoeff[1] = arrayLocal[0] |  |
| iCoeff[2] = arrayLocal[1] |  |
| iCoeff[13] = arrayLocal[2] |  |
| iCoeff[14] = arrayLocal[3] |  |
| arrayLocal[ ] = {iCoeff[4], iCoeff[7], iCoeff[8], iCoeff[11]} |  |
| T2x2h(arrayLocal[ ], 0) | 9.9.7.2 |
| iCoeff[4] = arrayLocal[0] |  |
| iCoeff[7] = arrayLocal[1] |  |
| iCoeff[8] = arrayLocal[2] |  |
| iCoeff[11] = arrayLocal[3] |  |
| } |  |

NOTE 2 – Each stage consists of four 2×2 transforms that may be done in any arbitrary sequence or concurrently within the stage. However, the first stage transforms must be complete before any of the second stage transforms are initiated.

#### T2x2h( )

The function T2x2h( ) is specified in Table 161.

NOTE – The variable valRound is a rounding control variable. The value of valRound is set to 0 or 1 by the function that invokes T2x2h( ). The inverse of T2x2Th( ) is two successive applications of T2x2Th, operating on variables of the array iCoeff[ ] with the same value of valRound.

Table 161 – Pseudocode for function T2x2h( )

|  |  |
| --- | --- |
| **T2x2h(iCoeff[ ], valRound)** { | **Reference** |
| iCoeff[0] += iCoeff[3] |  |
| iCoeff[1] −= iCoeff[2] |  |
| valT1 = ((iCoeff[0] − iCoeff[1] + valRound) >> 1) |  |
| valT2 = iCoeff[2] |  |
| iCoeff[2] = valT1 − iCoeff[3] |  |
| iCoeff[3] = valT1 − valT2 |  |
| iCoeff[0] −= iCoeff[3] |  |
| iCoeff[1] += iCoeff[2] |  |
| } |  |

#### InvTodd( )

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

Table 162 – Pseudocode for function InvTodd( )

|  |  |
| --- | --- |
| **InvTodd(iCoeff[ ])** { | **Reference** |
| iCoeff[1] += iCoeff[3] |  |
| iCoeff[0] −= iCoeff[2] |  |
| iCoeff[3] −= (iCoeff[1] >> 1) |  |
| iCoeff[2] += ((iCoeff[0] + 1) >> 1) |  |
| iCoeff[0] −= ((3\* iCoeff[1] + 4) >> 3) |  |
| iCoeff[1] += ((3\* iCoeff[0] + 4) >> 3) |  |
| iCoeff[2] −= ((3\* iCoeff[3] + 4) >> 3) |  |
| iCoeff[3] += ((3\* iCoeff[2] + 4) >> 3 |  |
| iCoeff[2] −= ((iCoeff[1] + 1) >> 1) |  |
| iCoeff[3] = ((iCoeff[0] + 1) >> 1) − iCoeff[3] |  |
| iCoeff[1] += iCoeff[2] |  |
| iCoeff[0] −= iCoeff[3] |  |
| } |  |

#### InvToddodd( )

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

Table 163 – Pseudocode for function InvToddodd

|  |  |
| --- | --- |
| **InvToddodd(**iCoeff[ ]**)** { | **Reference** |
| iCoeff[3] += iCoeff[0] |  |
| iCoeff[2] −= iCoeff[1] |  |
| valT1 = iCoeff[3] >> 1 |  |
| valT2 = iCoeff[2] >> 1 |  |
| iCoeff[0] −= valT1 |  |
| iCoeff[1] += valT2 |  |
| iCoeff[0] −= ((iCoeff[1] \* 3 + 3) >> 3) |  |
| iCoeff[1] += ((iCoeff[0] \* 3 + 3) >> 2) |  |
| iCoeff[0] −= ((iCoeff[1] \* 3 + 4) >> 3) |  |
| iCoeff[1] −= valT2 |  |
| iCoeff[0] += valT1 |  |
| iCoeff[2] += iCoeff[1] |  |
| iCoeff[3] −= iCoeff[0] |  |
| iCoeff[1] = −iCoeff[1] |  |
| iCoeff[2] = −iCoeff[2] |  |
| } |  |

#### InvPermute( )

The function InvPermute( ) operates on an ordered array of 16 sample values, producing a permuted list. The input to this function is the ordered array arrayInput[i], for i ranging from 0 to 15. The output of this function is the re-ordered array arrayInput[i].

To define the permutation, the array InvPermArr[i] is specified, for i ranging from 0 to 15, in Table 164.

Table 164 – Inverse Permutation

| **i** | **InvPermArr[i]** |
| --- | --- |
| 0 | 0 |
| 1 | 8 |
| 2 | 4 |
| 3 | 13 |
| 4 | 2 |
| 5 | 15 |
| 6 | 3 |
| 7 | 14 |
| 8 | 1 |
| 9 | 12 |
| 10 | 5 |
| 11 | 9 |
| 12 | 7 |
| 13 | 11 |
| 14 | 6 |
| 15 | 10 |

The function InvPermute( ) is specified in Table 165.

Table 165 – Pseudocode for function InvPermute( )

|  |  |
| --- | --- |
| **InvPermute(arrayInput[ ]) {** | **Reference** |
| for (i = 0; i <= 15; i++) |  |
| arrayTemp[InvPermArr[i]] = arrayInput[i] |  |
| for (i = 0; i <= 15; i++) |  |
| arrayInput[i] = arrayTemp[i] |  |
| } |  |

#### InvPermute2pt( )

The function InvPermute2pt( ) operates on an ordered array of 2 sample values, producing a permuted list. The input to this function is the ordered array arrayInput[i], for i ranging from 0 to 1. The output of this function is the re-ordered array arrayInput[i].

The function InvPermute2pt( ) is specified in Table 166.

Table 166 – Pseudocode for function InvPermute2pt( )

|  |  |
| --- | --- |
| **InvPermute2pt( ) {** | **Reference** |
| arrayTemp[0] = arrayInput[1] |  |
| arrayTemp[1] = arrayInput[0] |  |
| for (i = 0; i <= 1; i++) |  |
| arrayInput[i] = arrayTemp[i] |  |
| } |  |

#### T2pt( )

The function T2pt( ) is specified by the pseudocode in Table 167.

Table 167 – Pseudocode for function T2pt( )

|  |  |
| --- | --- |
| **T2pt(iCoeff[ ])** { | **Reference** |
| iCoeff[0] −= (iCoeff[1]+1) >> 1 |  |
| iCoeff[1] += iCoeff[0] |  |
| } |  |

### Overlap filtering functions

#### OverlapPostFilter4x4( )

The function OverlapPostFilter4x4( ) is specified in Table 168.

Table 168 – Pseudocode for function OverlapPostFilter4x4( )

| **OverlapPostFilter4x4(iCoeff[ ])** { | **Reference** |
| --- | --- |
| arrayLocal[ ] = {iCoeff[0], iCoeff[3], iCoeff[12], iCoeff[15]} |  |
| T2x2h(arrayLocal[ ], 0) | 9.9.7.2 |
| iCoeff[0] = arrayLocal[0] |  |
| iCoeff[3] = arrayLocal[1] |  |
| iCoeff[12] = arrayLocal[2] |  |
| iCoeff[15] = arrayLocal[3] |  |
| arrayLocal[ ] = {iCoeff[1], iCoeff[2], iCoeff[13], iCoeff[14]} |  |
| T2x2h(arrayLocal[ ], 0) | 9.9.7.2 |
| iCoeff[1] = arrayLocal[0] |  |
| iCoeff[2] = arrayLocal[1] |  |
| iCoeff[13] = arrayLocal[2] |  |
| iCoeff[14] = arrayLocal[3] |  |
| arrayLocal[ ] = {iCoeff[4], iCoeff[7], iCoeff[8], iCoeff[11]} |  |
| T2x2h(arrayLocal[ ], 0) | 9.9.7.2 |
| iCoeff[4] = arrayLocal[0] |  |
| iCoeff[7] = arrayLocal[1] |  |
| iCoeff[8] = arrayLocal[2] |  |
| iCoeff[11] = arrayLocal[3] |  |
| arrayLocal[ ] = {iCoeff[5], iCoeff[6], iCoeff[9], iCoeff[10]} |  |
| T2x2h(arrayLocal[ ], 0) | 9.9.7.2 |
| iCoeff[5] = arrayLocal[0] |  |
| iCoeff[6] = arrayLocal[1] |  |
| iCoeff[9] = arrayLocal[2] |  |
| iCoeff[10] = arrayLocal[3] |  |
| arrayLocal[ ] = {iCoeff[13], iCoeff[12]} |  |
| InvRotate(arrayLocal[ ]) | 9.9.8.5 |
| iCoeff[13] = arrayLocal[0] |  |
| iCoeff[12] = arrayLocal[1] |  |
| arrayLocal[ ] = {iCoeff[9], iCoeff[8]} |  |
| InvRotate(arrayLocal[ ]) | 9.9.8.5 |
| iCoeff[9] = arrayLocal[0] |  |
| iCoeff[8] = arrayLocal[1] |  |
| arrayLocal[ ] = {iCoeff[7], iCoeff[3]} |  |
| InvRotate(arrayLocal[ ]) | 9.9.8.5 |
| iCoeff[7] = arrayLocal[0] |  |
| iCoeff[3] = arrayLocal[1] |  |
| arrayLocal[ ] = {iCoeff[6], iCoeff[2]} |  |
| InvRotate(arrayLocal[ ]) | 9.9.8.5 |
| iCoeff[6] = arrayLocal[0] |  |
| iCoeff[2] = arrayLocal[1] |  |
| arrayLocal[ ] = {iCoeff[10], iCoeff[11], iCoeff[14], iCoeff[15]} |  |
| InvToddoddPOST(arrayLocal[ ]) | 9.9.8.8 |
| iCoeff[10] = arrayLocal[0] |  |
| iCoeff[11] = arrayLocal[1] |  |
| iCoeff[14] = arrayLocal[2] |  |
| iCoeff[15] = arrayLocal[3] |  |
| arrayLocal[ ] = {iCoeff[0], iCoeff[15]} |  |
| InvScale(arrayLocal[ ]) | 9.9.8.6 |
| iCoeff[0] = arrayLocal[0] |  |
| iCoeff[15] = arrayLocal[1] |  |
| arrayLocal[ ] = {iCoeff[1], iCoeff[14]} |  |
| InvScale(arrayLocal[ ]) | 9.9.8.6 |
| iCoeff[1] = arrayLocal[0] |  |
| iCoeff[14] = arrayLocal[1] |  |
| arrayLocal[ ] = {iCoeff[4], iCoeff[11]} |  |
| InvScale(arrayLocal[ ]) | 9.9.8.6 |
| iCoeff[4] = arrayLocal[0] |  |
| iCoeff[11] = arrayLocal[1] |  |
| arrayLocal[ ] = {iCoeff[5], iCoeff[10]} |  |
| InvScale(arrayLocal[ ]) | 9.9.8.6 |
| iCoeff[5] = arrayLocal[0] |  |
| iCoeff[10] = arrayLocal[1] |  |
| arrayLocal[ ] = {iCoeff[0], iCoeff[3], iCoeff[12], iCoeff[15]} |  |
| T2x2hPOST(arrayLocal[ ]) | 9.9.8.7 |
| iCoeff[0] = arrayLocal[0] |  |
| iCoeff[3] = arrayLocal[1] |  |
| iCoeff[12] = arrayLocal[2] |  |
| iCoeff[15] = arrayLocal[3] |  |
| arrayLocal[ ] = {iCoeff[1], iCoeff[2], iCoeff[13], iCoeff[14]} |  |
| T2x2hPOST(arrayLocal[ ]) | 9.9.8.7 |
| iCoeff[1] = arrayLocal[0] |  |
| iCoeff[2] = arrayLocal[1] |  |
| iCoeff[13] = arrayLocal[2] |  |
| iCoeff[14] = arrayLocal[3] |  |
| arrayLocal[ ] = {iCoeff[4], iCoeff[7], iCoeff[8], iCoeff[11]} |  |
| T2x2hPOST(arrayLocal[ ]) | 9.9.8.7 |
| iCoeff[4] = arrayLocal[0] |  |
| iCoeff[7] = arrayLocal[1] |  |
| iCoeff[8] = arrayLocal[2] |  |
| iCoeff[11] = arrayLocal[3] |  |
| arrayLocal[ ] = {iCoeff[5], iCoeff[6], iCoeff[9], iCoeff[10]} |  |
| T2x2hPOST(arrayLocal[ ]) | 9.9.8.7 |
| iCoeff[5] = arrayLocal[0] |  |
| iCoeff[6] = arrayLocal[1] |  |
| iCoeff[9] = arrayLocal[2] |  |
| iCoeff[10] = arrayLocal[3] |  |
| } |  |

#### OverlapPostFilter4( )

The function OverlapPostFilter4( ) is specified in Table 169.

Table 169 – Pseudocode for function OverlapPostFilter4( )

|  |  |
| --- | --- |
| **OverlapPostFilter4(iCoeff[ ])** { | **Reference** |
| iCoeff[0] += iCoeff[3] |  |
| iCoeff[1] += iCoeff[2] |  |
| iCoeff[3] −= ((iCoeff[0] + 1) >> 1) |  |
| iCoeff[2] −= ((iCoeff[1] + 1) >> 1) |  |
| InvScale(iCoeff[0], iCoeff[3]) | 9.9.8.6 |
| InvScale(iCoeff[1], iCoeff[2]) | 9.9.8.6 |
| iCoeff[0] += ((iCoeff[3] \* 3+ 4) >> 3) |  |
| iCoeff[1] += ((iCoeff[2] \* 3 + 4) >> 3) |  |
| iCoeff[3] −= ( iCoeff[0] >> 1) |  |
| iCoeff[2] −= ( iCoeff[1] >> 1) |  |
| iCoeff[0] += iCoeff[3] |  |
| iCoeff[1] += iCoeff[2] |  |
| iCoeff[3] = −iCoeff[3] |  |
| iCoeff[2] = −iCoeff[2] |  |
| InvRotate(iCoeff[2], iCoeff[3]) | 9.9.8.5 |
| iCoeff[3] += ((iCoeff[0] + 1) >> 1) |  |
| iCoeff[2] += ((iCoeff[1] + 1) >> 1) |  |
| iCoeff[0] −= iCoeff[3] |  |
| iCoeff[1] −= iCoeff[2] |  |
| } |  |

#### OverlapPostFilter2x2( )

The function OverlapPostFilter2x2( ) is specified in Table 170.

Table 170 – Pseudocode for function OverlapPostFilter2x2( )

|  |  |
| --- | --- |
| **OverlapPostFilter2x2(iCoeff[ ])** { | **Reference** |
| iCoeff[0] += iCoeff[3] |  |
| iCoeff[1] += iCoeff[2] |  |
| iCoeff[3] −= ((iCoeff[0] + 1) >> 1) |  |
| iCoeff[2] −= ((iCoeff[1] + 1) >> 1) |  |
| iCoeff[1] += ((iCoeff[0] + 2) >> 2) |  |
| iCoeff[0] += ((iCoeff[1] + 1) >> 1) |  |
| iCoeff[0] += (iCoeff[1] >> 5) |  |
| iCoeff[0] += (iCoeff[1] >> 9) |  |
| iCoeff[0] += (iCoeff[1] >> 13) |  |
| iCoeff[1] += ((iCoeff[0] + 2) >> 2) |  |
| iCoeff[3] += ((iCoeff[0] + 1) >> 1) |  |
| iCoeff[2] += ((iCoeff[1] + 1) >> 1) |  |
| iCoeff[0] −= iCoeff[3] |  |
| iCoeff[1] −= iCoeff[2] |  |
| } |  |

#### OverlapPostFilter2( )

The function OverlapPostFilter2( ) is specified in Table 171.

Table 171 – Pseudocode for function OverlapPostFilter2( )

|  |  |
| --- | --- |
| **OverlapPostFilter2(iCoeff[ ])** { | **Reference** |
| iCoeff[1] += ((iCoeff[0] + 2) >> 2) |  |
| iCoeff[0] += ((iCoeff[1] + 1) >> 1) |  |
| iCoeff[0] += (iCoeff[1] >> 5) |  |
| iCoeff[0] += (iCoeff[1] >> 9) |  |
| iCoeff[0] += (iCoeff[1] >> 13) |  |
| iCoeff[1] += ((iCoeff[0] + 2) >> 2) |  |
| } |  |

#### 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) |  |
| } |  |

#### 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) |  |
| } |  |

#### T2x2hPOST( )

The function T2x2hPOST( ) is specified by the pseudocode in Table 174.

Table 174 – Pseudocode for function T2x2hPOST( )

|  |  |
| --- | --- |
| **T2x2hPOST(iCoeff[ ])** { | **Reference** |
| iCoeff[1] −= iCoeff[2] |  |
| iCoeff[0] += (iCoeff[3] \* 3 + 4) >> 3 |  |
| iCoeff[3] −= (iCoeff[1] >> 1) |  |
| iCoeff[2] = ((iCoeff[0] − iCoeff[1]) >> 1) − iCoeff[2] |  |
| valT1 = iCoeff[2] |  |
| iCoeff[2] = iCoeff[3] |  |
| iCoeff[3] = valT1 |  |
| iCoeff[0] −= iCoeff[3] |  |
| iCoeff[1] += iCoeff[2] |  |
| } |  |

#### InvToddoddPOST( )

The function InvToddoddPOST( ) is specified by the pseudocode in Table 175.

Table 175 – Pseudocode for function InvToddoddPOST( )

|  |  |
| --- | --- |
| **InvToddoddPOST(iCoeff[ ])** { | **Reference** |
| iCoeff[3] += iCoeff[0] |  |
| iCoeff[2] −= iCoeff[1] |  |
| valT1 = iCoeff[3] >> 1 |  |
| valT2 = iCoeff[2] >> 1 |  |
| iCoeff[0] −= valT1 |  |
| iCoeff[1] += valT2 |  |
| iCoeff[0] −= (iCoeff[1] \* 3 + 6) >> 3 |  |
| iCoeff[1] += (iCoeff[0] \* 3 + 2) >> 2 |  |
| iCoeff[0] −= (iCoeff[1] \* 3 + 4) >> 3 |  |
| iCoeff[1] −= valT2 |  |
| iCoeff[0] += valT1 |  |
| iCoeff[2] += iCoeff[1] |  |
| iCoeff[3] −= iCoeff[0] |  |
| } |  |

## Output formatting

The final stage of the decoder consists of converting the sample values reconstructed in the internal format to the intended output format as specified in 9.10.2.

### 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, colour format conversion is applied to convert the internal colour formats to output formats. The colour format conversions are specified in the following. A bias is added to the sample values, to re-centre the values around the nominal value for a neutral or zero intensity representation. When the scaling mode is used, on the decoder side, the values are rounded down after colour conversion. For high numerical range formats (BD16, BD16S, BD32S and BD32F), the internal integer representations need to be converted to output representations. Finally, the values are clipped to fit the appropriate range.

### Output formatting stage

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

The output formatting process is specified for the combinations of OUTPUT\_BITDEPTH and OUTPUT\_CLR\_FMT that are listed in Table 176.

In Table 176, "+" indicates that output formatting is specified for the corresponding combinations of OUTPUT\_BIT\_DEPTH and OUTPUT\_CLR\_FMT. The combination of OUTPUT\_BIT\_DEPTH and OUTPUT\_CLR\_FMT shall not have the value corresponding to empty cells.

Table 176 – Conformance-specified output formatting combinations of OUTPUT\_BITDEPTH and OUTPUT\_CLR\_FMT

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| OUTPUT\_BITDEPTH | OUTPUT\_CLR\_FMT | | | | | | | | |
| YONLY | YUV420 | YUV422 | YUV444 | RGB | RGBE | CMYK | CMYKDIRECT | NCOMPONENT |
| BD1WHITE1 | + |  |  |  |  |  |  |  |  |
| BD1BLACK1 | + |  |  |  |  |  |  |  |  |
| BD5 |  |  |  |  | + |  |  |  |  |
| BD565 |  |  |  |  | + |  |  |  |  |
| BD8 | + | + | + | + | + | + | + | + | + |
| BD10 |  |  | + | + | + |  |  |  |  |
| BD16 | + |  | + | + | + |  | + | + | + |
| BD16S | + |  |  | + | + |  |  |  |  |
| BD16F | + |  |  |  | + |  |  |  |  |
| BD32S | + |  |  |  | + |  |  |  |  |
| BD32F | + |  |  |  | + |  |  |  |  |

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

Table 177 – Pseudocode for function OutputFormatting( )

|  |  |
| --- | --- |
| **OutputFormatting( ) {** | **Reference** |
| SamplingConversion( ) | ‎9.10.3.1 |
| ConvertInternalToOutputClrFmt( ) | ‎9.10.4.1 |
| AddBias( ) | ‎9.10.5 |
| ComputeScaling( ) | ‎9.10.6 |
| PostscalingProcess( ) | ‎9.10.7.1 |
| ClippingAndPackingStage( ) | ‎9.10.8.1 |
| } |  |

### Sampling conversion

#### General

The sampling conversion process is specified in Table 178.

The combinations of INTERNAL\_CLR\_FMT and OUTPUT\_CLR\_FMT for which sampling conversions are specified for conformance purposes are specified in Table 179. In Table 179, "+" indicates that no sampling conversion is required. It is a requirement of codestream conformance to this Specification that the combination of INTERNAL\_CLR\_FMT and OUTPUT\_CLR\_FMT shall not have a value corresponding to any empty cell in Table 179.

In the illustrated case in Table 178 in which upsampling is specified both vertically and horizontally for INTERNAL\_CLR\_FMT equal to YUV420, the upsampling process to be performed by the decoder shall produce an array of two-dimensionally upsampled samples at the index values for which such samples are produced as specified in Table 178. However, decoders may use alternative upsampling methods (different from that specified in 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 in 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 (INTERNAL\_CLR\_FMT = = YUV420) |  |
| Upsample( ) in the vertical direction | 9.10.3.2 |
| if ((INTERNAL\_CLR\_FMT = = YUV422) | | (INTERNAL\_CLR\_FMT = = YUV420)) |  |
| Upsample( ) in the horizontal direction | 9.10.3.2 |
| } |  |
| } |  |

Table 179 – Conformance-specified sampling conversions

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| OUTPUT\_CLR\_FMT | INTERNAL\_CLR\_FMT | | | | | |
| YONLY | YUV420 | YUV422 | YUV444 | YUVK | NCOMPONENT |
| YONLY | + |  |  |  |  |  |
| YUV420 |  | + |  |  |  |  |
| YUV422 |  |  | + |  |  |  |
| YUV444 | + | Upsample( ) in the vertical and horizontal directions | Upsample( ) in the horizontal direction | + |  |  |
| RGB with OUTPUT\_BITDEPTH equal to BD5, BD565, BD8, BD10, BD16, BD16S or BD32S | + | Upsample( ) in the vertical and horizontal directions | Upsample( ) in the horizontal direction | + |  |  |
| RGB with OUTPUT\_BITDEPTH equal to BD16F or BD32F |  |  |  | + |  |  |
| RGBE |  |  |  | + |  |  |
| CMYK |  |  |  |  | + |  |
| CMYKDIRECT |  |  |  |  | + |  |
| NCOMPONENT |  |  |  |  |  | + |

#### Upsample( )

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

The upsampling process to be performed by the decoder shall produce an array of upsampled samples at the index values for which such samples are produced as specified in Table 180. However, decoders may use alternative upsampling methods (different from that specified in Table 180) – 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 in 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 in Table 180.

Table 180 – Pseudocode for function Upsample( )

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

The values of the filter coefficients iH[0], iH[1], iH[2], iH[3]for the chroma positions are specified in 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 |

### Conversion from INTERNAL\_CLR\_FMT to OUTPUT\_CLR\_FMT

#### Overview

The conversion process proceeds as specified in Table 182.

Table 182 – Pseudocode for function ConvertInternalToOutputClrFmt( )

|  |  |
| --- | --- |
| **ConvertInternalToOutputClrFmt( ) {** | **Reference** |
| if ((INTERNAL\_CLR\_FMT = = YUVK) &&  (OUTPUT\_CLR\_FMT = = CMYK)) |  |
| InvColorFmtConvert3( ) | 9.10.4.4 |
| else if ((INTERNAL\_CLR\_FMT = = YUVK) &&   (OUTPUT\_CLR\_FMT = = CMYKDIRECT)) |  |
| InvColorFmtConvert4( ) | 9.10.4.5 |
| else if ((INTERNAL\_CLR\_FMT = = YONLY) &&  (OUTPUT\_CLR\_FMT = = RGB)) |  |
| InvColorFmtConvert1( ) | 9.10.4.2 |
| else if (((INTERNAL\_CLR\_FMT = = YUV444) | |  (INTERNAL\_CLR\_FMT = = YUV422) | |   (INTERNAL\_CLR\_FMT = = YUV420)) &&  ((OUTPUT\_CLR\_FMT = = RGB) | | (OUTPUT\_CLR\_FMT = = RGBE))) |  |
| InvColorFmtConvert2( ) | 9.10.4.3 |
| else if ((INTERNAL\_CLR\_FMT = = YONLY) &&  ((OUTPUT\_CLR\_FMT = = YUV444) | | (OUTPUT\_CLR\_FMT = = YUV422) | |  (OUTPUT\_CLR\_FMT = = YUV420))) { |  |
| if (OUTPUT\_CLR\_FMT = = YUV420) |  |
| chromaHeight = ExtendedHeight[0] / 2 |  |
| else |  |
| chromaHeight = ExtendedHeight[0] |  |
| if ((OUTPUT\_CLR\_FMT = = YUV422) | | (OUTPUT\_CLR\_FMT = = YUV420)) |  |
| chromaWidth = ExtendedWidth[0] / 2 |  |
| else |  |
| chromaWidth = ExtendedWidth[0] |  |
| for (i = 1; i < 3; i ++) |  |
| for (y = 0; y < chromaHeight; y++) |  |
| for (x = 0; x < chromaWidth; x++) |  |
| ImagePlane[i][x][y] = 0 /\* Ensure that chroma is inferred as zero \*/ |  |
| } |  |
| } |  |

The combinations of INTERNAL\_CLR\_FMT and OUTPUT\_CLR\_FMT for which colour format conversions are specified for conformance purposes are specified in Table 183. In Table 183, "+" indicates that no colour format conversion is required. For cases that require colour format conversion, the function name for the conversion process is specified in the table cell. It is a requirement of codestream conformance to this Specification that the combination of INTERNAL\_CLR\_FMT and OUTPUT\_CLR\_FMT shall not have a value corresponding to any empty cell in Table 183.

Table 183 – Conformance-specified colour format conversions

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| OUTPUT\_CLR\_FMT | INTERNAL\_CLR\_FMT | | | | | |
| YONLY | YUV420 | YUV422 | YUV444 | YUVK | NCOMPONENT |
| YONLY | + |  |  |  |  |  |
| YUV420 |  | + |  |  |  |  |
| YUV422 |  |  | + |  |  |  |
| YUV444 | + | + | + | + |  |  |
| RGB with OUTPUT\_BITDEPTH equal to BD5, BD565, BD8, BD10, BD16, BD16S or BD32S | InvColorFmtConvert1( ) | InvColorFmtConvert2( ) | InvColorFmtConvert2( ) | InvColorFmtConvert2( ) |  |  |
| RGB with OUTPUT\_BITDEPTH equal to BD16F or BD32F |  |  |  | InvColorFmtConvert2( ) |  |  |
| RGBE |  |  |  | InvColorFmtConvert2( ) |  |  |
| CMYK |  |  |  |  | InvColorFmtConvert3( ) |  |
| CMYKDIRECT |  |  |  |  | InvColorFmtConvert4( ) |  |
| NCOMPONENT |  |  |  |  |  | + |

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

#### InvColorFmtConvert1( )

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

Table 184 – Pseudocode for function InvColorFmtConvert1( )

|  |  |
| --- | --- |
| **InvColorFmtConvert1( )** { | **Reference** |
| for (y = 0; y < ExtendedHeight[0]; y++) |  |
| for (x = 0; x < ExtendedWidth[0]; x++) { |  |
| /\* ImagePlane[0][x][y] G = Y \*/ |  |
| ImagePlane[1][x][y] = ImagePlane[0][x][y] /\* R = Y \*/ |  |
| ImagePlane[2][x][y] = ImagePlane[0][x][y] /\* B = Y \*/ |  |
| } |  |
| } |  |

#### InvColorFmtConvert2( )

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

Table 185 – Pseudocode for function InvColorFmtConvert2( )

|  |  |
| --- | --- |
| **InvColorFmtConvert2( )** { | **Reference** |
| for (y = 0; y < ExtendedHeight[0]; y++) |  |
| for (x = 0; x < ExtendedWidth[0]; x++) { |  |
| tempT = −ImagePlane[1][x][y]   /\* t = −U \*/ |  |
| arrayOut[1] = ImagePlane[0][x][y] − Floor(tempT ÷ 2)   /\* G = Y − Floor(t ÷ 2) \*/ |  |
| arrayOut[0] = tempT + arrayOut[1] − Ceiling(ImagePlane[2][x][y] ÷ 2)   /\* R = t + G − Ceiling(V ÷ 2) \*/ |  |
| arrayOut[2] = ImagePlane[2][x][y] + arrayOut[0]   /\* B = V + R \*/ |  |
| if ((OUTPUT\_BITDEPTH = = BD5 | | OUTPUT\_BITDEPTH = = BD565 | |  OUTPUT\_BITDEPTH = = BD10) &&  !RED\_BLUE\_NOT\_SWAPPED\_FLAG) { |  |
| tempT = arrayOut[0] |  |
| arrayOut[0] = arrayOut[2] |  |
| arrayOut[2] = tempT |  |
| } |  |
| for (i = 0; i < 3; i++) |  |
| ImagePlane[i][x][y] = arrayOut[i] |  |
| } |  |
| } |  |

#### InvColorFmtConvert3( )

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

Table 186 – Pseudocode for function InvColorFmtConvert3( )

|  |  |
| --- | --- |
| **InvColorFmtConvert3( )** { | **Reference** |
| for (y = 0; y < ExtendedHeight[0]; y++) |  |
| for (x = 0; x < ExtendedWidth[0]; x++) { |  |
| arrayOut[3] = ImagePlane[3][x][y] + Floor(ImagePlane[0][x][y] ÷ 2)   /\* k = K + Floor(Y ÷ 2) \*/ |  |
| arrayOut[1] = arrayOut[3] − ImagePlane[0][x][y] −   Floor(ImagePlane[1][x][y] ÷ 2) /\* m = k − Y − Floor(U ÷ 2) \*/ |  |
| arrayOut[0] = ImagePlane[1][x][y] + arrayOut[1] +   Floor(ImagePlane[2][x][y] ÷ 2) /\* c = U + m + Floor(V ÷ 2) \*/ |  |
| arrayOut[2] = arrayOut[0] − ImagePlane[2][x][y]   /\* y = c − V \*/ |  |
| for (i = 0; i < 4; i++) |  |
| ImagePlane[i][x][y] = arrayOut[i] |  |
| } |  |
| } |  |

#### InvColorFmtConvert4( )

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

Table 187 – Pseudocode for function InvColorFmtConvert4( )

|  |  |
| --- | --- |
| **InvColorFmtConvert4( )** { | **Reference** |
| for (y = 0; y < ExtendedHeight[0]; y++) |  |
| for (x = 0; x < ExtendedWidth[0]; x++) { |  |
| arrayOut[3] = ImagePlane[0][x][y] /\* k = Y \*/ |  |
| arrayOut[1] = ImagePlane[2][x][y] /\* m = V \*/ |  |
| arrayOut[0] = ImagePlane[1][x][y] /\* c = U \*/ |  |
| arrayOut[2] = ImagePlane[3][x][y] /\* y = K \*/ |  |
| for (i = 0; i < 4; i++) |  |
| ImagePlane[i][x][y] = arrayOut[i] |  |
| } |  |
| } |  |

### AddBias( )

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

Table 188 – Pseudocode for function AddBias( )

|  |  |
| --- | --- |
| **AddBias( )** { | **Reference** |
| if (SCALED\_FLAG) |  |
| iScale = 3 |  |
| else |  |
| iScale = 0 |  |
| if (OUTPUT\_BITDEPTH = = BD5) |  |
| iBias = (1 << 4) |  |
| else if (OUTPUT\_BITDEPTH = = BD565) |  |
| iBias = (1 << 5) |  |
| else if (OUTPUT\_BITDEPTH = = BD8) |  |
| iBias = (1 << 7) |  |
| else if (OUTPUT\_BITDEPTH = = BD10) |  |
| iBias = (1 << 9) |  |
| else if (OUTPUT\_BITDEPTH = = BD16) |  |
| iBias = (1 << 15) |  |
| else |  |
| iBias = 0 |  |
| if ((OUTPUT\_BITDEPTH = = BD16) | | (OUTPUT\_BITDEPTH = = BD16S) | |   (OUTPUT\_BITDEPTH = = BD32S)) |  |
| iBias = (iBias >> SHIFT\_BITS) |  |
| if ((OUTPUT\_CLR\_FMT = = RGB) | | (OUTPUT\_CLR\_FMT = = YUV444) | |  (OUTPUT\_CLR\_FMT = = YUV422) | | (OUTPUT\_CLR\_FMT = = YUV420) | |  (OUTPUT\_CLR\_FMT = = YONLY) | | (OUTPUT\_CLR\_FMT = = NCOMPONENT) | |  (OUTPUT\_CLR\_FMT = = CMYKDIRECT)) { |  |
| if ((OUTPUT\_CLR\_FMT = = RGB) | | (OUTPUT\_CLR\_FMT = = YUV444) | |  (OUTPUT\_CLR\_FMT = = YUV422) | | (OUTPUT\_CLR\_FMT = = YUV420)) |  |
| outputComponents = 3 |  |
| else |  |
| outputComponents = NumComponents |  |
| for (i = 0; i < outputComponents; i++) { |  |
| if ((i > 0) && (OUTPUT\_CLR\_FMT = = YUV420)) |  |
| outputHeight = ExtendedHeight[0] / 2 |  |
| else |  |
| outputHeight = ExtendedHeight[0] |  |
| if ((i > 0) &&  ((OUTPUT\_CLR\_FMT = = YUV422) | | (OUTPUT\_CLR\_FMT = = YUV420))) |  |
| outputWidth = ExtendedWidth[0] / 2 |  |
| else |  |
| outputWidth = ExtendedWidth[0] |  |
| for (y = 0; y < outputHeight; y++) |  |
| for (x = 0; x < outputWidth; x++) |  |
| ImagePlane[i][x][y] += (iBias << iScale) |  |
| } |  |
| } else if (OUTPUT\_CLR\_FMT = = CMYK) { |  |
| for (i = 0; i < 3; i++) |  |
| for (y = 0; y < ExtendedHeight[0]; y++) |  |
| for (x = 0; x < ExtendedWidth[0]; x++) |  |
| ImagePlane[i][x][y] += ((iBias >> 1) << iScale) /\* c, m, y \*/ |  |
| for (y = 0; y < ExtendedHeight[0]; y++) |  |
| for (x = 0; x < ExtendedWidth[0]; x++) |  |
| ImagePlane[3][x][y] −= ((iBias >> 1) << iScale) /\* k \*/ |  |
| } |  |
| } |  |

### ComputeScaling( )

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

Table 189 – Pseudocode for function ComputeScaling( )

|  |  |
| --- | --- |
| **ComputeScaling( )** { | **Reference** |
| 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)) |  |
| iRoundingFactor = 3 |  |
| else if ((OUTPUT\_BITDEPTH = = BD1WHITE1) | |   (OUTPUT\_BITDEPTH = = BD1BLACK1) | | (OUTPUT\_BITDEPTH = = BD16)) |  |
| iRoundingFactor = 4 |  |
| } |  |
| 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] |  |
| 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) |  |
| } |  |
| } |  |

### Postscaling process

#### Overview

The function PostscalingProcess( ) is specified in Table 190.

Table 190 – Pseudocode for function PostscalingProcess( )

|  |  |
| --- | --- |
| **PostscalingProcess( ) {** | **Reference** |
| if (OUTPUT\_CLR\_FMT = = RGBE) |  |
| for (y = 0; y < ExtendedHeight[0]; y++) |  |
| for (x = 0; x < ExtendedWidth[0]; x++) { |  |
| for (k = 0; k < 3; k++) |  |
| localArrayIn[k] = ImagePlane[k][x][y] |  |
| PostScalingF2(localArrayOut[ ], localArrayIn[ ]) /\* Produces 4 outputs for 3 inputs \*/ | 9.10.7.4 |
| for (k = 0; k < 4; k++) |  |
| ImagePlane[k][x][y] = localArrayOut[k] |  |
| } |  |
| else { |  |
| if ((OUTPUT\_CLR\_FMT = = RGB) | | (OUTPUT\_CLR\_FMT = = YUV444) | |  (OUTPUT\_CLR\_FMT = = YUV422) | | (OUTPUT\_CLR\_FMT = = YUV420)) |  |
| outputComponents = 3 |  |
| else |  |
| outputComponents = NumComponents |  |
| for (i = 0; i < outputComponents; i++) { |  |
| if ((i > 0) && (OUTPUT\_CLR\_FMT = = YUV420)) |  |
| outputHeight = ExtendedHeight[0] / 2 |  |
| else |  |
| outputHeight = ExtendedHeight[0] |  |
| if ((i > 0) &&  ((OUTPUT\_CLR\_FMT = = YUV422) | | (OUTPUT\_CLR\_FMT = = YUV420))) |  |
| outputWidth = ExtendedWidth[0] / 2 |  |
| else |  |
| outputWidth = ExtendedWidth[0] |  |
| if ((OUTPUT\_BITDEPTH = = BD16) | | (OUTPUT\_BITDEPTH = = BD16S) | |  (OUTPUT\_BITDEPTH = = BD32S)) |  |
| for (y = 0; y < outputHeight; y++) |  |
| for (x = 0; x < outputWidth; x++) |  |
| ImagePlane[i][x][y] = PostScalingInt(ImagePlane[i][x][y]) | 9.10.7.2 |
| else if ((OUTPUT\_BITDEPTH = = BD32F) | | (OUTPUT\_BITDEPTH = = BD16F)) |  |
| for (y = 0; y < outputHeight; y++) |  |
| for (x = 0; x < outputWidth; x++) |  |
| ImagePlane[i][x][y] = PostScalingFl(ImagePlane[i][x][y]) | 9.10.7.3 |
| } |  |
| } |  |
| } |  |

#### 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.

#### PostScalingFl( )

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

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

Table 192 – Pseudocode for function PostScalingFl( )

|  |  |
| --- | --- |
| **PostScalingFl(iX) {** | **Reference** |
| if (iX < 0) |  |
| iS = 1 |  |
| else |  |
| iS = 0 |  |
| if (OUTPUT\_BITDEPTH = = BD16F) { |  |
| iEM = Min(Abs(iX), 32767) |  |
| fV = ((iS << 15) | iEM) /\* Concatenate these fields\*/ |  |
| } else { /\* OUTPUT\_BITDEPTH = = BD32F \*/ |  |
| iX = Abs(iX) |  |
| iE = (iX >> LEN\_MANTISSA) |  |
| iM = ((iX & ((1 << LEN\_MANTISSA) − 1)) | (1 << LEN\_MANTISSA)) |  |
| if (iE = = 0) { |  |
| iM ^= (1 << LEN\_MANTISSA) |  |
| iE = 1 |  |
| } |  |
| iE = iE − EXP\_BIAS + 127 |  |
| while ((iM < (1 << LEN\_MANTISSA)) && (iE > 1) && (iM > 0)) { |  |
| iE −= 1 |  |
| iM <<= 1 |  |
| } |  |
| if (iM < (1 << LEN\_MANTISSA)) |  |
| iE = 0 |  |
| else |  |
| iM ^= (1 << LEN\_MANTISSA) |  |
| iM <<= (23 − LEN\_MANTISSA) |  |
| fV = ((iS << 31) | (iE << 23) | iM) /\* Concatenate these fields \*/ |  |
| } |  |
| return fV |  |
| } |  |

#### PostScalingF2( )

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

Table 193 – Pseudocode for function PostScalingF2( )

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

### Clipping and packing stage

#### General

The ClippingAndPackingStage( ) process by which clipping, packing, and windowing are performed is specified in Table 194. The clipping ensures that the sample values are constrained to the appropriate range. The packing process packs multiple samples into single variables for some values of OUTPUT\_BITDEPTH. The windowing process uses the LEFT\_MARGIN, TOP\_MARGIN, WIDTH\_MINUS1 and HEIGHT\_MINUS1 syntax elements to discard the data outside of the image area that is to be output.

Table 194 – Pseudocode for function ClippingAndPackingStage( )

| **ClippingAndPackingStage( ) {** | **Reference** |
| --- | --- |
| if((OUTPUT\_CLR\_FMT = = RGB) &&  ((OUTPUT\_BITDEPTH = = BD5) | | (OUTPUT\_BITDEPTH = = BD565) | |  (OUTPUT\_BITDEPTH = = BD10))) /\* Packed RGB \*/ |  |
| outputArrays = 1 |  |
| else if ((OUTPUT\_CLR\_FMT = = RGB) | | (OUTPUT\_CLR\_FMT = = YUV444) | |  (OUTPUT\_CLR\_FMT = = YUV422) | | (OUTPUT\_CLR\_FMT = = YUV420)) |  |
| outputArrays = 3 |  |
| else if (OUTPUT\_CLR\_FMT = = RGBE) |  |
| outputArrays = 4 |  |
| else |  |
| outputArrays = NumComponents |  |
| for (i = 0; i < outputArrays; i++) { |  |
| if ((i > 0) && (OUTPUT\_CLR\_FMT = = YUV420)) { |  |
| outputHeight = (HEIGHT\_MINUS1 + 1) / 2 |  |
| n = TOP\_MARGIN / 2 |  |
| } else { |  |
| outputHeight = HEIGHT\_MINUS1 + 1 |  |
| n = TOP\_MARGIN |  |
| } |  |
| if ((i > 0) &&  ((OUTPUT\_CLR\_FMT = = YUV422) | | (OUTPUT\_CLR\_FMT = = YUV420))) { |  |
| outputWidth = (WIDTH\_MINUS1 + 1) / 2 |  |
| m = LEFT\_MARGIN / 2 |  |
| } else { |  |
| outputWidth = WIDTH\_MINUS1 + 1 |  |
| m = LEFT\_MARGIN |  |
| } |  |
| if ((OUTPUT\_BITDEPTH = = BD8) | | (OUTPUT\_BITDEPTH = = BD16) | |   (OUTPUT\_BITDEPTH = = BD16S)) |  |
| for (y = 0; y < outputHeight; y++) |  |
| for (x = 0; x < outputWidth; x++) |  |
| ImagePlane[i][x][y] = ClippingBasic(ImagePlane[i][x + m][y + n]) | 9.10.8.2 |
| else if (OUTPUT\_BITDEPTH = = BD565) |  |
| for (y = 0; y < outputHeight; y++) |  |
| for (x = 0; x < outputWidth; x++) |  |
| ImagePlane[i][x][y] = ClipAndPackBD565(ImagePlane[0][x + m][y + n],   ImagePlane[1][x + m][y + n], ImagePlane[2][x + m][y + n]) | 9.10.8.3 |
| else if (OUTPUT\_BITDEPTH = = BD5) |  |
| for (y = 0; y < outputHeight; y++) |  |
| for (x = 0; x < outputWidth; x++) |  |
| ImagePlane[i][x][y] = ClipAndPackBD5(ImagePlane[0][x + m][y + n],   ImagePlane[1][x + m][y + n], ImagePlane[2][x + m][y + n]) | 9.10.8.4 |
| else if (OUTPUT\_BITDEPTH = = BD10) |  |
| if (OUTPUT\_CLR\_FMT = = RGB) |  |
| for (y = 0; y < outputHeight; y++) |  |
| for (x = 0; x < outputWidth; x++) |  |
| ImagePlane[i][x][y] = ClipAndPackBD10(ImagePlane[0][x + m][y + n],   ImagePlane[1][x + m][y + n], ImagePlane[2][x + m][y + n]) | 9.10.8.5 |
| else |  |
| for (y = 0; y < outputHeight; y++) |  |
| for (x = 0; x < outputWidth; x++) |  |
| ImagePlane[i][x][y] = ClipAndPackBD10(ImagePlane[i][x + m][y + n], 0, 0) | ‎9.10.8.5 |
| else if ((OUTPUT\_BITDEPTH = = BD1WHITE1) | | (OUTPUT\_BITDEPTH = = BD1BLACK1)) |  |
| for (y = 0; y < outputHeight; y++) |  |
| for (x = 0; x < outputWidth; x+=8) { /\* Up to 8 samples are packed into each output byte \*/ |  |
| pNum = Min(outputWidth − x, 8) /\* Number of values to pack into current output byte \*/ |  |
| for (p = pNum; m < 8; p++) /\* Prevent junk beyond valid image data in array \*/ |  |
| valList[p] = 0 /\* Actual value does not matter in this region \*/ |  |
| for (p = 0; p < pNum; p++) |  |
| valList[p] = ImagePlane[i][x + m + p][y + n] |  |
| ImagePlane[i][x >> 3][y] = ClipAndPackBD1BorW(valList) | 9.10.8.6 |
| } |  |
| else /\* OUTPUT\_BITDEPTH equal to BD16F, BD32F, or BD32S \*/ |  |
| for (y = 0; y < outputHeight; y++) |  |
| for (x = 0; x < outputWidth; x++) |  |
| ImagePlane[i][x][y] = ImagePlane[i][x + m][y + n] |  |
| } |  |
| } |  |

#### ClippingBasic( )

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

Table 195 – Pseudocode for function ClippingBasic( )

| **ClippingBasic(iSample)** { |
| --- |
| if (OUTPUT\_BITDEPTH = = BD8) { |
| iLow = 0 |
| iHigh = 255 |
| } else if (OUTPUT\_BITDEPTH = = BD16) { |
| iLow = 0 |
| iHigh = 65535 |
| } else if (OUTPUT\_BITDEPTH = = BD16S) { |
| iLow = −32768 |
| iHigh = 32767 |
| } |
| iResult = Clip(iSample, iLow, iHigh) /\* Clip within the range iLow to iHigh \*/ |
| return iResult |
| } |

#### ClipAndPackBD565( )

The pseudocode function ClipAndPackBD565( ) is specified in Table 196.

Table 196 – Pseudocode for function ClipAndPackBD565( )

|  |
| --- |
| **ClipAndPackBD565(i0, i1, i2)** { |
| iLow = 0 |
| iHigh = 31 |
| i0 = Clip(i0, iLow, iHigh) |
| i2 = Clip(i2, iLow, iHigh) |
| iLow = 0 |
| iHigh = 63 |
| i1 = Clip(i1, iLow, iHigh) |
| iResult = i0 + (i1 << 5) + (i2 << 11) |
| return iResult |
| } |

#### ClipAndPackBD5( )

The pseudocode function ClipAndPackBD5( ) is specified in Table 197.

Table 197 – Pseudocode for function ClipAndPackBD5( )

|  |
| --- |
| **ClipAndPackBD5(i0, i1, i2)** { |
| iLow = 0 |
| iHigh = 31 |
| i0 = Clip(i0, iLow, iHigh) |
| i1 = Clip(i1, iLow, iHigh) |
| i2 = Clip(i2, iLow, iHigh) |
| iResult = i0 + (i1 << 5) + (i2 << 10) |
| return iResult |
| } |

#### ClipAndPackBD10( )

The pseudocode function ClipAndPackBD10( ) is specified in Table 198.

Table 198 – Pseudocode for function ClipAndPackBD10( )

|  |
| --- |
| **ClipAndPackBD10(iSample0, iSample1, iSample2)** { |
| iLow = 0 |
| iHigh = 1023 |
| if (OUTPUT\_CLR\_FMT = = RGB) { |
| i0 = Clip(iSample0, iLow, iHigh) |
| i1 = Clip(iSample1, iLow, iHigh) |
| i2 = Clip(iSample2, iLow, iHigh) |
| iResult = i0 + (i1 << 10) + (i2 << 20) |
| } else |
| iResult = Clip(iSample0, iLow, iHigh) |
| return iResult |
| } |

#### ClipAndPackBD1BorW( )

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

Table 199 – Pseudocode for function ClipAndPackBD1BorW( )

|  |
| --- |
| **ClipAndPackBD1BorW(valList)** { |
| /\* valList[0] holds the value associated with the first sample value in the scan order, and   valList[7] holds the value associated with the last sample value in the scan order \*/ |
| valList[0] = Clip(valList[0], 0, 1) |
| valList[1] = Clip(valList[1], 0, 1) |
| valList[2] = Clip(valList[2], 0, 1) |
| valList[3] = Clip(valList[3], 0, 1) |
| valList[4] = Clip(valList[4], 0, 1) |
| valList[5] = Clip(valList[5], 0, 1) |
| valList[6] = Clip(valList[6], 0, 1) |
| valList[7] = Clip(valList[7], 0, 1) |
| if (OUTPUT\_BITDEPTH = = BD1BLACK1) |
| iResult = (1 − valList[7]) + ((1 − valList[6]) << 1) + ((1 − valList[5]) << 2) +   ((1 − valList[4]) << 3) + ((1 − valList[3]) << 4) + ((1 − valList[2]) << 5) +   ((1 − valList[1]) << 6) + ((1 − valList[0]) << 7) |
| else /\* OUTPUT\_BITDEPTH = = BD1WHITE1 \*/ |
| iResult = valList[7] + (valList[6] << 1) + (valList[5] << 2) +   (valList[4] << 3) + (valList[3] << 4) + (valList[2] << 5) +   (valList[1] << 6) + (valList[0] << 7) |
| return iResult |
| } |

1. Tag-based file format

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

* 1. General

This annex specifies a format for files containing JPEG XR images. It uses syntax structures (IFD\_ENTRY( ) structures as specified in A.7) that each contain a syntax element (FIELD\_TAG as specified in A.7.2) that can be referred to as a tag. Therefore, this file format is referred to as being tag-based. The value of the tag serves as an identifier of the type of data contained in the syntax structure that is associated with the tag.

NOTE 1 – The file format specified in this annex is based on that specified for use in ISO 12234-2, ISO 12639, TIFF Revision 6.0, and EXIF 2.2 (JEITA CP-3451), and is intended to provide a form of consistency and compatibility with those documents – e.g. to enable the sharing of some functional components designed for reading, writing, and otherwise making use of such files.

NOTE 2 – This specification of this file format does not preclude the existence of alternative file format specifications for files containing JPEG XR images.

NOTE 3 – When a file is formatted as specified in this annex, in addition to the syntax structures that are specified by this annex, arbitrary data (formatted in a manner not specified by this annex) may also be present at locations within the file that lie between or beyond the locations in the file that contain the syntax structures specified by this annex.

NOTE 4 – The use of the filename extension ".jxr" is suggested for files conforming to the file format specified in this annex.

The FILE\_HEADER( ) syntax structure specified in clause A.5 shall be present at the beginning of the file (at byte position 0).

The variable FileSizeInBytes is considered to be equal to the total number of bytes in the file. The method of determining the value of FileSizeInBytes is determined by the application and is not specified in this Specification. The value of FileSizeInBytes shall not exceed 232 − 1.

For purposes of this Specification, a decoder is assumed to be capable of either storing the entire file in random access memory or performing random access seek operations to access the data at arbitrary specified positions in the file.

* 1. Normative references

The following Recommendations and International Standards contain provisions which, through reference in this text, constitute provisions of this annex to this Recommendation | International Standard. At the time of publication, the editions indicated were valid. All Recommendations and Standards are subject to revision, and parties to agreements based on this Recommendation | International Standard are encouraged to investigate the possibility of applying the most recent edition of the Recommendations and Standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards. The Telecommunication Standardization Bureau of the ITU maintains a list of currently valid ITU-T Recommendations.

* + 1. Identical Recommendations | International Standards

None.

* + 1. Paired Recommendations | International Standards equivalent in technical content

None.

* + 1. Additional references
* ISO/IEC 10646:2014, *Information technology – Universal Coded Character Set (UCS)*.
* ISO/IEC/IEEE 60559 (2011), *Information technology – Microprocessor Systems – Floating-Point arithmetic.*
  1. Definitions

For the purposes of this annex to this Recommendation | International Standard, the following definitions apply. In this clause, *italic* is used to identify all occurrences of terms that are defined in this subclause or in clause 3.

1. **interleaved alpha image plane**:  *Images* with the value of ALPHA\_IMAGE\_PLANE\_FLAG set equal to 1 have an *interleaved alpha image plane*.
2. **separate alpha image plane**:  *Images* with optional *alpha image plane* and the value of ALPHA\_IMAGE\_PLANE\_FLAG set equal to zero have a *separate alpha image plane*. For such *images*, data relating to the *alpha image plane* is present in the CODED\_IMAGE( ) syntax structure specified by the ALPHA\_OFFSET syntax element.
3. **universal multiple-octet coded character set transformation format 8 (UTF-8)**:  The 8-bit character set encoding specified in Annex D of ISO/IEC 10646.
   1. Abbreviations

For the purposes of this annex to this Recommendation | International Standard, the following abbreviations apply.

CIE Commission Internationale de l´Eclairage (International Commission on Illumination)

FCC Federal Communications Commission

ICC International Color Consortium

NTSC National Television System Committee

RP Recommended Practice

SMPTE Society of Motion Picture and Television Engineers

UTF Universal multiple-octet coded character set Transformation Format (as specified by ISO/IEC 10646)

UUID Universal Unique Identifier (as specified in Annex A of ISO/IEC 11578)

* 1. FILE\_HEADER( )
     1. Syntax structure

The FILE\_HEADER( ) syntax structure is specified in Table A.1.

Table A.1 – FILE\_HEADER( ) syntax structure

|  |  |  |
| --- | --- | --- |
| **FILE\_HEADER( ) {** | **Descriptor** | **Reference** |
| FIXED\_FILE\_HEADER\_II\_2BYTES | u(16) | A.5.2 |
| FIXED\_FILE\_HEADER\_0XBC\_BYTE | u(8) | A.5.3 |
| FILE\_VERSION\_ID | u(8) | A.5.4 |
| FIRST\_IFD\_OFFSET | le(32) | A.5.5 |
| } |  |  |

* + 1. FIXED\_FILE\_HEADER\_II\_2BYTES

FIXED\_FILE\_HEADER\_II\_2BYTES shall be equal to 0x4949.

* + 1. FIXED\_FILE\_HEADER\_0XBC\_BYTE

FIXED\_FILE\_HEADER\_0XBC\_BYTE shall be equal to 0xBC.

* + 1. FILE\_VERSION\_ID

FILE\_VERSION\_ID shall be equal to 1. Other values of FILE\_VERSION\_ID are reserved for future use, as modified in additional parts or amendments, by ITU-T | ISO/IEC.

* + 1. FIRST\_IFD\_OFFSET

FIRST\_IFD\_OFFSET specifies the byte position, relative to the beginning of the file, of the first IMAGE\_FILE\_DIRECTORY( ) syntax structure (A.6) in the file. The value of FIRST\_IFD\_OFFSET shall be an integer multiple of 2.

* 1. IMAGE\_FILE\_DIRECTORY( )
     1. Syntax structure

The IMAGE\_FILE\_DIRECTORY( ) syntax structure is specified in Table A.2.

Table A.2 – IMAGE\_FILE\_DIRECTORY( ) syntax structure

|  |  |  |
| --- | --- | --- |
| **IMAGE\_FILE\_DIRECTORY( ) {** | **Descriptor** | **Reference** |
| NUM\_ENTRIES | le(16) | A.6.2 |
| for (iNumEntries = 0;  iNumEntries < NUM\_ENTRIES;  iNumEntries++) |  |  |
| IFD\_ENTRY( ) |  | A.7 |
| ZERO\_OR\_NEXT\_IFD\_OFFSET | le(32) | A.6.3 |
| } |  |  |

* + 1. NUM\_ENTRIES

NUM\_ENTRIES specifies the number of entries in the IMAGE\_FILE\_DIRECTORY( ) syntax structure. NUM\_ENTRIES shall not be equal to 0. The value 0 for NUM\_ENTRIES is reserved for future use by ITU-T | ISO/IEC.

* + 1. ZERO\_OR\_NEXT\_IFD\_OFFSET

ZERO\_OR\_NEXT\_IFD\_OFFSET is interpreted as follows.

* If ZERO\_OR\_NEXT\_IFD\_OFFSET is equal to 0, this indicates that no additional IMAGE\_FILE\_DIRECTORY( ) syntax structures are present in the file.
* Otherwise, ZERO\_OR\_NEXT\_IFD\_OFFSET specifies the byte position, relative to the beginning of the file, to the next IMAGE\_FILE\_DIRECTORY( ) syntax structure in the file.

The value of ZERO\_OR\_NEXT\_IFD\_OFFSET shall be an integer multiple of 2.

Decoders may ignore any IMAGE\_FILE\_DIRECTORY( ) syntax structures at locations in the file specified by any ZERO\_OR\_NEXT\_IFD\_OFFSET syntax element.

* 1. IFD\_ENTRY( )
     1. Syntax structure

The IFD\_ENTRY( ) syntax structure is specified in Table A.3.

Table A.3 – IFD\_ENTRY( ) syntax structure

|  |  |  |
| --- | --- | --- |
| **IFD\_ENTRY( ) {** | **Descriptor** | **Reference** |
| FIELD\_TAG | le(16) | A.7.2 |
| ELEMENT\_TYPE | le(16) | A.7.3 |
| NUM\_ELEMENTS | le(32) | A.7.4 |
| VALUES\_OR\_OFFSET | le(32) | A.7.5 |
| } |  |  |

The interpretation and presence of syntax elements of the IFD\_ENTRY( ) syntax structure is specified in Table A.4. The data associated with a FIELD\_TAG value is interpreted as the value of the syntax element or syntax structure in the "Syntax element or syntax structure" column. The term "variable" is used in Table A.4 to indicate cases in which NUM\_ELEMENTS may have any value corresponding to the quantity of associated data. The "Presence" column is interpreted as follows.

* "Required" specifies that the FIELD\_TAG value shall be present in an IFD\_ENTRY( ) syntax structure of each IMAGE\_FILE\_DIRECTORY( ) syntax structure in the file.
* "Optional" indicates that the FIELD\_TAG value may or may not be present in an IFD\_ENTRY( ) syntax structure of each IMAGE\_FILE\_DIRECTORY( ) syntax structure in the file.

IFD entries with combinations of FIELD\_TAG, ELEMENT\_TYPE, and NUM\_ELEMENTS that do not appear in Table A.4, with the FIELD\_TAG value in the range of 0x1000 to 0x3FFF and 0x5000 to 0x7FFF are reserved for future use by ITU-T | ISO/IEC. Decoders shall ignore (parse and discard) any IFD\_ENTRY( ) syntax structures in which such combinations appear.

NOTE 1 – The purpose of the specification for decoders to ignore IFD entries with such combinations of FIELD\_TAG, ELEMENT\_TYPE, and NUM\_ELEMENTS is to enable the future definition of a backward-compatible usage of additional combinations.

IFD entries with combinations of FIELD\_TAG, ELEMENT\_TYPE and NUM\_ELEMENTS that do not appear in Table A.4, with the FIELD\_TAG value in the range of 0x0000 to 0x0FFF, 0x4000 to 0x4FFF, and 0x8000 to 0xFFFF are available for unspecified use and interpretation as determined by the application. Decoders shall parse any IFD\_ENTRY( ) syntax structures in which such combinations appear and, for purposes relevant to determining conformance to this Specification, shall ignore these syntax structures. Any use of such FIELD\_TAG values shall not affect the expressed requirements for conformance to this Specification. Additionally, ITU-T and ISO/IEC reserve the ability to potentially specify uses for such FIELD\_TAG values in future revisions of this Specification.

NOTE 2 – Since interpretation of such FIELD\_TAG values may be application specific, it is recommended that any such IFD\_ENTRY( ) syntax structures that have unknown interpretations be removed when transferring files between differing application domains.

NOTE 3 – The use of a field tag value equal to 0x02BC in tag-based encoded files [such as files formatted according to ISO 12234-2, ISO 12639, TIFF Revision 6.0 or EXIF 2.2 (JEITA CP-3451)], is specified in ISO 16684-1. The use of a field tag value equal to 0x8769 is specified in section 4.6.3 of the EXIF 2.2 (JEITA CP-3451) specification. The use of a field tag value equal to 0x8773 is specified in clause B.3 of the ICC.1;2001-04 (ICC version 2.4.0.0) specification and in clause B.3 of ISO 15076-1:2010, which specify two versions of ICC profile data. The provision to allow these field tag values to be present is intended to allow the use of ISO 16684-1, EXIF 2.2 (JEITA CP-3451), ICC.1:2001-04 and ISO 15076-1:2010 specifications with files encoded according to this Specification (without imposing normative conformance requirements related to such use). The use of ICC profile data is further discussed in Annex C.

Table A.4 – Interpretation, allowed combinations, and presence of  
syntax elements of the IFD\_ENTRY( ) syntax structure

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **FIELD\_TAG** | **ELEMENT\_TYPE** | **NUM\_ELEMENTS** | **Syntax element or syntax structure** | **Presence** |
| 0x010D | UTF8 | variable | DOCUMENT\_NAME | Optional |
| 0x010E | UTF8 | variable | IMAGE\_DESCRIPTION | Optional |
| 0x010F | UTF8 | variable | EQUIPMENT\_MAKE | Optional |
| 0x0110 | UTF8 | variable | EQUIPMENT\_MODEL | Optional |
| 0x011D | UTF8 | variable | PAGE\_NAME | Optional |
| 0x0129 | USHORT | 2 | PAGE\_NUMBER | Optional |
| 0x0131 | UTF8 | variable | SOFTWARE\_NAME\_VERSION | Optional |
| 0x0132 | UTF8 | 20 | DATE\_TIME | Optional |
| 0x013B | UTF8 | variable | ARTIST\_NAME | Optional |
| 0x013C | UTF8 | variable | HOST\_COMPUTER | Optional |
| 0x8298 | UTF8 | variable | COPYRIGHT\_NOTICE | Optional |
| 0xA001 | USHORT | 1 | COLOR\_SPACE | Optional |
| 0xBC01 | BYTE | 16 | PIXEL\_FORMAT | Required |
| 0xBC02 | BYTE, USHORT, or ULONG | 1 | SPATIAL\_XFRM\_PRIMARY | Optional |
| 0xBC04 | ULONG | 1 | IMAGE\_TYPE | Optional |
| 0xBC05 | BYTE | 4 | PTM\_COLOR\_INFO( ) | Optional |
| 0xBC06 | BYTE | variable | PROFILE\_LEVEL\_CONTAINER( ) | Optional |
| 0xBC80 | BYTE, USHORT, or ULONG | 1 | IMAGE\_WIDTH | Required |
| 0xBC81 | BYTE, USHORT, or ULONG | 1 | IMAGE\_HEIGHT | Required |
| 0xBC82 | FLOAT | 1 | WIDTH\_RESOLUTION | Optional |
| 0xBC83 | FLOAT | 1 | HEIGHT\_RESOLUTION | Optional |
| 0xBCC0 | BYTE, USHORT, or ULONG | 1 | IMAGE\_OFFSET | Required |
| 0xBCC1 | BYTE, USHORT, or ULONG | 1 | IMAGE\_BYTE\_COUNT | Required |
| 0xBCC2 | BYTE, USHORT, or ULONG | 1 | ALPHA\_OFFSET | Optional |
| 0xBCC3 | BYTE, USHORT, or ULONG | 1 | ALPHA\_BYTE\_COUNT | Optional |
| 0xBCC4 | BYTE | 1 | IMAGE\_BAND\_PRESENCE | Optional |
| 0xBCC5 | BYTE | 1 | ALPHA\_BAND\_PRESENCE | Optional |
| 0xEA1C | UNDEFINED | variable | PADDING\_DATA | Optional |

* + 1. FIELD\_TAG

FIELD\_TAG identifies the data contained in the IFD\_ENTRY( ) syntax structure. When the IFD\_ENTRY( ) syntax structure is not the first IFD\_ENTRY( ) syntax structure of the IMAGE\_FILE\_DIRECTORY( ) syntax structure, the value of FIELD\_TAG shall be greater than the value of FIELD\_TAG in the preceding IFD\_ENTRY( ) syntax structure of the IMAGE\_FILE\_DIRECTORY( ) syntax structure.

* + 1. ELEMENT\_TYPE

ELEMENT\_TYPE identifies the type of data contained in the IFD\_ENTRY( ) syntax structure as specified in Table A.5.

Table A.5 – ELEMENT\_TYPE

|  |  |  |
| --- | --- | --- |
| **ELEMENT\_TYPE** | **Mnemonic** | **SizeOfElement** |
| **0** | RESERVED | Not specified |
| **1** | BYTE | 1 |
| **2** | UTF-8 | 1 |
| **3** | USHORT | 2 |
| **4** | ULONG | 4 |
| **5** | URATIONAL | 8 |
| **6** | SBYTE | 1 |
| **7** | UNDEFINED | 1 |
| **8** | SSHORT | 2 |
| **9** | SLONG | 4 |
| **10** | SRATIONAL | 8 |
| **11** | FLOAT | 4 |
| **12** | DOUBLE | 8 |
| **13-65535** | RESERVED | Not specified |

The value of ELEMENT\_TYPE shall not be equal to RESERVED in files conforming to this version of this annex. Such values are reserved for use in future versions of this Specification. Decoders that encounter files containing ELEMENT\_TYPE equal to RESERVED shall ignore the associated data.

When ELEMENT\_TYPE is equal to UNDEFINED, the value of FIELD\_TAG shall be equal to 0x8773 if it is not equal to 0xEA1C in files conforming to this version of this annex. Other values of ELEMENT\_TYPE are reserved for use in future versions of this Specification.

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 Annex D of ISO/IEC 10646, 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 clause A.7.33.
* Otherwise, the interpretation of the data elements is not specified by this Specification.
* Otherwise, if ELEMENT\_TYPE is equal to URATIONAL, each data element is interpreted as a rational number having a numerator equal to the first 4 bytes of the data element value, interpreted as an unsigned integer of the specified length in little-endian form, and a denominator equal to the remaining 4 bytes of the data element value, interpreted as an unsigned integer of the specified length in little-endian form.
* Otherwise, if ELEMENT\_TYPE is equal to SRATIONAL, each data element is interpreted as a rational number having a numerator equal to the first 4 bytes of the data element value, interpreted as a two's complement signed integer of the specified length in little-endian form, and a denominator equal to the remaining 4 bytes of the data element value, interpreted as a two's complement signed integer of the specified length in little-endian form.
* Otherwise, if ELEMENT\_TYPE is equal to FLOAT, each data element is interpreted as a 4-byte ISO/IEC/IEEE 60559 floating-point number in little-endian form.
* Otherwise (ELEMENT\_TYPE is equal to DOUBLE), each data element is interpreted as an 8-byte ISO/IEC/IEEE 60559 floating-point number in little-endian form.
  + 1. NUM\_ELEMENTS

NUM\_ELEMENTS specifies the number of data elements associated with the IFD\_ENTRY( ) syntax structure. Let iSizeOfElement be equal to the value in the SizeOfElement column of Table A.5 for the value of ELEMENT\_TYPE. The total number of bytes of data associated with the IFD\_ENTRY( ) syntax structure is NUM\_ELEMENTS \* iSizeOfElement.

NOTE – NUM\_ELEMENTS must be set to the total quantity of data elements, including termination indicator values such as null-valued terminators as applicable.

* + 1. VALUES\_OR\_OFFSET

VALUES\_OR\_OFFSET is interpreted as follows:

* Let iSizeOfElement be equal to the value in the SizeOfElement column of Table A.5 for the value of ELEMENT\_TYPE. If NUM\_ELEMENTS \* iSizeOfElement is less than or equal to 4, VALUES\_OR\_OFFSET contains the data elements associated with the IFD\_ENTRY( ) syntax structure.
* Otherwise, VALUES\_OR\_OFFSET specifies the byte position, relative to the beginning of the file, of the data elements associated with the IFD\_ENTRY( ) syntax structure. In this case, the value of the VALUES\_OR\_OFFSET syntax element shall be an integer multiple of 2.
  + 1. DOCUMENT\_NAME

DOCUMENT\_NAME (when present) provides, as a UTF-8 character string, a name for the image. There are no specific conformance requirements for the content of the associated character string (other than that it follows the format conventions of the ELEMENT\_TYPE definition of the UTF-8 data type as specified in A.7.3).

NOTE 1 – The name of this syntax element alludes to the intended interpretation instead of any normative meaning.

NOTE 2 – DOCUMENT\_NAME is specified here in a manner that is intended to be consistent with the "DocumentName" tag specified in ISO 12639 and TIFF Revision 6.0.

* + 1. IMAGE\_DESCRIPTION

IMAGE\_DESCRIPTION (when present) provides, as a UTF-8 character string, a title or description of the image subject matter. There are no specific conformance requirements for the content of the associated character string (other than that it follows the format conventions of the ELEMENT\_TYPE definition of the UTF8 data type as specified in A.7.3).

NOTE 1 – The name of this syntax element alludes to the intended interpretation instead of any normative meaning.

NOTE 2 – IMAGE\_DESCRIPTION is specified here in a manner that is intended to be consistent with the "ImageDescription" tag specified in ISO 12234-2, ISO 12639, TIFF Revision 6.0, and EXIF 2.2 (JEITA CP-3451).

* + 1. EQUIPMENT\_MAKE

EQUIPMENT\_MAKE (when present) provides, as a UTF-8 character string, a name for the manufacturer or vendor of the camera, image-capturing device, scanner or other equipment that generated the image. There are no specific conformance requirements for the content of the associated character string (other than that it follows the format conventions of the ELEMENT\_TYPE definition of the UTF8 data type as specified in A.7.3).

NOTE 1 – The name of this syntax element alludes to the intended interpretation instead of any normative meaning.

NOTE 2 – EQUIPMENT\_MAKE is specified here in a manner that is intended to be consistent with the "Make" tag specified in ISO 12234-2, ISO 12639, TIFF Revision 6.0, and EXIF 2.2 (JEITA CP-3451).

* + 1. EQUIPMENT\_MODEL

When present, EQUIPMENT\_MODEL provides, as a UTF-8 character string, a model name or model number of the camera, image-capturing device, scanner or other equipment that generated the image. There are no specific conformance requirements for the content of the associated character string (other than that it follows the format conventions of the ELEMENT\_TYPE definition of the UTF8 data type as specified in A.7.3).

NOTE 1 – The name of this syntax element alludes to the intended interpretation instead of any normative meaning.

NOTE 2 – EQUIPMENT\_MODEL is specified here in a manner that is intended to be consistent with the "Model" tag specified in ISO 12234-2, ISO 12639, TIFF Revision 6.0, and EXIF 2.2 (JEITA CP-3451).

* + 1. PAGE\_NAME

PAGE\_NAME (when present) provides, as a UTF-8 character string, a name to be considered to be the name of a page to which the image is considered to belong. There are no specific conformance requirements for the content of the associated character string (other than that it follows the format conventions of the ELEMENT\_TYPE definition of the UTF8 data type as specified in A.7.3).

NOTE 1 – The name of this syntax element alludes to the intended interpretation instead of any normative meaning.

NOTE 2 – PAGE\_NAME is specified here in a manner that is intended to be consistent with the "PageName" tag specified in ISO 12639 and TIFF Revision 6.0.

* + 1. PAGE\_NUMBER

PAGE\_NUMBER (when present) provides, as a pair of unsigned integers, a "page number" (the first number) to be considered the page number of a page from a sequence of pages (the second number) to which the image is considered to belong. There are no specific conformance requirements for the content of the associated character string (other than that it follows the format conventions of the ELEMENT\_TYPE definition of the USHORT data type as specified in A.7.3).

NOTE 1 – The name of this syntax element alludes to the intended interpretation instead of any normative meaning.

NOTE 2 – PAGE\_NUMBER is specified here in a manner that is intended to be consistent with the "PageNumber" tag specified in TIFF Revision 6.0.

* + 1. SOFTWARE\_NAME\_VERSION

SOFTWARE\_NAME\_VERSION (when present) provides, as a UTF-8 character string, name and version number information for one or more software packages used in generating the image or file. There are no specific conformance requirements for the content of the associated character string (other than that it follows the format conventions of the ELEMENT\_TYPE definition of the UTF8 data type as specified in A.7.3).

NOTE 1 – The name of this syntax element alludes to the intended interpretation instead of any normative meaning.

NOTE 2 – SOFTWARE\_NAME\_VERSION is specified here in a manner that is intended to be consistent with the "Software" tag specified in ISO 12234-2, ISO 12639, TIFF Revision 6.0, and EXIF 2.2 (JEITA CP-3451).

* + 1. DATE\_TIME

DATE\_TIME (when present) provides, as a UTF-8 character string, the date and time of the creation or last modification of the image.

The value of NUM\_ELEMENTS for the DATE\_TIME tag shall be equal to 20.

The interpretation of the UTF-8 character string associated with the DATE\_TIME tag shall be as follows.

* If the UTF-8 character string associated with the DATE\_TIME tag is the string "0000:00:00 00:00:00" or this string with all "0" characters replaced by space (0x20) characters as in the string "    :  :     :  :  ", followed by a byte equal to 0, this indicates that the date and time are unknown.
* Otherwise, the UTF-8 character string associated with the DATE\_TIME tag shall be formatted as "YYYY:MM:DD HH:MM:SS", followed by a byte equal to 0, where the formatting of this string shall be as follows.
* The "YYYY" part of the string shall contain a character string representation of a four-digit decimal number for a calendar year.
* Each colon character ":" part of the string shall be a colon ":" character (a byte equal to 0x3A).
* The "MM" part of the string shall contain a character string representation of a two-digit decimal number in the range from "01" to "12" for a month within the year (where "01" indicates January, "02" indicates February, etc.) or two space (0x20) characters.
* The "DD" part of the string shall contain a character string representation of a two-digit decimal number in the range from "01" to "31" for a day within the month (where "01" indicates the first day of the month, "02" indicates the second day of the month, etc.).
* The character separating the "DD" and "HH" parts of the string shall be a space character " " (a byte equal to 0x20).
* The "HH" part of the string shall contain a character string representation of a two-digit decimal number in the range from "00" to "23" for an hour within the day, following a 24-hour time of day convention (where "00" indicates a time in the first hour of the day, "01" indicates a time in the second hour of the day, etc.).

NOTE 1 – The time 00:00:00 is used to indicate midnight. Similarly, 00:00:01 indicates one second after midnight and 23:59:59 indicates one second prior to midnight.

* The "MM" part of the string shall contain a character string representation of a two-digit decimal number in the range from "00" to "59" for a minute within the hour (where "00" indicates a time in the first minute of the hour, "01" indicates a time within the second minute of the hour, etc.).
* The "SS" part of the string shall contain a character string representation of a two-digit decimal number in the range from "00" to "59" for a second within the minute (where "00" indicates a time in the first second of the minute, "01" indicates a time within the second second of the minute, etc.).

The time zone for the date and time are not specified in this version of this Specification.

NOTE 2 – For example, the date and time may be referenced to the local time zone in which the encoding equipment is operated or is ordinarily kept.

There are no additional specific conformance requirements for the content of the associated character string.

NOTE 3 – The name of this syntax element alludes to the intended interpretation instead of any normative meaning.

NOTE 4 – DATE\_TIME is specified here in a manner that is intended to be consistent with the "DateTime" tag specified in ISO 12234-2, ISO 12639, TIFF Revision 6.0, and EXIF 2.2 (JEITA CP-3451).

* + 1. ARTIST\_NAME

ARTIST\_NAME (when present) provides, as a UTF-8 character string, one or more names and possibly additional information relating to persons involved in the creation of the image. There are no specific conformance requirements for the content of the associated character string (other than that it follows the format conventions of the ELEMENT\_TYPE definition of the UTF8 data type as specified in A.7.3).

NOTE 1 – The name of this syntax element alludes to the intended interpretation instead of any normative meaning.

NOTE 2 – ARTIST\_NAME is specified here in a manner that is intended to be consistent with the "Artist" tag specified in ISO 12234-2, ISO 12639, TIFF Revision 6.0, and EXIF 2.2 (JEITA CP-3451). An example preferred use of ARTIST\_NAME is the string value "Camera owner, John Smith; Photographer, Michael Brown; Image creator, Ken James" (followed by a byte equal to 0).

* + 1. HOST\_COMPUTER

HOST\_COMPUTER (when present) provides, as a UTF-8 character string, an indication of a computer, operating system or other system used in generating the image or file. There are no specific conformance requirements for the content of the associated character string (other than that it follows the format conventions of the ELEMENT\_TYPE definition of the UTF8 data type as specified in A.7.3).

NOTE 1 – The name of this syntax element alludes to the intended interpretation instead of any normative meaning.

NOTE 2 – HOST\_COMPUTER is specified here in a manner that is intended to be consistent with the "HostComputer" tag specified in ISO 12639 and TIFF Revision 6.0.

* + 1. COPYRIGHT\_NOTICE

COPYRIGHT\_NOTICE (when present) provides, as a UTF-8 character string, an indication of copyright rights associated with the image or file. There are no specific conformance requirements for the content of the associated character string (other than that it follows the format conventions of the ELEMENT\_TYPE definition of the UTF8 data type as specified in A.7.3).

NOTE 1 – The name of this syntax element alludes to the intended interpretation instead of any normative meaning.

NOTE 2 – COPYRIGHT\_NOTICE is specified here in a manner that is intended to be consistent with the "Copyright" tag specified in ISO 12234-2, ISO 12639, TIFF Revision 6.0, and EXIF 2.2 (JEITA CP-3451). A complete copyright statement should be listed in the character string including any dates and statements of claims – e.g., "Copyright, John Smith, 2009. All rights reserved" (followed by a byte equal to 0). When desired, this character string can also list a royalty clearing house. The string may be used to indicate either or both photographer and editor copyrights as well as any additional or alternative copyrights that apply to the image or file. When a photographer statement is included, it should appear first. When no photographer copyright statement is included and an editor or additional or alternative copyright statement is included, the absence of a photographer copyright statement should be indicated by a space character (0x20) followed by a byte equal to 0 (null) at the beginning of the character string. When an editor copyright statement is included, it should appear next. When no editor copyright statement is included and some additional or alternative copyright statement is included, the absence of an editor copyright statement should be indicated by a space character (0x20) followed by a byte equal to 0 (null) at the position in the character string at which the editor copyright statement would otherwise appear. When some additional or alternative copyright statement is included, it should appear next. When present, each additional or alternative copyright statement should end with a byte equal to 0 (null). When a royalty clearing house listing is included, this listing should appear last.

* + 1. COLOR\_SPACE

COLOR\_SPACE (when present) equal to 1, when the value in the Num column in Table A.6 for the line corresponding to the value of PIXEL\_FORMAT is equal to UINT, indicates that the associated image should be interpreted according to the sRGB specification IEC 61966-2-1. COLOR\_SPACE equal to 0xFFFF, when used with an unsigned pixel format as specified by PIXEL\_FORMAT, indicates that the sRGB specification may not be the preferred method of interpretation of the associated image.

COLOR\_SPACE shall not be equal to 1 when PIXEL\_FORMAT specifies a fixed or floating point pixel format.

When present, COLOR\_SPACE shall be equal to 1 or 0xFFFF. All other values are reserved for future use by ITU-T | ISO/IEC. All values other than 1 and 0xFFFF shall be treated the same as the value 0xFFFF by decoders.

NOTE – sRGB image data is used in many systems, being interpreted directly by various receiving devices.

* + 1. PIXEL\_FORMAT

PIXEL\_FORMAT consists of a string of 16 BYTE values. The values of the bytes ordered from first to last in this string are interpreted as digit pairs appearing from left to right in the hexadecimal integer values specified in Table A.6.

NOTE 1 – The PIXEL\_FORMAT values specified in Table A.6 originated as UUID values as specified by ISO/IEC 11578. UUID values are commonly shown as a string of hexadecimal digits separated by dash "-" characters. The bytes of such a UUID value are typically stored in a file using a rearranged form of such a typical dashed-form description. In Table A.6, that convention is not followed. Instead, the values are specified in a manner intended to ease the interpretation of the value as a string of 16 BYTE values that can be directly compared to the provided strings of hexadecimal digits. The rearrangement of the common form of UUID specification to produce the equivalent ordered string of BYTE values as shown in Table A.6 can be performed as follows.

* The first (leftmost) 4 bytes (which precede the first dash character in the common form of UUID value specification) are ordered first, represented in little-endian (byte-reversed) form.
* The next two bytes (which precede the second dash character in the common form of UUID value specification) are ordered next, represented in little-endian (byte-reversed) form.
* The next two bytes (which precede the third dash character in the common form of UUID value specification) are ordered next, represented in little-endian (byte-reversed) form.
* The remaining bytes in the common form of UUID value specification follow, in left-to-right order.

The headings of Table A.6 are interpreted as follows:

* "Mnemonic name" indicates a string associated with the syntax element corresponding to the value of PIXEL\_FORMAT.
* "NC" specifies the following.
* If the value of PIXEL\_FORMAT specifies that no alpha channel is present (as specified by the entry in the "Alpha" column), it is a requirement for file format conformance to this annex that the value of NumComponents as calculated in Table 31 shall be equal to the value of the entry in the "NC" column.
* Otherwise, if the alpha channel is present as a separate alpha image plane (as specified by the entry in the "Alpha" column and the presence of ALPHA\_OFFSET), it is a requirement for file format conformance to this annex that the value of NumComponents as calculated in Table 31 for the CODED\_IMAGE( ) syntax structure at the position specified by IMAGE\_OFFSET shall be equal to the value of the entry in the "NC" column minus 1, and the value of NumComponents as calculated in Table 31 for the CODED\_IMAGE( ) syntax structure at the position specified by ALPHA\_OFFSET shall be equal to 1.
* Otherwise (the alpha channel is present as an interleaved alpha image plane), it is a requirement for file format conformance to this annex that the value of NumComponents as calculated in Table 31 for the primary image plane shall be equal to the value of the entry in the "NC" column minus 1, and the value of NumComponents as calculated in Table 31 for the alpha image plane shall be equal to 1.
* "Alpha" specifies whether the PIXEL\_FORMAT specifies the presence of an alpha channel. An entry marked with "✓" indicates that an alpha channel is present and an entry not marked with "✓" indicates that an alpha channel is not present.
* "BPC" specifies a requirement for file format conformance to this annex that the value of the OUTPUT\_BITDEPTH syntax element of the CODED\_IMAGE( ) syntax structure at the position specified by IMAGE\_OFFSET shall be equal to the value of the entry in the "BPC" column and, when present, the value of the OUTPUT\_BITDEPTH syntax element of the CODED\_IMAGE( ) syntax structure at the position specified by ALPHA\_OFFSET shall be equal to value of the entry in the "BPC" column.
* "Num" specifies the numeric interpretation of values in the decoded image.
* "Colour" specifies a requirement for file format conformance to this annex that the value of the OUTPUT\_CLR\_FMT syntax element of the CODED\_IMAGE( ) syntax structure at the position specified by IMAGE\_OFFSET shall be equal to the value of the entry in the "Colour" column.

NOTE 2 – Image data may be structured as single channel monochrome "Gray", three-channel "RGB", four-channel "CMYK" or n-Channel ("NCOMPONENT") with n in the range of 2 to 16, inclusive.

Table A.6 – Interpretation of PIXEL\_FORMAT

| **PIXEL\_FORMAT value** | **Mnemonic** | **NC** | **Alpha** | **BPC** | **Num** | **Colour** |
| --- | --- | --- | --- | --- | --- | --- |
| 0x24C3DD6f034EFE4Bb1853d77768dc90d | 24bppRGB | 3 |  | BD8 | UINT | RGB |
| 0x24C3DD6f034EFE4Bb1853d77768dc90c | 24bppBGR | 3 |  | BD8 | UINT | RGB |
| 0x24C3DD6f034EFE4Bb1853d77768dc90e | 32bppBGR | 3 |  | BD8 | UINT | RGB |
| 0x24C3DD6f034EFE4Bb1853d77768dc915 | 48bppRGB | 3 |  | BD16 | UINT | RGB |
| 0x24C3DD6f034EFE4Bb1853d77768dc912 | 48bppRGBFixedPoint | 3 |  | BD16S | SINT | RGB |
| 0x24C3DD6f034EFE4Bb1853d77768dc93b | 48bppRGBHalf | 3 |  | BD16F | Float | RGB |
| 0x24C3DD6f034EFE4Bb1853d77768dc918 | 96bppRGBFixedPoint | 3 |  | BD32S | SINT | RGB |
| 0x24C3DD6f034EFE4Bb1853d77768dc940 | 64bppRGBFixedPoint | 3 |  | BD16S | SINT | RGB |
| 0x24C3DD6f034EFE4Bb1853d77768dc942 | 64bppRGBHalf | 3 |  | BD16F | Float | RGB |
| 0x24C3DD6f034EFE4Bb1853d77768dc941 | 128bppRGBFixedPoint | 3 |  | BD32S | SINT | RGB |
| 0x24C3DD6f034EFE4Bb1853d77768dc91b | 128bppRGBFloat | 3 |  | BD32F | Float | RGB |
| 0x24C3DD6f034EFE4Bb1853d77768dc90f | 32bppBGRA | 4 | ✓ | BD8 | UINT | RGB |
| 0x24C3DD6f034EFE4Bb1853d77768dc916 | 64bppRGBA | 4 | ✓ | BD16 | UINT | RGB |
| 0x24C3DD6f034EFE4Bb1853d77768dc91d | 64bppRGBAFixedPoint | 4 | ✓ | BD16S | SINT | RGB |
| 0x24C3DD6f034EFE4Bb1853d77768dc93a | 64bppRGBAHalf | 4 | ✓ | BD16F | Float | RGB |
| 0x24C3DD6f034EFE4Bb1853d77768dc91e | 128bppRGBAFixedPoint | 4 | ✓ | BD32S | SINT | RGB |
| 0x24C3DD6f034EFE4Bb1853d77768dc919 | 128bppRGBAFloat | 4 | ✓ | BD32F | Float | RGB |
| 0x24C3DD6f034EFE4Bb1853d77768dc910 | 32bppPBGRA | 4 | ✓ | BD8 | UINT | RGB |
| 0x24C3DD6f034EFE4Bb1853d77768dc917 | 64bppPRGBA | 4 | ✓ | BD16 | UINT | RGB |
| 0x24C3DD6f034EFE4Bb1853d77768dc91a | 128bppPRGBAFloat | 4 | ✓ | BD32F | Float | RGB |
| 0x24C3DD6f034EFE4Bb1853d77768dc91c | 32bppCMYK | 4 |  | BD8 | UINT | CMYK |
| 0x24C3DD6f034EFE4Bb1853d77768dc92c | 40bppCMYKAlpha | 5 | ✓ | BD8 | UINT | CMYK |
| 0x24C3DD6f034EFE4Bb1853d77768dc91f | 64bppCMYK | 4 |  | BD16 | UINT | CMYK |
| 0x24C3DD6f034EFE4Bb1853d77768dc92d | 80bppCMYKAlpha | 5 | ✓ | BD16 | UINT | CMYK |
| 0x24C3DD6f034EFE4Bb1853d77768dc920 | 24bpp3Channels | 3 |  | BD8 | UINT | NCOMPONENT |
| 0x24C3DD6f034EFE4Bb1853d77768dc921 | 32bpp4Channels | 4 |  | BD8 | UINT | NCOMPONENT |
| 0x24C3DD6f034EFE4Bb1853d77768dc922 | 40bpp5Channels | 5 |  | BD8 | UINT | NCOMPONENT |
| 0x24C3DD6f034EFE4Bb1853d77768dc923 | 48bpp6Channels | 6 |  | BD8 | UINT | NCOMPONENT |
| 0x24C3DD6f034EFE4Bb1853d77768dc924 | 56bpp7Channels | 7 |  | BD8 | UINT | NCOMPONENT |
| 0x24C3DD6f034EFE4Bb1853d77768dc925 | 64bpp8Channels | 8 |  | BD8 | UINT | NCOMPONENT |
| 0x24C3DD6f034EFE4Bb1853d77768dc92e | 32bpp3ChannelsAlpha | 4 | ✓ | BD8 | UINT | NCOMPONENT |
| 0x24C3DD6f034EFE4Bb1853d77768dc92f | 40bpp4ChannelsAlpha | 5 | ✓ | BD8 | UINT | NCOMPONENT |
| 0x24C3DD6f034EFE4Bb1853d77768dc930 | 48bpp5ChannelsAlpha | 6 | ✓ | BD8 | UINT | NCOMPONENT |
| 0x24C3DD6f034EFE4Bb1853d77768dc931 | 56bpp6ChannelsAlpha | 7 | ✓ | BD8 | UINT | NCOMPONENT |
| 0x24C3DD6f034EFE4Bb1853d77768dc932 | 64bpp7ChannelsAlpha | 8 | ✓ | BD8 | UINT | NCOMPONENT |
| 0x24C3DD6f034EFE4Bb1853d77768dc933 | 72bpp8ChannelsAlpha | 9 | ✓ | BD8 | UINT | NCOMPONENT |
| 0x24C3DD6f034EFE4Bb1853d77768dc926 | 48bpp3Channels | 3 |  | BD16 | UINT | NCOMPONENT |
| 0x24C3DD6f034EFE4Bb1853d77768dc927 | 64bpp4Channels | 4 |  | BD16 | UINT | NCOMPONENT |
| 0x24C3DD6f034EFE4Bb1853d77768dc928 | 80bpp5Channels | 5 |  | BD16 | UINT | NCOMPONENT |
| 0x24C3DD6f034EFE4Bb1853d77768dc929 | 96bpp6Channels | 6 |  | BD16 | UINT | NCOMPONENT |
| 0x24C3DD6f034EFE4Bb1853d77768dc92A | 112bpp7Channels | 7 |  | BD16 | UINT | NCOMPONENT |
| 0x24C3DD6f034EFE4Bb1853d77768dc92B | 128bpp8Channels | 8 |  | BD16 | UINT | NCOMPONENT |
| 0x24C3DD6f034EFE4Bb1853d77768dc934 | 64bpp3ChannelsAlpha | 4 | ✓ | BD16 | UINT | NCOMPONENT |
| 0x24C3DD6f034EFE4Bb1853d77768dc935 | 80bpp4ChannelsAlpha | 5 | ✓ | BD16 | UINT | NCOMPONENT |
| 0x24C3DD6f034EFE4Bb1853d77768dc936 | 96bpp5ChannelsAlpha | 6 | ✓ | BD16 | UINT | NCOMPONENT |
| 0x24C3DD6f034EFE4Bb1853d77768dc937 | 112bpp6ChannelsAlpha | 7 | ✓ | BD16 | UINT | NCOMPONENT |
| 0x24C3DD6f034EFE4Bb1853d77768dc938 | 128bpp7ChannelsAlpha | 8 | ✓ | BD16 | UINT | NCOMPONENT |
| 0x24C3DD6f034EFE4Bb1853d77768dc939 | 144bpp8ChannelsAlpha | 9 | ✓ | BD16 | UINT | NCOMPONENT |
| 0x24C3DD6f034EFE4Bb1853d77768dc908 | 8bppGray | 1 |  | BD8 | UINT | YONLY |
| 0x24C3DD6f034EFE4Bb1853d77768dc90b | 16bppGray | 1 |  | BD16 | UINT | YONLY |
| 0x24C3DD6f034EFE4Bb1853d77768dc913 | 16bppGrayFixedPoint | 1 |  | BD16S | SINT | YONLY |
| 0x24C3DD6f034EFE4BB1853d77768dc93e | 16bppGrayHalf | 1 |  | BD16F | Float | YONLY |
| 0x24C3DD6f034EFE4Bb1853d77768dc93f | 32bppGrayFixedPoint | 1 |  | BD32S | SINT | YONLY |
| 0x24C3DD6f034EFE4Bb1853d77768dc911 | 32bppGrayFloat | 1 |  | BD32F | Float | YONLY |
| 0x24C3DD6f034EFE4Bb1853d77768dc905 | BlackWhite | 1 |  | BD1WHITE1 or BD1BLACK1 | UINT | YONLY |
| 0x24C3DD6f034EFE4Bb1853d77768dc909 | 16bppBGR555 | 3 |  | BD5 | UINT | RGB |
| 0x24C3DD6f034EFE4Bb1853d77768dc90a | 16bppBGR565 | 3 |  | BD565 | UINT | RGB |
| 0x24C3DD6F034EFE4BB1853D77768DC914 | 32bppBGR101010 | 3 |  | BD10 | UINT | RGB |
| 0x24C3DD6f034EFE4Bb1853d77768dc93d | 32bppRGBE | 3 |  | BD8 | Float | RGBE |
| 0x24C3DD6F034EFE4BB1853D77768DC954 | 32bppCMYKDIRECT | 4 |  | BD8 | UINT | CMYKDIRECT |
| 0x24C3DD6F034EFE4BB1853D77768DC955 | 64bppCMYKDIRECT | 4 |  | BD16 | UINT | CMYKDIRECT |
| 0x24C3DD6F034EFE4BB1853D77768DC956 | 40bppCMYKDIRECTAlpha | 5 | ✓ | BD8 | UINT | CMYKDIRECT |
| 0x24C3DD6f034EFE4Bb1853d77768dc943 | 80bppCMYKDIRECTAlpha | 5 | ✓ | BD16 | UINT | CMYKDIRECT |
| 0x24C3DD6f034EFE4Bb1853d77768dc944 | 12bppYCC420 | 3 |  | BD8 | UINT | YUV420 |
| 0x24C3DD6f034EFE4Bb1853d77768dc945 | 16bppYCC422 | 3 |  | BD8 | UINT | YUV422 |
| 0x24C3DD6f034EFE4Bb1853d77768dc946 | 20bppYCC422 | 3 |  | BD10 | UINT | YUV422 |
| 0x24C3DD6f034EFE4Bb1853d77768dc947 | 32bppYCC422 | 3 |  | BD16 | UINT | YUV422 |
| 0x24C3DD6f034EFE4Bb1853d77768dc948 | 24bppYCC444 | 3 |  | BD8 | UINT | YUV444 |
| 0x24C3DD6f034EFE4Bb1853d77768dc949 | 30bppYCC444 | 3 |  | BD10 | UINT | YUV444 |
| 0x24C3DD6f034EFE4Bb1853d77768dc94A | 48bppYCC444 | 3 |  | BD16 | UINT | YUV444 |
| 0x24C3DD6f034EFE4Bb1853d77768dc94B | 48bppYCC444FixedPoint | 3 |  | BD16S | SINT | YUV444 |
| 0x24C3DD6f034EFE4Bb1853d77768dc94C | 20bppYCC420Alpha | 4 | ✓ | BD8 | UINT | YUV420 |
| 0x24C3DD6f034EFE4Bb1853d77768dc94D | 24bppYCC422Alpha | 4 | ✓ | BD8 | UINT | YUV422 |
| 0x24C3DD6f034EFE4Bb1853d77768dc94E | 30bppYCC422Alpha | 4 | ✓ | BD10 | UINT | YUV422 |
| 0x24C3DD6f034EFE4Bb1853d77768dc94F | 48bppYCC422Alpha | 4 | ✓ | BD16 | UINT | YUV422 |
| 0x24C3DD6f034EFE4Bb1853d77768dc950 | 32bppYCC444Alpha | 4 | ✓ | BD8 | UINT | YUV444 |
| 0x24C3DD6f034EFE4Bb1853d77768dc951 | 40bppYCC444Alpha | 4 | ✓ | BD10 | UINT | YUV444 |
| 0x24C3DD6f034EFE4Bb1853d77768dc952 | 64bppYCC444Alpha | 4 | ✓ | BD16 | UINT | YUV444 |
| 0x24C3DD6f034EFE4Bb1853d77768dc953 | 64bppYCC444AlphaFixedPoint | 4 | ✓ | BD16S | SINT | YUV444 |

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

* If PIXEL\_FORMAT is equal to 24bppRGB, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is R, then G, then B, and each sample value for each channel is output as a single byte.
* Otherwise, if PIXEL\_FORMAT is equal to 32bppRGBE, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is B, then G, then R, then E, and each sample value for each channel is output as a single byte.
* Otherwise, if PIXEL\_FORMAT is equal to 48bppRGB, 48bppRGBFixedPoint or 48bppRGBHalf, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is R, then G, then B, and each sample value for each channel is output using 16 bits in little-endian form.
* Otherwise, if PIXEL\_FORMAT is equal to 96bppRGBFixedPoint, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is R, then G, then B, and each sample value for each channel is output using 32 bits in little-endian form.
* Otherwise, if PIXEL\_FORMAT is equal to 64bppRGBFixedPoint or 64bppRGBHalf, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is R, then G, then B, then a padding channel, and each sample value for each channel is output using 16 bits in little-endian form.
* Otherwise, if PIXEL\_FORMAT is equal to 128bppRGBFixedPoint or 128bppRGBFloat, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is R, then G, then B, then a padding channel, and each sample value for each channel is output using 32 bits in little-endian form.
* Otherwise, if PIXEL\_FORMAT is equal to 64bppRGBA, 64bppRGBAFixedPoint, 64bppRGBAHalf or 64bppPRGBA, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is R, then G, then B, then Alpha, and each sample value for each channel is output using 16 bits in little-endian form.
* Otherwise, if PIXEL\_FORMAT is equal to 128bppRGBAFixedPoint, 128bppRGBAFloat or 128bppPRGBAFloat, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is R, then G, then B, then Alpha, and each sample value for each channel is output using 32 bits in little-endian form.
* Otherwise, if PIXEL\_FORMAT is equal to 24bppBGR, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is B, then G, then R, and each sample value for each channel is output as a single byte.
* Otherwise, if PIXEL\_FORMAT is equal to 32bppBGR, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is B, then G, then R, then a padding channel, and each sample value for each channel is output as a single byte.
* Otherwise, if PIXEL\_FORMAT is equal to 32bppPBGRA, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is B, then G, then R, then Alpha, and each sample value for each channel is output as a single byte.
* Otherwise, if PIXEL\_FORMAT is equal to 16bppBGR565, the output is in the form of packed bit fields within the integer values specified in Table 196, and these integer values are output in raster scan order using 16 bits in little-endian form. (See Note 3.)
* Otherwise, if PIXEL\_FORMAT is equal to 16bppBGR555, the output is in the form of packed bit fields within the integer values specified in Table 197, and these integer values are output in raster scan order using 16 bits in little-endian form. (See Note 3.)
* Otherwise, if PIXEL\_FORMAT is equal to 32bppBGR101010, the output is in the form of packed bit fields within the integer values specified in Table 198, and these integer values are output in raster scan order using 32 bits in little-endian form. (See Note 3.)

NOTE 3 – When the PIXEL\_FORMAT is equal to 16bppBGR565, 16bppBGR555 or 32bppBGR101010, the positions of the bits representing the actual R and B components depend on the value of the RED\_BLUE\_NOT\_SWAPPED\_FLAG. In these cases, if RED\_BLUE\_NOT\_SWAPPED\_FLAG is equal to 0, the R component bits are in the MSBs of the packed bit fields and the B component bits are in the LSBs; otherwise, in these cases the R component bits are in the LSBs of the packed bit fields and the B component bits are in the MSBs.

* Otherwise, if PIXEL\_FORMAT is equal to BlackWhite, the output is in the form of packed bit fields within the integer values specified in Table 199, and these integer values are output as bytes in raster scan order.
* Otherwise, if PIXEL\_FORMAT is equal to 32bppCMYK or 32bppCMYKDIRECT, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is C, then M, then Y, then K, and each sample value for each channel is output as a single byte.
* Otherwise, if PIXEL\_FORMAT is equal to 40bppCMYKAlpha or 40bppCMYKDIRECTAlpha, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is C, then M, then Y, then K, then Alpha, and each sample value for each channel is output as a single byte.
* Otherwise, if PIXEL\_FORMAT is equal to 64bppCMYK or 64bppCMYKDIRECT, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is C, then M, then Y, then K, and each sample value for each channel is output using 16 bits in little-endian form.
* Otherwise, if PIXEL\_FORMAT is equal to 80bppCMYKAlpha or 80bppCMYKDIRECTAlpha, the decoded channels are interleaved on a sample-by-sample basis in raster scan order, where the ordering of the interleaving is C, then M, then Y, then K, then Alpha, and each sample value for each channel is output using 16 bits in little-endian form.
* Otherwise, if PIXEL\_FORMAT is equal to 12bppYCC420, 16bppYCC422 or 24bppYCC444, the Y samples are output first in raster scan order, then the U samples are output in raster scan order, then the V samples are output in raster scan order, and each sample value for each channel is output as a single byte.
* Otherwise, if PIXEL\_FORMAT is equal to 20bppYCC422, 32bppYCC422, 30bppYCC444, 48bppYCC444 or 48bppYCC444FixedPoint, the Y samples are output first in raster scan order, then the U samples are output in raster scan order, then the V samples are output in raster scan order, and each sample value for each channel is output using 16 bits in little-endian form.
* Otherwise, if PIXEL\_FORMAT is equal to 20bppYCC420Alpha or 24bppYCC422Alpha, the Y samples are output first in raster scan order, then the U samples are output in raster scan order, then the V samples are output in raster scan order, then the Alpha samples are output in raster scan order, and each sample value for each channel is output as a single byte.
* Otherwise, if PIXEL\_FORMAT is equal to 32bppYCC444Alpha, the Y samples are output first in raster scan order, then the U samples are output in raster scan order, then the V samples are output in raster scan order, then the Alpha samples are output in raster scan order, and each sample value for each channel is output as a single byte.
* Otherwise, if PIXEL\_FORMAT is equal to 30bppYCC422Alpha, 48bppYCC422Alpha, 40bppYCC444Alpha, 64bppYCC444Alpha or 64bppYCC444AlphaFixedPoint, the Y samples are output first in raster scan order, then the U samples are output in raster scan order, then the V samples are output in raster scan order, then the Alpha samples are output in raster scan order, and each sample value for each channel is output using 16 bits in little-endian form.
* Otherwise, if PIXEL\_FORMAT is equal to 24bpp3Channels, 32bpp4Channels, 40bpp5Channels, 48bpp6Channels, 56bpp7Channels or 64bpp8Channels, the decoded channels are ordered by increasing channel number, and the samples of each channel are output in raster scan order, and each sample value for each channel is output as a single byte.
* Otherwise, if PIXEL\_FORMAT is equal to 48bpp3Channels, 64bpp4Channels, 80bpp5Channels, 96bpp6Channels, 112bpp7Channels or 128bpp8Channels, the decoded channels are ordered by increasing channel number, and the samples of each channel are output in raster scan order, and each sample value for each channel is output using 16 bits in little-endian form.
* Otherwise, if PIXEL\_FORMAT is equal to 32bpp3ChannelsAlpha, 40bpp4ChannelsAlpha, 48bpp5ChannelsAlpha, 56bpp6ChannelsAlpha, 64bpp7ChannelsAlpha or 72bpp8ChannelsAlpha, the decoded channels are ordered by increasing channel number with the Alpha channel being considered as having the highest channel number and the samples of each channel are output in raster scan order, and each sample value for each channel is output as a single byte.
* Otherwise, if PIXEL\_FORMAT is equal to 64bpp3ChannelsAlpha, 80bpp4ChannelsAlpha, 96bpp5ChannelsAlpha, 112bpp6ChannelsAlpha, 128bpp7ChannelsAlpha or 144bpp8ChannelsAlpha, the decoded channels are ordered by increasing channel number and the last channel is considered the Alpha channel, and the samples of each channel are output in raster scan order, and each sample value for each channel is output using 16 bits in little-endian form.
* Otherwise, if PIXEL\_FORMAT is equal to 8bppGray, there is only one decoded channel, and the samples of that channel are output in raster scan order, and each sample value is output as a single byte.
* Otherwise, if PIXEL\_FORMAT is equal to 16bppGray, 16bppGrayFixedPoint or 16bppGrayHalf, there is only one decoded channel, and the samples of that channel are output in raster scan order, and each sample value is output using 16 bits in little-endian form.
* Otherwise (PIXEL\_FORMAT is equal to 32bppGrayFixedPoint or 32bppGrayFloat), there is only one decoded channel, and the samples of that channel are output in raster scan order, and each sample value is output using 32 bits in little-endian form.

NOTE 4 – Particular care should be taken with regard to the interpretation of this defined output format for PIXEL\_FORMAT values with mnemonic names that include "RGB", "BGR" or "CMYK" as part of the mnemonic name due to the specification of little-endian form for packed bit fields. In particular, the first byte of each packed bit field for the PIXEL\_FORMAT values 16bppBGR565, 16bppBGR555, and 32bppBGR101010 actually contains the LSBs of the packed bit fields due to the use of little-endian form. Similarly, if the 4 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 4 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 5 – The designation of an alpha channel as pre-multiplied indicates that the decoded sample values do not require multiplication by the alpha channel values when performing compositing (as any such necessary multiplication process was performed as a pre-processing step prior to encoding).

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

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

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

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

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

PIXEL\_FORMAT values having mnemonic names that end in "FixedPoint" specify fixed point representations.

NOTE 6 – A fixed point numerical representation is not commonly supported in prior image file formats. It is supported in JPEG XR to encode an extended range of numerical values more directly while retaining the performance advantages of integer processing.

JPEG XR supports fixed point numerical encoding for 16-bit and 32-bit signed values. In this Specification, 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 13 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 24 fractional bits. Using this interpretation, a numerical range of −128.0 to +127.999… can be represented, with the value of 1.0 represented by the signed integer value 16 777 216 (0x01000000).

NOTE 7 – JPEG XR does not enable fully lossless compression for 32-bit data in general. The encoding and decoding algorithms use 32-bit computations, and some dynamic range is lost to necessary headroom for signal processing calculations such as overlap and core transform computations. A minimum of 22 bits and typically 24 bits or more precision is retained through the end-to-end encoding and decoding process.

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 4 bytes. The bytes include unsigned 8-bit mantissas for the red, green and blue channels, plus a shared 8-bit exponent.

NOTE 8 – 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 ISO/IEC/IEEE 60559 floating point number in little-endian form. This encoding uses 1 sign bit, 8 exponent bits and 23 normalized mantissa bits.

NOTE 9 – 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 10 – 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.

* + 1. SPATIAL\_XFRM\_PRIMARY

SPATIAL\_XFRM\_PRIMARY (when present) indicates a preferred spatial transformation that should be applied to the decoded image. The value of SPATIAL\_XFRM\_PRIMARY (when present) shall be in the range of 0 to 7, inclusive, in files conforming to this version of this Specification. Decoders conforming to this version of this Specification should interpret values outside the range from 0 to 7 as equivalent to the value 0. When SPATIAL\_XFRM\_PRIMARY is not present, its value shall be inferred to be equal to 0.

The interpretation of the value of SPATIAL\_XFRM\_PRIMARY is the same as the interpretation specified in clause8.3.8 for the syntax element SPATIAL\_XFRM\_SUBORDINATE.

When the file format specified in this annex is used, the value of SPATIAL\_XFRM\_SUBORDINATE should be ignored.

* + 1. IMAGE\_TYPE

IMAGE\_TYPE indicates an attribute of an individual image in a multi-image file. When present, the value of IMAGE\_TYPE shall be in the range of 0 to 3, inclusive. All other values are reserved for future use by ITU-T | ISO/IEC. Decoders shall ignore the most significant 30 bits of IMAGE\_TYPE.

If IMAGE\_TYPE & 1 is equal to 1, the accompanying image is indicated to be a "preview" image; otherwise the accompanying image is not indicated to be a preview image. A preview image is an alternate representation of another image in the file. The n-th preview image in the file is an alternate representation of the n-th image in the file that is not indicated to be a preview image, where n indicates the ordering in the sequence of IMAGE\_FILE\_DIRECTORY structures in the file. A preview image should be encoded using a reduced resolution or a reduced quantity of encoded data relative to another image in the file.

NOTE 1 – The only constraint on ordering of images indicated to be a preview image and images not indicated to be a preview image aside is the ordering correspondence constraint. That is, the first preview image is a preview of the first image that is not a preview image; the second preview image is a preview of the second image that is not a preview image; and so on. Provided this constraint is met, preview images and images that are not preview images can otherwise be arranged in any order.

A preview image shall be encoded using the mnemonic value of PIXEL\_FORMAT equal to the mnemonic value of PIXEL\_FORMAT for the corresponding image that is not indicated to be a preview image or a mnemonic value of PIXEL\_FORMAT from the following list: 8bppGray, 24bppBGR, 12bppYCC420, 16bppYCC422, 24bppYCC444, 32bppCMYK, 32bppCMYKDIRECT, 24bpp3Channels, 32bpp4Channels, 40bpp5Channels, 48bpp6Channels, 56bpp7Channels, 64bpp8Channels, 32bppBGRA, 20bppYCC420Alpha, 24bppYCC422Alpha, 32bppYCC444Alpha, 40bppCMYKAlpha, 40bppCMYKDIRECTAlpha, 32bpp3ChannelsAlpha, 40bpp4ChannelsAlpha, 48bpp5ChannelsAlpha, 56bpp6ChannelsAlpha, 64bpp7ChannelsAlpha, 72bpp8ChannelsAlpha.

NOTE 2 – A preview image is intended to be used to enable faster, reduced-complexity access to a basic representation of the image content. If multiple images in a file are indicated to be preview images, the first such image should ordinarily be used by applications.

If (IMAGE\_TYPE >> 1) & 1 is equal to 1, the accompanying image is indicated to be an individual "page" of a sequence of "page" images in the file; otherwise the accompanying image is indicated not to be an individual page image of a sequence of page images in the file. When (IMAGE\_TYPE >> 1) & 1 is equal to 1, there should be more than one image in the file for which (IMAGE\_TYPE >> 1) & 1 is equal to 1. When (IMAGE\_TYPE >> 1) & 1 is equal to 1, there shall not be any images in the file for which (IMAGE\_TYPE >> 1) & 1 is equal to 0.

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 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 equal to 1.
* For all images with IMAGE\_TYPE & 1 equal to 1 and (IMAGE\_TYPE >> 1) & 1 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 and (IMAGE\_TYPE >> 1) & 1 equal to 1.

When a file contains only a single image, IMAGE\_TYPE should not be present.

* + 1. PTM\_COLOR\_INFO( )

The PTM\_COLOR\_INFO( ) syntax structure (when present) is specified in Table A.7.

Table A.7 – PTM\_COLOR\_INFO( ) syntax structure

|  |  |  |
| --- | --- | --- |
| **PTM\_COLOR\_INFO( ) {** | **Descriptor** | **Reference** |
| COLOR\_PRIMARIES | u(8) | A.7.21.1 |
| TRANSFER\_CHARACTERISTICS | u(8) | A.7.21.2 |
| MATRIX\_COEFFICIENTS | u(8) | A.7.21.3 |
| RESERVED\_K | u(7) | A.7.21.4 |
| FULL\_RANGE\_FLAG | u(1) | A.7.21.5 |
| } |  |  |

* + - 1. COLOR\_PRIMARIES

COLOR\_PRIMARIES (when present) indicates the chromaticity coordinates of the colour primaries for the image as specified in Table A.8, in terms of the CIE 1931 definition of x and y as specified by ISO 11664-1.

Table A.8 – Interpretation of COLOR\_PRIMARIES syntax element

| Value | Primaries | Informative remark |
| --- | --- | --- |
| 0 | Reserved | For future use by ITU-T | ISO/IEC |
| 1 | primary x y  green 0.300 0.600  blue 0.150 0.060  red 0.640 0.330  white D65 0.3127 0.3290 | Rec. ITU-R BT.709-6  Rec. ITU-R BT.1361-0 (historical) conventional colour gamut system and extended colour gamut system  IEC 61966-2-1 (sRGB or sYCC)  IEC 61966-2-4 (xvYCC)  Annex B of SMPTE RP 177 |
| 2 | Unspecified | Image characteristics are unknown or are determined by the application. |
| 3 | Reserved | For future use by ITU-T | ISO/IEC |
| 4 | primary x y  green 0.21 0.71  blue 0.14 0.08  red 0.67 0.33  white C 0.310 0.316 | Rec. ITU-R BT.470-6 System M (historical)  NTSC 1953  FCC (2003), *Title 47* |
| 5 | primary x y  green 0.29 0.60  blue 0.15 0.06  red 0.64 0.33  white D65 0.3127 0.3290 | Rec. ITU-R BT.470-6 System B, G (historical)  Rec. ITU-R BT.601-7 625  Rec. ITU-R BT.1358-0 625 (historical)  Rec. ITU-R BT.1700-0 625 PAL and 625 SECAM |
| 6 | primary x y  green 0.310 0.595  blue 0.155 0.070  red 0.630 0.340  white D65 0.3127 0.3290 | Rec. ITU-R BT.601-7 525  Rec. ITU-R BT.1358-1 525 or 625  Rec. ITU-R BT.1700-0 NTSC  SMPTE 170M (2004)  (functionally the same as the value 7) |
| 7 | primary x y  green 0.310 0.595  blue 0.155 0.070  red 0.630 0.340  white D65 0.3127 0.3290 | SMPTE 240M (1999) (historical)  (functionally the same as the value 6) |
| 8-255 | Reserved | For future use by ITU-T | ISO/IEC |

* + - 1. TRANSFER\_CHARACTERISTICS

TRANSFER\_CHARACTERISTICS (when present) indicates the opto-electronic transfer characteristic of the image colour components as specified in Table A.9 as a function of a real-valued linear optical intensity input vLc in the range of 0 to 1.

NOTE – Although the transfer characteristics equations found in Table A.9 are provided in the form of a mapping from a real-valued source signal intensity to a real-valued output value as might occur in an analog image capture system, it is intended to be interpreted as guidance to aid in the rendering of the decoded image (by providing a model of a hypothetical capturing system) rather than as a description of the actual image scene signal capturing process.

Table A.9 – Interpretation of TRANSFER\_CHARACTERISTICS syntax element

| Value | Transfer characteristics | Informative remark |
| --- | --- | --- |
| 0 | Reserved | For future use by ITU-T | ISO/IEC |
| 1 | vV = 1.099 \* vLc0.45 − 0.099 for 1 >= vLc >= 0.018  vV = 4.500 \* vLc for 0.018 > vLc >= 0 | Rec. ITU-R BT.709-6  Rec. ITU-R BT.1361-0 (historical) conventional colour gamut system  (functionally the same as the value 6) |
| 2 | Unspecified | Image characteristics are unknown or are determined by the application. |
| 3 | Reserved | For future use by ITU-T | ISO/IEC |
| 4 | Assumed display gamma 2.2 | Rec. ITU-R BT.470-6 System M (historical)  NTSC1953  FCC (2003), *Title 47* |
| 5 | Assumed display gamma 2.8 | Rec. ITU-R BT.470-6 System B, G (historical) |
| 6 | vV = 1.099 \* vLc0.45 − 0.099 for 1 >= vLc >= 0.018  vV = 4.500 \* vLc for 0.018 > vLc >= 0 | Rec. ITU-R BT.601-7 525 or 625  Rec. ITU-R BT.1358-1 525 or 625  Rec. ITU-R BT.1700-0 NTSC  Rec. ITU-R BT.1700-0 625 PAL and 625 SECAM  SMPTE 170M (2004)  (functionally the same as the value 1) |
| 7 | vV = 1.1115 \* vLc0.45 − 0.1115 for 1 >= vLc >= 0.0228  vV = 4.0 \* vLc for 0.0228 > vLc >= 0 | SMPTE 240M (1999) (historical) |
| 8 | vV = vLc for 1 > vLc >= 0 | Linear transfer characteristics |
| 9 | Reserved | For future use by ITU-T | ISO/IEC |
| 10 | Reserved | For future use by ITU-T | ISO/IEC |
| 11 | vV = 1.099 \* vLc0.45 − 0.099 for vLc >= 0.018  vV = 4.500 \* vLc for 0.018 > vLc > −0.018  vV = −1.099 \* (− vLc)0.45 + 0.099 for −0.018 >= vLc | IEC 61966-2-4 (xvYCC) |
| 12 | vV = 1.099 \* vLc0.45 − 0.099 for 1.33 > vLc >= 0.018  vV = 4.500 \* vLc for 0.018 > vLc >= −0.0045  vV = −(1.099 \* (−4 \* vLc) 0.45 − 0.099) ÷ 4 for −0.0045 > vLc >= −0.25 | Rec. ITU-R BT.1361-0 (historical) extended colour gamut system |
| 13 | vV = 1.055 \* vLc(1/2.4) − 0.055 for 1 > vLc >= 0.0031308  vV = 12.92 \* vLc for 0.0031308 > vLc >= 0 | IEC 61966-2-1 (sRGB or sYCC) |
| 14..255 | Reserved | For future use by ITU-T | ISO/IEC |

* + - 1. MATRIX\_COEFFICIENTS

MATRIX\_COEFFICIENTS (when present) indicates the matrix coefficients associated with derivation of luma and chroma signals from the green, blue and red primaries, as specified using Table A.10 and the value of FULL\_RANGE\_FLAG as specified in the following.

Table A.10 – Interpretation of MATRIX\_COEFFICIENTS syntax element

|  |  |  |
| --- | --- | --- |
| Value | Matrix | Informative remark |
| 0 | RGB, GBR | No matrix transformation. See A.7.21.5.  IEC 61966-2-1 (sRGB) |
| 1 | vKr = 0.2126; vKb = 0.0722 | Rec. ITU-R BT.709-6  Rec. ITU-R BT.1361-0 (historical) conventional colour gamut system and extended colour gamut system  IEC 61966-2-1 (sYCC)  IEC 61966-2-4 xvYCC709  Annex B of SMPTE RP 177 |
| 2 | Unspecified | Image characteristics are unknown or are determined by the application. |
| 3 | Reserved | For future use by ITU-T | ISO/IEC |
| 4 | vKr = 0.30; vKb = 0.11 | FCC (2003) *Title 47*) |
| 5 | vKr = 0.299; vKb = 0.114 | Rec. ITU-R BT.470-6 System B, G (historical)  Rec. ITU-R BT.601-7 625  Rec. ITU-R BT.1358-0 625 (historical)  Rec. ITU-R BT.1700-0 625 PAL and 625 SECAM  IEC 61966-2-4 xvYCC601  (functionally the same as the value 6) |
| 6 | vKr = 0.299; vKb = 0.114 | Rec. ITU-R BT.601-7 525  Rec. ITU-R BT.1358-1 525 or 625  Rec. ITU-R BT.1700-0 NTSC  SMPTE 170M (2004)  (functionally the same as the value 5) |
| 7 | vKr = 0.212; vKb = 0.087 | SMPTE 240M (1999) (historical) |
| 8 | YCgCo | See A.7.21.5. |
| 9-255 | Reserved | For future use by ITU-T | ISO/IEC |

* + - 1. RESERVED\_K

RESERVED\_K (when present) is a 7-bit syntax element that shall be equal to 0 in all codestreams conforming to this version of this Specification. All other values are reserved. Decoders conforming to this version of this Specification shall ignore the value of RESERVED\_K.

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

* + - 1. FULL\_RANGE\_FLAG

FULL\_RANGE\_FLAG (when present) is used together with TRANSFER\_CHARACTERISTICS and MATRIX\_COEFFICIENTS to provide an indication of the preferred interpretation of the image colour components.

NOTE 1 – Although the equations found in the following are provided in the form of a mapping from source signals ER, EG and EB to integer output values iR, iG, and iB or Y, Cb, and Cr, they are intended to be interpreted as guidance to aid in the rendering of the decoded image (by providing a model of a hypothetical capturing system) rather than as a description of the actual image scene signal capturing process.

These syntax elements are interpreted as follows:

* If TRANSFER\_CHARACTERISTICS is not equal to 11 or 12, ER, EG and EB are considered to be real-valued signals with values between 0 and 1.
* Otherwise {TRANSFER\_CHARACTERISTICS is equal to 11 (IEC 61966-2-4) or 12 [Rec. ITU-R BT.1361-0 (historical) extended colour gamut system]}, ER, EG and EB are considered to be real-valued signals with a larger range not specified in this Specification.
* Nominal black is considered to have the property ER = 0, EG = 0 and EB = 0.
* Nominal white is considered to have the property ER = 1, EG = 1 and EB = 1.
* The value of valueBitDepthWhite is calculated as follows.
  + If OUTPUT\_BITDEPTH is equal to BD8, the value of valueBitDepthWhite is set equal to 8.
  + Otherwise, if OUTPUT\_BITDEPTH is equal to BD10, the value of valueBitDepthWhite is set equal to 10.
  + Otherwise, if OUTPUT\_BITDEPTH is equal to BD16S, the value of valueBitDepthWhite is set equal to 13.
  + Otherwise, if OUTPUT\_BITDEPTH is equal to BD16, the value of valueBitDepthWhite is set equal to 16.
* The value of valueWhite is set equal to the result of (1 << valueBitDepthWhite) taken as real-value.
* The value of chromaOffset is calculated as follows:
  + If OUTPUT\_BITDEPTH is equal to BD8, BD10 or BD16, the value of chromaOffset is set equal to valueWhite / 2.
  + Otherwise, if OUTPUT\_BITDEPTH is equal to BD16S, the value of chromaOffset is set equal to 0.
* If MATRIX\_COEFFICIENTS is equal to 0 or 8, the following applies.
* If FULL\_RANGE\_FLAG is equal to FALSE, the following equations apply.

RS = valueWhite ÷ 256 \* (219 \* E'R + 16)  
GS = valueWhite ÷ 256 \* (219 \* E'G + 16)  
BS = valueWhite ÷ 256 \* (219 \* E'B + 16)

* Otherwise (FULL\_RANGE\_FLAG is equal to TRUE), the following equations apply.

RS = (valueWhite − 1) \* E'R  
GS = (valueWhite − 1) \* E'G  
BS = (valueWhite − 1) \* E'B

* If MATRIX\_COEFFICIENTS is equal to 0, the following equations apply.

iR = Round(RS)  
iG = Round(GS)  
iB = Round(BS)

* Otherwise (MATRIX\_COEFFICIENTS is equal to 8), the following equations apply.

Y  Round(0.5 \* GS + 0.25 \* (RS + BS))  
Cb  Round(0.5 \* GS − 0.25 \* (RS + BS)) + chromaOffset  
Cr  Round(0.5 \* (RS − BS)) + chromaOffset

NOTE 2 – For purposes of the YCgCo nomenclature used in Table A.10, Cb and Cr of the foregoing equations may be referred to as Cg and Co, respectively. If OUTPUT\_BITDEPTH is equal to BD8, BD10 or BD16, the inverse conversion for the above three equations should be computed as

t = Y − (Cb − chromaOffset)  
iG = t + 2 \* (Cb − chromaOffset)  
iB = t − (Cr − chromaOffset)  
iR = iB + 2 \* (Cr − chromaOffset)

Otherwise, if OUTPUT\_BITDEPTH is equal to BD16S, the inverse conversion for the above three equations should be computed as.

t = Y − (Cb >> 1)  
iG = t + Cb  
iB = t − (Cr >> 1)  
iR = iB + Cr

* Otherwise, if MATRIX\_COEFFICIENTS is equal to 1, 4, 5, 6 or 7, the following applies:
* The following equations apply.

E'Y = vKr \* E'R + (1 − vKr − vKb) \* E'G + vKb \* E'B

E'PB = 0.5 \* (E'B − E'Y) ÷ (1 − vKb)

E'PR = 0.5 \* (E'R − E'Y) ÷ (1 − vKr)

NOTE 3 – E'Y is real-valued with the value 0 associated with nominal black and the value 1 associated with nominal white. E'PB and E'PR are real-valued with the value 0 associated with both nominal black and nominal white. When TRANSFER\_CHARACTERISTICS is not equal to 11 or 12, E'Y is real-valued with values in the range of 0 to 1. When TRANSFER\_CHARACTERISTICS is not equal to 11 or 12, E'PB and E'PR are real-valued with values in the range of −0.5 to 0.5. When TRANSFER\_CHARACTERISTICS is equal to 11 (IEC 61966-2-4) or 12 [ITU‑R BT.1361-0 (historical) extended colour gamut system], E'Y, E'PB and E'PR are real-valued with a larger range not specified in this Specification.

* If FULL\_RANGE\_FLAG is equal to FALSE and OUTPUT\_BITDEPTH is equal to BD8, BD10 or BD16, Y, Cb and Cr are related to EY, EPB and EPR by the following.

Y = Clip(Round(valueWhite ÷ 256 \* (219 \* E'Y + 16)), 0, valueWhite – 1)

Cb = Clip(Round(valueWhite ÷ 256 \* (224 \* E'PB) + chromaOffset), 0, valueWhite – 1)

Cr = Clip(Round(valueWhite ÷ 256 \* (224 \* E'PR) + chromaOffset), 0, valueWhite – 1)

* If FULL\_RANGE\_FLAG is equal to FALSE and OUTPUT\_BITDEPTH is equal to BD16S, Y, Cb and Cr are related to EY, EPB and EPR by the following.

Y = Round(valueWhite ÷ 256 \* (219 \* E'Y + 16))

Cb = Round(valueWhite ÷ 256 \* (224 \* E'PB) + chromaOffset)

Cr = Round(valueWhite ÷ 256 \* (224 \* E'PR) + chromaOffset)

* Otherwise (FULL\_RANGE\_FLAG is equal to TRUE), Y, Cb and Cr are related to EY, EPB and EPR by the following.

Y = Round((valueWhite − 1) \* E'Y)

Cb = Round((valueWhite − 1) \* E'PB + chromaOffset)

Cr = Round((valueWhite − 1) \* E'PR + chromaOffset)

* Otherwise, no interpretation of FULL\_RANGE\_FLAG, TRANSFER\_CHARACTERISTICS, and MATRIX\_COEFFICIENTS is specified in this version of this Specification.
  + 1. PROFILE\_LEVEL\_CONTAINER( )

PROFILE\_LEVEL\_CONTAINER( ) (when present) has the same syntax and semantics as PROFILE\_LEVEL\_INFO( ) in ‎8.6.

* + 1. IMAGE\_WIDTH

IMAGE\_WIDTH specifies the width of the image in units of sample columns after decoding and before application of the spatial transformation indicated by the (actual or inferred) value of SPATIAL\_XFRM\_PRIMARY. The value of IMAGE\_WIDTH shall be equal to WIDTH\_MINUS1 + 1.

NOTE – When ALPHA\_OFFSET is present, the value of IMAGE\_WIDTH is equal to the value of WIDTH\_MINUS1 + 1 from the CODED\_IMAGE( ) syntax structure at the position specified by ALPHA\_OFFSET and is also equal to the value of WIDTH\_MINUS1 + 1 from the CODED\_IMAGE( ) syntax structure at the position specified by IMAGE\_OFFSET.

* + 1. IMAGE\_HEIGHT

IMAGE\_HEIGHT specifies the height of the image in units of sample rows after decoding and before application of the spatial transformation indicated by the (actual or inferred) value of SPATIAL\_XFRM\_PRIMARY. The value of IMAGE\_HEIGHT shall be equal to HEIGHT\_MINUS1 + 1.

NOTE – When ALPHA\_OFFSET is present, the value of IMAGE\_HEIGHT is equal to the value of HEIGHT\_MINUS1 + 1 from the CODED\_IMAGE( ) syntax structure at the position specified by ALPHA\_OFFSET and is also equal to the value of HEIGHT\_MINUS1 + 1 from the CODED\_IMAGE( ) syntax structure at the position specified by IMAGE\_OFFSET.

* + 1. WIDTH\_RESOLUTION

WIDTH\_RESOLUTION (when present) indicates a nominal width of the associated image after decoding and before application of the spatial transformation indicated by the (actual or inferred) value of SPATIAL\_XFRM\_PRIMARY. WIDTH\_RESOLUTION is in units of columns per inch (columns per 2.54 cm). When WIDTH\_RESOLUTION is equal to 0 or is not present, a value of 96 columns per inch should be inferred. The nominal width of the image (before application of the spatial transformation indicated by SPATIAL\_XFRM\_PRIMARY) is indicated to be IMAGE\_WIDTH ÷ WIDTH\_RESOLUTION inches.

NOTE – The nominal width may be interpreted as an indication of a suggested width for printing or display of the image. The combination of IMAGE\_WIDTH, IMAGE\_HEIGHT, WIDTH\_RESOLUTION and HEIGHT\_RESOLUTION may be used to determine an indication of a suggested display or printing aspect ratio for the image.

* + 1. HEIGHT\_RESOLUTION

HEIGHT\_RESOLUTION (when present) indicates a nominal height of the associated image after decoding and before application of the spatial transformation indicated by the (actual or inferred) value of SPATIAL\_XFRM\_PRIMARY. HEIGHT\_RESOLUTION is in units of rows per inch (rows per 2.54 cm). When HEIGHT\_RESOLUTION is equal to 0 or is not present, a value of 96 rows per inch should be inferred. The nominal height of the image (before application of the spatial transformation indicated by SPATIAL\_XFRM\_PRIMARY) is indicated to be IMAGE\_HEIGHT ÷ HEIGHT\_RESOLUTION inches.

NOTE – The nominal height may be interpreted as an indication of a suggested height for printing or display of the image. The combination of IMAGE\_WIDTH, IMAGE\_HEIGHT, WIDTH\_RESOLUTION and HEIGHT\_RESOLUTION may be used to determine an indication of a suggested display or printing aspect ratio for the image.

* + 1. IMAGE\_OFFSET

IMAGE\_OFFSET specifies the byte position, relative to the beginning of the file, of the start of a CODED\_IMAGE( ) syntax structure in the file (see 8.2) containing the primary image plane in the IMAGE\_FILE\_DIRECTORY( ). When ALPHA\_OFFSET is not present, but an alpha plane is present, the CODED\_IMAGE( ) syntax structure at the position specified by IMAGE\_OFFSET also contains the alpha image plane.

* + 1. IMAGE\_BYTE\_COUNT

IMAGE\_BYTE\_COUNT specifies the quantity of the image codestream data in the file as follows.

* If IMAGE\_BYTE\_COUNT is not equal to 0, the quantity of the image codestream data in the file is indicated to be equal to the value of IMAGE\_BYTE\_COUNT.
* Otherwise (IMAGE\_BYTE\_COUNT is equal to 0), the quantity of image codestream data in the file is indicated to be equal to FileSizeInBytes − IMAGE\_OFFSET.

NOTE – When IMAGE\_BYTE\_COUNT is equal to 0, no additional syntax structures are present in the file after the CODED\_IMAGE( ) syntax structure at the position specified by IMAGE\_OFFSET.

* + 1. 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 (‎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.

* + 1. ALPHA\_BYTE\_COUNT

ALPHA\_BYTE\_COUNT (when present) specifies the quantity of the alpha plane codestream data in the file. When ALPHA\_OFFSET is present, ALPHA\_BYTE\_COUNT shall also be present.

* + 1. IMAGE\_BAND\_PRESENCE

IMAGE\_BAND\_PRESENCE (when present) indicates which frequency bands are present in the accompanying image data as specified in Table A.11. When IMAGE\_BAND\_PRESENCE is not present, its value shall be inferred to be equal to 0. The value of IMAGE\_BAND\_PRESENCE, when present, shall be equal to the value of BANDS\_PRESENT of the primary image plane of the CODED\_IMAGE( ) syntax structure at the position specified by IMAGE\_OFFSET.

Table A.11 – Interpretation of IMAGE\_BAND\_PRESENCE and ALPHA\_BAND\_PRESENCE

|  |  |
| --- | --- |
| **IMAGE\_BAND\_PRESENCE or ALPHA\_BAND\_PRESENCE** | **Bands present** |
| **0** | DC, LP, HP, and Flexbits |
| **1** | DC, LP, and HP only |
| **2** | DC and LP only |
| **3** | DC only |

* + 1. ALPHA\_BAND\_PRESENCE

ALPHA\_BAND\_PRESENCE (when present) indicates which frequency bands are represented in the accompanying alpha channel data as specified in Table A.11. When ALPHA\_OFFSET is not present, ALPHA\_BAND\_PRESENCE shall not be present. When ALPHA\_OFFSET is present and ALPHA\_BAND\_PRESENCE is not present, its value shall be inferred to be equal to 0. If the alpha channel is present as an interleaved alpha image plane, the value of ALPHA\_BAND\_PRESENCE, when present, shall be equal to the value of BANDS\_PRESENT of the alpha image plane of the CODED\_IMAGE( ) syntax structure at the position specified by IMAGE\_OFFSET. If the alpha channel is present as a separate alpha image plane, the value of ALPHA\_BAND\_PRESENCE, when present, shall be equal to the value of BANDS\_PRESENT of the CODED\_IMAGE( ) syntax structure at the position specified by ALPHA\_OFFSET.

* + 1. PADDING\_DATA

PADDING\_DATA (when present) consists of a string of byte values of the indicated quantity. PADDING\_DATA is formatted as follows.

* The first byte shall be equal to 0x1C.
* The second byte (when present) shall be equal to 0xEA.
* The remaining bytes (when present) shall be equal to 0 in all codestreams conforming to this version of this Specification. Decoders conforming to this version of this Specification shall ignore the value of the remaining bytes.

NOTE – One use for PADDING\_DATA is to reserve space within a file for potential later replacement by other data.

1. Profiles and levels

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

* 1. General

This annex specifies sets of constraints on the syntax and values of derived parameters that enhance interoperability by establishing bounds on the capabilities that are needed for the decoding of images encoded according to this Recommendation | International Standard. These sets of constraints are classified into profile and level constraints.

NOTE – Typically, a profile constraint restricts the use of algorithmic features, while a level constraint restricts the range of values in a manner intended to constrain resource requirements such as memory storage capacity.

Decoders that claim conformance to a specified profile of this Specification at some specified level shall be capable of decoding any coded image that conforms to all specified constraints for the specified profile at the specified level or any lower specified level and has such conformance expressed syntactically as specified in this annex.

Conformance of a coded image to a particular profile and level is expressed by a PROFILE\_LEVEL\_INFO( ) syntax structure. When the PROFILE\_LEVEL\_INFO( ) syntax structure is not present, the following inference rules apply:

* PROFILE\_IDC shall be inferred to be equal to 111.
* LEVEL\_IDC shall be inferred to be equal to 255.
  1. Profiles
     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 32bppBGR101010 encoding to avoid exceeding the dynamic range constraints imposed when LONG\_WORD\_FLAG is equal to FALSE.

Conformance of a coded image to the sub-baseline profile is indicated by PROFILE\_IDC equal to 44. The value 44 for PROFILE\_IDC shall not be used in a PROFILE\_LEVEL\_INFO( ) syntax structure unless the associated coded image conforms to the constraints specified for the sub-baseline profile.

Values of PROFILE\_IDC that are less than 44 may additionally be interpreted by decoders as expressing conformance of the associated coded image to the sub-baseline profile.

* + 1. Baseline profile

The baseline profile is defined as the set of coded image syntax that is consistent with the use of the following PIXEL\_FORMAT mnemonic values (specified in Table A.6): 24bppRGB, 24bppBGR, 32bppBGR, 48bppRGB, 48bppRGBFixedPoint, 64bppRGBFixedPoint, 8bppGray, 16bppGray, 16bppGrayFixedPoint, BlackWhite, 16bppBGR555, 16bppBGR565 or 32bppBGR101010.

NOTE – Although this constraint is specified in terms of concepts defined in Annex A, it also applies when the file formas 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.

Conformance of a coded image to the baseline profile is indicated by PROFILE\_IDC equal to 55. The value 55 for PROFILE\_IDC shall not be used in a PROFILE\_LEVEL\_INFO( ) syntax structure unless the associated coded image conforms to the constraints specified for the baseline profile.

Values of PROFILE\_IDC that are less than 55 may additionally be interpreted by decoders as expressing conformance of the associated coded image to the baseline profile.

* + 1. Main profile

The Main profile is defined as the set of coded image syntax that is consistent with the use of the following PIXEL\_FORMAT mnemonic values (specified in Table A.6): 24bppRGB, 24bppBGR, 32bppBGR, 48bppRGB, 48bppRGBFixedPoint, 48bppRGBHalf, 96bppRGBFixedPoint, 64bppRGBFixedPoint, 64bppRGBHalf, 128bppRGBFixedPoint, 128bppRGBFloat, 32bppBGRA, 64bppRGBA, 64bppRGBAFixedPoint, 64bppRGBAHalf, 128bppRGBAFixedPoint, 128bppRGBAFloat, 32bppPBGRA, 64bppPRGBA, 128bppPRGBAFloat, 32bppCMYK, 40bppCMYKAlpha, 64bppCMYK, 80bppCMYKAlpha, 24bpp3Channels, 32bpp4Channels, 40bpp5Channels, 48bpp6Channels, 56bpp7Channels, 64bpp8Channels, 32bpp3ChannelsAlpha, 40bpp4ChannelsAlpha, 48bpp5ChannelsAlpha, 56bpp6ChannelsAlpha, 64bpp7ChannelsAlpha, 72bpp8ChannelsAlpha, 48bpp3Channels, 64bpp4Channels, 80bpp5Channels, 96bpp6Channels, 112bpp7Channels, 128bpp8Channels, 64bpp3ChannelsAlpha, 80bpp4ChannelsAlpha, 96bpp5ChannelsAlpha, 112bpp6ChannelsAlpha, 128bpp7ChannelsAlpha, 144bpp8ChannelsAlpha, 8bppGray, 16bppGray, 16bppGrayFixedPoint, 16bppGrayHalf, 32bppGrayFixedPoint, 32bppGrayFloat, BlackWhite, 16bppBGR555, 16bppBGR565, 32bppBGR101010 or 32bppRGBE.

NOTE – Although this constraint is specified in terms of concepts defined in Annex A, it also applies when the file formas 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.

Conformance of a coded image to the Main profile is indicated by PROFILE\_IDC equal to 66. The value 66 for PROFILE\_IDC shall not be used in a PROFILE\_LEVEL\_INFO( ) syntax structure unless the associated coded image conforms to the constraints specified for the Main profile.

Values of PROFILE\_IDC that are less than 66 may additionally be interpreted by decoders as expressing conformance of the associated coded image to the Main profile.

* + 1. Advanced profile

The Advanced profile is defined as the set of all coded image syntax that is specified in this Specification.

NOTE – Some image codestreams that conform to the Advanced profile cannot be stored in a file as specified in Annex A due to the lack of specification of any PIXEL\_FORMAT value in Annex A that is consistent with the content of the image codestream. For example, a coded image with NumComponents equal to 32 cannot be stored in a file as specified in Annex A.

Conformance of a coded image to the Advanced profile is indicated by PROFILE\_IDC equal to 111. The value 111 for PROFILE\_IDC shall not be used in a PROFILE\_LEVEL\_INFO( ) syntax structure unless the associated coded image conforms to the constraints specified for the Advanced profile.

Values of PROFILE\_IDC that are less than 111 may additionally be interpreted by decoders as expressing conformance of the associated coded image to the Advanced profile.

* + 1. Reserved values of PROFILE\_IDC

The value of PROFILE\_IDC in a PROFILE\_LEVEL\_INFO( ) syntax structure shall not be equal to any value other than 44, 55, 66 or 111. Such other values are reserved for future use by ITU-T | ISO/IEC.

* 1. Levels

Conformance of a coded image to a particular level is indicated by LEVEL\_IDC equal to the value shown in the "Level" column of Table B.1.

For a value of LEVEL\_IDC specified by the "Level" column of Table B.1, the associated coded image shall obey the following constraints using values obtained from the corresponding row of Table B.1.

* WIDTH\_MINUS1 + LEFT\_MARGIN + RIGHT\_MARGIN shall be less than MaxImageDimension.
* HEIGHT\_MINUS1 + TOP\_MARGIN + BOTTOM\_MARGIN shall be less than MaxImageDimension.
* NUM\_HOR\_TILES\_MINUS1 shall be less than MaxDimensionInTiles.
* NUM\_VER\_TILES\_MINUS1 shall be less than MaxDimensionInTiles.
* TILE\_WIDTH\_IN\_MB[n] \* 16, when present, shall be less than MaxTileDimension for all values of n greater than or equal to 0 and less than NUM\_VER\_TILES\_MINUS1.
* (MBWidth – LeftMBIndexOfTile[NUM\_VER\_TILES\_MINUS1]) \* 16 shall be less than MaxTileDimension.
* TILE\_HEIGHT\_IN\_MB[n] \* 16, when present, shall be less than MaxTileDimension for all values of n greater than or equal to 0 and less than NUM\_HOR\_TILES\_MINUS1.
* (MBHeight – TopMBIndexOfTile[NUM\_HOR\_TILES\_MINUS1]) \* 16 shall be less than MaxTileDimension.
* When LEVEL\_IDC is not equal to 255, the value of ImageBufferBytes(valNC) as specified in Table B.2 shall be less than MaxBufferSizeInBytes, where the value of valNC is determined as follows.
* If LEVEL\_IDC is within a PROFILE\_LEVEL\_CONTAINER( ) syntax structure, valNC is set equal to the value in the "NC" column of Table A.6 for the corresponding value of PIXEL\_FORMAT.
* Otherwise, if ALPHA\_IMAGE\_PLANE\_FLAG is equal to FALSE, valNC is set equal to the value returned by DetermineNumComponents( ) for the primary image plane.
* Otherwise, valNC is set equal to the value returned by DetermineNumComponents( ) for the primary image plane plus 1.

When LEVEL\_IDC is less than or equal to 128, SHORT\_HEADER\_FLAG shall be equal to 1.

Table B.1 – Parameters of levels of conformance to this Specification

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Level** | **MaxImageDimension** | **MaxDimensionInTiles** | **MaxTileDimension** | **MaxBufferSizeInBytes** |
| 4 | 210 | 24 | 210 | 222 |
| 8 | 211 | 25 | 211 | 224 |
| 16 | 212 | 26 | 212 | 226 |
| 32 | 213 | 27 | 212 | 228 |
| 64 | 214 | 28 | 212 | 230 |
| 128 | 216 | 210 | 212 | 232 |
| 255 | 232 | 212 | 232 | Not applicable |

NOTE – When LEVEL\_IDC is equal to 255 and TILING\_FLAG is equal to TRUE, the number of bits used for the syntax elements TILE\_WIDTH\_IN\_MB[n] and TILE\_HEIGHT\_IN\_MB[n] cannot express a value greater than 216−1.

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

|  |  |
| --- | --- |
| **ImageBufferBytes(valNC)** { | **Reference** |
| numBytes = 0 |  |
| if (OUTPUT\_BITDEPTH = = BD8) |  |
| numBytes = 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 |  |
| } |  |

The value of LEVEL\_IDC in a PROFILE\_LEVEL\_INFO( ) syntax structure shall not be equal to any value other than a value specified in Table B.1. Such other values are reserved for future use by ITU-T | ISO/IEC. Any reserved value of LEVEL\_IDC that is lower than some value that is specified in Table B.1 shall be interpreted by decoders as expressing conformance of the associated image to the higher value of LEVEL\_IDC that is specified in Table B.1 (in the context of the associated value of PROFILE\_IDC).

1. Colour imagery representation and colour management

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

* 1. Background information

While it might be theoretically possible to agree on one method for assigning specific numerical values to real world colours, doing so is not practical. Since any specific device has its own limited range for colour reproduction, the device's range may be a small portion of the agreed-upon universal colour 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 colours or gamut. In addition, a colour profile is often provided that describes the numeric encoding for the specific device relative to some pre-defined reference standard. For example, a colour profile may include a specification of a non-linear transformation from the range of integer values to the luminance (or radiance) of the colour values as reproduced. This non-linear transformation is called the colour 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. Colour profiles deal with all aspects of the colour 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. Colour 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. Colour profiles make it possible to convert image data between different colour encodings, thereby controlling the capture and production of colours using a variety of devices. This science is known as "colour management".

Most common photo and image formats use an 8-bit or 16-bit unsigned integer value to represent some function of the intensity of each colour channel. The minimum value (0) often represents the minimum intensity in a single channel. The encoding of black is achieved in such usage when all channels are zero. The maximum value (2n − 1 for an unsigned bit depth of n bits) often represents 100% intensity or full saturation of an individual channel. When all channels have this maximal value, this typically corresponds to an encoding of white. Various alternative encoding conventions also exist, such as several colour representations specified by the ITU-R for which black may be represented by the value 16 and white may be represented by the value 235 for each colour channel.

The exact meaning of "black", "full saturation" and "white" as well as the specific colours produced by all the intermediate numerical steps in the numerical representation range is dependent on how these values are initially created or captured or how these values are rendered. Different source or destination devices (including cameras, scanners, displays or printers) may use different numerical values to represent the same "real world" colour.

The process of converting colours, and their numerical representation, to the ranges covered by different devices creates some significant complications. First, colours optimized for the gamut of the source device are re-optimized for the gamut of the destination device. This process involves some artistic judgment, as there is no single correct method. Then, the converted colours are encoded using the destination device encoding.

In addition, any intermediate image processing has the potential to produce values that extend beyond the black (low limit) or white (high limit) point of the particular colour encoding, often resulting in these calculated values being clipped to the associated limits. A working space colour encoding with a limited numerical range can cause values to be corrupted during intermediate calculations, even though subsequent image processing may bring these values back within the displayable (black to white) range of the target rendering destination. Because of this issue, many modern image processing software applications use a much larger gamut, including representation of numerical values of greater bit depths, for all intermediate image processing calculations. (It is not uncommon to use 32-bit floating point values for intermediate calculations, minimizing any image corruption caused by rounding errors and clipping at the range limits during the intermediate steps of image processing calculations.) However, most common image formats today require that this image data be converted back to a range-limited, unsigned integer representation, limited to the gamut as defined by a specific colour profile. So once again, the potential for data corruption exists.

* 1. Colour interpretation in the JPEG XR context

To address these challenges, JPEG XR provides a much more flexible approach to the numerical encoding of image data by supporting a wide range of different pixel formats. JPEG XR supports three basic types of numerical encoding, each at a variety of bit depths – specifically, these types are unsigned integer, fixed-point, and floating point representations.

A variety of pixel formats is supported by JPEG XR using unsigned integer representations in bit depths of 8 and 16, as well as smaller bit depths for specialized applications. JPEG XR also supports a number of additional pixel formats that avoid some problematic aspects involved in the unsigned integer representations.

ICC profile data as specified by ICC.1:2001-04 (ICC version 2.4.0.0) or ISO 15076-1:2010 (ICC version 4.3.0.0) can provide at least one unambiguous interpretation of the colours 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 15076-1:2010 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 15076-1:2010) transform should be used for interpretation of the colour imagery.

Unsigned integer RGB formats, as may be specified using the PIXEL\_FORMAT tag defined in A.7.18, are a common form of imagery representation. When present with unsigned integer pixel formats, embedded ICC Profile data provides a non-ambiguous definition of the colour interpretation of an image. The colour interpretation of JPEG XR image data should be defined either explicitly using embedded ICC profile data. When, for an unsigned pixel format, ICC profile data is not present, COLOR\_SPACE is not present or not equal to 1, PTM\_COLOR\_INFO( ) is not present, and colour interpretation information is not available by other means, unsigned integer RGB data should be assumed to use the sRGB colour encoding specified in IEC 61966-2-1.

Fixed point values are essentially signed, scaled integer values. Rather than interpreting the numbers as the integer step from the minimum limit to the maximum limit for a particular colour encoding, a fixed point number is scaled. In this fixed point colour space, zero could represent either the minimum device value or it could represent absolute black (zero photons). A value of 1.0 could represent the maximum device value or when applied to all channels that make up a pixel, it could represent a perfect diffuse white. The specific scaling for each bit depth specifies exactly what point in the entire signed integer range is interpreted as a value of 1.0. The type of black encoded at zero (e.g. the device black or zero photons), and the type of white encoded at 1.0 (e.g. some encoding maximum or a perfect diffuse white) are determined from the colour profile associated with the encoding.

Unlike unsigned integer values that are often interpreted based on a colour component transfer function (CCTF) that may be different for different colour encodings, it is preferred that the fixed point colour encodings used for the JPEG XR context should be based on a linear CCTF (equivalent to a gamma equal to 1.0).

Floating point numerical formats, as may be specified using the PIXEL\_FORMAT tag defined in A.7.18, should also be based on a linear CCTF (gamma = 1.0) colour encoding. ICC profiles embedded in images using a floating point pixel format as specified by PIXEL\_FORMAT in A.7.18 should therefore have a linear CCTF in the relative colorimetric rendering intent (0xA2B1 in ISO 15076-1:2010) transform.

Fixed point or floating point RGB data, as may be specified using the PIXEL\_FORMAT tag defined in A.7.18, should use the scRGB colour 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 colour space specified in ISO 11664-1). scRGB is a linear CCTF (gamma = 1.0) colour encoding that uses the same colour 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 colour channels set to a value of 1.0.

Since the colour rendering of scene colorimetry is often scene-specific, the perceptual intent transforms found in embedded scRGB-float ICC profiles may be image-specific. However, the use of a default perceptual rendering may be appropriate under some circumstances.

Devices and software receiving scRGB image data may also provide users with the opportunity to custom colour render scene-referred colorimetry. If such edits are to be saved, it is recommended that where possible they should be incorporated as changes to the perceptual rendering intent (0xA2B0 in ISO 15076-1:2010) transform in the embedded ICC profile, with the scRGB image data left unchanged.

For Gray pixel formats, as may be specified using the PIXEL\_FORMAT tag defined in A.7.18, in the absence of ICC profile data, the represented Gray value should be inferred as the value of all three colour components of the corresponding RGB format.

In the absence of an ICC profile, when no other indication of the colour interpretation is available by other means, any CMYK pixel format (as defined in A.7.18) should be assumed to be encoded according to ANSI CGATS/SWOP[[2]](#footnote-2) TR003-2007 CMYK.

NOTE 1 – A profile for ANSI CGATS/SWOP TR003-2007 CMYK is available from the ICC Profile Registry. The profile name in the registry is SWOP2006\_Coated3v2.icc.

There is no inherent description of the colour context for n-channel data, so when using an n-channel pixel format, it is recommended that an ICC profile should always be included (when applicable). However, if an ICC profile is not present and no other indication of the colour interpretation is available by other means, the following default colour context assumptions are recommended.

* When n = 3, the three channels should be assumed to be red, green, and blue (RGB) encoded using the sRGB colour space.
* When n > 3, the first four channels should be assumed to be CMYK encoded according to CGATS/SWOP TR003 2007 CMYK. Any additional channels should be discarded.

NOTE 2 – A profile for ANSI CGATS/SWOP TR003-2007 CMYK is available from the ICC Profile Registry. The profile name in the registry is SWOP2006\_Coated3v2.icc.

1. Encoder processing

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

* 1. Overview

The encoder carries out the following steps:

* pre-scaling;
* colour conversion;
* transform;
* outer pre-filter;
* outer forward core transform (FCT);
* inner pre-filter;
* inner FCT;
* coefficient prediction;
* quantization;
* coefficient scanning;
* entropy coding.

These steps are explained in D.2 to D.9.

* 1. Pre-scaling

The pre-scaling steps are applied for 16-bit unsigned integer (BD16), 16-bit signed integer (BD16S), 16-bit float (BD16F), 32-bit signed integer (BD32S), 32-bit float (BD32F) and RGBE. They are usually used when the input data range is greater than 27/24 bits. The 27-bit limit is used when data is scaled, and the 24-bit limit applies when the data is unscaled. For the most common cases such as for 16-bit data, the pre-scaling steps are omitted although they may still be used.

For OUTPUT\_BITDEPTH equal to BD16, BD16S or BD32S, input values are right-shifted by the value specified by SHIFT\_BITS. The function PreScaling1( ) is specified in Table D.1.

Table D.1 – Pseudocode for function PreScaling1( )

|  |  |
| --- | --- |
| **PreScaling1(iInput) {** | **Reference** |
| iOutput = (iInput >> SHIFT\_BITS) |  |
| return iOutput |  |
| } |  |

For OUTPUT\_BITDEPTH equal to BD16F, a sign bit extension is applied. The function PreScalingBD16F( ) is specified in Table D.2.

Table D.2 – Pseudocode for function PreScalingBD16F( )

|  |  |
| --- | --- |
| **PreScalingBD16F(fV) {** | **Reference** |
| if (fV < 0) |  |
| iS = 1 |  |
| else |  |
| iS = 0 |  |
| iX = (fV & 0x7fff) ^ iS − iS |  |
| return iX |  |
| } |  |

For OUTPUT\_BITDEPTH equal to BD32F, the encoder first selects a value of mantissa length LEN\_MANTISSA, and an exponent bias EXP\_BIAS. LEN\_MANTISSA is less than or equal to 23. The function PreScalingBD32F( ) is specified in Table D.3.

Table D.3 – Pseudocode for function PreScalingBD32F( )

|  |  |
| --- | --- |
| **PreScalingBD32F(fV)** { | Reference |
| if (fV = = 0) |  |
| iX = 0 |  |
| else { |  |
| iS = (fV >> 31) & 0x00000001 |  |
| iE = (fV >> 23) & 0x000000FF |  |
| iM = (fV & 0x007FFFFF) | 0x800000 |  |
| if (iE = = 0) { |  |
| iM ^= 0x800000 |  |
| iE++ |  |
| } |  |
| iE = iE − 127 + EXP\_BIAS |  |
| if (iE <= 1) { |  |
| if (iE < 1) |  |
| iM >>= (1 − iE) |  |
| iE = 1 |  |
| if ((iM & 0x800000) = = 0) |  |
| iE = 0 |  |
| } |  |
| iM = (((iM & 0x007FFFFF) + (1 << (23 − LEN\_MANTISSA − 1))) >> (23 − LEN\_MANTISSA)) |  |
| iX = ((iE << LEN\_MANTISSA) | iM) ^ iS − iS |  |
| } |  |
| return iX |  |
| } |  |

For OUTPUT\_CLR\_FMT equal to RGBE, the function PreScalingRGBE( ) is specified in Table D.4 and Table D.5.

Table D.4 – Pseudocode for function PreScalingRGBE( )

|  |  |
| --- | --- |
| **PreScalingRGBE(arrayOut[ ], arrayIn[ ]) {** | **Reference** |
| arrayOut[0] = RGBEForwardConversion(arrayIn[0], arrayIn[3]) |  |
| arrayOut[1] = RGBEForwardConversion(arrayIn[1], arrayIn[3]) |  |
| arrayOut[2] = RGBEForwardConversion(arrayIn[2], arrayIn[3]) |  |
| **}** |  |

Table D.5 – Pseudocode for function RGBEForwardConversion( )

|  |  |
| --- | --- |
| **RGBEForwardConversion(iRGB, iE) {** | **Reference** |
| if (iE = = 0) |  |
| return 0 |  |
| iAppend = 1 |  |
| iE −= 1 |  |
| while (((iRGB & 0x80) = = 0) && (iE > 0)) { |  |
| iRGB = (iRGB << 1) + iAppend |  |
| iAppend = 0 |  |
| iE −= 1 |  |
| } |  |
| if (iE = = 0) |  |
| iX = iRGB |  |
| else { |  |
| iE++ |  |
| iX = (iRGB & 0x7f) + (iE << 7) |  |
| } |  |
| return iX |  |
| } |  |

* 1. Colour conversion
     1. General

The encoder uses a reversible colour format conversion to convert between the OUTPUT\_CLR\_FMT and INTERNAL\_CLR\_FMT.

RGB to YUV444 colour format conversion is performed using FwdColorFmtConvert1( ) that is specified in D.3.2. In order to convert from RGB to YUV422 or YUV420, downsampling must be performed after colour 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 colour format conversion is performed using FwdColorFmtConvert2( ) that is specified in D.3.3. CMYKDIRECT to YUVK colour format conversion is performed using FwdColorFmtConvert3( ) that is specified in D.3.4.

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

* + 1. FwdColorFmtConvert1( )

The function FwdColorFmtConvert1( ) implements the following operations to convert from RGB to YUV444 in Table D.6. When the decoded image is intended to be formatted using OUTPUT\_CLR\_FMT equal to RGB and OUTPUT\_BITDEPTH equal to BD5, BD565 or BD10, the process depends on the selected behaviour for the decoder with respect to the RED\_BLUE\_NOT\_SWAPPED\_FLAG syntax element.

NOTE – The specification of semantics for RED\_BLUE\_NOT\_SWAPPED\_FLAG was not included in the original edition of this Specification. The specification of RED\_BLUE\_NOT\_SWAPPED\_FLAG was added later to address a problem with respect to the observed behaviour of existing implementations. In principle, when OUTPUT\_CLR\_FMT is equal to RGB and OUTPUT\_BITDEPTH is equal to BD5, BD565, or BD10, the value 1 for RED\_BLUE\_NOT\_SWAPPED\_FLAG should ordinarily provide better compression capability than the value 0. However, early product implementations of this Specification have operated in a manner corresponding to the value 0. Encoders that are designed primarily for interoperability with early product implementations of this Specification should therefore use the value 0 for RED\_BLUE\_NOT\_SWAPPED\_FLAG when OUTPUT\_CLR\_FMT is equal to RGB and OUTPUT\_BITDEPTH is equal to BD5, BD565 or BD10. It is anticipated that, as time passes, decoder implementations will also properly support operation with the value 1 for RED\_BLUE\_NOT\_SWAPPED\_FLAG when OUTPUT\_CLR\_FMT is equal to RGB and OUTPUT\_BITDEPTH is equal to BD5, BD565 or BD10.

Table D.6 – Pseudocode for function FwdColorFmtConvert1( )

|  |  |
| --- | --- |
| **FwdColorFmtConvert1(arrayIn[ ], arrayOut[ ])** { | **Reference** |
| /\* arrayIn[ ] = {R, G, B} \*/ |  |
| /\* arrayOut[ ] = {Y, U, V} \*/ |  |
| if ((OUTPUT\_CLR\_FMT = = RGB) &&  (OUTPUT\_BITDEPTH = = BD5 | | OUTPUT\_BITDEPTH = = BD565 | |  OUTPUT\_BITDEPTH = = BD10) && !RED\_BLUE\_NOT\_SWAPPED\_FLAG) { |  |
| /\* swappedRBflag = 1 \*/ |  |
| arrayTemp[0] = arrayIn[2] |  |
| arrayTemp[1] = arrayIn[1] |  |
| arrayTemp[2] = arrayIn[0] |  |
| } else { |  |
| /\* swappedRBflag = 0 \*/ |  |
| arrayTemp[0] = arrayIn[0] |  |
| arrayTemp[1] = arrayIn[1] |  |
| arrayTemp[2] = arrayIn[2] |  |
| } |  |
| arrayOut[2] = arrayTemp[2] − arrayTemp[0]  /\* V = B − R, except when swappedRBflag is equal to 1 \*/ |  |
| tempT = arrayTemp[0] − arrayTemp[1] + Ceiling(arrayOut[2] ÷ 2)  /\* t = R − G + Ceiling(V ÷ 2), except when swappedRBflag is equal to 1 \*/ |  |
| arrayOut[0] = arrayTemp[1] + Floor(tempT ÷ 2)  /\* Y = G + Floor(t ÷ 2) \*/ |  |
| arrayOut[1] = −tempT /\* U = −t \*/ |  |
| } |  |

* + 1. 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) \*/ |  |
| } |  |

* + 1. FwdColorFmtConvert3( )

The function FwdColorConvert3( ) implements the following operations to convert from CMYKDIRECT to YUVK in Table D.8.

Table D.8 – Pseudocode for function FwdColorFmtConvert3( )

|  |  |
| --- | --- |
| **FwdColorFmtConvert3(arrayIn[ ], arrayOut[ ])** { | **Reference** |
| /\* arrayIn[ ] = {c, m, y, k} \*/ |  |
| /\* arrayOut[ ] = {Y, U, V, K} \*/ |  |
| arrayOut[2] = arrayIn[1] /\* V = m \*/ |  |
| arrayOut[1] = arrayIn[0] /\* U = c \*/ |  |
| arrayOut[0] = arrayIn[3] /\* Y = k \*/ |  |
| arrayOut[3] = arrayIn[2] /\* K = y \*/ |  |
| } |  |

* + 1. Macroblock alignment and padding

The width and height of images are not always integer multiples of 16. When an image width or height is not an integer multiple of 16, the image is extended by some means so that the extended image has dimensions that are multiples of 16.

A suggested method of image extension is to perform horizontal replication of the bottom and right edge sample values to the extent necessary to form an input array having a conforming (multiple of 16) extended image width and height.

* 1. Transform

The encoder uses a two level lapped transform, which is structured as follows:

* A pre-filter operation is optionally applied to 4×4 areas evenly straddling blocks in two dimensions. For images with soft tiles, this filter is applied to all such blocks. For images with hard tiles, this filter is applied only to the interior of tiles. Furthermore, a pre-filter is applied to boundary areas which are 2×4 or 4×2 in size, and a pre-filter is applied to the four 2×2 corners on each colour component. For images with hard tiles, these filters are additionally applied at tile boundaries.
* An FCT is applied to all 4×4 blocks. This completes the first transform level.
* DC coefficients of the 4×4 blocks are grouped together. 16 Such coefficients are present for each macroblock colour component, and these are structured as 4×4 blocks in the DC-LP array.
* A second level pre-filter operation is optionally applied to 4×4 areas evenly straddling blocks in two dimensions in the DC-LP array. For images with soft tiles, this filter is applied to all such blocks. For images with hard tiles, this filter is applied only to the interior of tiles. Furthermore, a pre-filter is applied to boundary 2×4 and 4×2 areas, and a pre-filter is applied to the four 2×2 corner areas. For images with hard tiles, these filters are additionally applied at tile boundaries.
* The FCT is applied to each 4×4 block corresponding to DC coefficients in a macroblock colour component.

The pre-filtering operation is applied across macroblock boundaries and optionally across tile boundaries. If HARD\_TILING\_FLAG is set equal to FALSE, the pre-filtering is applied across tile boundaries. Otherwise, the pre-filtering operation is not applied across tile boundaries and tile boundaries are treated as image edges instead.

These operations are repeated for all colour components. For the special cases of INTERNAL\_CLR\_FMT equal to YUV422 or YUV420, appropriately modified transforms are applied (for instance a 2×2 transform is used as the block transform of the chroma component DC-LP arrays of YUV420).

"OverlapPreFilter1" in Figure D.1 is used to denote the corner prediction process, and the process is equivalent to "OverlapPostFilter1" detailed in Table 154 and Table 155.

An overview of this process is shown in Figure D.1.



Figure D.1 – Encoder transform

The application of overlap filters is governed by the OVERLAP\_MODE syntax element. Lossless coding is possible with all overlap modes, although OVERLAP\_MODE equal to 0 is usually best or sufficient for lossless coding. OVERLAP\_MODE equal to 1 produces the shortest codestream for a large class of images and quantization levels. OVERLAP\_MODE equal to 2 is recommended for high quantization levels, but with slightly higher complexity. OVERLAP\_MODE equal to 0 is recommended for lowest complexity encoding/decoding; however, this mode implements a hierarchical block transform and potentially introduces blocking at low bit rates.

The forward core transform (FCT) process can be written as the cascade of three elementary 2×2 transform operations, which are:

1. 2×2 Hadamard transform: T2x2h( )
2. 1D rotate: TOdd( )
3. 2D rotate: TOddOdd( )
   * 1. T2x2h( )

As specified in 9.9.7.2.

* + 1. TOdd( )

TOdd( ) is specified by the pseudocode in Table D.9.

Table D.9 – Pseudocode for function TOdd( )

|  |
| --- |
| **TOdd(iCoeff[ ])** { |
| iCoeff[1] −= iCoeff[2] |
| iCoeff[0] += iCoeff[3] |
| iCoeff[2] += ((iCoeff[1] + 1) >> 1) |
| iCoeff[3] = ((iCoeff[0] + 1) >> 1) − iCoeff[3] |
| iCoeff[1] −= ((3\* iCoeff[0] + 4) >> 3) |
| iCoeff[0] += ((3\* iCoeff[1] + 4) >> 3) |
| iCoeff[3] −= ((3\* iCoeff[2] + 4) >> 3) |
| iCoeff[2] += ((3\* iCoeff[3] + 4) >> 3) |
| iCoeff[3] += (iCoeff[1] >> 1) |
| iCoeff[2] −= ((iCoeff[0] + 1) >> 1) |
| iCoeff[1] −= iCoeff[3] |
| iCoeff[0] += iCoeff[2] |
| } |

* + 1. TOddOdd( )

TOddOdd( ) is specified by the pseudocode in Table D.10.

Table D.10 – Pseudocode for function TOddOdd( )

|  |
| --- |
| **TOddOdd(iCoeff[ ])** { |
| iCoeff[1] = −iCoeff[1] |
| iCoeff[2] = −iCoeff[2] |
| iCoeff[3] += iCoeff[0] |
| iCoeff[2] −= iCoeff[1] |
| iCoeff[0] −= (valT1 = iCoeff[3] >> 1) |
| iCoeff[1] += (valT2 = iCoeff[2] >> 1) |
| iCoeff[0] += ((iCoeff[1] \* 3 + 4) >> 3) |
| iCoeff[1] −= ((iCoeff[0] \* 3 + 3) >> 2) |
| iCoeff[0] += ((iCoeff[1] \* 3 + 3) >> 3) |
| iCoeff[1] −= valT2 |
| iCoeff[0] += valT1 |
| iCoeff[2] += iCoeff[1] |
| iCoeff[3] −= iCoeff[0] |
| } |

* + 1. FwdPermute( )

The forward permutation function FwdPermute( ) operates on an ordered array of 16 sample values, producing a permuted list. The input to this function is the ordered array arrayInput[i], for i ranging from 0 to 15. The output of this function is the re-ordered array arrayInput[i].

To define the permutation, the array FwdPermArr[i] is specified, for i ranging from 0 to 15, in Table D.11.

Table D.11 – Forward Permutation

|  |  |
| --- | --- |
| **i** | **FwdPermArr[i]** |
| 0 | 0 |
| 1 | 8 |
| 2 | 4 |
| 3 | 6 |
| 4 | 2 |
| 5 | 10 |
| 6 | 14 |
| 7 | 12 |
| 8 | 1 |
| 9 | 11 |
| 10 | 15 |
| 11 | 13 |
| 12 | 9 |
| 13 | 3 |
| 14 | 7 |
| 15 | 5 |

The function FwdPermute( ) is specified in Table D.12.

Table D.12 – Pseudocode for function FwdPermute( )

|  |  |
| --- | --- |
| **FwdPermute(arrayInput[ ]) {** | **Reference** |
| for (i = 0; i <= 15; i++) |  |
| arrayTemp[FwdPermArr[i]] = arrayInput[i] |  |
| for (i = 0; i <= 15; i++) |  |
| arrayInput[i] = arrayTemp[i] |  |
| } |  |

* + 1. FCT operations
       1. FCT4x4( )

The input to the FCT4x4 function is an array of length 16, denoted iCoeff[ ]. The output of this function is a modified array of length 16 denoted iCoeff[ ].

The FCT4x4( ) is built using the three operators: T2x2h( ), TOdd( ) and TOddOdd( ), followed by the permutation function FwdPermute( ). The FCT begins with two transform stages. Each stage consists of four 2×2 transforms which may be done in any arbitrary sequence or concurrently within the stage. However, the first stage transforms must be complete before any of the second stage transforms are initiated. Also, for each function called in the FCT, the inverse of that function is called in the ICT pseudocode (9.9.7.1), with the exact same set of parameters.

When these two transform stages are complete, the coefficients need to be re-ordered by the FwdPermute( ) function.

The FCT4x4( ) function is specified by the pseudocode in Table D.13.

Table D.13 – Pseudocode for function FCT4x4( )

| **FCT4x4(iCoeff[ ])** { | **Reference** |
| --- | --- |
| /\* First stage \*/ |  |
| arrayLocal[ ] = {iCoeff[0], iCoeff[3], iCoeff[12], iCoeff[15]} |  |
| T2x2h(arrayLocal[ ], 0) | 9.9.7.2 |
| iCoeff[0] = arrayLocal[0] |  |
| iCoeff[3] = arrayLocal[1] |  |
| iCoeff[12] = arrayLocal[2] |  |
| iCoeff[15] = arrayLocal[3] |  |
| arrayLocal[ ] = {iCoeff[5], iCoeff[6], iCoeff[9], iCoeff[10]} |  |
| T2x2h(arrayLocal[ ], 0) | 9.9.7.2 |
| iCoeff[5] = arrayLocal[0] |  |
| iCoeff[6] = arrayLocal[1] |  |
| iCoeff[9] = arrayLocal[2] |  |
| iCoeff[10] = arrayLocal[3] |  |
| arrayLocal[ ] = {iCoeff[1], iCoeff[2], iCoeff[13], iCoeff[14]} |  |
| T2x2h(arrayLocal[ ], 0) | 9.9.7.2 |
| iCoeff[1] = arrayLocal[0] |  |
| iCoeff[2] = arrayLocal[1] |  |
| iCoeff[13] = arrayLocal[2] |  |
| iCoeff[14] = arrayLocal[3] |  |
| arrayLocal[ ] = {iCoeff[4], iCoeff[7], iCoeff[8], iCoeff[11] } |  |
| T2x2h(arrayLocal[ ], 0) | 9.9.7.2 |
| iCoeff[4] = arrayLocal[0] |  |
| iCoeff[7] = arrayLocal[1] |  |
| iCoeff[8] = arrayLocal[2] |  |
| iCoeff[11] = arrayLocal[3] |  |
| /\* Second stage \*/ |  |
| arrayLocal[ ] = {iCoeff[0], iCoeff[1], iCoeff[4], iCoeff[5]} |  |
| T2x2h(arrayLocal[ ], 1) | 9.9.7.2 |
| iCoeff[0] = arrayLocal[0] |  |
| iCoeff[1] = arrayLocal[1] |  |
| iCoeff[4] = arrayLocal[2] |  |
| iCoeff[5] = arrayLocal[3] |  |
| arrayLocal[ ] = {iCoeff[2], iCoeff[3], iCoeff[6], iCoeff[7]} |  |
| TOdd(arrayLocal[ ]) | D.4.3 |
| iCoeff[2] = arrayLocal[0] |  |
| iCoeff[3] = arrayLocal[1] |  |
| iCoeff[6] = arrayLocal[2] |  |
| iCoeff[7] = arrayLocal[3] |  |
| arrayLocal[ ] = {iCoeff[8], iCoeff[12], iCoeff[9], iCoeff[13]} |  |
| TOdd(arrayLocal[ ]) | D.4.3 |
| iCoeff[8] = arrayLocal[0] |  |
| iCoeff[12] = arrayLocal[1] |  |
| iCoeff[9] = arrayLocal[2] |  |
| iCoeff[13] = arrayLocal[3] |  |
| arrayLocal[ ] = {iCoeff[10], iCoeff[11], iCoeff[14], iCoeff[15]} |  |
| TOddOdd(arrayLocal[ ]) | D.4.3 |
| iCoeff[10] = arrayLocal[0] |  |
| iCoeff[11] = arrayLocal[1] |  |
| iCoeff[14] = arrayLocal[2] |  |
| iCoeff[15] = arrayLocal[3] |  |
| /\* Permute the coefficients \*/ |  |
| FwdPermute(iCoeff[ ]) | D.4.4 |
| } |  |

* + - 1. FCT2x2( )

The function FCT2x2( ) is identical to the T2x2h( ) function specified in Table 161.

* 1. Overlap pre-filtering
     1. Overview

Four operators determine the overlap pre-filters used in the transform. These are:

1. Overlap Prefilter4x4
2. OverlapPrefilter4
3. OverlapPreFilter2x2
4. OverlapPreFilter2

The pre-filter makes use of the operators T2x2hEnc( ), FwdRotate( ), FwdScale( ) and FwdTOddOdd( ) that are specified in Table D.14, Table D.15, Table D.16, and Table D.17, respectively.

* + - 1. T2x2hEnc( )

T2x2hEnc( ) is specified by the pseudocode in Table D.14.

Table D.14 – Pseudocode for function T2x2hEnc( )

|  |
| --- |
| **T2x2hEnc(iCoeff[ ])** { |
| iCoeff[0] += iCoeff[3] |
| iCoeff[1] −= iCoeff[2] |
| valT1 = iCoeff[3] |
| valT2 = iCoeff[2] |
| iCoeff[2] = ((iCoeff[0] − iCoeff[1]) >> 1) − valT1 |
| iCoeff[3] = valT2 + (iCoeff[1] >> 1) |
| iCoeff[1] += iCoeff[2] |
| iCoeff[0] −= (iCoeff[3] \* 3 + 4) >> 3 |
| } |

* + - 1. FwdRotate( )

FwdRotate( ) is specified by the pseudocode in Table D.15.

Table D.15 – Pseudocode for function FwdRotate( )

|  |
| --- |
| **FwdRotate(iCoeff[ ])** { |
| iCoeff[1] −= (iCoeff[0] + 1) >> 1 |
| iCoeff[0] += (iCoeff[1] + 1) >> 1 |
| } |

* + - 1. FwdScale( )

FwdScale( ) is specified by the pseudocode in Table D.16.

Table D.16 – Pseudocode for function FwdScale( )

|  |
| --- |
| **FwdScale(iCoeff[ ])** { |
| iCoeff[1] −= (iCoeff[0] \* 3 + 0) >> 4 |
| iCoeff[1] −= (iCoeff[0] >> 7) |
| iCoeff[1] += (iCoeff[0] >> 10) |
| iCoeff[0] −= (iCoeff[1] \* 3 + 0) >> 3 |
| iCoeff[1] = (iCoeff[0] >> 1) − iCoeff[1] |
| iCoeff[0] −= iCoeff[1] |
| } |

* + - 1. FwdTOddOdd( )

FwdTOddOdd( ) is specified by the pseudocode in Table D.17.

Table D.17 – Pseudocode for function FwdTOddOdd( )

|  |
| --- |
| **FwdTOddOdd(iCoeff[ ])** { |
| iCoeff[3] += iCoeff[0] |
| iCoeff[2] −= iCoeff[1] |
| valT1 = iCoeff[3] >> 1 |
| valT2 = iCoeff[2] >> 1 |
| iCoeff[0] −= valT1 |
| iCoeff[1] += valT2 |
| iCoeff[0] += (iCoeff[1] \* 3 + 4) >>3 |
| iCoeff[1] −= (iCoeff[0] \* 3 + 2) >> 2 |
| iCoeff[0] += (iCoeff[1] \* 3 + 6) >> 3 |
| iCoeff[1] −= valT2 |
| iCoeff[0] += valT1 |
| iCoeff[2] += iCoeff[1] |
| iCoeff[3] −= iCoeff[0] |
| } |

* + - 1. OverlapPreFilter4x4( )

The 4×4 pre-filter is applied to all block junctions (areas straddling 4 blocks evenly) in all colour components when OVERLAP\_MODE is 1 or 2. Also, the 4×4 filter is applied to all block junctions in the DC-LP array for all planes when OVERLAP\_MODE is 2, and for only the luma plane when OVERLAP\_MODE is 2 and INTERNAL\_CLR\_FMT is equal to YUV420 or YUV422.

The function OverlapPreFilter4x4( ) is specified in Table D.18.

Table D.18 – Pseudocode for function OverlapPreFilter4x4( )

|  |  |
| --- | --- |
| **OverlapPreFilter4x4(iCoeff[ ])** { | **Reference** |
| arrayLocal[ ] = {iCoeff[0], iCoeff[3], iCoeff[12], iCoeff[15]} |  |
| T2x2hEnc(arrayLocal[ ]) | D.5.1.1 |
| iCoeff[0] = localArray[0] |  |
| iCoeff[3] = localArray[1] |  |
| iCoeff[12] = localArray[2] |  |
| iCoeff[15] = localArray[3] |  |
| arrayLocal[ ] = {iCoeff[1], iCoeff[2], iCoeff[13], iCoeff[14]} |  |
| T2x2hEnc(arrayLocal[ ]) | D.5.1.1 |
| iCoeff[1] = localArray[0] |  |
| iCoeff[2] = localArray[1] |  |
| iCoeff[13] = localArray[2] |  |
| iCoeff[14] = localArray[3] |  |
| arrayLocal[ ] = {iCoeff[4], iCoeff[7], iCoeff[8], iCoeff[11]} |  |
| T2x2hEnc(arrayLocal[ ]) | D.5.1.1 |
| iCoeff[4] = localArray[0] |  |
| iCoeff[7] = localArray[1] |  |
| iCoeff[8] = localArray[2] |  |
| iCoeff[11] = localArray[3] |  |
| arrayLocal[ ] = {iCoeff[5], iCoeff[6], iCoeff[9], iCoeff[10]} |  |
| T2x2hEnc(arrayLocal[ ]) | D.5.1.1 |
| iCoeff[5] = localArray[0] |  |
| iCoeff[6] = localArray[1] |  |
| iCoeff[9] = localArray[2] |  |
| iCoeff[10] = localArray[3] |  |
| arrayLocal[ ] = {iCoeff[0], iCoeff[15]} |  |
| FwdScale(arrayLocal[ ]) | D.5.1.3 |
| iCoeff[0] = localArray[0] |  |
| iCoeff[15] = localArray[1] |  |
| arrayLocal[ ] = {iCoeff[1], iCoeff[14]} |  |
| FwdScale(arrayLocal[ ]) | D.5.1.3 |
| iCoeff[1] = localArray[0] |  |
| iCoeff[14] = localArray[1] |  |
| arrayLocal[ ] = {iCoeff[4], iCoeff[11]} |  |
| FwdScale(arrayLocal[ ]) | D.5.1.3 |
| iCoeff[4] = localArray[0] |  |
| iCoeff[11] = localArray[1] |  |
| arrayLocal[ ] = {iCoeff[5], iCoeff[10]} |  |
| FwdScale(arrayLocal[ ]) | D.5.1.3 |
| iCoeff[5] = localArray[0] |  |
| iCoeff[10] = localArray[1] |  |
| arrayLocal[ ] = {iCoeff[13], iCoeff[12]} |  |
| FwdRotate(arrayLocal[ ]) | D.5.1.2 |
| iCoeff[13] = localArray[0] |  |
| iCoeff[12] = localArray[1] |  |
| arrayLocal[ ] = {iCoeff[9], iCoeff[8]} |  |
| FwdRotate(arrayLocal[ ]) | D.5.1.2 |
| iCoeff[9] = localArray[0] |  |
| iCoeff[8] = localArray[1] |  |
| arrayLocal[ ] = {iCoeff[7], iCoeff[3]} |  |
| FwdRotate(arrayLocal[ ]) | D.5.1.2 |
| iCoeff[7] = localArray[0] |  |
| iCoeff[3] = localArray[1] |  |
| arrayLocal[ ] = {iCoeff[6], iCoeff[2]} |  |
| FwdRotate(arrayLocal[ ]) | D.5.1.2 |
| iCoeff[6] = localArray[0] |  |
| iCoeff[2] = localArray[1] |  |
| arrayLocal[ ] = {iCoeff[10], iCoeff[11], iCoeff[14], iCoeff[15]} |  |
| FwdTOddOdd(arrayLocal[ ]) | D.5.1.4 |
| iCoeff[10] = localArray[0] |  |
| iCoeff[11] = localArray[1] |  |
| iCoeff[14] = localArray[2] |  |
| iCoeff[15] = localArray[3] |  |
| arrayLocal[ ] = {iCoeff[0], iCoeff[12], iCoeff[3], iCoeff[15]} |  |
| T2x2h(arrayLocal[ ], 0) | 9.9.7.2 |
| iCoeff[0] = localArray[0] |  |
| iCoeff[12] = localArray[1] |  |
| iCoeff[3] = localArray[2] |  |
| iCoeff[15] = localArray[3] |  |
| arrayLocal[ ] = {iCoeff[1], iCoeff[2], iCoeff[13], iCoeff[14]} |  |
| T2x2h(arrayLocal[ ], 0) | 9.9.7.2 |
| iCoeff[1] = localArray[0] |  |
| iCoeff[2] = localArray[1] |  |
| iCoeff[13] = localArray[2] |  |
| iCoeff[14] = localArray[3] |  |
| arrayLocal[ ] = {iCoeff[4], iCoeff[7], iCoeff[8], iCoeff[11]} |  |
| T2x2h(arrayLocal[ ], 0) | 9.9.7.2 |
| iCoeff[4] = localArray[0] |  |
| iCoeff[7] = localArray[1] |  |
| iCoeff[8] = localArray[2] |  |
| iCoeff[11] = localArray[3] |  |
| arrayLocal[ ] = {iCoeff[5], iCoeff[6], iCoeff[9], iCoeff[10]} |  |
| T2x2h(arrayLocal[ ], 0) | 9.9.7.2 |
| iCoeff[5] = localArray[0] |  |
| iCoeff[6] = localArray[1] |  |
| iCoeff[9] = localArray[2] |  |
| iCoeff[10] = localArray[3] |  |
| } |  |

* + 1. OverlapPreFilter4( )

Linear 4-point filters are applied to edge straddling 2×4 and 4×2 areas on the boundary of the image. If the input data is an array of length 4, denoted iCoeff[ ], the 4-point pre-filter, OverlapPreFilter4( ), is specified in Table D.19.

Table D.19 – Pseudocode for function OverlapPreFilter4( )

|  |  |
| --- | --- |
| **OverlapPreFilter4(iCoeff[ ])** { | **Reference** |
| iCoeff[0] += iCoeff[3] |  |
| iCoeff[1] += iCoeff[2] |  |
| iCoeff[3] −= ((iCoeff[0] + 1) >> 1) |  |
| iCoeff[2] −= ((iCoeff[1] + 1) >> 1) |  |
| FwdRotate(iCoeff[2], iCoeff[3]) | D.5.1.2 |
| iCoeff[3] = −iCoeff[3] |  |
| iCoeff[2] = −iCoeff[2] |  |
| iCoeff[0] −= iCoeff[3] |  |
| iCoeff[1] −= iCoeff[2] |  |
| iCoeff[3] += ( iCoeff[0] >> 1) |  |
| iCoeff[2] += ( iCoeff[1] >> 1) |  |
| iCoeff[0] −= ((iCoeff[3] \* 3 + 4) >> 3) |  |
| iCoeff[1] −= ((iCoeff[2] \* 3 + 4) >> 3) |  |
| FwdScale(iCoeff[0], iCoeff[3]) | D.5.1.3 |
| FwdScale(iCoeff[1], iCoeff[2]) | D.5.1.3 |
| iCoeff[3] += ((iCoeff[0] + 1) >> 1) |  |
| iCoeff[2] += ((iCoeff[1] + 1) >> 1) |  |
| iCoeff[0] −= iCoeff[3] |  |
| iCoeff[1] −= iCoeff[2] |  |
| } |  |

* + 1. OverlapPrefilter2x2

The 2×2 pre-filter is applied to areas straddling blocks in the DC-LP array for the chroma components when INTERNAL\_CLR\_FMT is equal to YUV420 or YUV422 data. The OverlapPreFilter2x2( ) function is specified in Table D.20.

Table D.20 – Pseudocode for function OverlapPreFilter2x2( )

|  |
| --- |
| **OverlapPreFilter2x2(iCoeff[ ])** { |
| iCoeff[0] += iCoeff[3] |
| iCoeff[1] += iCoeff[2] |
| iCoeff[3] −= ((iCoeff[0] + 1) >> 1) |
| iCoeff[2] −= ((iCoeff[1] + 1) >> 1) |
| iCoeff[1] −= ((iCoeff[0] + 2) >> 2) |
| iCoeff[0] −= (iCoeff[1] >> 5) |
| iCoeff[0] −= (iCoeff[1] >> 9) |
| iCoeff[0] −= (iCoeff[1] >> 13) |
| iCoeff[0] −= ((iCoeff[1] + 1) >> 1) |
| iCoeff[1] −= ((iCoeff[0] + 2) >> 2) |
| iCoeff[3] += ((iCoeff[0] + 1) >> 1) |
| iCoeff[2] += ((iCoeff[1] + 1) >> 1) |
| iCoeff[0] −= iCoeff[3] |
| iCoeff[1] −= iCoeff[2] |
| } |

* + 1. OverlapPreFilter2

As for the 2-point filters, the 2-point pre-filter is applied to boundary 2×1 and 1×2 samples that straddle blocks. The 2‑point pre-filter function OverlapPreFilter2( ) is specified in Table D.21.

Table D.21 – Pseudocode for function OverlapPreFilter2( )

|  |
| --- |
| **OverlapPreFilter2(iCoeff[ ])** { |
| iCoeff[1] −= ((iCoeff[0] + 2) >> 2) |
| iCoeff[0] −= (iCoeff[1] >> 13) |
| iCoeff[0] −= (iCoeff[1] >> 9) |
| iCoeff[0] −= (iCoeff[1] >> 5) |
| iCoeff[0] −= ((iCoeff[1] + 1) >> 1) |
| iCoeff[1] −= ((iCoeff[0] + 2) >> 2) |
| } |

* 1. Coefficient prediction

The coefficient prediction process is the same as the corresponding process in the decoder. See 9.6.

* 1. Coefficient scanning

The coefficient scanning process is the same as the corresponding process in the decoder.

* 1. Quantization

Quantization is a process whereby the transform coefficients are divided by the quantization scaling factor and rounded to an integer value, called the *quantized value*.

For lossless coding, the quantization scaling factor is selected to be equal to 1. For lossy coding, the quantization scaling factor is selected to be greater than 1.

The quantization scaling factor used for quantization is determined as a function of the quantization factor in the same manner as for the decoding process. Larger values of the quantization parameter correspond to larger quantization scaling factors and (typically) correspond to lower fidelity representation of the image content.

On the encoder side, the specific rounding method used in the process of quantization is implementation specific, and is not covered in this Specification.

The quantization parameter is allowed to differ across HP, LP and DC bands. The DC QP index within a tile is fixed. The DC QP index across tiles may vary. The HP and LP QP indices within a tile may take on either the same value or one of a multiple value set. The QP index used may change at every macroblock, and is specified in the codestream.

* 1. Adaptive coefficient normalization

Adaptive coefficient normalization partitions the transform coefficients into a VLC-coded part and an FLC part (called FLEXBITS). The VLC-coded part is obtained from a coarse quantization of the coefficient referred to as the normalized coefficient, while the FLEXBITS represents the LSBs and provides a refinement layer over the VLC-coded part. Sign information may also be present as part of FLEXBITS when the normalized coefficient is equal to 0. The partition between the VLC coded and the FLC coded parts is performed as follows.

1. The functions MB\_HP( ) (8.7.18.2) and MB\_HP\_FLEX( ) (8.7.18.3) specify how iLapMean[0] and iLapMean[1] are used to compute the total number of non-zero coefficients that have been coded for the luma and chroma components, respectively.
2. The UpdateModelMB( ) function (8.12.2) specifies how this information is used to compute **Model**.MBits[0] and **Model**.MBits[1] which are the number of LSBs that form the FLEXBITS portion of the luma and chroma components, respectively. The more significant bits are VLC coded.

In effect, the number of non-zero coded coefficients is used as a measure of the statistical variance of the block. Thus, a larger number of non-zero coded coefficients in a block produces higher values of iLapMean[ ], and thus result in FLEXBITS forming a larger fraction of the partition. As a result of this partition, entropy coding is applied only to the normalized coefficients with reduced dynamic range, and thus reduces the computational and memory complexity of entropy coding at both encoder and decoder. An overview of the adaptive coefficient normalization process at the encoder is shown in Figure D.2.



Figure D.2 – Representation of the adaptive normalization process and partitioning of coefficients into VLC coded and FLC coded parts

1. Media type specification

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

* 1. General

This annex is a media type specification for images encoded according to the tag-based format specified in Annex A of this Recommendation | International Standard for use in various Internet protocols, according to the scheme defined in IETF RFC 6838. Recent Internet protocols have been carefully designed to be easily extensible in certain areas, and many such protocols are capable of carrying arbitrary labelled content. The mechanism used to label such content is a media type, consisting of a top-level type, a subtype, and in some instances, optional parameters. This annex specifies such a content labelling scheme for JPEG XR coded images, to correspond with a registration in the IANA[[3]](#footnote-3) central registry.

* 1. Media type specification template information

Media type name: image

Subtype name: jxr

Required parameters: none

Optional parameters: none

Encoding considerations: files are binary and should be transmitted in a suitable encoding without CR/LF conversion, 7-bit stripping etc.; base64 is a suitable encoding.

Security considerations: The conveyed coded image files use a structure that can store image data, metadata corresponding to this image data, and other user-defined data. It should be noted that selected metadata fields may encompass information partly intended to protect the image against unauthorized use or distribution. In this case, the intention may be that alteration or removal of the data in the fields would be treated as an offence. Metadata fields may also contain information about the source of the image content. The data files have an extensible structure, so that it is theoretically possible that metadata fields could be defined in the future that could be used to induce particular actions on the part of the recipient, thus presenting additional security risks, but this type of capability is currently not supported in the current referenced specifications.

Interoperability considerations: JPEG XR coded image files can conform to one of several profiles and levels of capabilities (as specified in Annex B of this Recommendation | International Standard) – not all of which may be supported by a receiving decoder. As a result, implementations may attempt to decode and display an encoded JPEG XR image only to determine that the image cannot be rendered either partially or in full.

Published specification:

(this Recommendation | International Standard) Rec. ITU-T T.832 (2016) | ISO/IEC 29199-2 (20126), *Information technology – JPEG XR image coding system – Image coding specification*

Applications that use this media type: Imaging, fax, messaging and multi-media.

Fragment identifier considerations: N/A

Additional information:

Deprecated alias names for this type: N/A

Magic number(s):

As specified in Annex A of this Recommendation | International Standard, the data begins with a FILE\_HEADER( ) data structure, which begins with a FIXED\_FILE\_HEADER\_II\_2BYTES field equal to 0x4949, followed by a FIXED\_FILE\_HEADER\_0XBC\_BYTE field equal to 0xBC, followed by a FILE\_VERSION\_ID which is equal to 1 for the current version of this Recommendation | International Standard (with other values reserved for future use, as modified in additional parts or amendments, by ITU-T | ISO/IEC).

Within the payload data, JPEG XR IMAGE\_HEADER( ) data structures begin with a GDI\_SIGNATURE, which is a 64-bit syntax element that has the value 0x574D50484F544F00 that corresponds to "WMPHOTO" using the UTF-8 character set encoding specified in Annex D of ISO/IEC 10646, followed by a byte equal to 0.

File extension(s): jxr

Person and email address to contact for further information:

Touradj Ebrahimi, JPEG convenor (convenor@jpeg.org or touradj.ebrahimi@epfl.ch)

Gary J. Sullivan, JPEG XR image coding specification editor (garysull@microsoft.com)

Intended usage: COMMON

Restrictions on usage: N/A

Author: ITU-T and ISO/IEC JTC 1

Change controller: ITU-T and ISO/IEC JTC 1

Provisional registration? (standards tree only): Same (image/jxr)

Bibliography

* Recommendation ITU-R BT.470-6 (1998) (historical), *Conventional television systems*.
* Recommendation ITU-R BT.601-7 (2011), *Studio encoding parameters of digital television for standard 4:3 and wide screen 16:9 aspect ratios*.
* Recommendation ITU-R BT.709-6 (2015), *Parameter values for the HDTV standards for production and international programme exchange*.
* Recommendation ITU-R BT.1358-0 (1998) (historical), *Studio parameters of 625 and 525 line progressive scan television systems*.
* Recommendation ITU-R BT.1358-1 (2007), *Studio parameters of 625 and 525 line progressive scan television systems*.
* Recommendation ITU-R BT.1361-0 (1998) (historical), Worldwide *unified colorimetry and related characteristics of future television and imaging systems*.
* Recommendation ITU-R BT.1700-0 (2005), *Characteristics of composite video signals for conventional analogue television systems*.
* Recommendation ITU-T T.81 (1992) | ISO/IEC 10918-1:1994, *Information technology – Digital compression and coding of continuous-tone still images – Requirements and guidelines*.
* ISO/IEC 9899:2011, *Information technology – Programming languages – C*.
* ISO/IEC 11578:1996, *Information technology – Open Systems Interconnection – Remote Procedure Call (RPC)*.
* ISO 11664-1:2007 (CIE S 014-1/E:2006), *Colorimetry* *– Part 1: CIE standard colorimetric observers*.
* ISO 12234-2:2001, *Electronic still-picture imaging – Removable memory – Part 2: TIFF/EP image data format*.
* ISO 12639:2004, *Graphic technology – Prepress digital data exchange – Tag image file format for image technology (TIFF/IT)*.
* ISO 15076-1:2010, *Image technology colour management – Architecture, profile format and data structure – Part 1: Based on ICC.1:2010*.
* ISO 16684-1:2012, *Graphic technology – Extensible metadata platform (XMP) specification – Part 1: Data model, serialization and core properties.*
* IEC 61966-2-1:1999 + Amd.1:2003, *Multimedia systems and equipment – Colour measurement and management – Part 2-1: Colour management – Default RGB colour space – sRGB*.
* IEC 61966-2-2:2003, *Multimedia systems and equipment – Colour measurement and management – Part 2-2: Colour management – Extended RGB colour space - scRGB*
* IEC 61966-2-4:2006+Amd.1:2016, *Multimedia systems and equipment – Colour measurement and management – Part 2-4: Colour management – Extended-gamut YCC colour space for video applications – xvYCC*.
* ANSI CGATS/SWOP TR 003-2007 CMYK, *Graphic technology - Color characterization data for SWOP proofing and printing on U.S. Grade 3 coated publication paper*.
* FCC (2003), *Title 47 Code of Federal Regulations* 73.682 (a) (20).
* ICC.1:2001, *File format for color profiles*.
* IETF RFC 6838, *Media type specifications and registration procedures*.
* JEITA CP-3451 (2002), *Exchangeable image file format for digital still cameras: Exif version 2.2.*
* NTSC (1953), *Recommendation for transmission standards for colour television*.
* SMPTE RP 177-1993, *Derivation of basic television color equations*.
* SMPTE 170M-2004, *Television – Composite analog video signal – NTSC for studio applications*.
* SMPTE 240M-1999 (historical), *Television – 1125-line high-definition production systems – Signal parameters*.
* SWOP2006\_Coated3v2.icc (IDEAlliance), Profile ID: F7B89B0445878F17EBB85D7515DA9A98  
  <<http://www.color.org/registry/SWOP2006_Coated3v2.xalter>>
* [TIFF Revision 6.0](https://partners.adobe.com/public/developer/en/tiff/TIFF6.pdf) (1992), Adobe Systems Incorporated   
  https://partners.adobe.com/public/developer/en/tiff/TIFF6.pdf

|  |  |
| --- | --- |
| **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 | Environment and ICTs, climate change, e-waste, energy efficiency; 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, Internet of Things and smart cities |
| Series Z | Languages and general software aspects for telecommunication systems |
|  |  |

1. \* To access the Recommendation, type the URL http://handle.itu.int/ in the address field of your web browser, followed by the Recommendation's unique ID. For example, <http://handle.itu.int/11.1002/1000/11830-en>. [↑](#footnote-ref-1)
2. Committee for Graphics Arts Technologies Standards/Specifications for Web Offset Publications. [↑](#footnote-ref-2)
3. Internet Assigned Numbers Authority. [↑](#footnote-ref-3)