|  |  |
| --- | --- |
| **Joint Collaborative Team on Video Coding (JCT-VC)**  **of ITU-T SG16 WP3 and ISO/IEC JTC1/SC29/WG11**  7th Meeting: Geneva, CH, 21–30 November, 2011 | Document: JCTVC-G1103\_d9 |

|  |  |  |  |
| --- | --- | --- | --- |
| *Title:* | **WD5: Working Draft 5 of High-Efficiency Video Coding** | | |
| *Status:* | Output Document of JCT-VC | | |
| *Purpose:* | Working Draft of HEVC | | |
| *Author(s) or Contact(s):* | Benjamin Bross Fraunhofer HHI  Woo-Jin Han Gachon University  Jens-Rainer Ohm RWTH Aachen  Gary J. Sullivan Microsoft  Thomas Wiegand Fraunhofer HHI / TU Berlin | Email:  Email:  Email:  Email:  Email: | benjamin.bross@hhi.fraunhofer.de  hurumi@gmail.com  ohm@ient.rwth-aachen.de  garysull@microsoft.com  thomas.wiegand@hhi.fraunhofer.de |
| *Source:* | Editor | | |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Abstract

Working Draft 5 of High-Efficiency Video Coding.

Ed. Notes (WD5):

* Incorporated weighted prediction (JCTVC-F265)
* Removed CAVLC
* Incorporated wavefront parallel processing (JCTVC-F274)
* Incorporated wavefront CABAC flush (JCTVC-F275)
* Incorporated tiles (JCTVC-F335)
* Removed ClipMv (JCTVC-G134)
* Removed merge partition redundancy check (JCTVC-G681)
* Incorporated simplified merge pruning (JCTVC-G1006)
* Incorporated extend scaling factor clipping to 16 (JCTVC-G223)
* Incorporated amvp position dependency removal (JCTVC-G542)
* Incorporated simplified TMVP refidx derivation (JCTVC-G163)
* Incorporated MaxNumMergeCand signaling in slice header (JCTVC-G091)
* Incorporated modified H and center TMVP positions (JCTVC-G082)
* Incorporated intra smoothing for horizontal and vertical directions (G457)
* Incorporated removal of ALF DC offset (JCTVC-G445)
* Incorporated line buffer elimination (JCTVC-G145)
* Incorporated simplified intra mode mapping (JCTVC-G418/G109/G144)
* Incorporated modified intra mode coding (JCTVC-G119)
* Incorporated luma interpolation filter (JCTVC-G778)
* Incorporated chroma interpolation filter (JCTVC-G778)
* Incorporated simplified intra padding (JCTVC-G812)
* Incorporated modified cRiceParam update (JCTVC-G700)
* Incorporated 8bit init values for CABAC (JCTVC-G633)
* Incorporated harmonized pred and part mode binarization (JCTVC-G1042)
* Incorporated significant map context reduction (JCTVC-G1015)
* Incorporated level chroma context reduction (JCTVC-G783)
* Incorporated diagonal sub-block scan for residual coding (JCTVC-G323)
* Removed NSQT remapping and transform reordering (JCTVC-G1038)
* Incorporated modified last\_significant\_coeff\_x/y coding (JCTVC-G201/G704)
* Incorporated shared chroma CBF contexts (JCTVC-G718)
* Incorporated luma intra mode bypass coding (JCTVC-G767)
* Incorporated multi-level significant map (JCTVC-G644)
* Incorporated WD and HM mismatch for LM prediction (JCTVC-G1034)
* Revert the slice boundary padding for adaptive loop filter to WD4
* Incorporated 4x4 BA classification (JCTVC-G609)
* Incorporated virtual boundary processing (JCTVC-G212)
* Incorporated fixed K-table for ALF (JCTVC-G610)
* Incorporated removing 15th merge flag for BA mode in ALF (JCTVC-G216)
* Incorporated prediction of ALF coefficients (JCTVC-G665)
* Incorporated deblocking clean-up (JCTVC-G1035/G620)
* Revert the deblocking decision to HM3 (JCTVC-G088)
* Incorporated support of varying QP in deblocking (JCTVC-G1031)
* Incorporated reducing motion data line buffers (JCTVC-G229)
* Incorporated reference picture set (RPS) (JCTVC-G1002)
* Incorporated reference picture set prediction (JCTVC-G198)
* Incorporated separate decisions for each half (4 lines) of a length 8 block boundaries (JCTVC-G590)
* Incorporated BoG on deblocking fix (JCTVC-G1035)
* Incorporated core transform (JCTVC-G495)
* Incorporated clipping at the output of the first inverse transform (JCTVC-G782)
* Incorporated forbidding level values outside of 16b (JCTVC-G719)
* Incorporated reducing cbf flag signalling redundancy (JCTVC-G444)
* Incorporated changing luma/chroma coefficient interleaving from CU to TU level (JCTVC-G381)
* Incorporated defining MaxIPCMCUSize, MinChromaTrafoSize (JCTVC-G112)
* Incorporated harmonization of implicit TU, AMP and NSQT (JCTVC-G519)
* Incorporated improved weighted prediction (JCTVC-G065)
* Incorporated redundancy removal of explicit weighted prediction syntax (JCTVC-G441)
* Incorporated non-cross-tiles loop filtering for independent tiles (JCTVC-G194)
* Incorporated low latency CABAC initialization for dependent tiles (JCTVC-G197)
* Incorporated AVC-based quantization matrices syntax (JCTVC-G434)
* Incorporated HVS-based quantization matrices (JCTVC-G880)
* Incorporated APS quantization matrices and parameter set extension syntax (JCTVC-G1016)
* Incorporated nal\_unit\_type value of 14 for APS
* Incorporated SPS syntax for chroma\_format\_idc from AVC
* Incorporated pure VLC for SAO and ALF (JCTVC-G220)
* Moved slice address and put slice\_type and cabac\_init\_idc into slice and entropy slice header. (JCTVC-G1025)
* Incorporated picture width and height coding using ue(v) rather than u(16) (JCTVC-G325)
* Incorporated ALF and SAO flags in slice header (JCTVC-G566)
* Incorporated marking process for non-TMVP pictures (JCTVC-G398)
* Incorporated max\_dec\_frame\_buffering, num\_reorder\_frames, and use max\_latency\_increase (JCTVC-G546)
* Incorporated high level syntax clean up (JCTVC-G507)
* Incorporated chroma QP offset (JCTVC-G509)

Working Draft 4 of High-Efficiency Video Coding.

Ed. Notes (WD4):

* Removed inferred merge (JCTVC-F082)
* Incorporated slice header flag to disable 4x4 inter partitions (JCTVC-F744)
* Incorporated modified rounding in MV scaling (JCTVC-F142)
* Removed intermediate amvp spatial candidates redundancy check (JCTVC-F050)
* Incorporated reducing the number of spatial mv scalings to 1 (JCTVC-F088)
* Incorporated spatial merge candidate positions unification (JCTVC-F419)
* Incorporated one reference list check for temporal mvp (JCTVC-F587)
* Incorporated AMVP/merge parsing robustness with simplifications (JCTVC-F470)
* Incorporated unified availability check for intra (JCTVC-F477)
* Incorporated generic interpolation filter (JCTVC-F537)
* Incorporated non-square quadtree transform NSQT (JCTVC-F412)
* Incorporated asymmetric motion partitions AMP (JCTVC-F379)
* Incorporated CBF redundancy reduction (JCTVC-C277)
* Incorporated modified delta QP binarization (JCTVC-F745)
* Incorporated diagonal coefficient scanning in CABAC (JCTVC-F129)
* Incorporated parallel context processing for coefficient levels in CABAC (JCTVC-F130)
* Incorporated context sharing for significant\_coeff\_flag of 16x16 and 32x32 transforms (JCTVC-F132)
* Incorporated unified scans (JCTVC-F288)
* Incorporated sample adaptive offset (JCTVC-E049)
* Incorporated sample adaptive offset for chroma (JCTVC-F057)
* Incorporated sample adaptive offset offset accuracy (JCTVC-F396)
* Incorporated updated ALF slice padding (JCTVC-D128)
* Incorporated updated ALF slice padding due to ALF filter shape change (JCTVC-F303/F042)
* Incorporated updated ALF slice padding due to unified luma and chroma filter shapes (JCTVC-F157)
* Incorporated ALF filter using subset of pixels (JCTVC-F301)
* Incorporated modified deblocking process for luma (JCTVC-F118)
* Incorporated modified tc\_offset in deblocking process (JCTVC-F143)
* Incorporated MDIS and pixel position change of planar (JCTVC-F483)
* Incorporated availability check removal for intra DC filtering (JCTVC-F178)
* Incorporated size-independent intra DC filtering (JCTVC-F252)
* Incorporated modified MDIS table (JCTVC-F126)
* Incorporated simplified intra\_FromLuma prediction (JCTVC-F760)
* Incorporated fixed number of MPM (JCTVC-F765)
* Incorporated SAO boundary processing (JCTVC-F232)
* Minor bug in deriving sample positions in SAO process was fixed
* Bug in coding tree syntax table related to the initialization of variable IsCuQpDeltaCoded was fixed
* Incorporated modified last significant coefficient position coding in CABAC (JCTVC-F375)
* Incorporated modified mvd coding in CABAC (JCTVC-F455)
* Incorporated reduced number of contexts in CABAC (JCTVC-F746)
* Incorporated high-level syntax cleanup (JCTVC-F714)
* Incorporated NAL unit type and CDR (CRA) (JCTVC-F462/464)
* Incorporated adaptation parameter set (APS) (JCTVC-F747)

Ed. Notes (WD3):

* Added Residual coding CABAC syntax and semantics
* Added Zig-zag scanning process
* Added CABAC Binarization processes
* Incorporated MV coding (JCTVC-E481)
* Incorporated Compression of reference indices (JCTVC-E059)
* Incorporated Zero merge candidate (JCTVC-E146)
* Incorporated Intra mode coding (JCTVC-E088/E131) (Inserted by TK 31/3/2011 with notes)
* Fixed the CABAC coefficients syntax, semantics and inverse scanning process
* Incorporated CABAC coeffs (JCTVC-E253)
* Moved the EGk binarization from the UEGk subclause in a separate subclause
* Added text representing CABAC entropy coding context initialisation
* Added text representing CABAC entropy coding context derivation.
* Mode-dependent 3- scan for intra (JCTVC-D393)
* Incorporated CABAC: Context size reduction (JCTVC-E227/E489)
* Incorporated CABAC: significance map coding simplification (JCTVC-E227/E338/E344/E494)
* Incorporated CABAC: Contexts for MVD (JCTVC-E324)
* Incorporated initial draft of CAVLC text
* CAVLC for 16x16 & 32x32 (JCTVC-E383)
* CAVLC table size reduction (JCTVC-E384)
* CAVLC for RQT (JCTVC-E404)
* CAVLC: counters (JCTVC-E143)
* CAVLC: Intra prediction mode coding in LCEC (JCTVC-D366)
* CAVLC: Inter prediction mode coding in LCEC (JCTVC- D370)
* CAVLC: 4x4 and 8x8 transform coefficient coding in LCEC (JCTVC- D374)
* Block-based ALF (JCTVC-E046/E323)
* ALF parameters to PPS (JCTVC-E045)
* Parallel deblocking (JCTVC-E496/E181/E224)
* Clipping for bi-pred averaging (JCTVC-E242)
* Reference sample padding (JCTVC-E488)
* Transformation processes are replaced by [TBD] mark (meeting note, JCTVC-E243)
* Sub-LCU-level dQP (JCTVC-E051/E220)
* Temporal layer switching and reference list management based on temporal\_id (JCTVC-E279/D081/D200)
* Improved text of entropy slice (JCTVC-D070) [Ed. (WJ): the term entropy slice is replaced by lightweight slice according to proponent’s text]
* Slice independent deblocking and adaptive loop filtering (JCTVC-D128)
* Fine-granularity slices (JCTVC-E483)
* PCM mode (JCTVC-E057 and JCTVC-E192)
* CAVLC: Inter pred coding (JCTVC-E381)
* CAVLC: Combined coding of inter prediction direction and reference frame index (JCTVC-D141)
* 4x4 DST (JCTVC-E125)
* Planar mode (JCTVC-E321)
* Luma-based chroma intra prediction (JCTVC-E266)
* Modification of DC predictor (JCTVC-E069)
* Bug in mapping table and the corresponding text for mostProbableIntra was fixed. (64x64 uses 3-directions, but the table was specified for 5-directions)
* Non-existing cases of Intra\_DC, Intra\_Planar and Intra\_FromLuma are removed (due to reference sample padding, JCTVC-E488)

Ed. Notes (WD2):

* Incorporated Partial Merging according to JCTVC-D441
  + removed direct mode
  + moved merge to prediction\_unit and added candidates
  + added partial merge restrictions
  + inter NxN partitioning only for smallest coding\_unit
* Updated transform\_tree and transform\_coeff syntax
* Added transform\_coeff to coding\_unit syntax (Fix)
* Incorporated intra NxN partitioning only for smallest coding\_unit according to JCTVC-D432
* Incorporated modified temporal motion vector predition according to JCTVC-D164
* Incorporated simplified motion vector prediction according to JCTVC-D231
  + removed median
  + removed pruning process
  + changed the selection manner of left/top predictor
* 8-tap luma interpolation filter according to JCTVC-D344
* 4-tap chroma interpolation filter according to JCTVC-D347
* Improved deblocking filter text according to JCTVC-D395
* IBDI syntax is removed
* Updated syntax and semantics
  + Two tool-enabling flags (adaptive\_loop\_filter\_enabled\_flag and cu\_qp\_delta\_enabled\_flag) are added in SPS according to software. However, low\_delay\_coding\_enabled\_flag is not added – it could be handled by more general reference management scheme. merging\_enabled\_flag is not added – partial merging (JCTVC-D441) was adopted thus merging cannot be turned off any more. amvp\_mode[] is not added since amvp cannot be turned off any more due to absence of median predictor (JCTVC-D231). Note that software has all switches.
  + cu\_qp\_delta (coding unit layer), syntax and semantics are added. (JCTVC-D258)
  + collocated\_from\_l0 (slice header), syntax and semantics are added.
* Clean decoding refresh (CDR) (JCTVC-D234).
* Temporal motion vector memory compression (JCTVC-D072)
* Constrained intra prediction (JCTVC-D086)
* Mode-dependent intra smoothing (JCTVC-D282)
* Merging chroma intra prediction process into luma intra prediction process
* Combined reference list (JCTVC-D421)
* Chroma intra prediction mode reordering (JCTVC-D255/D278/D166)
* Adaptive loop filter text is added
* Entropy slice is added (JCTVC-D070)
* High precision bi-directional averaging (JCTVC-D321)
* Reduction of number of intra prediction modes for 64x64 blocks (JCTVC-D100)
* Misc.
  + TPE bits are reduced from 4 to 2
  + Clipping is applied to (temporally) scaled mv – revisit

Ed. Notes (WD1):

* Incorporated the decisions on high-level syntax according to JCTVC-B121
* Incorporated text from JCTVC-B205revision7
* Incorporated text from JCTVC-C319 (as found to be stable)
* Revised coding tree, coding unit and prediction unit syntaxes (coding tree syntax is newly added. needs to be confirmed)
* Initial drafting of decoding process of coding units in intra prediction mode (luma part, JCTVC-B100 and JCTVC-C042)
* Initial drafting of decoding process of coding units in inter prediction mode
* Initial drafting of scaling and transformation process
* Added text, transform 16T and 32T
* Initial drafting of deblocking process
* Improving the text, derivation process for motion vector components and reference indices
* Added text, boundary filtering strength

Open issues:

* Should support for monochrome, 4:2:2 and 4:4:4 (with and w/o separate color planes) be included from the start? Currently, it has been left in the text as it doesn't seem to affect much text.
* Handling of the term "frame". We are currently leaning towards changing all occurrences of "frame" to "picture" (removed or marked all occurrences of "field")
* Use of bin string and bit string should be consistent.
* MD5 hash (JCTVC-E490)
* ALF coefficient derivation from syntax elements
* Improve text quality by considering: strict use of terms “unit” and “block” – block = a rectangular 2D array (one component), unit = collective term for specifying information for both luma and chroma. Don’t use ther term ‘unit’ by itself – always use the term ‘unit’ with prefix – coding unit, prediction unit or transform unit.
* Syntax functions cuAddress and granularity\_block\_boundary (fine granularity slices, JCTVC-E483) are missing
* Software seems to have SPS syntax to turn on/off combined reference list (JCTVC-D421), but currently text does not.
* Both variables IntraPredMode and IntraPredModeC are used in the syntax table but the actual derivations are specified in the decoding process. Maybe it’s better to move them to the semantics section.
* Binarization of intra\_chroma\_pred\_mode reflecting codeword switching and luma-based chroma intra prediction (JCTVC-E266) is missing.
* Clarification on the use of intra\_chroma\_pred\_mode and IntraPredModeC is needed. The former specifies the syntax item to indicate how to determine the chroma intra prediction mode (IntraPredModeC) as 0 (to Intra\_FromLuma), 1 (to Intra\_Vertical), 2 (to Intra\_Horizontal), 3 (Intra\_DC or Intra\_Planar) or 4 (re-use luma mode). The latter specifies chroma intra prediction mode, which is actually mapped to the specific prediction process.
* The variable qPL used in deblocking filter process has to be defined by, e.g. setting it to QPY.
* The Chroma QP should be considered in chroma edge filtering instad of luma QP.
* The tc and beta values of the deblocking filter should be explicitly scaled in the text since their values are provided in the table for the 8-bit case.
* Deblocking filter text seems incorrect. Some horizontal edges are not filtered mainly due to the parallel deblocking process. Usage of indices xDk and XPOS also has to be cross-verified with the software behaviour.
* Software-text mismatch of JCTVC-F252: SW applies DC filtering to 64x64 intra block but WD does not.
* There are input/output parameter inconsistencies between function call and actual functions of intra prediction. Function call uses sample position, block, size and chroma index while some functions do not use some parameters.
* No semantics for deblocking filter control syntaxes (deblocking\_filter\_in\_APS\_enabled\_flag, aps\_deblocking\_filter\_flag, inherit\_dbl\_params\_from\_APS\_flag, beta\_offset\_div2, and tc\_offset\_div2)
* Absolute operation |·| is not defined. It could be replaced by Abs(·) or defined separately in arithmetic operators subclause.
* No semantics for slice\_sample\_adaptive\_offset\_flag and slice\_adaptive\_loop\_filter\_flag (value of duplicate flag must always match).

**CONTENTS**

*Page*

Abstract i

0 Introduction 11

0.1 Prologue 11

0.2 Purpose 11

0.3 Applications 11

0.4 Publication and versions of this Specification 11

0.5 Profiles and levels 11

0.6 Overview of the design characteristics 12

0.7 How to read this Specification 12

1 Scope 13

2 Normative references 13

3 Definitions 13

4 Abbreviations 18

5 Conventions 19

5.1 Arithmetic operators 19

5.2 Logical operators 19

5.3 Relational operators 20

5.4 Bit-wise operators 20

5.5 Assignment operators 20

5.6 Range notation 20

5.7 Mathematical functions 20

5.8 Order of operation precedence 21

5.9 Variables, syntax elements, and tables 22

5.10 Text description of logical operations 23

5.11 Processes 24

6 Source, coded, decoded and output data formats, scanning processes, and neighbouring relationships 24

6.1 Bitstream formats 24

6.2 Source, decoded, and output picture formats 24

6.3 Spatial subdivision of pictures and slices 26

6.4 Inverse scanning processes and derivation processes for neighbours 27

6.4.1 Inverse treeblock scanning process 27

6.5 Zig-zag scanning array initialisation process 28

6.6 Scanning order array initialisation process 28

7 Syntax and semantics 29

7.1 Method of specifying syntax in tabular form 29

7.2 Specification of syntax functions, categories, and descriptors 30

7.3 Syntax in tabular form 32

7.3.1 NAL unit syntax 32

7.3.2 Raw byte sequence payloads and RBSP trailing bits syntax 33

7.3.2.1 Sequence parameter set RBSP syntax 33

7.3.2.2 Picture parameter set RBSP syntax 33

7.3.2.3 Supplemental enhancement information RBSP syntax 34

7.3.2.4 Access unit delimiter RBSP syntax 34

7.3.2.5 Filler data RBSP syntax 34

7.3.2.6 Slice layer RBSP syntax 35

7.3.2.7 RBSP slice trailing bits syntax 35

7.3.2.8 RBSP trailing bits syntax 35

7.3.3 Slice header syntax 36

7.3.3.1 Reference picture list modification syntax 37

7.3.3.2 Reference picture lists combination syntax 37

7.3.3.3 Decoded reference picture marking syntax 38

7.3.3.4 Adaptive loop filter parameter syntax 39

7.3.4 Slice data syntax 40

7.3.5 Coding tree syntax 41

7.3.6 Coding unit syntax 41

7.3.7 Prediction unit syntax 43

7.3.8 Transform tree syntax 45

7.3.9 Transform coefficient syntax 47

7.3.10 Residual coding CABAC syntax 48

7.3.11 Residual coding CAVLC syntax 48

7.3.12 Level and sign CAVLC syntax 49

7.4 Semantics 50

7.4.1 NAL unit semantics 50

7.4.1.1 Encapsulation of an SODB within an RBSP (informative) 52

7.4.1.2 Order of NAL units and association to coded pictures, access units, and video sequences 53

7.4.2 Raw byte sequence payloads and RBSP trailing bits semantics 57

7.4.2.1 Sequence parameter set RBSP semantics 57

7.4.2.2 Picture parameter set RBSP semantics 58

7.4.2.3 Supplemental enhancement information RBSP semantics 59

7.4.2.4 Access unit delimiter RBSP semantics 59

7.4.2.5 Filler data RBSP semantics 59

7.4.2.6 Slice layer RBSP semantics 59

7.4.2.7 RBSP slice trailing bits semantics 59

7.4.2.8 RBSP trailing bits semantics 60

7.4.3 Slice header semantics 60

7.4.3.1 Reference picture list modification semantics 62

7.4.3.2 Reference picture lists combination semantics 63

7.4.3.3 Decoded reference picture marking semantics 63

7.4.3.4 Adaptive loop filter parameter semantics 66

7.4.4 Slice data semantics 67

7.4.5 Coding tree semantics 67

7.4.6 Coding unit semantics 68

7.4.7 Prediction unit semantics 69

7.4.8 Transform tree semantics 71

7.4.9 Transform coefficient semantics 72

7.4.10 Residual coding CABAC semantics 73

7.4.11 Residual coding CAVLC semantics 74

8 Decoding process 75

8.1 NAL unit decoding process 76

8.2 Slice decoding process 76

8.2.1 Decoding process for picture order count 76

8.2.1.1 Decoding process for picture order count type 0 76

8.2.1.2 Decoding process for picture order count type 1 76

8.2.1.3 Decoding process for picture order count type 2 76

8.2.2 Decoding process for reference picture lists construction 76

8.2.2.1 Decoding process for picture numbers 77

8.2.2.2 Initialisation process for reference picture lists 77

8.2.2.3 Modification process for reference picture lists 78

8.2.2.4 Mapping process for reference picture lists combination in B slices 80

8.2.3 Decoded reference picture marking process 81

8.2.3.1 Sequence of operations for decoded reference picture marking process 81

8.2.3.2 Decoding process for gaps in frame\_num 82

8.2.3.3 Sliding window decoded reference picture marking process 82

8.2.3.4 Adaptive memory control decoded reference picture marking process 82

8.3 Decoding process for coding units coded in intra prediction mode 83

8.3.1 Derivation process for luma intra prediction mode 84

8.3.2 Derivation process for chroma intra prediction mode 86

8.3.3 Decoding process for intra blocks 86

8.3.3.1 Intra sample prediction 87

8.4 Decoding process for coding units coded in inter prediction mode 93

8.4.1 Inter prediction process 93

8.4.2 Decoding process for prediction units in inter prediction mode 95

8.4.2.1 Derivation process for motion vector components and reference indices 96

8.4.2.2 Decoding process for inter prediction samples 107

8.4.3 Decoding process for the residual signal of coding units coded in inter prediction mode 114

8.4.3.1 Decoding process for luma residual blocks 114

8.4.3.2 Decoding process for chroma residual blocks 115

8.5 Scaling, transformation and array construction process prior to deblocking filter process 116

8.5.1 Scaling and transformation process 116

8.5.2 Inverse scanning process for transform coefficients 117

8.5.3 Scaling process for transform coefficients 117

8.5.4 Transformation process for scaled transform coefficients 118

8.5.4.1 Transformation process for 4 samples 118

8.5.4.2 Transformation process for 8 samples 118

8.5.4.3 Transformation process for 16 samples 118

8.5.4.4 Transformation process for 32 samples 121

8.5.5 Array construction process 128

8.6 In-loop filter process 128

8.6.1 Deblocking filter process 128

8.6.1.1 Derivation process of transform unit boundary 129

8.6.1.2 Derivation process of prediction unit boundary 129

8.6.1.3 Derivation process of boundary filtering strength 130

8.6.1.4 Filtering process for coding unit 131

8.6.2 Adaptive loop filter process 136

8.6.2.1 Derivation process for filter coefficients 136

8.6.2.2 Derivation process for filter index array for luma samples 137

8.6.2.3 Filtering process for luma samples 138

8.6.2.4 Filtering process for chroma samples 139

9 Parsing process 139

9.1 Parsing process for Exp-Golomb codes 140

9.1.1 Mapping process for signed Exp-Golomb codes 141

9.2 CAVLC parsing process for slice data 142

9.2.1 Parsing process for VLC codes 142

9.2.2 Initialisation process 144

9.2.2.1 Counter initialisation for codeword index adaptive mapping 144

9.2.2.2 Initialisation process for lastPosVlcNumIndex 145

9.2.2.3 Initialisation process for lastPosTable 145

9.2.2.4 Initialisation process for splitPredPartModeTable 145

9.2.2.5 Initialisation process for intraModeTable 145

9.2.2.6 Initialisation process for cbpSplitTransTable 146

9.2.3 Codeword index mapping update process 146

9.2.4 Parsing process for transform coefficient syntax elements 147

9.2.4.1 Parsing process for last\_pos\_level\_one 147

9.2.4.2 Parsing process for level\_minus2\_and\_sign 148

9.2.4.3 Parsing process for run\_level\_one 148

9.2.4.4 Parsing process for level 152

9.2.4.5 Parsing process for cu\_split\_pred\_part\_mode 152

9.2.4.6 Parsing process for rem\_intra\_luma\_pred\_mode 153

9.2.4.7 Parsing process for cbp\_and\_split\_transform 153

9.3 CABAC parsing process for slice data 154

9.3.1 Initialisation process 155

9.3.1.1 Initialisation process for context variables 155

9.3.1.2 Initialisation process for the arithmetic decoding engine 165

9.3.2 Binarization process 166

9.3.2.1 Unary (U) binarization process 169

9.3.2.2 Truncated unary (TU) binarization process 170

9.3.2.3 Truncated Rice (TR) binarization process 170

9.3.2.4 k-th order Exp-Golomb (EGk) binarization process 170

9.3.2.5 Concatenated unary/ k-th order Exp-Golomb (UEGk) binarization process 171

9.3.2.6 Fixed-length (FL) binarization process 171

9.3.2.7 Binarization process for cu\_qp\_delta 171

9.3.2.8 Binarization process for pred\_type 171

9.3.2.9 Binarization process for rem\_intra\_luma\_pred\_mode 172

9.3.2.10 Binarization process for coeff\_abs\_level\_minus3 172

9.3.3 Decoding process flow 173

9.3.3.1 Derivation process for ctxIdx 173

9.3.3.2 Arithmetic decoding process 182

9.3.4 Arithmetic encoding process (informative) 189

9.3.4.1 Initialisation process for the arithmetic encoding engine (informative) 189

9.3.4.2 Encoding process for a binary decision (informative) 190

9.3.4.3 Renormalization process in the arithmetic encoding engine (informative) 191

9.3.4.4 Bypass encoding process for binary decisions (informative) 193

9.3.4.5 Encoding process for a binary decision before termination (informative) 194

9.3.4.6 Byte stuffing process (informative) 196

Annex A Profiles and levels 197

A.1 Requirements on video decoder capability 197

A.2 Profiles 197

A.2.1 ABC profile 197

A.3 Levels 197

Annex B Byte stream format 199

B.1 Byte stream NAL unit syntax and semantics 199

B.1.1 Byte stream NAL unit syntax 199

B.1.2 Byte stream NAL unit semantics 199

B.2 Byte stream NAL unit decoding process 200

B.3 Decoder byte-alignment recovery (informative) 200

Annex C Hypothetical reference decoder 201

C.1 Operation of coded picture buffer (CPB) 202

C.1.1 Timing of bitstream arrival 202

C.1.2 Timing of coded picture removal 202

C.2 Operation of the decoded picture buffer (DPB) 202

C.3 Bitstream conformance 202

C.4 Decoder conformance 202

Annex D Supplemental enhancement information 203

D.1 SEI payload syntax 203

D.2 SEI payload semantics 203

Annex E Video usability information 204

E.1 VUI syntax 204

E.1.1 VUI parameters syntax 204

E.2 VUI semantics 204

**List of figures**

[Figure 6‑2 – Nominal vertical and horizontal locations of 4:2:2 luma and chroma samples in a picture [Ed. Re-draw figure] 26](#_Toc293649324)

[Figure 6‑3 – Nominal vertical and horizontal locations of 4:4:4 luma and chroma samples in a picture [Ed. Re-draw figure] 26](#_Toc293649325)

[Figure 6‑4 – A picture with 11 by 9 treeblocks that is partitioned into two slices 27](#_Toc293649326)

[Figure 7‑1 – Structure of an access unit not containing any NAL units with nal\_unit\_type equal to 0, 7, 8, or in the range of 12 to 18, inclusive, or in the range of 20 to 31, inclusive 56](#_Toc293649327)

[Figure 8‑2 – Intra prediction angle definition (informative) 92](#_Toc293649328)

[Figure 8‑3 – Spatial neighbours that can be used as merging candidates (informative) illustrates the position of the spatial neighbours A, B, C and D relative to the current prediction unit. 99](#_Toc293649329)

[Figure 8‑4 – Spatial neighbours, that can be used to derive reference indices for temporal merging candidates, (informative) illustrates the position of the spatial neighbours A, B, C, D and E relative to the current prediction unit. 101](#_Toc293649330)

[Figure 8‑5 – Spatial motion vector neighbours 103](#_Toc293649331)

[Figure 8‑6 – Integer samples (shaded blocks with upper-case letters) and fractional sample positions (un-shaded blocks with lower-case letters) for quarter sample luma interpolation 109](#_Toc293649332)

[Figure 8‑8 Mapping between geometric position and luma adaptive loop filter index according to alfTap (informative) 139](#_Toc293649333)

[Figure 9‑1 – Overview of the arithmetic decoding process for a single bin (informative) 183](#_Toc293649334)

[Figure 9‑2 – Flowchart for decoding a decision 185](#_Toc293649335)

[Figure 9‑3 – Flowchart of renormalization 187](#_Toc293649336)

[Figure 9‑4 – Flowchart of bypass decoding process 188](#_Toc293649337)

[Figure 9‑5 – Flowchart of decoding a decision before termination 189](#_Toc293649338)

[Figure 9‑6 – Flowchart for encoding a decision 191](#_Toc293649339)

[Figure 9‑7 – Flowchart of renormalization in the encoder 192](#_Toc293649340)

[Figure 9‑8 – Flowchart of PutBit(B) 193](#_Toc293649341)

[Figure 9‑9 – Flowchart of encoding bypass 194](#_Toc293649342)

[Figure 9‑10 – Flowchart of encoding a decision before termination 195](#_Toc293649343)

[Figure 9‑11 – Flowchart of flushing at termination 196](#_Toc293649344)

[Figure C‑9‑12 – Structure of byte streams and NAL unit streams for HRD conformance checks 201](#_Toc293649345)

[Figure C‑9‑13 – HRD buffer model 202](#_Toc293649346)

**List of tables**

[Table 5‑1 – Operation precedence from highest (at top of table) to lowest (at bottom of table) 23](#_Toc293649349)

[Table 6‑1 – SubWidthC, and SubHeightC values derived from chroma\_format\_idc and separate\_colour\_plane\_flag 26](#_Toc293649350)

[Table 7‑1 – NAL unit type codes, syntax element categories, and NAL unit type classes 52](#_Toc293649351)

[Table 7‑2 – Meaning of primary\_pic\_type 60](#_Toc293649352)

[Table 7‑3 – Name association to slice\_type 61](#_Toc293649353)

[Table 7‑4 – modification\_of\_pic\_nums\_idc operations for modification of reference picture lists 64](#_Toc293649354)

[Table 7‑5 – Interpretation of adaptive\_ref\_pic\_marking\_mode\_flag 65](#_Toc293649355)

[Table 7‑6 – Memory management control operation (memory\_management\_control\_operation) values 66](#_Toc293649356)

[Table 7‑7 – Specification of alf\_chroma\_idc 68](#_Toc293649357)

[Table 7‑8 – Specification of cu\_split\_pred\_part\_mode when log2CUSize is greater than Log2MinCUSize 69](#_Toc293649358)

[Table 7‑9 – Specification of cu\_split\_pred\_part\_mode when log2CUSize is equal to Log2MinCUSize 69](#_Toc293649359)

[Table 7‑10 ‑ Name association to prediction mode and partitioning type 70](#_Toc293649360)

[Table 7‑11 – Number of bins and number of mode for rem\_intra\_luma\_pred\_mode associated to PuSize 70](#_Toc293649361)

[Table 7‑12 – Name association to inter prediction mode 71](#_Toc293649362)

[Table 7‑13 – Specification of cbp\_and\_split\_transform 73](#_Toc293649363)

[Table 7‑14 – Specification of ScanType[ log2TrafoSize – 2 ][ IntraPredMode ] 74](#_Toc293649364)

[Table 7‑15 – Specification of ScanTypeC[ log2TrafoSize – 2 ][ IntraPredModeC ] 74](#_Toc293649365)

[Table 8‑1 – Specification of intraPredModeNum 85](#_Toc293649366)

[Table 8‑2 – Specification of mapIntraPredMode5 and mapIntraPredMode9 86](#_Toc293649367)

[Table 8‑3 – Specification of IntraPredModeC according to the values of intra\_chroma\_pred\_mode and IntraPredMode[ xB ][ yB ] 87](#_Toc293649368)

[Table 8‑4 – Specification of intraPredMode and associated names 89](#_Toc293649369)

[Table 8‑5 – Specification of intraFilterType[ nS ][ IntraPredMode ] for various prediction unit sizes 90](#_Toc293649370)

[Table 8‑6 – Specification of intraPredOrder 92](#_Toc293649371)

[Table 8‑7 – Specification of intraPredAngle 93](#_Toc293649372)

[Table 8‑8 – Specification of invAngle 93](#_Toc293649373)

[Table 8‑9 – Assignment of the luma prediction sample predSampleLXL[ xL, yL ] 112](#_Toc293649374)

[Table 8‑10 – Assignment of the chroma prediction sample predSampleLXC[ xC, yC ] for ( X, Y ) being replaced by ( 1, b ), ( 2, c ), ( 3, d ), ( 4, e ), ( 5, f ), ( 6, g ), and ( 7, h ), respectively 114](#_Toc293649375)

[Table 8‑11 – Derivation of threshold variables β and tC from input Q 136](#_Toc293649376)

[Table 8‑12 – Specification of horPos[ i ] according to alfTap for adaptive loop filter process of luma samples 139](#_Toc293649377)

[Table 8‑13 – Specification of verPos[ i ] according to alfTap for adaptive loop filter process of luma samples 139](#_Toc293649378)

[Table 9‑1 – Bit strings with "prefix" and "suffix" bits and assignment to codeNum ranges (informative) 141](#_Toc293649379)

[Table 9‑2 – Exp-Golomb bit strings and codeNum in explicit form and used as ue(v) (informative) 142](#_Toc293649380)

[Table 9‑3 – Assignment of syntax element to codeNum for signed Exp-Golomb coded syntax elements se(v) 143](#_Toc293649381)

[Table 9‑4 – codeNum and codeword with vlcNum equal to 15 145](#_Toc293649382)

[Table 9‑5 – Specification of initial values of lastPosTable[tableNum][codeNum] 146](#_Toc293649383)

[Table 9‑6 – Specification of intraModeTable[k][codeNum] 146](#_Toc293649384)

[Table 9‑7 – Specification of initial values of cbpSplitTransTable[flagPattern-8][n][codeNum] 147](#_Toc293649385)

[Table 9‑8 – Specification of lastPosVlcNumTable[blockType][vlcNumIdx] 149](#_Toc293649386)

[Table 9‑9 – Derivation of vlcNum from tableIdx and maxRunIdx 150](#_Toc293649387)

[Table 9‑10 – Derivation of largeOnePos from tableIdx and maxRunIdx when N is equal to 4 151](#_Toc293649388)

[Table 9‑11 – Derivation of largeOnePos from tableIdx and maxRunIdx when N is greater than 4 151](#_Toc293649389)

[Table 9‑12 – Derivation of runOfZeros from maxRun and codeNum 151](#_Toc293649390)

[Table 9‑13 – Derivation of levelThreshold from vlcIdx 153](#_Toc293649391)

[Table 9‑14 – Specification of cbpVlcNumTable[cbpVlcNumIdx] 155](#_Toc293649392)

[Table 9‑15 – Derivation of cMax from flagPattern and n 155](#_Toc293649393)

[Table 9‑16 – Specification of cbpSplitTransTable and counterNum associated 155](#_Toc293649394)

[Table 9‑17 – Association of ctxIdx and syntax elements for each slice type in the initialisation process 157](#_Toc293649395)

[Table 9‑18 – Values of variables m and n for alf\_cu\_flag ctxIdx 157](#_Toc293649396)

[Table 9‑19 – Values of variables m and n for split\_coding\_unit\_flag ctxIdx 158](#_Toc293649397)

[Table 9‑20 – Values of variables m and n for skip\_flag ctxIdx 158](#_Toc293649398)

[Table 9‑21 – Values of variables m and n for cu\_qp\_delta ctxIdx 158](#_Toc293649399)

[Table 9‑22 – Values of variables m and n for pred\_type 158](#_Toc293649400)

[Table 9‑23 – Values of variable m and n for prev\_intra\_luma\_pred\_flag ctxIdx 158](#_Toc293649401)

[Table 9‑24 – Values of variable m and n for rem\_intra\_luma\_pred\_mode ctxIdx 158](#_Toc293649402)

[Table 9‑25 – Values of variable m and n for intra\_chroma\_pred\_mode ctxIdx 159](#_Toc293649403)

[Table 9‑26 – Values of variable m and n for merge\_flag ctxIdx 159](#_Toc293649404)

[Table 9‑27 – Values of variable m and n for merge\_idx ctxIdx 159](#_Toc293649405)

[Table 9‑28 – Values of variable m and n for inter\_pred\_flag ctxIdx 159](#_Toc293649406)

[Table 9‑29 – Values of variable m and n for ref\_idx\_lc, ref\_idx\_l0, ref\_idx\_l1 ctxIdx 159](#_Toc293649407)

[Table 9‑30 – Values of variable m and n for mvd\_lc, mvd\_l0, mvd\_l1 ctxIdx 159](#_Toc293649408)

[Table 9‑31 – Values of variable m and n for mvp\_idx\_lc, mvp\_idx\_l0, mvp\_idx\_l1 ctxIdx 160](#_Toc293649409)

[Table 9‑32 – Values of variable m and n for no\_residual\_data\_flag ctxIdx 160](#_Toc293649410)

[Table 9‑33 – Values of variable m and n for split\_transform\_flag ctxIdx 160](#_Toc293649411)

[Table 9‑34 – Values of variable m and n for cbf\_luma ctxIdx 160](#_Toc293649412)

[Table 9‑35 – Values of variable m and n for cbf\_cb ctxIdx 160](#_Toc293649413)

[Table 9‑36 – Values of variable m and n for cbf\_cr ctxIdx 160](#_Toc293649414)

[Table 9‑37 – Values of variable m and n for last\_significant\_coefficient\_x ctxIdx 161](#_Toc293649415)

[Table 9‑38 – Values of variable m and n for last\_significant\_coefficient\_y ctxIdx 162](#_Toc293649416)

[Table 9‑39 – Values of variable m and n for significant\_coeff\_flag ctxIdx (A) 163](#_Toc293649417)

[Table 9‑40 – Values of variable m and n for significant\_coeff\_flag ctxIdx (B) 163](#_Toc293649418)

[Table 9‑41 – Values of variable m and n for significant\_coeff\_flag ctxIdx (C) 164](#_Toc293649419)

[Table 9‑42 – Values of variable m and n for coeff\_abs\_level\_greater1\_flag ctxIdx 165](#_Toc293649420)

[Table 9‑43 – Values of variable m and n for coeff\_abs\_level\_greater2\_flag ctxIdx 166](#_Toc293649421)

[Table 9‑44 – Syntax elements and associated types of binarization, maxBinIdxCtx, ctxIdxTable, and ctxIdxOffset 168](#_Toc293649422)

[Table 9‑45 – Bin string of the unary binarization (informative) 171](#_Toc293649423)

[Table 9‑46 – Binarization for pred\_type 173](#_Toc293649424)

[Table 9‑47 – Binarization for rem\_intra\_luma\_pred\_mode 173](#_Toc293649425)

[Table 9‑48 – Specifcation of cTRMax depending on cRiceParam 174](#_Toc293649426)

[Table 9‑49 – Assignment of ctxIdxInc to binIdx for all ctxIdxTable and ctxIdxOffset values 176](#_Toc293649427)

[Table 9‑50 – Specifcation of ctxIdxInc using left and above syntax elements 179](#_Toc293649428)

[Table 9‑51 – Specifcation of lastCtx depending on binIdx 180](#_Toc293649429)

[Table 9‑52 – Specification of rangeTabLPS depending on pStateIdx and qCodIRangeIdx 186](#_Toc293649430)

[Table 9‑53 – State transition table 187](#_Toc293649431)

[Table A‑9‑54 – Level limits [Ed. (TW) Kept for convenience] 199](#_Toc293649432)

**Foreword**

The International Telecommunication Union (ITU) is the United Nations specialized agency in the field of telecommunications. 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 standardising telecommunications on a world-wide basis. The World Telecommunication Standardization Assembly (WTSA), which meets every four years, establishes the topics for study by the ITU-T study groups that, 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 that fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialised system for world-wide standardisation. National Bodies that are members of ISO and IEC participate in the development of International Standards through technical committees established by the respective organisation to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organisations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75% of the national bodies casting a vote.

This Recommendation | International Standard was prepared jointly by ITU-T SG 16 Q.6, also known as VCEG (Video Coding Experts Group), and by ISO/IEC JTC 1/SC 29/WG 11, also known as MPEG (Moving Picture Experts Group). VCEG was formed in 1997 to maintain prior ITU-T video coding standards and develop new video coding standard(s) appropriate for a wide range of conversational and non-conversational services. MPEG was formed in 1988 to establish standards for coding of moving pictures and associated audio for various applications such as digital storage media, distribution, and communication.

In this Recommendation | International Standard Annexes  A through E contain normative requirements and are an integral part of this Recommendation | International Standard.

**INTERNATIONAL STANDARD**

**ITU-T RECOMMENDATION**

High-Efficiency Video Coding

# Introduction

This clause does not form an integral part of this Recommendation | International Standard.

## Prologue

This subclause does not form an integral part of this Recommendation | International Standard.

As the costs for both processing power and memory have reduced, network support for coded video data has diversified, and advances in video coding technology have progressed, the need has arisen for an industry standard for compressed video representation with substantially increased coding efficiency and enhanced robustness to network environments. Toward these ends the ITU-T Video Coding Experts Group (VCEG) and the ISO/IEC Moving Picture Experts Group (MPEG) formed a Joint Collaborative Team on Video Coding (JCT-VC) in 2010 for development of a new Recommendation | International Standard.

## Purpose

This subclause does not form an integral part of this Recommendation | International Standard.

[Ed. TW: revise the following] This Recommendation | International Standard was developed in response to the growing need for higher compression of moving pictures for various applications such as videoconferencing, digital storage media, television broadcasting, internet streaming, and communication. It is also designed to enable the use of the coded video representation in a flexible manner for a wide variety of network environments. The use of this Recommendation | International Standard allows motion video to be manipulated as a form of computer data and to be stored on various storage media, transmitted and received over existing and future networks and distributed on existing and future broadcasting channels.

## Applications

This subclause does not form an integral part of this Recommendation | International Standard.

[Ed. TW: revise the following] This Recommendation | International Standard is designed to cover a broad range of applications for video content including but not limited to the following:

CATV Cable TV on optical networks, copper, etc.

DBS Direct broadcast satellite video services

DSL Digital subscriber line video services

DTTB Digital terrestrial television broadcasting

ISM Interactive storage media (optical disks, etc.)

MMM Multimedia mailing

MSPN Multimedia services over packet networks

RTC Real-time conversational services (videoconferencing, videophone, etc.)

RVS Remote video surveillance

SSM Serial storage media (digital VTR, etc.)

## Publication and versions of this Specification

This subclause does not form an integral part of this Recommendation | International Standard.

This Specification has been jointly developed by ITU‑T Video Coding Experts Group (VCEG) and the ISO/IEC Moving Picture Experts Group. It is published as technically-aligned twin text in both organizations ITU-T and ISO/IEC.

## Profiles and levels

This subclause does not form an integral part of this Recommendation | International Standard.

This Recommendation | International Standard is designed to be generic in the sense that it serves a wide range of applications, bit rates, resolutions, qualities, and services. Applications should cover, among other things, digital storage media, television broadcasting and real-time communications. In the course of creating this Specification, various requirements from typical applications have been considered, necessary algorithmic elements have been developed, and these have been integrated into a single syntax. Hence, this Specification will facilitate video data interchange among different applications.

Considering the practicality of implementing the full syntax of this Specification, however, a limited number of subsets of the syntax are also stipulated by means of "profiles" and "levels". These and other related terms are formally defined in clause 3.

A "profile" is a subset of the entire bitstream syntax that is specified by this Recommendation | International Standard. Within the bounds imposed by the syntax of a given profile it is still possible to require a very large variation in the performance of encoders and decoders depending upon the values taken by syntax elements in the bitstream such as the specified size of the decoded pictures. In many applications, it is currently neither practical nor economic to implement a decoder capable of dealing with all hypothetical uses of the syntax within a particular profile.

In order to deal with this problem, "levels" are specified within each profile. A level is a specified set of constraints imposed on values of the syntax elements in the bitstream. These constraints may be simple limits on values. Alternatively they may take the form of constraints on arithmetic combinations of values (e.g., picture width multiplied by picture height multiplied by number of pictures decoded per second).

Coded video content conforming to this Recommendation | International Standard uses a common syntax. In order to achieve a subset of the complete syntax, flags, parameters, and other syntax elements are included in the bitstream that signal the presence or absence of syntactic elements that occur later in the bitstream.

## Overview of the design characteristics

This subclause does not form an integral part of this Recommendation | International Standard.

[Ed. TW: revise the following] The coded representation specified in the syntax is designed to enable a high compression capability for a desired image or video quality. The algorithm is typically not lossless, as the exact source sample values are typically not preserved through the encoding and decoding processes. A number of techniques may be used to achieve highly efficient compression. Encoding algorithms (not specified in this Recommendation | International Standard) may select between inter and intra coding for block-shaped regions of each picture. Inter coding uses motion vectors for block-based inter prediction to exploit temporal statistical dependencies between different pictures. Intra coding uses various spatial prediction modes to exploit spatial statistical dependencies in the source signal for a single picture. Motion vectors and intra prediction modes may be specified for a variety of block sizes in the picture. The prediction residual is then further compressed using a transform to remove spatial correlation inside the transform block before it is quantised, producing an irreversible process that typically discards less important visual information while forming a close approximation to the source samples. Finally, the motion vectors or intra prediction modes are combined with the quantised transform coefficient information and encoded using either variable length coding or arithmetic coding.

## How to read this Specification

This subclause does not form an integral part of this Recommendation | International Standard.

It is suggested that the reader starts with clause 1 (Scope) and moves on to clause 3 (Definitions). Clause 6 should be read for the geometrical relationship of the source, input, and output of the decoder. Clause 7 (Syntax and semantics) specifies the order to parse syntax elements from the bitstream. See subclauses 7.1-7.3 for syntactical order and see subclause 7.4 for semantics; i.e., the scope, restrictions, and conditions that are imposed on the syntax elements. The actual parsing for most syntax elements is specified in clause 9 (Parsing process). Finally, clause 8 specifies how the syntax elements are mapped into decoded samples. Throughout reading this Specification, the reader should refer to clauses 2 (Normative references), 4 (Abbreviations), and 5 (Conventions) as needed. Annexes A through E also form an integral part of this Recommendation | International Standard.

Annex A specifies profiles each being tailored to certain application domains, and defines the so-called levels of the profiles. Annex B specifies syntax and semantics of a byte stream format for delivery of coded video as an ordered stream of bytes. Annex C specifies the hypothetical reference decoder and its use to check bitstream and decoder conformance. Annex D specifies syntax and semantics for supplemental enhancement information message payloads. Annex E specifies syntax and semantics of the video usability information parameters of the sequence parameter set.

Throughout this Specification, statements appearing with the preamble "NOTE -" are informative and are not an integral part of this Recommendation | International Standard.

# Scope

This document specifies High-Efficiency Video Coding.

# Normative references

The following Recommendations and International Standards contain provisions which, through reference in this text, constitute provisions of 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. [Ed. TW: revise the following]

– ITU-T Recommendation T.35 (2000), *Procedure for the allocation of ITU-T defined codes for non‑standard facilities*.

– ISO/IEC 11578:1996, Annex A, *Universal Unique Identifier*.

– ISO/CIE 10527:2007, *Colorimetric Observers*.

# Definitions

[Ed. (TW) adpated definitions so far. Needs more work including turning them into 1 sentence each.]

For the purposes of this Recommendation | International Standard, the following definitions apply:

* 1. **access unit**: A set of *NAL units* that are consecutive in *decoding order* and contain exactly one *primary coded picture*. In addition to the *primary coded picture* one *auxiliary coded picture*, or other *NAL units* not containing *slices* of a *primary coded picture*. The decoding of an access unit always results in a *decoded picture*.
  2. **AC transform coefficient**: Any *transform coefficient* for which the *frequency index* in one or both dimensions is non-zero.
  3. **adaptive binary arithmetic decoding process**: An entropy *decoding process* that derives the values of *bins* from a *bitstream* produced by an *adaptive binary arithmetic encoding process*.
  4. **adaptive binary arithmetic encoding process**: An entropy *encoding process*, not normatively specified in this Recommendation | International Standard, that codes a sequence of *bins* and produces a *bitstream* that can be decoded using the *adaptive binary arithmetic decoding process*.
  5. **B slice**: A *slice* that may be decoded using *intra* *prediction* or *inter prediction* using at most two *motion vectors* and *reference indices* to *predict* the sample values of each *block*.
  6. **bin**: One bit of a *bin string*.
  7. **binarization**: A set of *bin strings* for all possible values of a *syntax element*.
  8. **binarization process**: A unique mapping process of all possible values of a *syntax element* onto a set of *bin strings*.
  9. **bin string**: A string of *bins*. A bin string is an intermediate binary representation of values of *syntax elements* from the *binarization* of the *syntax element*.
  10. **bi-predictive slice:** See *B slice***.**
  11. **bitstream**: A sequence of bits that forms the representation of *coded pictures* and associated data forming one or more *coded video sequences*. Bitstream is a collective term used to refer either to a *NAL unit stream* or a *byte stream*.
  12. **block**: An MxN (M-column by N-row) array of samples, or an MxN array of *transform coefficients*.
  13. **broken link**: A location in a *bitstream* at which it is indicated that some subsequent *pictures* in *decoding order* may contain serious visual artefacts due to unspecified operations performed in the generation of the *bitstream*.
  14. **byte**: A sequence of 8 bits, written and read with the most significant bit on the left and the least significant bit on the right. When represented in a sequence of data bits, the most significant bit of a byte is first.
  15. **byte-aligned**: A position in a *bitstream* is byte-aligned when the position is an integer multiple of 8 bits from the position of the first bit in the *bitstream*. A bit or *byte* or *syntax element* is said to be byte-aligned when the position at which it appears in a *bitstream* is byte-aligned.
  16. **byte stream**: An encapsulation of a *NAL unit stream* containing *start code prefixes* and *NAL units* as specified in Annex B.
  17. **can**: A term used to refer to behaviour that is allowed, but not necessarily required*.*
  18. **chroma**: An adjective specifying that a sample array or single sample is representing one of the two colour difference signals related to the primary colours. The symbols used for a chroma array or sample are Cb and Cr.

NOTE – The term chroma is used rather than the term chrominance in order to avoid the implication of the use of linear light transfer characteristics that is often associated with the term chrominance.

* 1. **clean random access (CRA) access unit**: An *access unit* in which the *primary coded picture* is a *CRA picture*.
  2. **clean random access (CRA) picture**: A *coded picture* containing only *I slices* and for which each *slice* has nal\_unit\_type equal to 4; all *coded pictures* that follow the CRA picture both in *decoding order* and *output order* shall not use *inter prediction* from any *picture* that precedes the CRA picture either in *decoding order* or *output order*; and any *picture* that precedes the CRA picture in *decoding order* also precedes the CRA picture in *output order*.
  3. **coded picture**: A *coded representation* of a *picture*.
  4. **coded picture buffer (CPB)**: A first-in first-out buffer containing *access units* in *decoding order* specified in the *hypothetical reference decoder* in Annex 0.
  5. **coded representation**: A data element as represented in its coded form.
  6. **coded slice NAL unit**: A *NAL unit* containing a *slice*.
  7. **coded video sequence**: A sequence of *access units* that consists, in decoding order, of an *IDR access unit* followed by zero or more non-IDR *access* *units* including all subsequent *access units* up to but not including any subsequent *IDR access unit*.
  8. **column**: An integer number of *treeblocks*. *Columns* are delineated from one another by vertical boundaries that extend from the top boundary to the bottom boundary of the *picture* and are ordered consecutively from left to right in the *picture*. For the *primary coded picture*, the division of each *picture* into *columns* is a *partitioning*.
  9. **component**: An array or single sample from one of the three arrays (*luma* and two *chroma*) that make up a *picture* in 4:2:0, 4:2:2, or 4:4:4 colour format or the array or a single sample of the array that make up a *picture* in monochrome format.
  10. **context variable**: A variable specified for the *adaptive binary arithmetic decoding* *process* of a *bin* by an equation containing recently decoded *bins*.
  11. **DC transform coefficient**: A *transform coefficient* for which the *frequency index* is zero in all dimensions.
  12. **decoded picture**: A *decoded picture* is derived by decoding a *coded picture*.
  13. **decoded picture buffer (DPB)**: A buffer holding *decoded pictures* for reference, output reordering, or output delay specified for the *hypothetical reference decoder* in Annex 0.
  14. **decoder**: An embodiment of a *decoding process*.
  15. **decoder under test (DUT)**: A *decoder* that is tested for conformance to this Recommendation | International Standard by operating the *hypothetical stream scheduler* to deliver a conforming *bitstream* to the *decoder* and to the *hypothetical reference decoder* and comparing the values and timing of the output of the two *decoders*.
  16. **decoding order**: The order in which *syntax elements* are processed by the *decoding process*.
  17. **decoding process**: The process specified in this Recommendation | International Standard that reads a *bitstream* and derives *decoded* *pictures* from it.
  18. **display process**: A process not specified in this Recommendation | International Standard having, as its input, the cropped decoded *pictures* that are the output of the *decoding process.*
  19. **emulation prevention byte**: A *byte* equal to 0x03 that may be present within a *NAL unit*. The presence of emulation prevention bytes ensures that no sequence of consecutive *byte-aligned* *bytes* in the *NAL unit* contains a *start code prefix*.
  20. **encoder**: An embodiment of an *encoding process*.
  21. **encoding process**: A process, not specified in this Recommendation | International Standard, that produces a *bitstream* conforming to this Recommendation | International Standard.
  22. **flag**: A variable that can take one of the two possible values 0 and 1.
  23. **frequency index**: A one-dimensional or two-dimensional index associated with a *transform coefficient* prior to an *inverse transform* part of the *decoding process.*
  24. **hypothetical reference decoder (HRD)**: A hypothetical *decoder* model that specifies constraints on the variability of conforming *NAL unit streams* or conforming *byte streams* that an encoding process may produce.
  25. **hypothetical stream scheduler (HSS)**: A hypothetical delivery mechanism for the timing and data flow of the input of a *bitstream* into the *hypothetical reference decoder*. The HSS is used for checking the conformance of a *bitstream* or a *decoder*.
  26. **I slice**: A *slice* that is decoded using *intra prediction* only.
  27. **informative**: A term used to refer to content provided in this Recommendation | International Standard that is not an integral part of this Recommendation | International Standard. Informative content does not establish any mandatory requirements for conformance to this Recommendation | International Standard.
  28. **instantaneous decoding refresh (IDR) access unit**: An *access unit* in which the *primary coded picture* is an *IDR picture*.
  29. **instantaneous decoding refresh (IDR) picture**: A *coded* *picture* for which the variable IdrPicFlag is equal to 1. An IDR picture causes the *decoding process* to mark all *reference pictures* as "unused for reference" immediately after the decoding of the IDR picture. All *coded pictures* that follow an IDR picture in *decoding order* can be decoded without *inter prediction* from any *picture* that precedes the IDR picture in *decoding order*. The first *picture* of each *coded video sequence* in *decoding order* is an IDR picture.
  30. **inter coding**: Coding of a *block*, *macroblock*, *slice*, or *picture* that uses *inter prediction*.
  31. **inter prediction**: A *prediction* derived from only data elements (e.g. sample value or motion vector) of *reference pictures* other than the current *decoded picture*.
  32. **intra coding**: Coding of a *block, macroblock*, *slice*, or *picture* that uses *intra prediction*.
  33. **intra prediction**: A *prediction* derived from only data elements (e.g. sample value) of the same decoded *slice*.
  34. **intra slice**: See *I slice*.
  35. **inverse transform**: A part of the *decoding process* by which a set of *transform coefficients* are converted into spatial-domain values, or by which a set of *transform coefficients* are converted into *DC transform coefficients*.
  36. **layer**: One of a set of syntactical structures in a non-branching hierarchical relationship. Higher layers contain lower layers. The coding layers are the *coded video sequence*, *picture*, *slice*, and *treeblock* layers.
  37. **leading picture**: A *coded* *picture* associated with a *CRA picture*, that follows the *CRA picture* in *decoding order* and precedes the *CRA picture* in *output order*.
  38. **leaf**: A terminating node of a tree that is a root node of a tree of depth 0.
  39. **level**: A defined set of constraints on the values that may be taken by the *syntax elements* and variables of this Recommendation | International Standard. The same set of levels is defined for all *profiles*, with most aspects of the definition of each level being in common across different *profiles.* Individual implementations may, within specified constraints, support a different level for each supported *profile*. In a different context, level is the value of a *transform coefficient* prior to *scaling*.
  40. **list 0 (list 1) motion vector**: A *motion vector* associated with a *reference index* pointing into *reference picture list 0* (*list 1*).
  41. **list 0 (list 1, list combination) prediction**: *Inter prediction* of the content of a *slice* using a *reference index* pointing into *reference picture list 0* (*list 1, combination*).
  42. **luma**: An adjective specifying that a sample array or single sample is representing the monochrome signal related to the primary colours. The symbol or subscript used for luma is Y or L.

NOTE – The term luma is used rather than the term luminance in order to avoid the implication of the use of linear light transfer characteristics that is often associated with the term luminance. The symbol L is sometimes used instead of the symbol Y to avoid confusion with the symbol y as used for vertical location.

* 1. **may**: A term used to refer to behaviour that is allowed, but not necessarily required*.* In some places where the optional nature of the described behaviour is intended to be emphasized, the phrase "may or may not" is used to provide emphasis.
  2. **motion vector**: A two-dimensional vector used for *inter prediction* that provides an offset from the coordinates in the *decoded picture* to the coordinates in a *reference picture*.
  3. **must**: A term used in expressing an observation about a requirement or an implication of a requirement that is specified elsewhere in this Recommendation | International Standard. This term is used exclusively in an *informative* context.
  4. **NAL unit**: A *syntax structure* containing an indication of the type of data to follow and *bytes* containing that data in the form of an *RBSP* interspersed as necessary with *emulation prevention bytes*.
  5. **NAL unit stream**: A sequence of *NAL units*.
  6. **non-reference picture**: A *picture* coded with nal\_ref\_flag equal to 0. A *non-reference picture* is not used for *inter prediction* of any other *pictures*.
  7. **note**: A term used to prefix *informative* remarks. This term is used exclusively in an *informative* context.
  8. **output order**: The order in which the *decoded* *pictures* are output from the *decoded picture buffer*.
  9. **P slice**: A *slice* that may be decoded using *intra* *prediction* or *inter prediction* using at most one *motion vector* and *reference index* to *predict* the sample values of each *block*.
  10. **parameter**: A *syntax element* of a *sequence parameter set* or a *picture parameter set*. Parameter is also used as part of the defined term *quantisation parameter*.
  11. **partitioning**: The division of a set into subsets such that each element of the set is in exactly one of the subsets.
  12. **picture**: [Ed. (TW) define]
  13. **picture parameter set**: A *syntax structure* containing *syntax elements* that apply to zero or more entire *coded pictures* as determined by the pic\_parameter\_set\_id *syntax element* found in each *slice header.*
  14. **picture order count**: A variable that is associated with each *coded picture* and has a value that is increasing with increasing *picture* position in *output order* relative to the *IDR picture* in *decoding order*.
  15. **prediction**: An embodiment of the *prediction process*.
  16. **prediction process**: The use of a *predictor* to provide an estimate of the data element (e.g. sample value or motion vector) currently being decoded.
  17. **predictive slice**: See *P slice*.
  18. **predictor**: A combination of specified values or previously decoded data elements (e.g. sample value or motion vector) used in the *decoding process* of subsequent data elements.
  19. **primary coded picture**: The coded representation of a *picture* to be used by the *decoding process* for a bitstream conforming to this Recommendation | International Standard. The primary coded picture contains all *treeblocks* of the *picture.* The only *pictures* that have a normative effect on the *decoding process* are primary coded pictures.
  20. **profile**: A specified subset of the syntax of this Recommendation | International Standard.
  21. **quadtree**: A *tree* in which a parent node can be split into four child nodes. A child node may become parent node for another plit into four child nodes.
  22. **quantisation parameter**: A variable used by the *decoding process* for *scaling* of *transform coefficient levels*.
  23. **random access**: The act of starting the decoding process for a *bitstream* at a point other than the beginning of the stream.
  24. **raster scan**: A mapping of a rectangular two-dimensional pattern to a one-dimensional pattern such that the first entries in the one-dimensional pattern are from the first top row of the two-dimensional pattern scanned from left to right, followed similarly by the second, third, etc., rows of the pattern (going down) each scanned from left to right.
  25. **raw byte sequence payload (RBSP):** A *syntax structure* containing an integer number of *bytes* that is encapsulated in a *NAL unit*. An RBSP is either empty or has the form of a *string of data bits* containing *syntax elements* followed by an *RBSP stop bit* and followed by zero or more subsequent bits equal to 0.
  26. **raw byte sequence payload (RBSP) stop bit:** A bit equal to 1 present within a *raw byte sequence payload (RBSP)* after a *string of data bits*. The location of the end of the *string of data bits* within an *RBSP* can be identified by searching from the end of the *RBSP* for the *RBSP stop bit*, which is the last non-zero bit in the *RBSP.*
  27. **recovery point**: A point in the *bitstream* at which the recovery of an exact or an approximate representation of the *decoded pictures* represented by the *bitstream* is achieved after a *random access* or *broken link*.
  28. **reference index**: An index into a *reference picture list*.
  29. **reference picture**: A *picture* with nal\_ref\_flag equal to 1. A *reference picture* contains samples that may be used for *inter prediction* in the *decoding process* of subsequent *pictures* in *decoding order*.
  30. **reference picture list**: A list of *reference pictures* that is used for *uni-prediction* of a *P* or *B slice.* For the *decoding process* of a *P slice,* there is one reference picture list*.* For the *decoding process* of a *B slice*, there are two reference picture lists (list 0 and list 1) and a reference picture lists combination*.* [Ed. (TW) the latter may not be true anymore]
  31. **reference picture list 0**: A *reference picture list* used for *inter prediction* of a *P* or *B* *slice*. All *inter prediction* used for *P* *slices* uses reference picture list 0. Reference picture list 0 is one of two *reference picture lists* used for *bi-prediction* for a *B slice*, with the other being *reference picture list*. [Ed. (TW) the latter may not be true anymore]
  32. **reference picture list 1**: A *reference picture list* used for *bi-prediction* of a *B slice*. Reference picture list 1 is one of two *reference picture lists* used for *bi-prediction* for a *B slice*, with the other being *reference picture list 0*.
  33. **reference picture lists combination**: A reference picture list used for uni-prediction of a B slice. Reference picture lists combination is derived from the entries of the reference picture lists 0 and reference picture list 1.
  34. **reference picture set**: A set of *reference pictures* associated with a *picture*, consisting of all *reference pictures* that are prior to the associated *picture* in decoding order, that may be used for *inter prediction* of the associated *picture* or any *picture* following the associated *picture* in *decoding order*.
  35. **reserved**: The term reserved, when used in the clauses specifying some values of a particular *syntax element*, are for future use by ITU-T | ISO/IEC. These values shall not be used in *bitstreams* conforming to this Recommendation | International Standard, but may be used in future extensions of this Recommendation | International Standard by ITU‑T | ISO/IEC.
  36. **residual**: The decoded difference between a *prediction* of a sample or data element and its decoded value.
  37. **row**: An integer number of *treeblocks*. *Rows* are delineated from one another by horizontal boundaries that extend from the left boundary to the right boundary of the *picture* and are ordered consecutively from top to bottom in the *picture*. For the *primary coded picture*, the division of each *picture* into *rows* is *a partitioning*.
  38. **sample aspect ratio**: Specifies, for assisting the *display process*, which is not specified in this Recommendation | International Standard, the ratio between the intended horizontal distance between the columns and the intended vertical distance between the rows of the *luma* sample array in a *picture*. Sample aspect ratio is expressed as *h*:*v*, where *h* is horizontal width and *v* is vertical height (in arbitrary units of spatial distance).
  39. **scaling**: The process of multiplying *transform coefficient levels* by a factor, resulting in *transform coefficients*.
  40. **sequence parameter set**: A *syntax structure* containing *syntax elements* that apply to zero or more entire *coded video sequences* as determined by the content of a seq\_parameter\_set\_id *syntax element* found in the *picture parameter set* referred to by the pic\_parameter\_set\_id *syntax element* found in each *slice header.*
  41. **shall**: A term used to express mandatory requirements for conformance to this Recommendation | International Standard. When used to express a mandatory constraint on the values of *syntax elements* or on the results obtained by operation of the specified *decoding process*, it is the responsibility of the *encoder* to ensure that the constraint is fulfilled. When used in reference to operations performed by the *decoding process*, any *decoding process* that produces identical results to the *decoding process* described herein conforms to the *decoding process* requirements of this Recommendation | International Standard*.*
  42. **should**: A term used to refer to behaviour of an implementation that is encouraged to be followed under anticipated ordinary circumstances, but is not a mandatory requirement for conformance to this Recommendation | International Standard.
  43. **slice**: An integer number of *treeblocks* ordered consecutively in the *raster scan*. For the *primary coded picture*, the division of each *picture* into slices is a *partitioning*. The *treeblock addresses* are derived from the first *treeblock address* in a slice (as represented in the *slice header*)*.*
  44. **slice header**: A part of a coded *slice* containing the data elements pertaining to the first or all *macroblocks* represented in the *slice*.
  45. **source**: Term used to describe the video material or some of its attributes before encoding.
  46. **start code prefix**: A unique sequence of three *bytes* equal to 0x000001 embedded in the *byte stream* as a prefix to each *NAL unit.* The location of a start code prefix can be used by a *decoder* to identify the beginning of a new *NAL unit* and the end of a previous *NAL unit*. Emulation of start code prefixes is prevented within *NAL units* by the inclusion of *emulation prevention bytes*.
  47. **string of data bits (SODB)**: A sequence of some number of bits representing *syntax elements* present within a *raw byte sequence payload* prior to the *raw byte sequence payload stop bit.* Within an SODB, the left-most bit is considered to be the first and most significant bit, and the right-most bit is considered to be the last and least significant bit.
  48. **syntax element**: An element of data represented in the *bitstream*.
  49. **syntax structure**: Zero or more *syntax elements* present together in the *bitstream* in a specified order*.*
  50. **tile**: An integer number of *treeblocks* co-occurring in one *column* and one *row*, ordered consecutively in the *raster scan* within the *tile*. For the *primary coded picture*, the division of each *picture* into *tiles* is a *partitioning*. *Tiles* are ordered consecutively in the *raster scan* within the *picture*. Although a *slice* contains treeblocks that are consecutive in the *raster scan* within a *tile*, these *treeblocks* are not necessarily consecutive in the *raster scan* within the picture.
  51. **transform coefficient**: A scalar quantity, considered to be in a frequency domain, that is associated with a particular one-dimensional or two-dimensional *frequency index* in an *inverse transform* part of the *decoding process*.
  52. **transform coefficient level**: An integer quantity representing the value associated with a particular two‑dimensional frequency index in the *decoding process* prior to *scaling* for computation of a *transform coefficient* value.
  53. **tree**: A tree is a finite ste of nodes with a unique root node. A terminating node is called a *leaf*.
  54. **treeblock**: A NxN *block* of *luma* samples and two corresponding *blocks* of *chroma* samples of a *picture* that has three sample arrays, or a NxN *block* of samples of a monochrome *picture* or a *picture* that is coded using three separate colour planes. The division of a *slice* into treeblocks is a *partitioning*.
  55. **treeblock address**: A treeblock address is the index of a *treeblock* in a *treeblock* *raster scan* of the *picture* starting with zero for the top-left *treeblock* in a *picture*.
  56. **treeblock location**: The two-dimensional coordinates of a *treeblock* in a *picture* denoted by ( x, y ). For the top left *treeblock* of the *picture* ( x, y ) is equal to ( 0, 0 ). x is incremented by 1 for each *treeblock* column from left to right. The value of y is incremented by 1 for each *treeblock* row from top to bottom.
  57. **treeblock partition**: A *block* of *luma* samples and two corresponding *blocks* of *chroma* samples resulting from a *partitioning* of a *treeblock* for *inter prediction* for a *picture* that has three sample arrays or a *block* of *luma* samples resulting from a *partitioning* of a *treeblock* for *inter prediction* for a monochrome *picture* or a *picture* that is coded using three separate colour planes.
  58. **universal unique identifier (UUID)**: An identifier that is unique with respect to the space of all universal unique identifiers.
  59. **unspecified:** The term unspecified, when used in the clauses specifying some values of a particular *syntax element*, indicates that the values have no specified meaning in this Recommendation | International Standard and will not have a specified meaning in the future as an integral part of this Recommendation | International Standard.
  60. **variable length coding (VLC)**: A reversible procedure for entropy coding that assigns shorter bit strings to *symbols* expected to be more frequent and longer bit strings to *symbols* expected to be less frequent.
  61. **VCL NAL unit**: A collective term for *coded slice NAL units*.
  62. **zig-zag scan**: A specific sequential ordering of *transform coefficient levels* from (approximately) the lowest spatial frequency to the highest.

# Abbreviations

For the purposes of this Recommendation | International Standard, the following abbreviations apply:

CABAC Context-based Adaptive Binary Arithmetic Coding

CBR Constant Bit Rate

CRA Clean Random Access

CPB Coded Picture Buffer

DPB Decoded Picture Buffer

DUT Decoder under test

FIFO First-In, First-Out

HRD Hypothetical Reference Decoder

HSS Hypothetical Stream Scheduler

IDR Instantaneous Decoding Refresh

LSB Least Significant Bit

MSB Most Significant Bit

NAL Network Abstraction Layer

RBSP Raw Byte Sequence Payload

SEI Supplemental Enhancement Information

SODB String Of Data Bits

TB Treeblock

UUID Universal Unique Identifier

VBR Variable Bit Rate

VCL Video Coding Layer

VLC Variable Length Coding

VUI Video Usability Information

# Conventions

NOTE – The mathematical operators used in this Specification are similar to those used in the C programming language. However, integer division and arithmetic shift operations are specifically defined. Numbering and counting conventions generally begin from 0.

## Arithmetic operators

The following arithmetic operators are defined as follows:

 Addition

− Subtraction (as a two-argument operator) or negation (as a unary prefix operator)

\* Multiplication, including matrix multiplication

x y Exponentiation. Specifies x to the power of y. In other contexts, such notation is used for superscripting not intended for interpretation as exponentiation.

/ Integer division with truncation of the result toward zero. For example, 7/4 and −7/−4 are truncated to 1 and −7/4 and 7/−4 are truncated to −1.

 Used to denote division in mathematical equations where no truncation or rounding is intended.

 Used to denote division in mathematical equations where no truncation or rounding is intended.

 The summation of f( i ) with i taking all integer values from x up to and including y.

x % y Modulus. Remainder of x divided by y, defined only for integers x and y with x >= 0 and y > 0.

## Logical operators

The following logical operators are defined as follows:

x && y Boolean logical "and" of x and y.

x | | y Boolean logical "or" of x and y.

! Boolean logical "not".

x ? y : z If x is TRUE or not equal to 0, evaluates to the value of y; otherwise, evaluates to the value of z.

## Relational operators

The following relational operators are defined as follows:

 Greater than.

 Greater than or equal to.

 Less than.

 Less than or equal to.

  Equal to.

! Not equal to.

When a relational operator is applied to a syntax element or variable that has been assigned the value "na" (not applicable), the value "na" is treated as a distinct value for the syntax element or variable. The value "na" is considered not to be equal to any other value.

## Bit-wise operators

The following bit-wise operators are defined as follows:

& Bit-wise "and". When operating on integer arguments, operates on a two's complement representation of the integer value. When operating on a binary argument that contains fewer bits than another argument, the shorter argument is extended by adding more significant bits equal to 0.

| Bit-wise "or". When operating on integer arguments, operates on a two's complement representation of the integer value. When operating on a binary argument that contains fewer bits than another argument, the shorter argument is extended by adding more significant bits equal to 0.

^ Bit-wise "exclusive or". When operating on integer arguments, operates on a two's complement representation of the integer value. When operating on a binary argument that contains fewer bits than another argument, the shorter argument is extended by adding more significant bits equal to 0.

x >> y Arithmetic right shift of a two's complement integer representation of x by y binary digits. This function is defined only for positive integer values of y. 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 << y Arithmetic left shift of a two's complement integer representation of x by y binary digits. This function is defined only for positive integer values of y. Bits shifted into the LSBs as a result of the left shift have a value equal to 0.

## Assignment operators

The following arithmetic operators are defined as follows:

 Assignment operator.

  Increment, i.e., *x*  is equivalent to *x*  *x*  1; when used in an array index, evaluates to the value of the variable prior to the increment operation.

− − Decrement, i.e., *x*− − is equivalent to *x*  *x* − 1; when used in an array index, evaluates to the value of the variable prior to the decrement operation.

+= Increment by amount specified, i.e., x += 3 is equivalent to x = x + 3, and x += (−3) is equivalent to x = x + (−3).

−= Decrement by amount specified, i.e., x −= 3 is equivalent to x = x − 3, and x −= (−3) is equivalent to x = x − (−3).

## Range notation

The following notation is used to specify a range of values:

x = y..z x takes on integer values starting from y to z, inclusive, with x, y, and z being integer numbers.

## Mathematical functions

The following mathematical functions are defined as follows:

Abs( x )   (5‑1)

Ceil( x ) the smallest integer greater than or equal to x. (5‑2)

Clip1Y( x ) = Clip3( 0, ( 1 << BitDepthY ) − 1, x ) (5‑3)

Clip1C( x ) = Clip3( 0, ( 1 << BitDepthC ) − 1, x ) (5‑4)

Clip3( x, y, z ) =  (5‑5)

Floor( x ) the greatest integer less than or equal to x. (5‑6)

InverseRasterScan( a, b, c, d, e ) =  (5‑7)

Log2( x ) returns the base-2 logarithm of x. (5‑8)

Log10( x ) returns the base-10 logarithm of x. (5‑9)

Median( x, y, z ) = x + y + z − Min( x, Min( y, z ) ) − Max( x, Max( y, z ) ) (5‑10)

Min( x, y ) =  (5‑11)

Max( x, y ) =  (5‑12)

Round( x ) = Sign( x ) \* Floor( Abs( x ) + 0.5 ) (5‑13)

Sign( x )   (5‑14)

Sqrt( x ) =  (5‑15)

## Order of operation precedence

When order of precedence in an expression is not indicated explicitly by use of parentheses, the following rules apply:

– operations of a higher precedence are evaluated before any operation of a lower precedence,

– operations of the same precedence are evaluated sequentially from left to right.

Table 5‑1 specifies the precedence of operations from highest to lowest; a higher position in the table indicates a higher precedence.

NOTE – For those operators that are also used in the C programming language, the order of precedence used in this Specification is the same as used in the C programming language.

Table 5‑1 – Operation precedence from highest (at top of table) to lowest (at bottom of table)

|  |
| --- |
| **operations (with operands x, y, and z)** |
| "x++", "x− −" |
| "!x", "−x" (as a unary prefix operator) |
| xy |
| "x \* y", "x / y", "x  y""", "x % y" |
| "x + y", "x − y" (as a two-argument operator), "" |
| "x << y", "x >> y" |
| "x < y", "x <= y", "x > y", "x >= y" |
| "x = = y", "x != y" |
| "x & y" |
| "x | y" |
| "x && y" |
| "x | | y" |
| "x ? y : z" |
| "x = y", "x += y", "x −= y" |

## Variables, syntax elements, and tables

Syntax elements in the bitstream are represented in **bold** type. Each syntax element is described by its name (all lower case letters with underscore characters), its one or two syntax categories, and one or two descriptors for its method of coded representation. The decoding process behaves according to the value of the syntax element and to the values of previously decoded syntax elements. When a value of a syntax element is used in the syntax tables or the text, it appears in regular (i.e., not bold) type.

In some cases the syntax tables may use the values of other variables derived from syntax elements values. Such variables appear in the syntax tables, or text, named by a mixture of lower case and upper case letter and without any underscore characters. Variables starting with an upper case letter are derived for the decoding of the current syntax structure and all depending syntax structures. Variables starting with an upper case letter may be used in the decoding process for later syntax structures without mentioning the originating syntax structure of the variable. Variables starting with a lower case letter are only used within the subclause in which they are derived.

In some cases, "mnemonic" names for syntax element values or variable values are used interchangeably with their numerical values. Sometimes "mnemonic" names are used without any associated numerical values. The association of values and names is specified in the text. The names are constructed from one or more groups of letters separated by an underscore character. Each group starts with an upper case letter and may contain more upper case letters.

NOTE – The syntax is described in a manner that closely follows the C-language syntactic constructs.

Functions that specify properties of the current position in the bitstream are referred to as syntax functions. These functions are specified in subclause 0 and assume the existence of a bitstream pointer with an indication of the position of the next bit to be read by the decoding process from the bitstream. Syntax functions are described by their names, which are constructed as syntax element names and end with left and right round parentheses including zero or more variable names (for definition) or values (for usage), separated by commas (if more than one variable).

Functions that are not syntax functions (including mathematical functions specified in subclause 5.7) are described by their names, which start with an upper case letter, contain a mixture of lower and upper case letters without any underscore character, and end with left and right parentheses including zero or more variable names (for definition) or values (for usage) separated by commas (if more than one variable).

A one-dimensional array is referred to as a list. A two-dimensional array is referred to as a matrix. Arrays can either be syntax elements or variables. Subscripts or square parentheses are used for the indexing of arrays. In reference to a visual depiction of a matrix, the first subscript is used as a row (vertical) index and the second subscript is used as a column (horizontal) index. The indexing order is reversed when using square parentheses rather than subscripts for indexing. Thus, an element of a matrix s at horizontal position x and vertical position y may be denoted either as s[ x, y ] or as syx.

Binary notation is indicated by enclosing the string of bit values by single quote marks. For example, '01000001' represents an eight-bit string having only its second and its last bits (counted from the most to the least significant bit) equal to 1.

Hexadecimal notation, indicated by prefixing the hexadecimal number by "0x", may be used instead of binary notation when the number of bits is an integer multiple of 4. For example, 0x41 represents an eight-bit string having only its second and its last bits (counted from the most to the least significant bit) equal to 1.

Numerical values not enclosed in single quotes and not prefixed by "0x" are decimal values.

A value equal to 0 represents a FALSE condition in a test statement. The value TRUE is represented by any value different from zero.

## Text description of logical operations

In the text, a statement of logical operations as would be described in pseudo-code as

if( condition 0 )  
 statement 0  
else if ( condition 1 )  
 statement 1  
…  
else /\* informative remark on remaining condition \*/  
 statement n

may be described in the following manner:

... as follows / ... the following applies.

– If condition 0, statement 0

– Otherwise, if condition 1, statement 1

– …

– Otherwise (informative remark on remaining condition), statement n

Each "If ... Otherwise, if ... Otherwise, ..." statement in the text is introduced with "... as follows" or "... the following applies" immediately followed by "If ... ". The last condition of the "If ... Otherwise, if ... Otherwise, ..." is always an "Otherwise, ...". Interleaved "If ... Otherwise, if ... Otherwise, ..." statements can be identified by matching "... as follows" or "... the following applies" with the ending "Otherwise, ...".

In the text, a statement of logical operations as would be described in pseudo-code as

if( condition 0a && condition 0b )  
 statement 0  
else if ( condition 1a | | condition 1b )  
 statement 1  
…  
else  
 statement n

may be described in the following manner:

... as follows / ... the following applies.

– If all of the following conditions are true, statement 0

– condition 0a

– condition 0b

– Otherwise, if any of the following conditions are true, statement 1

– condition 1a

– condition 1b

– …

– Otherwise, statement n

In the text, a statement of logical operations as would be described in pseudo-code as:

if( condition 0 )  
 statement 0  
if ( condition 1 )  
 statement 1

may be described in the following manner:

When condition 0, statement 0

When condition 1, statement 1

## Processes

Processes are used to describe the decoding of syntax elements. A process has a separate specification and invoking. All syntax elements and upper case variables that pertain to the current syntax structure and depending syntax structures are available in the process specification and invoking. A process specification may also have a lower case variable explicitly specified as the input. Each process specification has explicitly specified an output. The output is a variable that can either be an upper case variable or a lower case variable.

When invoking a process, the assignment of variables is specified as follows.

– If the variables at the invoking and the process specification do not have the same name, the variables are explicitly assigned to lower case input or output variables of the process specification.

– Otherwise (the variables at the invoking and the process specification have the same name), assignment is implied.

In the specification of a process, a specific macroblock may be referred to by the variable name having a value equal to the address of the specific macroblock.

# Source, coded, decoded and output data formats, scanning processes, and neighbouring relationships

## Bitstream formats

This subclause specifies the relationship between the NAL unit stream and byte stream, either of which are referred to as the bitstream.

The bitstream can be in one of two formats: the NAL unit stream format or the byte stream format. The NAL unit stream format is conceptually the more "basic" type. It consists of a sequence of syntax structures called NAL units. This sequence is ordered in decoding order. There are constraints imposed on the decoding order (and contents) of the NAL units in the NAL unit stream.

The byte stream format can be constructed from the NAL unit stream format by ordering the NAL units in decoding order and prefixing each NAL unit with a start code prefix and zero or more zero-valued bytes to form a stream of bytes. The NAL unit stream format can be extracted from the byte stream format by searching for the location of the unique start code prefix pattern within this stream of bytes. Methods of framing the NAL units in a manner other than use of the byte stream format are outside the scope of this Recommendation | International Standard. The byte stream format is specified in Annex B.

## Source, decoded, and output picture formats

This subclause specifies the relationship between source and decoded pictures that is given via the bitstream.

The video source that is represented by the bitstream is a sequence of pictures in decoding order.

The source and decoded pictures are each comprised of one or more sample arrays:

– Luma (Y) only (monochrome).

– Luma and two chroma (YCbCr or YCgCo).

– Green, Blue and Red (GBR, also known as RGB).

– Arrays representing other unspecified monochrome or tri-stimulus colour samplings (for example, YZX, also known as XYZ).

For convenience of notation and terminology in this Specification, the variables and terms associated with these arrays are referred to as luma (or L or Y) and chroma, where the two chroma arrays are referred to as Cb and Cr; regardless of the actual colour representation method in use. The actual colour representation method in use can be indicated in syntax that is specified in Annex E.

The variables SubWidthC, and SubHeightC are specified in Table 6‑1, depending on the chroma format sampling structure, which is specified through chroma\_format\_idc and separate\_colour\_plane\_flag. An entry marked as "-" in Table 6‑1 denotes an undefined value for SubWidthC or SubHeightC. Other values of chroma\_format\_idc, SubWidthC, and SubHeightC may be specified in the future by ITU‑T | ISO/IEC.

Table 6‑1 – SubWidthC, and SubHeightC values derived from   
chroma\_format\_idc and separate\_colour\_plane\_flag

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **chroma\_format\_idc** | **separate\_colour\_plane\_flag** | **Chroma Format** | **SubWidthC** | **SubHeightC** |
| 0 | 0 | monochrome | - | - |
| 1 | 0 | 4:2:0 | 2 | 2 |
| 2 | 0 | 4:2:2 | 2 | 1 |
| 3 | 0 | 4:4:4 | 1 | 1 |
| 3 | 1 | 4:4:4 | - | - |

In monochrome sampling there is only one sample array, which is nominally considered the luma array.

In 4:2:0 sampling, each of the two chroma arrays has half the height and half the width of the luma array.

In 4:2:2 sampling, each of the two chroma arrays has the same height and half the width of the luma array.

In 4:4:4 sampling, depending on the value of separate\_colour\_plane\_flag, the following applies.

– If separate\_colour\_plane\_flag is equal to 0, each of the two chroma arrays has the same height and width as the luma array.

– Otherwise (separate\_colour\_plane\_flag is equal to 1), the three colour planes are separately processed as monochrome sampled pictures.

The number of bits necessary for the representation of each of the samples in the luma and chroma arrays in a video sequence is in the range of 8 to 14, and the number of bits used in the luma array may differ from the number of bits used in the chroma arrays.

When the value of chroma\_format\_idc is equal to 1, the nominal vertical and horizontal relative locations of luma and chroma samples in pictures are shown in Figure 6‑1. Alternative chroma sample relative locations may be indicated in video usability information (see Annex E).



Figure 6‑1 – Nominal vertical and horizontal locations of 4:2:0 luma and chroma samples in a picture [Ed. Re-draw figure]

When the value of chroma\_format\_idc is equal to 2, the chroma samples are co-sited with the corresponding luma samples and the nominal locations in a picture are as shown in Figure 6‑2.



Figure 6‑2 – Nominal vertical and horizontal locations of 4:2:2 luma and chroma samples in a picture [Ed. Re-draw figure]

When the value of chroma\_format\_idc is equal to 3, all array samples are co-sited for all cases of pictures and the nominal locations in a picture are as shown in Figure 6‑3.



Figure 6‑3 – Nominal vertical and horizontal locations of 4:4:4 luma and chroma samples in a picture [Ed. Re-draw figure]

The samples are processed in units of treeblocks. The luma array for each treeblock in samples in both width and height is

TbSize = 64 (6‑1)  
Log2TbSize = 6 (6‑2)

The variables TbWidthC and TbHeightC, which specify the width and height, respectively, of the chroma arrays for each treeblock, are derived as follows.

– If chroma\_format\_idc is equal to 0 (monochrome) or separate\_colour\_plane\_flag is equal to 1, TbWidthC and TbHeightC are both equal to 0.

– Otherwise, TbWidthC and TbHeightC are derived as

TbWidthC = TbSize / SubWidthC (6‑3)  
TbHeightC = TbSize / SubHeightC (6‑4)

## Spatial subdivision of pictures, slices and tiles

This subclause specifies how a picture is partitioned into slices, tiles and treeblocks. Pictures are divided into slices and tiles. A slice is a sequence of treeblocks. Likewise, a tile is a sequence of treeblocks.

Each treeblock is comprised of one Ntb x Ntb luma array and, when the chroma sampling format is not equal to 4:0:0 and separate\_colour\_plane\_flag is equal to 0, two corresponding chroma sample arrays. When separate\_colour\_plane\_flag is equal to 1, each treeblock is comprised of one Ntb x Ntb luma or chroma sample array. For example, a picture may be divided into two slices as shown in Figure 6‑4. As another example, a picture may be divided into three tiles as shown in Figure 6‑5.

Unlike slices, tiles are always rectangular and always contain an integer number of treeblocks in treeblock raster scan. In general, slices and tiles do not contain the same sequence of treeblocks. A tile may comprise treeblocks contained in more than one slice as shown in Figure 6‑6. Similarly, a slice may comprise treeblocks contained in several tiles.

When a picture is coded using three separate colour planes (separate\_colour\_plane\_flag is equal to 1), a slice contains only treeblocks of one colour component being identified by the corresponding value of colour\_plane\_id, and each colour component array of a picture consists of slices having the same colour\_plane\_id value. Coded slices with different values of colour\_plane\_id within an access unit can be interleaved with each other under the constraint that for each value of colour\_plane\_id, the coded slice NAL units with that value colour\_plane\_id shall be in the order of increasing treeblock address for the first treeblock of each coded slice NAL unit.

NOTE – When separate\_colour\_plane\_flag is equal to 0, each treeblock of a picture is contained in exactly one slice. When separate\_colour\_plane\_flag is equal to 1, each treeblock of a colour component is contained in exactly one slice (i.e., information for each treeblock of a picture is present in exactly three slices and these three slices have different values of colour\_plane\_id).



Figure 6‑4 – A picture with 11 by 9 treeblocks that is partitioned into two slices



Figure ‑ – A picture with 13 by 8 treeblocks that is partitioned into three tiles

****

Figure ‑ – A picture with 13 by 8 treeblocks that is partitioned into three tiles and three slices

Each treeblock is assigned a partition signalling to identify the block sizes for intra or inter prediction and for transform coding. The partitioning is a recursive quadtree partitioning. The root of the quadtree is associated with the treeblock. The quadtree is split until a leaf is reached, which is referred to as the coding node. The coding node is the root node of two trees, the prediction tree and the transform tree.

The prediction tree specifies the position and size of prediction blocks. The prediction tree and associated prediction data are referred to as prediction unit.

The transform tree specifies the position and size of transform blocks. The transform tree and associated transform data are referred to as transform unit.

The splitting information for luma and chroma is identical for the prediction tree and may or may not be identical for the transform tree.

The coding node and the associated prediction and transform units form together a coding unit.

[Ed.: (WJ) block = a rectangular 2D array (one component), unit = collective term for specifying information for both luma and chroma. Don’t use ther term ‘unit’ by itself – always use the term ‘unit’ with prefix – coding unit, prediction unit or transform unit.]

## Inverse scanning processes and derivation processes for neighbours

This subclause specifies inverse scanning processes; i.e., the mapping of indices to locations, and derivation processes for neighbours.

### Treeblock address conversion process

Inputs to this process is a treeblock address tbAddr, the vector colBd[i], the vector rowBd[i], the picture width in LCUs picWidthInLCUs, and the current tile index tileIdx.

Output of the process is a converted treeblock address tbAddrRasterScan, the address of the first treeblock in a tile firstTbInTileAddr, and the tile width in LCUs tileWidthInLCUs.

The tile width in LCUs tileWidthInLCUs is derived as described by the following pseudo code.

vertTileIdx = 1  
while ( tbAddr >= rowBd[vertTileIdx] \* picWidthInLCUs )   
 vertTileIdx ++  
tileHeightInLCUs = rowBd[vertTileIdx] − rowBd[vertTileIdx − 1]   
horTileIdx = 1  
while ( ( tbAddr − rowBd[vertTileIdx − 1] \* picWidthInLCUs ) >= ( colBd[horTileIdx] \* tileHeightInLCUs ) )   
 horTileIdx ++  
tileWidthInLCUs = colBd[horTileIdx] − colBd[horTileIdx − 1]

The address of the first treeblock in a tile firstTbInTileAddr is derived as follows.

firstTbInTileAddr = rowBd[vertTileIdx − 1] \* picWidthInLCUs + (6‑5)  
 colBd[horTileIdx − 1] \* tileHeightInLCUs

The converted treeblock address tbAddrRasterScan is derived as follows.

tbVerOffsetInTileInLCUs = (tbAddr – firstTbInTileAddr) / tileWidthInLCUs  
tbHorOffsetInTileInLCUs = (tbAddr – firstTbInTileAddr) % tileWidthInLCUs  
tbAddrRasterScan = ( rowBd[vertTileIdx − 1] + tbVerOffsetInTileInLCUs ) \* picWidthInLCUs + (6‑5)  
 ( colBd[horTileIdx − 1] + tbHorOffsetInTileInLCUs )

The current tile index tileIdx is derived as follows.

tileIdx = ( vertTileIdx − 1 ) \* ( num\_tile\_columns\_minus1 +1 ) + ( horTileIdx − 1 ) (6‑5)

### Inverse treeblock scanning process

Input to this process is a converted treeblock address tbAddrRasterScan.

Output of this process is the location ( x, y ) of the upper-left luma sample for the treeblock with address tbAddrRasterScan relative to the upper-left sample of the picture.

The inverse treeblock scanning process is specified as follows.

x = InverseRasterScan( tbAddrRasterScan, 16, 16, PicWidthInSamplesL, 0 ) (6‑5)

y = InverseRasterScan( tbAddrRasterScan, 16, 16, PicWidthInSamplesL, 1 ) (6‑6)

[Ed. (TW): The text found in JCVTVC-B205 that relates to the rest of this clause requires substantial editorial and formatting work. It cannot be imported as is.]

### Derivation process for the availability of treeblock addresses

Input to this process is a treeblock address tbAddr.

Output of this process is the availability of the treeblock tbAddr.

NOTE – The meaning of availability is determined when this process is invoked.

The treeblock is marked as available, unless one of the following conditions is true in which case the treeblock shall be marked as not available:

– tbAddr < 0

– tbAddr > CurrTbAddr

– the treeblock with address tbAddr belongs to a different slice than the treeblock with address CurrTbAddr and the entropy\_slice\_flag of the slice containing the treeblock with address CurrTbAddr is equal to 0.

– tile\_boundary\_independence\_flag is equal to 1 and the treeblock with address tbAddr is contained in a different tile than the treeblock with address CurrTbAddr.

[Ed. (WJ): do we need to separate the use of slice and entropy slice? E.g. Is entropy slice a kind of slice or sub-slice? We may need the concrete definitions]

## Scanning array initialisation process

### Up-right diagonal scanning array initialisation process

Input to this process is a block width blkWidth and a block height blkHeight.

Output of this process is the array DiagScan[ pos ][ comp ]. The array index pos specify the scan position ranging from 0 to ( blkWidthSize \* blkHeightSize ) − 1. The array index comp equal to 0 specifies the horizontal component and the array index comp equal to 1 specifies the vertical component. Depending on the values of blkWidth and blkHeight, the array DiagScan is derived as follows.

– If blkWidth is less than 16 and blkHeight is less than 16, the following applies.

i = 0  
x = 0  
y = 0  
stopLoop = false  
while( !stopLoop ) {  
 while( y >= 0 ) {  
 if ( x < blkWidth && y < blkHeight ) {  
 DiagScan[ i ][ 0 ] = x  
 DiagScan[ i ][ 1 ] = y  
 i++  
 }  
 y− −  
 x++  
 }  
 y = x  
 x = 0  
 if ( i >= blkWidth \* blkHeight )   
 stopLoop = true  
}

– Otherwise (blkWidth is greater than 8 or blkHeight is greater than 8), the following applies.

x\_off = 0  
y\_off = 0  
i\_off = 0  
stopLoopSubblocks = false  
while( !stopLoopSubblocks ) {   
 i = 0  
 x = 0  
 y = 0  
 stopLoop = false  
 while( !stopLoop ) {  
 while( y >= 0 ) {  
 if ( x < 4 && y < 4 ) {  
 DiagScan [ i + i\_off ][ 0 ] = x + x\_off  
 DiagScan [ i + i\_off ][ 1 ] = y + y\_off  
 i++  
 }  
 y− −  
 x++  
 }  
 y = x  
 x = 0  
 if ( i >= 16 )   
 stopLoop = true  
 }  
 i\_off = i\_off + 16  
 if ( i\_off >= blkWidth \* blkHeight ) {  
 stopLoopSubblocks = true  
 } else{  
 do {  
 y\_off = y\_off − 4  
 x\_off = x\_off + 4  
 if ( y\_off < 0 ) {  
 y\_off = x\_off  
 x\_off = 0  
 }  
 } while( !( x\_off < blkWidth && y\_off < blkHeight ) )   
 }  
}

## Scanning order array initialisation process

Input to this process is a block widthsize blkWidth and a block height blkHeight.

Output of this process is the array ScanOrder[ scanIdx ][ pos ][ comp ]. The array index scanIdx equal to 0 specifies the up-right diagonal scan as specified in subclause 6.5.2 with blkWidth and blkHeight as inputs, scanIdx equal to 1 specifies a horizontal scan, and scanIdx equal to 2 specifies a vertical scan. The array index pos specifies the scan position ranging from 0 to ( blkWidth \* blkHeight ) − 1. The array index comp equal to 0 specifies the horizontal component and the array index comp equal to 1 specifies the vertical component. The array ScanOrder is derived as follows.

ScanOrder[0] = DiagScan

i = 0  
y = 0  
while( y < blkHeight ) {  
 x = 0  
 while( x < blkWidth ) {  
 ScanOrder[ 1 ][ i ][ 0 ] = x  
 ScanOrder[ 1 ][ i ][ 1 ] = y  
 x++  
 i++  
 }  
 y++  
}

i = 0  
x = 0  
while( x < blkWidth ) {  
 y = 0  
 while( y < blkHeight ) {  
 ScanOrder[ 2 ][ i ][ 0 ] = x  
 ScanOrder[ 2 ][ i ][ 1 ] = y  
 y++  
 i++  
 }  
 x++  
}

# Syntax and semantics

## Method of specifying syntax in tabular form

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

NOTE – An actual decoder should implement means for identifying entry points into the bitstream and means to identify and handle non-conforming bitstreams. The methods for identifying and handling errors and other such situations are not specified here.

The following table lists examples of pseudo code used to describe the syntax. When **syntax\_element** appears, it specifies that a syntax element is parsed from the bitstream and the bitstream pointer is advanced to the next position beyond the syntax element in the bitstream parsing process.

|  |  |  |
| --- | --- | --- |
|  | C | Descriptor |
| /\* A statement can be a syntax element with an associated syntax category and descriptor or can be an expression used to specify conditions for the existence, type, and quantity of syntax elements, as in the following two examples \*/ |  |  |
| **syntax\_element** | 3 | ue(v) |
| conditioning statement |  |  |
|  |  |  |
| /\* A group of statements enclosed in curly brackets is a compound statement and is treated functionally as a single statement. \*/ |  |  |
| { |  |  |
| statement |  |  |
| statement |  |  |
| … |  |  |
| } |  |  |
|  |  |  |
| /\* A "while" structure specifies a test of whether a condition is true, and if true, specifies evaluation of a statement (or compound statement) repeatedly until the condition is no longer true \*/ |  |  |
| while( condition ) |  |  |
| statement |  |  |
|  |  |  |
| /\* A "do … while" structure specifies evaluation of a statement once, followed by a test of whether a condition is true, and if true, specifies repeated evaluation of the statement until the condition is no longer true \*/ |  |  |
| do |  |  |
| statement |  |  |
| while( condition ) |  |  |
|  |  |  |
| /\* An "if … else" structure specifies a test of whether a condition is true, and if the condition is true, specifies evaluation of a primary statement, otherwise, specifies evaluation of an alternative statement. The "else" part of the structure and the associated alternative statement is omitted if no alternative statement evaluation is needed \*/ |  |  |
| if( condition ) |  |  |
| primary statement |  |  |
| else |  |  |
| alternative statement |  |  |
|  |  |  |
| /\* A "for" structure specifies evaluation of an initial statement, followed by a test of a condition, and if the condition is true, specifies repeated evaluation of a primary statement followed by a subsequent statement until the condition is no longer true. \*/ |  |  |
| for( initial statement; condition; subsequent statement ) |  |  |
| primary statement |  |  |

## Specification of syntax functions and descriptors

The functions presented here are used in the syntactical description. These functions assume the existence of a bitstream pointer with an indication of the position of the next bit to be read by the decoding process from the bitstream.

byte\_aligned( ) is specified as follows.

– If the current position in the bitstream is on a byte boundary, i.e., the next bit in the bitstream is the first bit in a byte, the return value of byte\_aligned( ) is equal to TRUE.

– Otherwise, the return value of byte\_aligned( ) is equal to FALSE.

more\_data\_in\_byte\_stream( ), which is used only in the byte stream NAL unit syntax structure specified in Annex B, is specified as follows.

– If more data follow in the byte stream, the return value of more\_data\_in\_byte\_stream( ) is equal to TRUE.

– Otherwise, the return value of more\_data\_in\_byte\_stream( ) is equal to FALSE.

more\_rbsp\_data( ) is specified as follows.

– If there is no more data in the RBSP, the return value of more\_rbsp\_data( ) is equal to FALSE.

– Otherwise, the RBSP data is searched for the last (least significant, right-most) bit equal to 1 that is present in the RBSP. Given the position of this bit, which is the first bit (rbsp\_stop\_one\_bit) of the rbsp\_trailing\_bits( ) syntax structure, the following applies.

– If there is more data in an RBSP before the rbsp\_trailing\_bits( ) syntax structure, the return value of more\_rbsp\_data( ) is equal to TRUE.

– Otherwise, the return value of more\_rbsp\_data( ) is equal to FALSE.

The method for enabling determination of whether there is more data in the RBSP is specified by the application (or in Annex B for applications that use the byte stream format).

more\_rbsp\_trailing\_data( ) is specified as follows.

– If there is more data in an RBSP, the return value of more\_rbsp\_trailing\_data( ) is equal to TRUE.

– Otherwise, the return value of more\_rbsp\_trailing\_data( ) is equal to FALSE.

next\_bits( n ) provides the next bits in the bitstream for comparison purposes, without advancing the bitstream pointer. Provides a look at the next n bits in the bitstream with n being its argument. When used within the byte stream as specified in Annex B, next\_bits( n ) returns a value of 0 if fewer than n bits remain within the byte stream.

read\_bits( n ) reads the next n bits from the bitstream and advances the bitstream pointer by n bit positions. When n is equal to 0, read\_bits( n ) is specified to return a value equal to 0 and to not advance the bitstream pointer.

The following descriptors specify the parsing process of each syntax element.

– ae(v): context-adaptive arithmetic entropy-coded syntax element. The parsing process for this descriptor is specified in subclause 9.2.

– b(8): byte having any pattern of bit string (8 bits). The parsing process for this descriptor is specified by the return value of the function read\_bits( 8 ).

– f(n): fixed-pattern bit string using n bits written (from left to right) with the left bit first. The parsing process for this descriptor is specified by the return value of the function read\_bits( n ).

– i(n): signed integer using n bits. When n is "v" in the syntax table, the number of bits varies in a manner dependent on the value of other syntax elements. The parsing process for this descriptor is specified by the return value of the function read\_bits( n ) interpreted as a two's complement integer representation with most significant bit written first.

– me(v): mapped Exp-Golomb-coded syntax element with the left bit first. The parsing process for this descriptor is specified in subclause 9.1.

– se(v): signed integer Exp-Golomb-coded syntax element with the left bit first. The parsing process for this descriptor is specified in subclause 9.1.

– te(v): truncated Exp-Golomb-coded syntax element with left bit first. The parsing process for this descriptor is specified in subclause 9.1.

– u(n): unsigned integer using n bits. When n is "v" in the syntax table, the number of bits varies in a manner dependent on the value of other syntax elements. The parsing process for this descriptor is specified by the return value of the function read\_bits( n ) interpreted as a binary representation of an unsigned integer with most significant bit written first.

– ue(v): unsigned integer Exp-Golomb-coded syntax element with the left bit first. The parsing process for this descriptor is specified in subclause 9.1.

## Syntax in tabular form

### NAL unit syntax

|  |  |
| --- | --- |
| nal\_unit( NumBytesInNALunit ) { | Descriptor |
| **forbidden\_zero\_bit** | f(1) |
| **nal\_ref\_flag** | u(1) |
| **nal\_unit\_type** | u(6) |
| NumBytesInRBSP = 0 |  |
| nalUnitHeaderBytes = 1 |  |
| if( nal\_unit\_type = = 1 | | nal\_unit\_type = = 4 | | nal\_unit\_type = = 5 ) { |  |
| **temporal\_id** | u(3) |
| **output\_flag** | u(1) |
| **reserved\_one\_4bits** | u(4) |
| nalUnitHeaderBytes += 1 |  |
| } |  |
| for( i = nalUnitHeaderBytes; i < NumBytesInNALunit; i++ ) { |  |
| if( i + 2 < NumBytesInNALunit && next\_bits( 24 ) = = 0x000003 ) { |  |
| **rbsp\_byte[** NumBytesInRBSP++ **]** | b(8) |
| **rbsp\_byte[** NumBytesInRBSP++ **]** | b(8) |
| i += 2 |  |
| **emulation\_prevention\_three\_byte** /\* equal to 0x03 \*/ | f(8) |
| } else |  |
| **rbsp\_byte[** NumBytesInRBSP++ **]** | b(8) |
| } |  |
| } |  |

### Raw byte sequence payloads and RBSP trailing bits syntax

#### Sequence parameter set RBSP syntax

|  |  |
| --- | --- |
| seq\_parameter\_set\_rbsp( ) { | Descriptor |
| **profile\_idc** | u(8) |
| **reserved\_zero\_8bits** /\* equal to 0 \***/** | u(8) |
| **level\_idc** | u(8) |
| **seq\_parameter\_set\_id** | ue(v) |
| **chroma\_format\_idc** | ue(v) |
| [Ed. (BB): Not in HM, further discuss separate\_colour\_plane\_flag] |  |
| **max\_temporal\_layers\_minus1** | u(3) |
| **pic\_width\_in\_luma\_samples** | ue(v) |
| **pic\_height\_in\_luma\_samples** | ue(v) |
| **bit\_depth\_luma\_minus8** | ue(v) |
| **bit\_depth\_chroma\_minus8** | ue(v) |
| [Ed. (BB): chroma bit depth present in HM software but not used further ] |  |
| **pcm\_enabled\_flag** | u(1) |
| if ( pcm\_enabled\_flag ) { |  |
| **pcm\_bit\_depth\_luma\_minus1** | u(4) |
| **pcm\_bit\_depth\_chroma\_minus1** | u(4) |
| } |  |
| **log2\_max\_pic\_order\_cnt\_lsb\_minus4** | ue(v) |
| **max\_num\_ref\_frames** | ue(v) |
| **num\_reorder\_frames** | ue(v) |
| **max\_dec\_frame\_buffering** | ue(v) |
| **max\_latency\_increase** | ue(v) |
| **log2\_min\_coding\_block\_size\_minus3** | ue(v) |
| **log2\_diff\_max\_min\_coding\_block\_size** | ue(v) |
| **log2\_min\_transform\_block\_size\_minus2** | ue(v) |
| **log2\_diff\_max\_min\_transform\_block\_size** | ue(v) |
| if ( pcm\_enabled\_flag ) { |  |
| **log2\_min\_pcm\_coding\_block\_size\_minus3** | ue(v) |
| **log2\_diff\_max\_min\_pcm\_coding\_block\_size** | ue(v) |
| } |  |
| **max\_transform\_hierarchy\_depth\_inter** | ue(v) |
| **max\_transform\_hierarchy\_depth\_intra** | ue(v) |
| **scaling\_list\_enable\_flag** |  |
| **chroma\_pred\_from\_luma\_enabled\_flag** | u(1) |
| **deblocking\_filter\_in\_APS\_enabled\_flag** | u(1) |
| **loop\_filter\_across\_slice\_flag** | u(1) |
| **sample\_adaptive\_offset\_enabled\_flag** | u(1) |
| **adaptive\_loop\_filter\_enabled\_flag** | u(1) |
| if ( pcm\_enabled\_flag ) |  |
| **pcm\_loop\_filter\_disable\_flag** | u(1) |
| **temporal\_id\_nesting\_flag** | u(1) |
| [Ed. (BB): x y padding syntax missing here, present in HM software ] |  |
| if ( log2\_min\_coding\_block\_size\_minus3 = = 0 ) |  |
| **inter\_4x4\_enabled\_flag** | u(1) |
| **num\_tile\_columns\_minus1** | ue(v) |
| **num\_tile\_rows\_minus1** | ue(v) |
| if ( num\_tile\_columns\_minus1 != 0 | | num\_tile\_rows\_minus1 != 0 ) { |  |
| **uniform\_spacing\_flag** | u(1) |
| if ( !uniform\_spacing\_flag ) { |  |
| for ( i = 0; i < num\_tile\_columns\_minus1; i++ ) |  |
| **column\_width[**i**]** | ue(v) |
| for ( i = 0; i < num\_tile\_rows\_minus1; i++ ) |  |
| **row\_height[**i**]** | ue(v) |
| } |  |
| **tile\_boundary\_independence\_flag** | u(1) |
| if ( tile\_boundary\_independence\_flag = = 1 ) |  |
| **loop\_filter\_across\_tile\_flag** | u(1) |
| } |  |
| **sps\_extension\_flag** | u(1) |
| if( sps\_extension\_flag ) |  |
| while( more\_rbsp\_data( ) ) |  |
| **sps\_extension\_data\_flag** | u(1) |
| rbsp\_trailing\_bits( ) |  |
| } |  |

#### Picture parameter set RBSP syntax

|  |  |
| --- | --- |
| pic\_parameter\_set\_rbsp( ) { | Descriptor |
| **pic\_parameter\_set\_id** | ue(v) |
| **seq\_parameter\_set\_id** | ue(v) |
| **num\_short\_term\_ref\_pic\_sets** | ue(v) |
| for(idx = 0; idx < num\_short\_term\_ref\_pic\_sets; idx++) |  |
| short\_term\_ref\_pic\_set( idx ) |  |
| **long\_term\_ref\_pics\_present\_flag** | u(1) |
| **entropy\_coding\_synchro** | u(v) |
| **cabac\_istate\_reset\_flag** | u(1) |
| if( entropy\_coding\_synchro ) |  |
| **num\_substreams\_minus1** | ue(v) |
| **num\_temporal\_layer\_switching\_point\_flags** | ue(v) |
| for( i = 0; i < num\_temporal\_layer\_switching\_point\_flags; i++ ) |  |
| **temporal\_layer\_switching\_point\_flag**[ i ] | u(1) |
| **num\_ref\_idx\_l0\_default\_active\_minus1** | ue(v) |
| **num\_ref\_idx\_l1\_default\_active\_minus1** | ue(v) |
| [Ed. (BB): not present in HM software ] |  |
| **pic\_init\_qp\_minus26** | se(v) |
| **constrained\_intra\_pred\_flag** | u(1) |
| **enable\_temporal\_mvp\_flag** | u(1) |
| **slice\_granularity** | u(2) |
| **max\_cu\_qp\_delta\_depth** | ue(v) |
| **chroma\_cb\_qp\_offset** | se(v) |
| **chroma\_cr\_qp\_offset** | se(v) |
| **weighted\_pred\_flag** | u(1) |
| **weighted\_bipred\_idc** | u(2) |
| **tile\_info\_present\_flag** | u(1) |
| **tile\_control\_present\_flag** | u(1) |
| if( tile\_info\_present\_flag = = 1 **) {** |  |
| **num\_tile\_columns\_minus1** | ue(v) |
| **num\_tile\_rows\_minus1** | ue(v) |
| if( num\_tile\_columns\_minus1 != 0 | | num\_tile\_rows\_minus1 != 0 ) { |  |
| **uniform\_spacing\_flag** | u(1) |
| if( !uniform\_spacing\_flag ) { |  |
| for( i = 0; i < num\_tile\_columns\_minus1; i++ ) |  |
| **column\_width[**i**]** | ue(v) |
| for( i = 0; i < num\_tile\_rows\_minus1; i++ ) |  |
| **row\_height[**i**]** | ue(v) |
| } |  |
| } |  |
| **}** |  |
| if ( tile\_control\_present\_flag ) { |  |
| if ( num\_tile\_columns\_minus1 != 0 | | num\_tile\_rows\_minus1 != 0 ) { |  |
| **tile\_boundary\_independence\_flag** | u(1) |
| if ( tile\_boundary\_independence\_flag = = 1 ) |  |
| **loop\_filter\_across\_tile\_flag** | u(1) |
| } |  |
| } |  |
| **pps\_extension\_flag** | u(1) |
| if( pps\_extension\_flag ) |  |
| while( more\_rbsp\_data( ) ) |  |
| **pps\_extension\_data\_flag** | u(1) |
| rbsp\_trailing\_bits( ) |  |
| } |  |

#### Scaling list data syntax

|  |  |
| --- | --- |
| scaling\_list\_param( ) { | **Descriptor** |
| **scaling\_list\_present\_flag** | u(1) |
| if( scaling\_list\_present\_flag ) |  |
| for( SizeID = 0; SizeID < 4; SizeID++ ) |  |
| for( MatrixID = 0; MatrixID < (SizeID = = 3) ? 2 : 6; MatrixID++ ) { |  |
| **scaling\_list\_pred\_mode\_flag** | u(1) |
| if( !scaling\_list\_pred\_mode\_flag ) |  |
| **scaling\_list\_pred\_matrix\_id\_delta** | ue(v) |
| else |  |
| scaling\_list( ScalingList[ SizeID ][ MatrixID ][ 0 ], ( 1 << ( 4 + ( sizeID << 1) ) ) ) |  |
| } |  |
| } |  |

#### Scaling list syntax

|  |  |
| --- | --- |
| scaling\_list( ScalingList, coefNum ) { | Descriptor |
| nextcoef = 8 | u(1) |
| for( i=0; i < coefNum, i++) { |  |
| **scaling\_list\_delta\_coef** | se(v) |
| nextcoef = ( nextcoef + scaling\_list\_delta\_coef + 256 ) % 256 |  |
| ScalingList[ i ] = nextcoef |  |
| } |  |
| } |  |

#### Adaptation parameter set RBSP syntax

[Ed. (BB): In HM the flags aps\_scaling\_list\_data\_present\_flag, aps\_deblocking\_filter\_flag, aps\_sample\_adaptive\_offset\_flag and aps\_adaptive\_loop\_filter\_flag are parsed together before the parameters are parsed. This was motivated by using CABAC for the parameter. Now that VLC is used, the flags should be parsed before the corresponding parameters.]

|  |  |
| --- | --- |
| aps\_rbsp( ) { | Descriptor |
| **aps\_id** | ue(v) |
| **aps\_scaling\_list\_data\_present\_flag** | u(1) |
| if( aps\_scaling\_list\_data\_present\_flag ) |  |
| scaling\_list\_param( ) |  |
| **aps\_deblocking\_filter\_flag** | u(1) |
| if (aps\_deblocking\_filter\_flag) { |  |
| **disable\_deblocking\_filter\_flag** | u(1) |
| if ( !disable\_deblocking\_filter\_flag ) { |  |
| **beta\_offset\_div2** | se(v) |
| **tc\_offset\_div2** | se(v) |
| } |  |
| **aps\_sample\_adaptive\_offset\_flag** | u(1) |
| if( aps\_sample\_adaptive\_offset\_flag ) |  |
| sao\_param( ) |  |
| **aps\_adaptive\_loop\_filter\_flag** | u(1) |
| if( aps\_adaptive\_loop\_filter\_flag ) |  |
| alf\_param( ) |  |
| **aps\_extension\_flag** | u(1) |
| if( aps\_extension\_flag ) |  |
| while( more\_rbsp\_data( ) ) |  |
| **aps\_extension\_data\_flag** | u(1) |
| rbsp\_trailing\_bits( ) |  |
| } |  |

#### Supplemental enhancement information RBSP syntax

|  |  |
| --- | --- |
| sei\_rbsp( ) { | Descriptor |
| do |  |
| sei\_message( ) |  |
| while( more\_rbsp\_data( ) ) |  |
| rbsp\_trailing\_bits( ) |  |
| } |  |

##### Supplemental enhancement information message syntax

|  |  |
| --- | --- |
| sei\_message( ) { | Descriptor |
| payloadType = 0 |  |
| while( next\_bits( 8 ) = = 0xFF ) { |  |
| **ff\_byte** /\* equal to 0xFF \*/ | f(8) |
| payloadType += 255 |  |
| } |  |
| **last\_payload\_type\_byte** | u(8) |
| payloadType += last\_payload\_type\_byte |  |
| payloadSize = 0 |  |
| while( next\_bits( 8 ) = = 0xFF ) { |  |
| **ff\_byte** /\* equal to 0xFF \*/ | f(8) |
| payloadSize += 255 |  |
| } |  |
| **last\_payload\_size\_byte** | u(8) |
| payloadSize += last\_payload\_size\_byte |  |
| sei\_payload( payloadType, payloadSize ) |  |
| } |  |

#### Access unit delimiter RBSP syntax

|  |  |
| --- | --- |
| access\_unit\_delimiter\_rbsp( ) { | Descriptor |
| **primary\_pic\_type** | u(3) |
| rbsp\_trailing\_bits( ) |  |
| } |  |

#### Filler data RBSP syntax

|  |  |
| --- | --- |
| filler\_data\_rbsp( ) { | Descriptor |
| while( next\_bits( 8 ) = = 0xFF ) |  |
| **ff\_byte** /\* equal to 0xFF \*/ | f(8) |
| rbsp\_trailing\_bits( ) |  |
| } |  |

#### Slice layer RBSP syntax

|  |  |
| --- | --- |
| slice\_layer\_rbsp( ) { | Descriptor |
| slice\_header( ) |  |
| slice\_data( ) |  |
| rbsp\_slice\_trailing\_bits( ) |  |
| } |  |

#### RBSP slice trailing bits syntax

|  |  |
| --- | --- |
| rbsp\_slice\_trailing\_bits( ) { | Descriptor |
| rbsp\_trailing\_bits( ) |  |
| while( more\_rbsp\_trailing\_data( ) ) |  |
| **cabac\_zero\_word** /\* equal to 0x0000 \*/ | f(16) |
| } |  |

#### RBSP trailing bits syntax

|  |  |
| --- | --- |
| rbsp\_trailing\_bits( ) { | Descriptor |
| **rbsp\_stop\_one\_bit** /\* equal to 1 \*/ | f(1) |
| while( !byte\_aligned( ) ) |  |
| **rbsp\_alignment\_zero\_bit** /\* equal to 0 \*/ | f(1) |
| } |  |

#### RBSP byte alignment syntax

|  |  |
| --- | --- |
| byte\_align( ) { | Descriptor |
| while( !byte\_aligned( ) ) |  |
| **bit\_equal\_to\_one** | f(1) |
| } |  |

### Slice header syntax

|  |  |
| --- | --- |
| slice\_header( ) { | Descriptor |
| **first\_slice\_in\_pic\_flag** | u(1) |
| if( first\_slice\_in\_pic\_flag = = 0 ) |  |
| **slice\_address** | u(v) |
| **slice\_type** | ue(v) |
| **entropy\_slice\_flag** | u(1) |
| if( !entropy\_slice\_flag ) { |  |
| **pic\_parameter\_set\_id** | ue(v) |
| if( IdrPicFlag ) { |  |
| **idr\_pic\_id** | ue(v) |
| **no\_output\_of\_prior\_pics\_flag** | u(1) |
| } |  |
| else { |  |
| **pic\_order\_cnt\_lsb** | u(v) |
| **short\_term\_ref\_pic\_set\_pps\_flag** | u(1) |
| if( !short\_term\_ref\_pic\_set\_pps\_flag ) |  |
| short\_term\_ref\_pic\_set( num\_short\_term\_ref\_pic\_sets ) |  |
| else |  |
| **short\_term\_ref\_pic\_set\_idx** | u(v) |
| if( long\_term\_ref\_pics\_present\_flag ) { |  |
| **num\_long\_term\_pics** | ue(v) |
| for( i = 0; i < num\_long\_term\_pics; i++ ) { |  |
| **delta\_poc\_lsb\_lt\_minus1**[ i ] | ue(v) |
| **used\_by\_curr\_pic\_lt\_flag**[ i ] | u(1) |
| } |  |
| } |  |
| } |  |
| if( scaling\_list\_enable\_flag | |   deblocking\_filter\_in\_APS\_enabled\_flag | |   sample\_adaptive\_offset\_enabled\_flag | |   adaptive\_loop\_filter\_enabled\_flag ) { |  |
| if( sample\_adaptive\_offset\_enabled\_flag ) |  |
| **slice\_sample\_adaptive\_offset\_flag** | u(1) |
| if( adaptive\_loop\_filter\_enabled\_flag) |  |
| **slice\_adaptive\_loop\_filter\_flag** | u(1) |
| **aps\_id** | ue(v) |
| } |  |
| if( slice\_type = = P | | slice\_type = = B ) { |  |
| **num\_ref\_idx\_active\_override\_flag** | u(1) |
| if( num\_ref\_idx\_active\_override\_flag ) { |  |
| **num\_ref\_idx\_l0\_active\_minus1** | ue(v) |
| if( slice\_type = = B ) |  |
| **num\_ref\_idx\_l1\_active\_minus1** | ue(v) |
| } |  |
| } |  |
| ref\_pic\_list\_modification( ) |  |
| ref\_pic\_list\_combination( ) |  |
| } |  |
| if( slice\_type != I ) |  |
| **cabac\_init\_idc** | ue(v) |
| if( !entropy\_slice\_flag ) { |  |
| **slice\_qp\_delta** | se(v) |
| **inherit\_dbl\_params\_from\_APS\_flag** | u(1) |
| if ( !inherit\_dbl\_params\_from\_APS\_flag ) { |  |
| **disable\_deblocking\_filter\_flag** | u(1) |
| if ( !disable\_deblocking\_filter\_flag ) { |  |
| beta\_offset\_div2 | se(v) |
| tc\_offset\_div2 | se(v) |
| } |  |
| } |  |
| if( slice\_type = = B ) |  |
| **collocated\_from\_l0\_flag** | u(1) |
| if( ( weighted\_pred\_flag && slice\_type = = P) | |  ( weighted\_bipred\_idc = = 1 && slice\_type = = B ) ) |  |
| pred\_weight\_table( ) |  |
| } |  |
| if( slice\_type = = P | | slice\_type = = B ) |  |
| **5\_minus\_max\_num\_merge\_cand** | ue(v) |
| if( adaptive\_loop\_filter\_enabled\_flag && aps\_adaptive\_loop\_filter\_flag ) |  |
| alf\_cu\_control\_param( ) |  |
| for( i = 0; i <num\_substreams\_minus1 + 1; i++ ){ |  |
| **substream\_length\_mode** | u(2) |
| **substream\_length[i]** | u(v) |
| } |  |
| } |  |

#### Short-term reference picture set syntax

|  |  |
| --- | --- |
| short\_term\_ref\_pic\_set( idx ) { | Descriptor |
| **inter\_ref\_pic\_set\_prediction\_flag** | u(1) |
| if( inter\_ref\_pic\_set\_prediction\_flag) { |  |
| **delta\_idx\_minus1** | ue(v) |
| **delta\_rps\_sign** | u(1) |
| **abs\_delta\_rps\_minus1** | ue(v) |
| for( j = 0; j <= NumDeltaPocs[ RIdx ]; j++ ) { |  |
| **ref\_idc0**[ j ] | u(1) |
| if( !**ref\_idc0**[ j ]) |  |
| **ref\_idc1**[ j ] | u(1) |
| } |  |
| } |  |
| else { |  |
| **num\_negative\_pics** | ue(v) |
| **num\_positive\_pics** | ue(v) |
| for( i = 0; i < num\_negative\_pics; i++ ) { |  |
| **delta\_poc\_s0\_minus1**[ i ] | ue(v) |
| **used\_by\_curr\_pic\_s0\_flag**[ i ] | u(1) |
| } |  |
| for( i = 0; i < num\_positive\_pics; i++ ) { |  |
| **delta\_poc\_s1\_minus1**[ i ] | ue(v) |
| **used\_by\_curr\_pic\_s1\_flag**[ i ] | u(1) |
| } |  |
| } |  |
| } |  |

#### Reference picture list modification syntax

|  |  |
| --- | --- |
| ref\_pic\_list\_modification( ) { | Descriptor |
| if( slice\_type = = P | | slice\_type = = B ) { |  |
| **ref\_pic\_list\_modification\_flag\_l0** | u(1) |
| if( ref\_pic\_list\_modification\_flag\_l0 ) |  |
| do { |  |
| **ref\_pic\_list\_modification\_idc** | ue(v) |
| if( ref\_pic\_list\_modification\_idc != 3 ) |  |
| **ref\_pic\_set\_idx** | ue(v) |
| } while( ref\_pic\_list\_modification\_idc != 3 ) |  |
| } |  |
| if( slice\_type = = B ) { |  |
| **ref\_pic\_list\_modification\_flag\_l1** | u(1) |
| if( ref\_pic\_list\_modification\_flag\_l1 ) |  |
| do { |  |
| **ref\_pic\_list\_modification\_idc** | ue(v) |
| if( ref\_pic\_list\_modification\_idc != 3 ) |  |
| **ref\_pic\_set\_idx** | ue(v) |
| } while( ref\_pic\_list\_modification\_idc != 3 ) |  |
| } |  |
| } |  |

#### Reference picture lists combination syntax

|  |  |
| --- | --- |
| ref\_pic\_list\_combination( ) { | Descriptor |
| if( slice\_type = = B ) { |  |
| **ref\_pic\_list\_combination\_flag** | u(1) |
| if( ref\_pic\_list\_combination\_flag ) { |  |
| **num\_ref\_idx lc\_active\_minus1** | ue(v) |
| **ref\_pic\_list\_modification\_flag\_lc** | u(1) |
| if( ref\_pic\_list\_modification\_flag\_lc) |  |
| for ( i =0; i <= num\_ref\_idx\_lc\_active\_minus1; i++ ) { |  |
| **pic\_from\_list\_0\_flag** | u(1) |
| **ref\_idx\_list\_curr** | ue(v) |
| } |  |
| } |  |
| } |  |
| } |  |

#### Sample adaptive offset parameter syntax

|  |  |
| --- | --- |
| sao\_param( ) { | Descriptor |
| sao\_split\_param( 0, 0, 0, 0 ) |  |
| sao\_offset\_param( 0, 0, 0, 0 ) |  |
| **sao\_flag\_cb** | u(1) |
| if( sao\_flag\_cb ) { |  |
| sao\_split\_param( 0, 0, 0, 1 ) |  |
| sao\_split\_param( 0, 0, 0, 1 ) |  |
| } |  |
| **sao\_flag\_cr** | u(1) |
| if( sao\_flag\_cr ) { |  |
| sao\_split\_param( 0, 0, 0, 2 ) |  |
| sao\_split\_param( 0, 0, 0, 2 ) |  |
| } |  |
| } |  |

|  |  |
| --- | --- |
| sao\_split\_param( rx, ry, saoDepth , cIdx ) { | Descriptor |
| if( saoDepth < SaoMaxDepth ) |  |
| **sao\_split\_flag**[ cIdx ][ saoDepth ][ rx ][ ry ] | u(1) |
| else |  |
| sao\_split\_flag[ cIdx ][ saoDepth ][ rx ][ ry ] = 0 |  |
| if( sao\_split\_flag[ cIdx ][ saoDepth ][ rx ][ ry ] ) { |  |
| sao\_split\_param( 2\*rx + 0, 2\*ry + 0, saoDepth + 1 , cIdx ) |  |
| sao\_split\_param( 2\*rx + 1, 2\*ry + 0, saoDepth + 1 , cIdx ) |  |
| sao\_split\_param( 2\*rx + 0, 2\*ry + 1, saoDepth + 1 , cIdx ) |  |
| sao\_split\_param( 2\*rx + 1, 2\*ry + 1, saoDepth + 1 , cIdx ) |  |
| } |  |
| } |  |

|  |  |
| --- | --- |
| sao\_offset\_param( rx, ry, saoDepth , cIdx ) { | Descriptor |
| if( sao\_split\_flag[ cIdx ][ saoDepth ][ rx ][ ry ] ) { |  |
| sao\_offset\_param( 2\*rx + 0, 2\*ry + 0, saoDepth + 1 , cIdx ) |  |
| sao\_offset\_param( 2\*rx + 1, 2\*ry + 0, saoDepth + 1 , cIdx ) |  |
| sao\_offset\_param( 2\*rx + 0, 2\*ry + 1, saoDepth + 1 , cIdx ) |  |
| sao\_offset\_param( 2\*rx + 1, 2\*ry + 1, saoDepth + 1 , cIdx ) |  |
| } else { |  |
| **sao\_type\_idx**[ cIdx ][ saoDepth ][ rx ][ ry ] | ue(v) |
| if( sao\_type\_idx[ cIdx ][ saoDepth ][ rx ][ ry ] != 0 ) |  |
| for( i = 0; i < NumSaoClass[ sao\_type\_idx ]; i++ ) |  |
| **sao\_offset**[ cIdx ][ saoDepth ][ x0 ][ y0 ][ i ] | se(v) |
| } |  |
| } |  |

[Ed.: (WJ) is it better to assign separate section?]

#### Adaptive loop filter parameter syntax

|  |  |
| --- | --- |
| alf\_param() { | Descriptor |
| **alf\_region\_adaptation\_flag** | u(1) |
| **alf\_length\_luma\_minus\_5\_div2** | ue(v) |
| **alf\_no\_filters\_minus1** | ue(v) |
| if (alf\_no\_filters\_minus1 == 1) |  |
| **alf\_start\_second\_filter** | ue(v) |
| else if (alf\_no\_filters\_minus1 > 1) { |  |
| for (i=1; i< (alf\_region\_adaptation\_flag ? 16 : 15); i++) |  |
| **alf\_filter\_pattern[i]** | u(1) |
| } |  |
| if (AlfNumFilters > 1) |  |
| **alf\_pred\_method** | u(1) |
| for (i=0; i< AlfNumFilters; i++ ) |  |
| **alf\_nb\_pred\_luma[i]** | u(1) |
| for (i=0; i< AlfNumFilters; i++) |  |
| for (j=0; j< AlfCodedLengthLuma; j++) |  |
| **alf\_coeff\_luma**[i][j] | ge(v) |
| **alf\_chroma\_idc** | ue(v) |
| if ( alf\_chroma\_idc ) { |  |
| **alf\_length\_chroma\_minus\_5\_div2** | ue(v) |
| for( i = 0; i< AlfCodedLengthChroma; i++ ) |  |
| **alf\_coeff\_chroma**[i] | se(v) |
| } |  |
| } |  |

#### Adaptive loop filter coding unit control parameter syntax

|  |  |
| --- | --- |
| alf\_cu\_control\_param() { | Descriptor |
| **alf\_cu\_control\_flag** | u(1) |
| if( alf\_cu\_control\_flag ) { |  |
| **alf\_cu\_control\_max\_depth** | ue(v) |
| **alf\_length\_cu\_control\_info** | se(v) |
| for( i = 0; i < NumAlfCuFlag; i++ ) |  |
| **alf\_cu\_flag**[ i ] | u(1) |
| } |  |
| } |  |

#### Prediction weight table syntax

|  |  |
| --- | --- |
| pred\_weight\_table( ) { | Descriptor |
| **luma\_log2\_weight\_denom** | ue(v) |
| if( chroma\_format\_idc != 0 ) |  |
| **delta\_chroma\_log2\_weight\_denom** | se(v) |
| if ( slice\_type = = P | |  ( slice\_type = = B && ref\_pic\_list\_combination\_flag = = 0 ) ) { |  |
| for( i = 0; i <= num\_ref\_idx\_l0\_active\_minus1; i++ ) { |  |
| **luma\_weight\_l0\_flag** | u(1) |
| if( luma\_weight\_l0\_flag ) { |  |
| **delta\_luma\_weight\_l0[** i **]** | se(v) |
| **luma\_offset\_l0[** i **]** | se(v) |
| } |  |
| if ( chroma\_format\_idc != 0 ) { |  |
| **chroma\_weight\_l0\_flag** | u(1) |
| if( chroma\_weight\_l0\_flag ) |  |
| for( j =0; j < 2; j++ ) { |  |
| **delta\_chroma\_weight\_l0[** i **][** j **]** | se(v) |
| **delta\_chroma\_offset\_l0[** i **][** j **]** | se(v) |
| } |  |
| } |  |
| } |  |
| } |  |
| if( slice\_type = = B ) { |  |
| if ( ref\_pic\_list\_combination\_flag = = 0 ) { |  |
| for( i = 0; i <= num\_ref\_idx\_l1\_active\_minus1; i++ ) { |  |
| **luma\_weight\_l1\_flag** | u(1) |
| if( luma\_weight\_l1\_flag ) { |  |
| **delta\_luma\_weight\_l1[** i **]** | se(v) |
| **luma\_offset\_l1[** i **]** | se(v) |
| } |  |
| if( chroma\_format\_idc != 0 ) { |  |
| **chroma\_weight\_l1\_flag** | u(1) |
| if( chroma\_weight\_l1\_flag ) |  |
| for( j = 0; j < 2; j++ ) { |  |
| **delta\_chroma\_weight\_l1[** i **][** j **]** | se(v) |
| **delta\_chroma\_offset\_l1[** i **][** j **]** | se(v) |
| } |  |
| **}** |  |
| } |  |
| } else { |  |
| for( i = 0; i <= num\_ref\_idx\_lc\_active\_minus1; i++ ) { |  |
| **luma\_weight\_lc\_flag** | u(1) |
| if( luma\_weight\_l1\_flag ) { |  |
| **delta\_luma\_weight\_lc[** i **]** | se(v) |
| **luma\_offset\_lc[** i **]** | se(v) |
| } |  |
| if( chroma\_format\_idc != 0 ) { |  |
| **chroma\_weight\_lc\_flag** | u(1) |
| if( chroma\_weight\_lc\_flag ) |  |
| for( j = 0; j < 2; j++ ) { |  |
| **delta\_chroma\_weight\_lc[** i **][** j **]** | se(v) |
| **delta\_chroma\_offset\_lc[** i **][** j **]** | se(v) |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  |

### Slice data syntax

|  |  |
| --- | --- |
| slice\_data( ) { | Descriptor |
| CurrTbAddr = LCUAddress |  |
| moreDataFlag = 1 |  |
| if( adaptive\_loop\_filter\_flag && alf\_cu\_control\_flag ) |  |
| AlfCuFlagIdx = -1 |  |
| do { |  |
| XLCU = HorLumaLocation( CurrTbAddr ) |  |
| YLCU = VerLumaLocation( CurrTbAddr ) |  |
| moreDataFlag = coding\_tree( XLCU, YLCU, Log2TbSize, 0 ) |  |
| CurrTbAddr = NextTbAddress( CurrTbAddr ) |  |
| if( CurrTbAddr = = firstTbInTileAddr ) |  |
| rbsp\_trailingbits() |  |
| } while( moreDataFlag ) |  |
| } |  |

### Coding tree syntax

|  |  |
| --- | --- |
| coding\_tree( x0, y0, log2CUSize, cuDepth ) { | Descriptor |
| if( x0 + ( 1 << log2CUSize ) <= PicWidthInSamplesL &&  y0 + ( 1 << log2CUSize ) <= PicHeightInSamplesL &&  cuAddress( x0, y0 ) >= SliceAddress &&  log2CUSize > Log2MinCUSize ) { |  |
| **split\_coding\_unit\_flag[** x0 **][** y0 **]** | ae(v) |
| } |  |
| if( adaptive\_loop\_filter\_flag && alf\_cu\_control\_flag ) { |  |
| if( cuDepth <= alf\_cu\_control\_max\_depth ) |  |
| if( cuDepth == alf\_cu\_control\_max\_depth || |  |
| split\_coding\_unit\_flag[ x0 ][ y0 ] == 0 ) |  |
| AlfCuFlagIdx++ |  |
| } |  |
| if( cu\_qp\_delta\_enabled\_flag &&   log2CUSize >= log2MinCUDQPSize ) |  |
| IsCuQpDeltaCoded = 0 |  |
| if( split\_coding\_unit\_flag[ x0 ][ y0 ] ) { |  |
| x1 = x0 + ( ( 1 << log2CUSize ) >> 1 ) |  |
| y1 = y0 + ( ( 1 << log2CUSize ) >> 1 ) |  |
| if( cuAddress( x1, y0 ) > SliceAddress ) |  |
| moreDataFlag = coding\_tree( x0, y0, log2CUSize – 1, cuDepth + 1 ) |  |
| if( cuAddress( x0, y1 ) > SliceAddress && moreDataFlag &&  x1 < PicWidthInSamplesL ) |  |
| moreDataFlag = coding\_tree( x1, y0, log2CUSize − 1, cuDepth + 1 ) |  |
| if( cuAddress( x1, y1 ) > SliceAddress && moreDataFlag &&  y1 < PicHeightInSamplesL ) |  |
| moreDataFlag = coding\_tree( x0, y1, log2CUSize − 1, cuDepth + 1 ) |  |
| if( moreDataFlag &&   x1 < PicWidthInSamplesL && y1 < PicHeightInSamplesL ) |  |
| moreDataFlag = coding\_tree( x1, y1, log2CUSize − 1, cuDepth + 1 ) |  |
| } else { |  |
| if(adaptive\_loop\_filter\_flag && alf\_cu\_control\_flag ) |  |
| AlfCuFlag[ x0 ][ y0 ] = alf\_cu\_flag[ AlfCuFlagIdx ] |  |
| coding\_unit( x0, y0, log2CUSize ) |  |
| if( granularity\_block\_boundary( x0, y0, log2CUSize ) ) { |  |
| **end\_of\_slice\_flag** | ae(v) |
| moreDataFlag = !end\_of\_slice\_flag |  |
| } else |  |
| moreDataFlag = 1 |  |
| } |  |
| return moreDataFlag |  |
| } |  |

[Ed. (WJ): cuAddress( x, y ) returns the address of the smallest coding unit at (x, y), expressed in global decoding order smallest coding unit coordinates. granularity\_block\_boundary( x0, y0, log2CUSize ) returns true when the coding unit specified by ( x0, y0, log2CUSize ) is the last coding unit in the slice in global decoding order. These functions are not defined yet. They will be added later.]

[Ed.(MH): Tiles affect global decoding order and needs to be considered when defining cuAddres(x,y).]

### Coding unit syntax

|  |  |
| --- | --- |
| coding\_unit( x0, y0, log2CUSize ) { | Descriptor |
| if( slice\_type != I ) |  |
| **skip\_flag[** x0 **][** y0 **]** | ae(v) |
| if( skip\_flag[ x0 ][ y0 ] ) |  |
| prediction\_unit( x0, y0 , log2CUSize ) |  |
| else if( slice\_type != I | | log2CUSize = = Log2MinCUSize ) { |  |
| if( slice\_type != I ) |  |
| **pred\_mode\_flag** | ae(v) |
| if( PredMode != MODE\_INTRA | | log2CUSize = = Log2MinCUSize ) |  |
| **part\_mode** | ae(v) |
| x1 = x0 + ( ( 1 << log2CUSize ) >> 1 ) |  |
| y1 = y0 + ( ( 1 << log2CUSize ) >> 1 ) |  |
| x2 = x1 − ( ( 1 << log2CUSize ) >> 2 ) |  |
| y2 = y1 − ( ( 1 << log2CUSize ) >> 2 ) |  |
| x3 = x1 + ( ( 1 << log2CUSize ) >> 2 ) |  |
| y3 = y1 + ( ( 1 << log2CUSize ) >> 2 ) |  |
| if( PartMode == PART\_2Nx2N ) { |  |
| prediction\_unit( x0, y0 , log2CUSize ) |  |
| } else if( PartMode == PART\_2NxN ) { |  |
| prediction\_unit( x0, y0 , log2CUSize ) |  |
| prediction\_unit( x0, y1 , log2CUSize ) |  |
| } else if( PartMode == PART\_Nx2N ) { |  |
| prediction\_unit( x0, y0 , log2CUSize ) |  |
| prediction\_unit( x1, y0 , log2CUSize ) |  |
| } else if( PartMode == PART\_2NxnU ) { |  |
| prediction\_unit( x0, y0 , log2CUSize ) |  |
| prediction\_unit( x0, y2 , log2CUSize ) |  |
| } else if( PartMode == PART\_2NxnD ) { |  |
| prediction\_unit( x0, y0 , log2CUSize ) |  |
| prediction\_unit( x0, y3 , log2CUSize ) |  |
| } else if( PartMode == PART\_nLx2N ) { |  |
| prediction\_unit( x0, y0 , log2CUSize ) |  |
| prediction\_unit( x2, y0 , log2CUSize ) |  |
| } else if( PartMode == PART\_nRx2N ) { |  |
| prediction\_unit( x0, y0 , log2CUSize ) |  |
| prediction\_unit( x3, y0 , log2CUSize ) |  |
| } else { /\* PART\_NxN \*/ |  |
| prediction\_unit( x0, y0 , log2CUSize ) |  |
| prediction\_unit( x1, y0 , log2CUSize ) |  |
| prediction\_unit( x0, y1 , log2CUSize ) |  |
| prediction\_unit( x1, y1 , log2CUSize ) |  |
| } |  |
| if( !pcm\_flag ) { |  |
| transform\_tree( x0, y0, log2CUSize, log2CUSize, log2CUSize, 0, 0 ) |  |
| transform\_coeff( x0, y0, x0, y0, log2CUSize, log2CUSize, 0, 0 ) |  |
| } |  |
| } |  |
| } |  |

### Prediction unit syntax

|  |  |
| --- | --- |
| prediction\_unit( x0, y0, log2CUSize ) { | Descriptor |
| if( skip\_flag[ x0 ][ y0 ] ) { |  |
| if( MaxNumMergeCand > 1 ) |  |
| **merge\_idx[** x0 **][** y0 **]** | ae(v) |
| } else if( PredMode = = MODE\_INTRA ) { |  |
| if( PartMode == PART\_2Nx2N && pcm\_enabled\_flag &&  log2CUSize >= Log2MinIPCMCUSize &&  log2CUSize <= Log2MaxIPCMCUSize ) |  |
| **pcm\_flag** | ae(v) |
| if( pcm\_flag ) { |  |
| while ( !byte\_aligned( ) ) |  |
| **pcm\_alignment\_zero\_bit** | u(v) |
| for( i = 0; i < 1 << ( log2CUSize << 1 ); i++ ) |  |
| **pcm\_sample\_luma**[ i ] | u(v) |
| for( i = 0; i < ( 1 << ( log2CUSize << 1 ) ) >> 1; i++ ) |  |
| **pcm\_sample\_chroma**[ i ] | u(v) |
| } else { |  |
| **prev\_intra\_luma\_pred\_flag[** x0 **][** y0 **]** | ae(v) |
| if( prev\_intra\_luma\_pred\_flag[ x0 ][ y0 ] ) |  |
| **mpm\_flag[** x0 **][** y0 **]** | ae(v) |
| else |  |
| **rem\_intra\_luma\_pred\_mode[** x0 **][**y0 **]** | ae(v) |
| **intra\_chroma\_pred\_mode**[ x0 ][ y0 ] | ae(v) |
| SignaledAsChromaDC =   ( chroma\_pred\_from\_luma\_enabled\_flag ?  intra\_chroma\_pred\_mode[ x0 ][ y0 ] == 3 :  intra\_chroma\_pred\_mode[ x0 ][ y0 ] == 2 ) |  |
| } |  |
| } else { /\* MODE\_INTER \*/ |  |
| **merge\_flag[** x0 **][** y0 **]** | ae(v) |
| if( merge\_flag[ x0 ][ y0 ] ) { |  |
| if( MaxNumMergeCand > 1 ) |  |
| **merge\_idx[** x0 **][** y0 **]** | ae(v) |
| } else { |  |
| if( slice\_type = = B ) |  |
| **inter\_pred\_flag[** x0 **][** y0 **]** | ae(v) |
| if( inter\_pred\_flag[ x0 ][ y0 ] = = Pred\_LC ) { |  |
| if( num\_ref\_idx\_lc\_active\_minus1 > 0 ) |  |
| **ref\_idx\_lc[** x0 **][** y0 **]** | ae(v) |
| mvd\_coding(mvd\_lc[ x0 ][ y0 ][ 0 ],   mvd\_lc[ x0 ][ y0 ][ 1 ]) |  |
| **mvp\_lc\_flag[ x0 ][ y0 ]** | ae(v) |
| } |  |
| else { /\* Pred\_L0 or Pred\_BI \*/ |  |
| if( num\_ref\_idx\_l0\_active\_minus1 > 0 ) |  |
| **ref\_idx\_l0**[ x0 ][ y0 ] | ae(v) |
| mvd\_coding(mvd\_l0[ x0 ][ y0 ][ 0 ],   mvd\_l0[ x0 ][ y0 ][ 1 ]) |  |
| **mvp\_l0\_flag[ x0 ][ y0 ]** | ae(v) |
| } |  |
| if( inter\_pred\_flag[ x0 ][ y0 ] = = Pred\_BI ) { |  |
| if( num\_ref\_idx\_l1\_active\_minus1 > 0 ) |  |
| **ref\_idx\_l1[** x0 **][** y0 **]** | ae(v) |
| mvd\_coding(mvd\_l1[ x0 ][ y0 ][ 0 ],   mvd\_l1[ x0 ][ y0 ][ 1 ]) |  |
| **mvp\_l1\_flag[ x0 ][ y0 ]** | ae(v) |
| } |  |
| } |  |
| } |  |
| } |  |

#### Motion vector difference coding syntax

|  |  |
| --- | --- |
| mvd\_coding( mvd\_x, mvd\_y ) { | **Descriptor** |
| **abs\_mvd\_greater0\_flag[** 0 **]** | ae(v) |
| **abs\_mvd\_greater0\_flag[** 1 **]** | ae(v) |
| if(abs\_mvd\_greater0\_flag[ 0 ] ) |  |
| **abs\_mvd\_greater1\_flag[** 0 **]** | ae(v) |
| if(abs\_mvd\_greater0\_flag[ 1 ] ) |  |
| **abs\_mvd\_greater1\_flag[** 1 **]** | ae(v) |
| if(abs\_mvd\_greater0\_flag[ 0 ] ) { |  |
| if(abs\_mvd\_greater1\_flag[ 0 ] ) |  |
| **abs\_mvd\_minus2[** 0 **]** | ae(v) |
| **mvd\_sign\_flag[** 0 **]** | ae(v) |
| } |  |
| if(abs\_mvd\_greater0\_flag[ 1 ] ) { |  |
| if(abs\_mvd\_greater1\_flag[ 1 ] ) |  |
| **abs\_mvd\_minus2[** 1 **]** | ae(v) |
| **mvd\_sign\_flag[** 1 **]** | ae(v) |
| } |  |
| mvd\_x = abs\_mvd\_greater0\_flag[ 0 ] \* ( abs\_mvd\_minus2[ 0 ] + 2 ) \*   ( 1 – 2 \* mvd\_sign\_flag[ 0 ] ) |  |
| mvd\_y = abs\_mvd\_greater0\_flag[ 1 ] \* ( abs\_mvd\_minus2[ 1 ] + 2 ) \*   ( 1 – 2 \* mvd\_sign\_flag[ 1 ] ) |  |
| } |  |

### Transform tree syntax

|  |  |
| --- | --- |
| transform\_tree( x0, y0, log2CUSize, log2TrafoWidth, log2TrafoHeight, trafoDepth, blkIdx ) { | Descriptor |
| if( trafoDepth = = 0 && IntraSplitFlag = = 0 && PredMode != MODE\_INTRA &&   !(PartMode = = PART\_2Nx2N && merge\_flag[x0][y0]) ) |  |
| **no\_residual\_data\_flag** | ae(v) |
| if( !no\_residual\_data\_flag ) { |  |
| log2TrafoSize = ( log2TrafoWidth + log2TrafoHeight ) >> 1 |  |
| intraSplitFlag = ( IntraSplitFlag && trafoDepth = = 0 ? 1 : 0 ) |  |
| interSplitFlag = ( max\_transform\_hierarchy\_depth\_inter = =0 &&  PredMode = = MODE\_INTER && PartMode != PART\_2Nx2N &&  trafoDepth = = 0 ) |  |
| maxDepth = ( PredMode = = MODE\_INTRA ?   max\_transform\_hierarchy\_depth\_intra + IntraSplitFlag :   max\_transform\_hierarchy\_depth\_inter + InterSplitFlag ) |  |
| xBase = x0 − ( x0 & ( 1 << log2TrafoWidth ) ) |  |
| yBase = y0 − ( y0 & ( 1 << log2TrafoHeight ) ) |  |
| if( log2TrafoSize <= Log2MaxTrafoSize &&   log2TrafoSize > Log2MinTrafoSize &&  trafoDepth < maxDepth && !intraSplitFlag && !interSplitFlag ) |  |
| **split\_transform\_flag**[ x0 ][ y0 ][ trafoDepth ] | ae(v) |
| if( PredMode != MODE\_INTRA &&  log2TrafoSize <= Log2MaxTrafoSize ) { |  |
| firstChromaCbf = ( log2TrafoSize = = Log2MaxTrafoSize | |  trafoDepth = = 0 ) ? 1 : 0 |  |
| if( firstChromaCbf | | log2TrafoSize > Log2MinTrafoSize ) {  [Ed. (WJ): Log2MinTrafoSize or 2?] |  |
| if( firstChromaCbf | | cbf\_cb[ xBase ][ yBase ][ trafoDepth − 1 ] ) { |  |
| readCbf = true |  |
| if( blkIdx = = 3 && log2TrafoSize < Log2MaxTrafoSize ) |  |
| readCbf = cbf\_cb[ xBase ][ yBase ][ trafoDepth ] | |   cbf\_cb[ xBase + ( 1 << log2TrafoWidth ) ][ yBase ][ trafoDepth ] | |   cbf\_cb[ xBase ][ yBase + ( 1 << log2TrafoHeight ) ][ trafoDepth ] |  |
| if ( !readCbf ) |  |
| cbf\_cb[ x0 ][ y0 ][ trafoDepth ] = 1 |  |
| else |  |
| **cbf\_cb**[ x0 ][ y0 ][ trafoDepth ] | ae(v) |
| } |  |
| if( firstChromaCbf | | cbf\_cr[ xBase ][ yBase ][ trafoDepth − 1 ] ) { |  |
| readCbf = true |  |
| if( blkIdx = = 3 && log2TrafoSize < Log2MaxTrafoSize ) |  |
| readCbf = cbf\_cr[ xBase ][ yBase ][ trafoDepth ] | |   cbf\_cr[ xBase + ( 1 << log2TrafoWidth ) ][ yBase ][ trafoDepth ] | |   cbf\_cr[ xBase ][ yBase + ( 1 << log2TrafoHeight ) ][ trafoDepth ] |  |
| if ( !readCbf ) |  |
| cbf\_cr[ x0 ][ y0 ][ trafoDepth ] = 1 |  |
| else |  |
| **cbf\_cr**[ x0 ][ y0 ][ trafoDepth ] | ae(v) |
| } |  |
| } |  |
| } |  |
| if( split\_transform\_flag[ x0 ][ y0 ][ trafoDepth ] ) { |  |
| if( InterTUSplitDirection = = 2 ) { |  |
| x1 = x0 + ( ( 1 << log2TrafoWidth ) >> 1 ) |  |
| y1 = y0 |  |
| x2 = x0 |  |
| y2 = y0 + ( ( 1 << log2TrafoHeight ) >> 1 ) |  |
| x3 = x1 |  |
| y3 = y2 |  |
| } else { |  |
| x1 = x0 + ( ( 1 << log2TrafoWidth ) >> 2 ) \* InterTUSplitDirection |  |
| y1 = y0 + ( ( 1 << log2TrafoHeight) >> 2 ) \* ( 1 − InterTUSplitDirection ) |  |
| x2 = x1 + ( ( 1 << log2TrafoWidth ) >> 2 ) \* InterTUSplitDirection |  |
| y2 = y1 + ( ( 1 << log2TrafoHeight) >> 2 ) \* ( 1 − InterTUSplitDirection ) |  |
| x3 = x2 + ( ( 1 << log2TrafoWidth ) >> 2 ) \* InterTUSplitDirection |  |
| y3 = y2 + ( ( 1 << log2TrafoHeight) >> 2 ) \* ( 1 − InterTUSplitDirection ) |  |
| log2TrafoHeight = log2TrafoHeight + 2 \* InterTUSplitDirection − 1 |  |
| log2TrafoWidth = log2TrafoWidth − 2 \* InterTUSplitDirection + 1 |  |
| } |  |
| transform\_tree( x0, y0, log2CUSize, log2TrafoWidth − 1, log2TrafoHeight − 1,   trafoDepth + 1, 0 ) |  |
| transform\_tree( x1, y1, log2CUSize, log2TrafoWidth − 1, log2TrafoHeight − 1,   trafoDepth + 1, 1 ) |  |
| transform\_tree( x2, y2, log2CUSize, log2TrafoWidth − 1, log2TrafoHeight − 1,   trafoDepth + 1, 2 ) |  |
| transform\_tree( x3, y3, log2CUSize, log2TrafoWidth − 1, log2TrafoHeight − 1,   trafoDepth + 1, 3 ) |  |
| } else { |  |
| if( PredMode = = MODE\_INTRA | | trafoDepth != 0 | |  cbf\_cb[ x0 ][ y0 ][ trafoDepth ] | |  cbf\_cr[ x0 ][ y0 ][ trafoDepth ] ) { |  |
| readCbf = true |  |
| if( blkIdx = = 3 && PredMode != MODE\_INTRA &&   ( ( log2CUSize <= Log2MaxTrafoSize+1 ) || ( log2TrafoSize < Log2MaxTrafoSize ) ) |  |
| readCbf = cbf\_luma[ xBase ][ yBase ][ trafoDepth ] | |   cbf\_luma[ xBase + ( 1 << log2TrafoWidth ) ][ yBase ][ trafoDepth ] | |   cbf\_luma[ xBase ][ yBase + ( 1 << log2TrafoHeight ) ][ trafoDepth ] | |   cbf\_cb[ xBase ][ yBase ][ trafoDepth – 1] | |   cbf\_cr[ xBase ][ yBase ][ trafoDepth – 1] |  |
| if ( !readCbf ) |  |
| cbf\_luma[ x0 ][ y0 ][ trafoDepth ] = 1 |  |
| else |  |
| **cbf\_luma**[ x0 ][ y0 ][ trafoDepth ] | ae(v) |
| } |  |
| if( PredMode = = MODE\_INTRA ) |  |
| if( log2TrafoSize > Log2MinTrafoSize ) { [Ed. (WJ): Log2MinTrafoSize or 2?] |  |
| **cbf\_cb**[ x0 ][ y0 ][ trafoDepth ] | ae(v) |
| **cbf\_cr**[ x0 ][ y0 ][ trafoDepth ] | ae(v) |
| } else if( blkIdx = = 0 ) { |  |
| **cbf\_cb**[ x0 ][ y0 ][ trafoDepth − 1 ] | ae(v) |
| **cbf\_cr**[ x0 ][ y0 ][ trafoDepth − 1 ] | ae(v) |
| } |  |
| } |  |
| } |  |
| } |  |

### Transform coefficient syntax

|  |  |
| --- | --- |
| transform\_coeff( x0, y0, xC, yC, log2TrafoWidth, log2TrafoHeight, trafoDepth, blkIdx ) { | Descriptor |
| if( cbf\_luma[ x0 ][ y0 ][ trafoDepth ] | | cbf\_cb[ x0 ][ y0 ][ trafoDepth ] | |  cbf\_cr[ x0 ][ y0 ][ trafoDepth ] { |  |
| if( cu\_qp\_delta\_enabled\_flag && !IsCuQpDeltaCoded ) { |  |
| **cu\_qp\_delta** | ae(v) |
| IsCuQpDeltaCoded = 1 |  |
| } |  |
| if( split\_transform\_flag[ x0 ][ y0 ][ trafoDepth ] ) { |  |
| if( InterTUSplitDirection = = 2 ) { |  |
| x1 = x0 + ( ( 1 << log2TrafoWidth ) >> 1 ) |  |
| y1 = y0 |  |
| x2 = x0 |  |
| y2 = y0 + ( ( 1 << log2TrafoHeight ) >> 1 ) |  |
| x3 = x1 |  |
| y3 = y2 |  |
| } else { |  |
| x1 = x0 + ( ( 1 << log2TrafoWidth ) >> 2 ) \* InterTUSplitDirection |  |
| y1 = y0 + ( ( 1 << log2TrafoHeight) >> 2 ) \* ( 1 − InterTUSplitDirection ) |  |
| x2 = x1 + ( ( 1 << log2TrafoWidth ) >> 2 ) \* InterTUSplitDirection |  |
| y2 = y1 + ( ( 1 << log2TrafoHeight) >> 2 ) \* ( 1 − InterTUSplitDirection ) |  |
| x3 = x2 + ( ( 1 << log2TrafoWidth ) >> 2 ) \* InterTUSplitDirection |  |
| y3 = y2 + ( ( 1 << log2TrafoHeight) >> 2 ) \* ( 1 − InterTUSplitDirection ) |  |
| log2TrafoHeight = log2TrafoHeight + 2 \* InterTUSplitDirection − 1 |  |
| log2TrafoWidth = log2TrafoWidth − 2 \* InterTUSplitDirection + 1 |  |
| } |  |
| transform\_coeff( x0, y0, x0, y0, log2TrafoWidth − 1, log2TrafoHeight − 1, trafoDepth + 1, 0 ) |  |
| transform\_coeff( x1, y1, x0, y0, log2TrafoWidth − 1, log2TrafoHeight − 1, trafoDepth + 1, 1 ) |  |
| transform\_coeff( x2, y2, x0, y0, log2TrafoWidth − 1, log2TrafoHeight − 1, trafoDepth + 1, 2 ) |  |
| transform\_coeff( x3, y3, x0, y0, log2TrafoWidth − 1, log2TrafoHeight − 1, trafoDepth + 1, 3 ) |  |
| } else { |  |
| log2TrafoSize = ( ( log2TrafoWidth + log2TrafoHeight ) >> 1 ) |  |
| log2TrafoSizeC = ( ( log2TrafoSize = = Log2MinTrafoSizeC ) ?  log2TrafoSize : log2TrafoSize – 1 ) |  |
| if ( PredMode = = MODE\_INTRA ) { |  |
| scanIdx = ScanType[ log2TrafoSize – 2 ][ IntraPredMode ] |  |
| scanIdxC = ScanType[ log2TrafoSize – 2 ][ IntraPredModeC ] |  |
| } else { |  |
| scanIdx = 0 |  |
| scanIdxC = 0 |  |
| } |  |
| if ( cbf\_luma[ x0 ][ y0 ][ trafoDepth ] ) |  |
| residual\_coding( x0, y0, log2TrafoWidth, log2TrafoHeight, scanIdx, 0 ) |  |
| if ( log2TrafoSize > Log2MinTrafoSize ) { [Ed. (WJ): Log2MinTrafoSize or 2?] |  |
| if ( cbf\_cb[ x0 ][ y0 ][ trafoDepth ] ) |  |
| residual\_coding( x0, y0, log2TrafoSizeC, trafoDepth, scanIdxC, 1 ) |  |
| if ( cbf\_cr[ x0 ][ y0 ][ trafoDepth ] ) |  |
| residual\_coding( x0, y0, log2TrafoSizeC, trafoDepth, scanIdxC, 2 ) |  |
| } else if ( blkIdx == 3 ) { |  |
| if ( cbf\_cb[ x0 ][ y0 ][ trafoDepth ] ) |  |
| residual\_coding( xC, yC, log2TrafoSizeC, trafoDepth, scanIdxC, 1 ) |  |
| if ( cbf\_cr[ x0 ][ y0 ][ trafoDepth ] ) |  |
| residual\_coding( xC, yC, log2TrafoSizeC, trafoDepth, scanIdxC, 2 ) |  |
| } |  |
| } |  |
| } |  |
| } |  |

### Residual coding syntax

|  |  |
| --- | --- |
| residual\_coding ( x0, y0, log2TrafoWidth, log2TrafoHeight, scanIdx, cIdx ) { | Descriptor |
| **last\_significant\_coeff\_x\_prefix** | ae(v) |
| **last\_significant\_coeff\_y\_prefix** | ae(v) |
| if ( last\_significant\_coeff\_x\_prefix > 3 ) |  |
| **last\_significant\_coeff\_x\_suffix** | ae(v) |
| if ( last\_significant\_coeff\_y\_prefix > 3 ) |  |
| **last\_significant\_coeff\_y\_suffix** | ae(v) |
| numCoeff = 0 |  |
| do { |  |
| xC = ScanOrder[ log2TrafoWidth ][ log2TrafoHeight ][ scanIdx ][ numCoeff ][ 0 ] |  |
| yC = ScanOrder[ log2TrafoWidth ][ log2TrafoHeight ][ scanIdx ][ numCoeff ][ 1 ] |  |
| numCoeff++ |  |
| } while( ( xC != LastSignificantCoeffX ) || ( yC != LastSignificantCoeffY ) ) |  |
| numLastSubset = (numCoeff − 1) >> 4 |  |
| for( i = numLastSubset; i >= 0; i− − ) { |  |
| offset = i << 4 |  |
| if( max( log2TrafoWidth, log2TrafoHeight ) > 3) { |  |
| xCG = ScanOrder[ log2TrafoWidth – 2 ][ log2TrafoHeight – 2 ][ scanIdx ][ i ][ 0 ] |  |
| yCG = ScanOrder[ log2TrafoWidth – 2 ][ log2TrafoHeight – 2 ][ scanIdx ][ i ][ 1 ] |  |
| rightCGFlag = (xCG = = (1<< (log2TrafoWidth – 2)) − 1) ? 0 :   significant\_coeff\_group\_flag[ xCG + 1 ][ yCG ] |  |
| bottomCGFlag = (yCG = = (1 << (log2TrafoHeight – 2)) − 1) ? 0 :   significant\_coeff\_group\_flag[ xCG ][ yCG + 1 ] |  |
| if( (i < numLastSubset) && (rightCGFlag + bottomCGFlag < 2) && (i > 0) ) |  |
| **significant\_coeff\_group\_flag**[ xCG ][ yCG ] | ae(v) |
| for( n = 15; n >= 0; n− − ) { |  |
| xC = ScanOrder[ log2TrafoWidth ][ log2TrafoHeight ][ scanIdx ][ n + offset ][ 0 ] |  |
| yC = ScanOrder[ log2TrafoWidth ][ log2TrafoHeight ][ scanIdx ][ n + offset ][ 1 ] |  |
| if ( (n + offset) < (numCoeff − 1) && significant\_coeff\_group\_flag[ xCG ][ yCG ] ) { |  |
| numNZInCG = (i = = numLastSubset) ? 1 : 0 |  |
| if ( (n > 0) | | (rightCGFlag + bottomCGFlag = = 2) | | (i = = 0) | | (numNZInCG > 0) ) { |  |
| **significant\_coeff\_flag**[ xC ][ yC ] | ae(v) |
| numNZInCG += significant\_coeff\_flag[ xC ][ yC ] |  |
| else |  |
| significant\_coeff\_flag[ xC ][ yC ] = 1 |  |
| } |  |
| } |  |
| } else { |  |
| for( n = 15; n >= 0; n− − ) { |  |
| xC = ScanOrder[ log2TrafoWidth ][ log2TrafoHeight ][ scanIdx ][ n + offset ][ 0 ] |  |
| yC = ScanOrder[ log2TrafoWidth ][ log2TrafoHeight ][ scanIdx ][ n + offset ][ 1 ] |  |
| if( (n + offset) < (numCoeff − 1) ) |  |
| **significant\_coeff\_flag[** xC **][** yC **]** | ae(v) |
| } |  |
| } |  |
| for( n = 15; n >= 0; n− − ) { |  |
| xC = ScanOrder[ log2TrafoWidth ][ log2TrafoHeight ][ scanIdx ][ n + offset ][ 0 ] |  |
| yC = ScanOrder[ log2TrafoWidth ][ log2TrafoHeight ][ scanIdx ][ n + offset ][ 1 ] |  |
| if( significant\_coeff\_flag[ xC ][ yC ] ) |  |
| **coeff\_abs\_level\_greater1\_flag[** n **]** | ae(v) |
| } |  |
| for( n = 15; n >= 0; n− − ) { |  |
| if( coeff\_abs\_level\_greater1\_flag[ n ] ) |  |
| **coeff\_abs\_level\_greater2\_flag[** n **]** | ae(v) |
| } |  |
| for( n = 15; n >= 0; n− − ) { |  |
| xC = ScanOrder[ log2TrafoWidth ][ log2TrafoHeight ][ scanIdx ][ n + offset ][ 0 ] |  |
| yC = ScanOrder[ log2TrafoWidth ][ log2TrafoHeight ][ scanIdx ][ n + offset ][ 1 ] |  |
| if( significant\_coeff\_flag[ xC ][ yC ] ) { |  |
| **coeff\_sign\_flag[** n **]** | ae(v) |
| } |  |
| for( n = 15; n >= 0; n− − ) { |  |
| if( coeff\_abs\_level\_greater2\_flag[ n ] ) |  |
| **coeff\_abs\_level\_minus3[** n **]** | ae(v) |
| xC = ScanOrder[ log2TrafoWidth ][ log2TrafoHeight ][ scanIdx ][ n + offset ][ 0 ] |  |
| yC = ScanOrder[ log2TrafoWidth ][ log2TrafoHeight ][ scanIdx ][ n + offset ][ 1 ] |  |
| if( significant\_coeff\_flag[ xC ][ yC ] ) { |  |
| transCoeffLevel[ x0 ][ y0 ][ cIdx ][ xC ][ yC ] =   ( coeff\_abs\_level\_minus3[ n ] + 3 ) \* ( 1 − 2 \* coeff\_sign\_flag[ n ] ) |  |
| } else |  |
| transCoeffLevel[ x0 ][ y0 ][ cIdx ][ xC ][ yC ] = 0 |  |
| } |  |
| } |  |
| } |  |

## Semantics

Semantics associated with the syntax structures and with the syntax elements within these structures are specified in this subclause. When the semantics of a syntax element are specified using a table or a set of tables, any values that are not specified in the table(s) shall not be present in the bitstream unless otherwise specified in this Recommendation | International Standard.

### NAL unit semantics

NOTE 1 – The VCL is specified to efficiently represent the content of the video data. The NAL is specified to format that data and provide header information in a manner appropriate for conveyance on a variety of communication channels or storage media. All data are contained in NAL units, each of which contains an integer number of bytes. A NAL unit specifies a generic format for use in both packet-oriented and bitstream systems. The format of NAL units for both packet-oriented transport and byte stream is identical except that each NAL unit can be preceded by a start code prefix and extra padding bytes in the byte stream format.

NumBytesInNALunit specifies the size of the NAL unit in bytes. This value is required for decoding of the NAL unit. Some form of demarcation of NAL unit boundaries is necessary to enable inference of NumBytesInNALunit. One such demarcation method is specified in Annex B for the byte stream format. Other methods of demarcation may be specified outside of this Recommendation | International Standard.

**forbidden\_zero\_bit** shall be equal to 0.

**nal\_ref\_flag** equal to 1 specifies that the content of the NAL unit contains a sequence parameter set, a picture parameter set, an adaptation parameter set or a slice of a reference picture.

For coded video sequences conforming to one or more of the profiles specified in Annex A that are decoded using the decoding process specified in clauses 2-9, nal\_ref\_flag equal to 0 for a NAL unit containing a slice indicates that the slice is part of a non-reference picture.

nal\_ref\_flag shall be equal to 1 for sequence parameter set, picture parameter set or adaptation parameter set NAL units. When nal\_ref\_flag is equal to 0 for one NAL unit with nal\_unit\_type equal to 1 or 4 of a particular picture, it shall be equal to 0 for all NAL units with nal\_unit\_type equal to 1 or 4 of the picture.

nal\_ref\_flag shall be equal to 1 for NAL units with nal\_unit\_type equal to 5.

nal\_ref\_flag shall be equal to 0 for all NAL units having nal\_unit\_type equal to 6, 9, 10, 11, or 12.

**nal\_unit\_type** specifies the type of RBSP data structure contained in the NAL unit as specified in Table 7‑1.

NAL units that use nal\_unit\_type equal to 0 or in the range of 24..63, inclusive, shall not affect the decoding process specified in this Recommendation | International Standard.

NOTE 2 – NAL unit types 0 and 24..63 may be used as determined by the application. No decoding process for these values of nal\_unit\_type is specified in this Recommendation | International Standard. Since different applications might use NAL unit types 0 and 24..63 for different purposes, particular care must be exercised in the design of encoders that generate NAL units with nal\_unit\_type equal to 0 or in the range of 24 to 63, inclusive, and in the design of decoders that interpret the content of NAL units with nal\_unit\_type equal to 0 or in the range of 24 to 63, inclusive.

Decoders shall ignore (remove from the bitstream and discard) the contents of all NAL units that use reserved values of nal\_unit\_type.

NOTE 3 – This requirement allows future definition of compatible extensions to this Recommendation | International Standard.

Table 7‑1 – NAL unit type codes and NAL unit type classes

|  |  |  |
| --- | --- | --- |
| **nal\_unit\_type** | **Content of NAL unit and RBSP syntax structure** | **NAL unit type class** |
| 0 | Unspecified | non-VCL |
| 1 | Coded slice of a non-IDR and non-CRA picture slice\_layer\_rbsp( ) | VCL |
| 2-3 | Reserved | n/a |
| 4 | Coded slice of a CRA picture  slice\_layer\_rbsp( ) | VCL |
| 5 | Coded slice of an IDR picture slice\_layer\_rbsp( ) | VCL |
| 6 | Supplemental enhancement information (SEI) sei\_rbsp( ) | non-VCL |
| 7 | Sequence parameter set seq\_parameter\_set\_rbsp( ) | non-VCL |
| 8 | Picture parameter set pic\_parameter\_set\_rbsp( ) | non-VCL |
| 9 | Access unit delimiter access\_unit\_delimiter\_rbsp( ) | non-VCL |
| 10-11 | Reserved | n/a |
| 12 | Filler data filler\_data\_rbsp( ) | non-VCL |
| 13 | Reserved | n/a |
| 14 | Adaptation parameter set aps\_rbsp( ) | non-VCL |
| 15-23 | Reserved | n/a |
| 24..63 | Unspecified | non-VCL |

In the text, coded slice NAL unit collectively refers to a coded slice of a non-IDR picture NAL unit or to a coded slice of an IDR picture NAL unit. The variable IdrPicFlag is specified as

IdrPicFlag = ( ( nal\_unit\_type = = 5 ) ? 1 : 0 ) (7‑1)

When the value of nal\_unit\_type is equal to 5 for a NAL unit containing a slice of a particular picture, the picture shall not contain NAL units with nal\_unit\_type equal to 1 or 4. For coded video sequences conforming to one or more of the profiles specified in Annex A that are decoded using the decoding process specified in clauses 2-9, such a picture is referred to as an IDR picture.

**temporal\_id** specifies a temporal identifier for the NAL unit. The value of temporal\_id shall be the same for all NAL units of an access unit. When an access unit contains any NAL unit with nal\_unit\_type equal to 5, temporal\_id shall be equal to 0.

**output\_flag** affects the decoded picture output and removal processes as specified in Annex C.

**reserved\_one\_4bits** shall be equal to 1. Other values of reserved\_one\_4bits may be specified in the future by ITU‑T | ISO/IEC. Decoders shall ignore the value of reserved\_one\_4bits.

**rbsp\_byte[** i **]** is the i-th byte of an RBSP. An RBSP is specified as an ordered sequence of bytes as follows.

The RBSP contains an SODB as follows.

– If the SODB is empty (i.e., zero bits in length), the RBSP is also empty.

– Otherwise, the RBSP contains the SODB as follows:

1) The first byte of the RBSP contains the (most significant, left-most) eight bits of the SODB; the next byte of the RBSP shall contain the next eight bits of the SODB, etc., until fewer than eight bits of the SODB remain.

2) rbsp\_trailing\_bits( ) are present after the SODB as follows:

i) The first (most significant, left-most) bits of the final RBSP byte contains the remaining bits of the SODB (if any).

ii) The next bit consists of a single rbsp\_stop\_one\_bit equal to 1.

iii) When the rbsp\_stop\_one\_bit is not the last bit of a byte-aligned byte, one or more rbsp\_alignment\_zero\_bit is present to result in byte alignment.

3) One or more cabac\_zero\_word 16-bit syntax elements equal to 0x0000 may be present in some RBSPs after the rbsp\_trailing\_bits( ) at the end of the RBSP.

Syntax structures having these RBSP properties are denoted in the syntax tables using an "\_rbsp" suffix. These structures shall be carried within NAL units as the content of the rbsp\_byte[ i ] data bytes. The association of the RBSP syntax structures to the NAL units shall be as specified in Table 7‑1.

NOTE 6 – When the boundaries of the RBSP are known, the decoder can extract the SODB from the RBSP by concatenating the bits of the bytes of the RBSP and discarding the rbsp\_stop\_one\_bit, which is the last (least significant, right-most) bit equal to 1, and discarding any following (less significant, farther to the right) bits that follow it, which are equal to 0. The data necessary for the decoding process is contained in the SODB part of the RBSP.

**emulation\_prevention\_three\_byte** is a byte equal to 0x03. When an emulation\_prevention\_three\_byte is present in the NAL unit, it shall be discarded by the decoding process.

The last byte of the NAL unit shall not be equal to 0x00.

Within the NAL unit, the following three-byte sequences shall not occur at any byte-aligned position:

– 0x000000

– 0x000001

– 0x000002

Within the NAL unit, any four-byte sequence that starts with 0x000003 other than the following sequences shall not occur at any byte-aligned position:

– 0x00000300

– 0x00000301

– 0x00000302

– 0x00000303

NOTE 7 – When nal\_unit\_type is equal to 0, particular care must be exercised in the design of encoders to avoid the presence of the above-listed three-byte and four-byte patterns at the beginning of the NAL unit syntax structure, as the syntax element emulation\_prevention\_three\_byte cannot be the third byte of a NAL unit.

#### Encapsulation of an SODB within an RBSP (informative)

This subclause does not form an integral part of this Recommendation | International Standard.

The form of encapsulation of an SODB within an RBSP and the use of the emulation\_prevention\_three\_byte for encapsulation of an RBSP within a NAL unit is specified for the following purposes:

– to prevent the emulation of start codes within NAL units while allowing any arbitrary SODB to be represented within a NAL unit,

– to enable identification of the end of the SODB within the NAL unit by searching the RBSP for the rbsp\_stop\_one\_bit starting at the end of the RBSP,

– to enable a NAL unit to have a size larger than that of the SODB under some circumstances (using one or more cabac\_zero\_word).

The encoder can produce a NAL unit from an RBSP by the following procedure:

1. The RBSP data is searched for byte-aligned bits of the following binary patterns:

'00000000 00000000 000000xx' (where xx represents any 2 bit pattern: 00, 01, 10, or 11),

and a byte equal to 0x03 is inserted to replace these bit patterns with the patterns:

'00000000 00000000 00000011 000000xx',

and finally, when the last byte of the RBSP data is equal to 0x00 (which can only occur when the RBSP ends in a cabac\_zero\_word), a final byte equal to 0x03 is appended to the end of the data. The last zero byte of a byte‑aligned three-byte sequence 0x000000 in the RBSP (which is replaced by the four-byte sequence 0x00000300) is taken into account when searching the RBSP data for the next occurrence of byte-aligned bits with the binary patterns specified above.

1. The resulting sequence of bytes is then prefixed as follows.

– If nal\_unit\_type is not equal to 14 or 20, the sequence of bytes is prefixed with the first byte of the NAL unit containing the syntax elements forbidden\_zero\_bit, nal\_ref\_flag, and nal\_unit\_type, where nal\_unit\_type indicates the type of RBSP data structure the NAL unit contains.

– Otherwise (nal\_unit\_type is equal to 14 or 20), the sequence of bytes is prefixed with the first four bytes of the NAL unit, where the first byte contains the syntax elements forbidden\_zero\_bit, nal\_ref\_flag, and nal\_unit\_type and the following three bytes contain the syntax structure nal\_unit\_header\_svc\_extension( ). The syntax element nal\_unit\_type in the first byte indicates the presence of the syntax structure nal\_unit\_header\_svc\_extension( ) in the following three bytes and the type of RBSP data structure the NAL unit contains.

The process specified above results in the construction of the entire NAL unit.

This process can allow any SODB to be represented in a NAL unit while ensuring that

– no byte-aligned start code prefix is emulated within the NAL unit,

* no sequence of 8 zero-valued bits followed by a start code prefix, regardless of byte-alignment, is emulated within the NAL unit.

#### Order of NAL units and association to coded pictures, access units, and video sequences

This subclause specifies constraints on the order of NAL units in the bitstream.

Any order of NAL units in the bitstream obeying these constraints is referred to in the text as the decoding order of NAL units. Within a NAL unit, the syntax in subclauses 7.3, D.1, and E.1 specifies the decoding order of syntax elements. Decoders shall be capable of receiving NAL units and their syntax elements in decoding order.

##### Order of sequence, picture and adaptation parameter set RBSPs and their activation

This subclause specifies the activation process of picture, sequence and adaptation parameter sets for coded video sequences that conform to one or more of the profiles specified in Annex A that are decoded using the decoding process specified in clauses 2-9.

NOTE 1 – The sequence, picture and adaptation parameter set mechanism decouples the transmission of infrequently changing information from the transmission of coded macroblock data. Sequence, picture and adaptation parameter sets may, in some applications, be conveyed "out-of-band".

An adaptation parameter set RBSP includes parameters that can be referred to by the coded slice NAL units of one or more coded pictures when at least one of sample\_adaptive\_offset\_enabled\_flag or adaptive\_loop\_filter\_enabled\_flag are equal to 1. Each adaptation parameter set RBSP is initially considered not active at the start of the operation of the decoding process. At most one adaptation parameter set RBSP is considered active at any given moment during the operation of the decoding process, and the activation of any particular adaptation parameter set RBSP results in the deactivation of the previously-active adaptation parameter set RBSP (if any).

When an adaptation parameter set RBSP (with a particular value of aps\_id) is not active and it is referred to by a coded slice NAL unit (using that value of aps\_id), it is activated. This adaptation parameter set RBSP is called the active adaptation parameter set RBSP until it is deactivated by the activation of another adaptation parameter set RBSP. An adaptation parameter set RBSP, with that particular value of aps\_id, shall be available to the decoding process prior to its activation.

A picture parameter set RBSP includes parameters that can be referred to by the coded slice NAL units or coded slice data partition A NAL units of one or more coded pictures. Each picture parameter set RBSP is initially considered not active at the start of the operation of the decoding process. At most one picture parameter set RBSP is considered active at any given moment during the operation of the decoding process, and the activation of any particular picture parameter set RBSP results in the deactivation of the previously-active picture parameter set RBSP (if any).

When a picture parameter set RBSP (with a particular value of pic\_parameter\_set\_id) is not active and it is referred to by a coded slice NAL unit or coded slice data partition A NAL unit (using that value of pic\_parameter\_set\_id), it is activated. This picture parameter set RBSP is called the active picture parameter set RBSP until it is deactivated by the activation of another picture parameter set RBSP. A picture parameter set RBSP, with that particular value of pic\_parameter\_set\_id, shall be available to the decoding process prior to its activation.

Any picture parameter set NAL unit containing the value of pic\_parameter\_set\_id for the active picture parameter set RBSP for a coded picture shall have the same content as that of the active picture parameter set RBSP for the coded picture unless it follows the last VCL NAL unit of the coded picture and precedes the first VCL NAL unit of another coded picture.

A sequence parameter set RBSP includes parameters that can be referred to by one or more picture parameter set RBSPs or one or more SEI NAL units containing a buffering period SEI message. Each sequence parameter set RBSP is initially considered not active at the start of the operation of the decoding process. At most one sequence parameter set RBSP is considered active at any given moment during the operation of the decoding process, and the activation of any particular sequence parameter set RBSP results in the deactivation of the previously-active sequence parameter set RBSP (if any).

When a sequence parameter set RBSP (with a particular value of seq\_parameter\_set\_id) is not already active and it is referred to by activation of a picture parameter set RBSP (using that value of seq\_parameter\_set\_id) or is referred to by an SEI NAL unit containing a buffering period SEI message (using that value of seq\_parameter\_set\_id), it is activated. This sequence parameter set RBSP is called the active sequence parameter set RBSP until it is deactivated by the activation of another sequence parameter set RBSP. A sequence parameter set RBSP, with that particular value of seq\_parameter\_set\_id, shall be available to the decoding process prior to its activation. An activated sequence parameter set RBSP shall remain active for the entire coded video sequence.

NOTE 2 – Because an IDR access unit begins a new coded video sequence and an activated sequence parameter set RBSP must remain active for the entire coded video sequence, a sequence parameter set RBSP can only be activated by a buffering period SEI message when the buffering period SEI message is part of an IDR access unit.

Any sequence parameter set NAL unit containing the value of seq\_parameter\_set\_id for the active sequence parameter set RBSP for a coded video sequence shall have the same content as that of the active sequence parameter set RBSP for the coded video sequence unless it follows the last access unit of the coded video sequence and precedes the first VCL NAL unit and the first SEI NAL unit containing a buffering period SEI message (when present) of another coded video sequence.

NOTE 3 – If picture parameter set RBSP, sequence parameter set RBSP or adaptation parameter set RBSP are conveyed within the bitstream, these constraints impose an order constraint on the NAL units that contain the picture parameter set RBSP or sequence parameter set RBSP, respectively. Otherwise (picture parameter set RBSP, sequence parameter set RBSP or adaptation parameter set RBSP are conveyed by other means not specified in this Recommendation | International Standard), they must be available to the decoding process in a timely fashion such that these constraints are obeyed.

All constraints that are expressed on the relationship between the values of the syntax elements (and the values of variables derived from those syntax elements) in sequence parameter sets, picture parameter sets and adaptation parameter sets and other syntax elements are expressions of constraints that apply only to the active sequence parameter set, the active picture parameter set and the active adaptation parameter set. If any sequence parameter set RBSP is present that is not activated in the bitstream, its syntax elements shall have values that would conform to the specified constraints if it were activated by reference in an otherwise‑conforming bitstream. If any picture parameter set RBSP is present that is not ever activated in the bitstream, its syntax elements shall have values that would conform to the specified constraints if it were activated by reference in an otherwise-conforming bitstream. If any adaptation parameter set RBSP is present that is not ever activated in the bitstream, its syntax elements shall have values that would conform to the specified constraints if it were activated by reference in an otherwise-conforming bitstream.

During operation of the decoding process (see clause 0), the values of parameters of the active picture parameter set and the active sequence parameter set shall be considered in effect. For interpretation of SEI messages, the values of the parameters of the picture parameter set and sequence parameter set that are active for the operation of the decoding process for the VCL NAL units of the primary coded picture in the same access unit shall be considered in effect unless otherwise specified in the SEI message semantics.

##### Order of access units and association to coded video sequences

A bitstream conforming to this Recommendation | International Standard consists of one or more coded video sequences.

A coded video sequence consists of one or more access units. For coded video sequences that conform to one or more of the profiles specified in Annex A that are decoded using the decoding process specified in clauses 2-9, the order of NAL units and coded pictures and their association to access units is described in subclause 7.4.1.2.3.

The first access unit of each coded video sequence is an IDR access unit. All subsequent access units in the coded video sequence are non-IDR access units.

The values of picture order count for the coded pictures in consecutive access units in decoding order containing non‑reference pictures shall be increasing.

When present, an access unit following an access unit that contains an end of sequence NAL unit shall be an IDR access unit.

When an SEI NAL unit contains data that pertain to more than one access unit (for example, when the SEI NAL unit has a coded video sequence as its scope), it shall be contained in the first access unit to which it applies.

When an end of stream NAL unit is present in an access unit, this access unit shall be the last access unit in the bitstream and the end of stream NAL unit shall be the last NAL unit in that access unit.

##### Order of NAL units and coded pictures and association to access units

This subclause specifies the order of NAL units and coded pictures and association to access unit for coded video sequences that conform to one or more of the profiles specified in Annex A that are decoded using the decoding process specified in clauses 2-9.

An access unit consists of one primary coded picture, zero or more corresponding redundant coded pictures, and zero or more non-VCL NAL units. The association of VCL NAL units to primary or redundant coded pictures is described in subclause 7.4.1.2.5.

The first access unit in the bitstream starts with the first NAL unit of the bitstream.

The first of any of the following NAL units after the last VCL NAL unit of a primary coded picture specifies the start of a new access unit:

– access unit delimiter NAL unit (when present),

– sequence parameter set NAL unit (when present),

– picture parameter set NAL unit (when present),

– SEI NAL unit (when present),

– NAL units with nal\_unit\_type in the range of 14 to 18, inclusive (when present),

– first VCL NAL unit of a primary coded picture (always present).

The constraints for the detection of the first VCL NAL unit of a primary coded picture are specified in subclause 7.4.1.2.4.

The following constraints shall be obeyed by the order of the coded pictures and non-VCL NAL units within an access unit:

– When an access unit delimiter NAL unit is present, it shall be the first NAL unit. There shall be at most one access unit delimiter NAL unit in any access unit.

– When any SEI NAL units are present, they shall precede the primary coded picture.

– When an SEI NAL unit containing a buffering period SEI message is present, the buffering period SEI message shall be the first SEI message payload of the first SEI NAL unit in the access unit.

– The primary coded picture shall precede the corresponding redundant coded pictures.

– NAL units having nal\_unit\_type equal to 0, 12, or in the range of 20 to 31, inclusive, shall not precede the first VCL NAL unit of the primary coded picture.

NOTE 1 – Sequence parameter set NAL units or picture parameter set NAL units may be present in an access unit, but cannot follow the last VCL NAL unit of the primary coded picture within the access unit, as this condition would specify the start of a new access unit.

The structure of access units not containing any NAL units with nal\_unit\_type equal to 0, 7, 8, or in the range of 12 to 18, inclusive, or in the range of 20 to 31, inclusive, is shown in Figure 7‑1. [Ed. (TW):adjust text and figure]



Figure 7‑1 – Structure of an access unit not containing any NAL units with nal\_unit\_type equal to 0, 7, 8,   
or in the range of 12 to 18, inclusive, or in the range of 20 to 31, inclusive

##### Detection of the first VCL NAL unit of a primary coded picture

This subclause specifies constraints on VCL NAL unit syntax that are sufficient to enable the detection of the first VCL NAL unit of each primary coded picture for coded video sequences that conform to one or more of the profiles specified in Annex A that are decoded using the decoding process specified in clauses 2-9.

Any coded slice NAL unit or coded slice data partition A NAL unit of the primary coded picture of the current access unit shall be different from any coded slice NAL unit or coded slice data partition A NAL unit of the primary coded picture of the previous access unit in one or more of the following ways:

– pic\_parameter\_set\_id differs in value.

– nal\_ref\_flag differs in value with one of the nal\_ref\_flag values being equal to 0.

– pic\_order\_cnt\_lsb differs in value.

– IdrPicFlag differs in value.

– IdrPicFlag is equal to 1 for both and idr\_pic\_id differs in value.

##### Order of VCL NAL units and association to coded pictures

[Ed. (TW): insert text]

### Raw byte sequence payloads and RBSP trailing bits semantics

#### Sequence parameter set RBSP semantics

**profile\_idc** and **level\_idc** indicate the profile and level to which the coded video sequence conforms.

**reserved\_zero\_8bits** shall be equal to 0. Decoders shall ignore the value of reserved\_zero\_8bits.

**seq\_parameter\_set\_id** identifies the sequence parameter set that is referred to by the picture parameter set. The value of seq\_parameter\_set\_id shall be in the range of 0 to 31, inclusive.

**chroma\_format\_idc** specifies the chroma sampling relative to the luma sampling as specified in subclause 6.2. The value of chroma\_format\_idc shall be in the range of 0 to 3, inclusive. When chroma\_format\_idc is not present, it shall be inferred to be equal to 1 (4:2:0 chroma format).

**max\_temporal\_layers\_minus1** + 1 specifies the maximum number of temporal layers present in the sequence. The value of max\_temporal\_layers\_minus1 shall be in the range of 0 to 7, inclusive.

**pic\_width\_in\_luma\_samples** specifies the width of each decoded picture in luma samples.

**pic\_height\_in\_luma\_samples** specifies the height of each decoded picture in luma samples.

**bit\_depth\_luma\_minus8** + 8 specifies the bit depth of the samples of the luma array and the value of the luma quantisation parameter range offset QpBdOffsetY, as specified by

BitDepthY = 8 + bit\_depth\_luma\_minus8 (7‑2)  
QpBdOffsetY = 6 \* bit\_depth\_luma\_minus8 (7‑3)

bit\_depth\_luma\_minus8 shall be in the range of 0 to 6, inclusive.

**bit\_depth\_chroma\_minus8** + 8specifies the bit depth of the samples of the chroma arrays and the value of the chroma quantisation parameter range offset QpBdOffsetC, as specified by

BitDepthC = 8 + bit\_depth\_chroma\_minus8 (7‑4)  
QpBdOffsetC = 6 \* bit\_depth\_chroma\_minus8 (7‑5)

bit\_depth\_chroma\_minus8 shall be in the range of 0 to 6, inclusive.

**pcm\_enabled\_flag** equal to 0 specifies that PCM data shall not be present in the video sequence.

**pcm\_sample\_bit\_depth\_luma\_minus1** + 1 specifies the number of bits used to represent each of PCM sample values of luma component. The value of pcm\_sample\_bit\_depth\_luma\_minus1 + 1 shall be smaller than or equal to the value of BitDepthY.

PCMBitDepthY = 1 + pcm\_sample\_bit\_depth\_luma\_minus1 (7‑4)

**pcm\_sample\_bit\_depth\_chroma\_minus1** + 1 specifies the number of bits used to represent each of PCM sample values of chroma components. The value of pcm\_sample\_bit\_depth\_chroma\_minus1 + 1 shall be smaller than or equal to the value of BitDepthC.

PCMBitDepthC = 1 + pcm\_sample\_bit\_depth\_chroma\_minus1 (7‑4)

**log2\_max\_pic\_order\_cnt\_lsb\_minus4** specifies the value of the variable MaxPicOrderCntLsb that is used in the decoding process for picture order count as follows:

MaxPicOrderCntLsb = 2( log2\_max\_pic\_order\_cnt\_lsb\_minus4 + 4 ) (7‑7)

The value of log2\_max\_pic\_order\_cnt\_lsb\_minus4 shall be in the range of 0 to 12, inclusive.

**max\_num\_ref\_frames** specifies the maximum number of short-term and long-term reference frames, complementary reference field pairs, and non-paired reference fields that may be used by the decoding process for inter prediction of any picture in the sequence. max\_num\_ref\_frames also determines the size of the sliding window operation. The value of max\_num\_ref\_frames shall be in the range of 0 to MaxDpbFrames, inclusive.

**num\_reorder\_frames** indicates the maximum number of frames in decoding order and follow it in output order. The value of num\_reorder\_frames shall be in the range of 0 to max\_dec\_frame\_buffering, inclusive.

**max\_dec\_frame\_buffering** specifies the required size of the HRD decoded picture buffer (DPB) in units of frame buffers. The coded video sequence shall not require a decoded picture buffer with size of more than Max( 1, max\_dec\_frame\_buffering ) frame buffers to enable the output of decoded pictures at the output times specified by dpb\_output\_delay of the picture timing SEI messages. The value of max\_dec\_frame\_buffering shall be in the range of num\_ref\_frames to MaxDpbSize (as specified in subclause XXX or XXX), inclusive.

**max\_latency\_increase** when not equal to 0, is used to compute the value of MaxLatencyFrames as specified by MaxLatencyFrames = num\_reorder\_frames + max\_latency\_increase. When max\_latency\_increase is not equal to 0, the value of MaxLatencyFrames specifies the maximum number of frames that can precede any frame in the coded video sequence in output order and follow it in decoding order. When max\_latency\_increase is equal to 0, no corresponding limit is expressed. The value of max\_latency\_increase shall be in the range of 0 to 232 − 1, inclusive.

**log2\_min\_coding\_block\_size\_minus3** specifies the minimum size of a coding block.

The variable Log2MinCUSize is set equal to log2\_min\_coding\_block\_size\_minus3 + 3.

**log2\_diff\_max\_min\_coding\_block\_size** specifies the difference between the maximum and minimum coding block size.

The variable Log2MaxCUSize is set equal to log2\_min\_coding\_block\_size\_minus 3 + 3 + log2\_diff\_max\_min\_coding\_block\_size.

**log2\_min\_transform\_block\_size\_minus2** specifies the minimum size of a transform block.

The variable Log2MinTrafoSize is set equal to log2\_min\_transform\_block\_size\_minus2 + 2. The variable Log2MinTrafoSizeC is set equal to Max( Log2MinTrafoSize – 1, 2 ) if chroma\_format\_idc is equal to 1 or 2, otherwise, it is set equal to Log2MinTrafoSize.

**log2\_diff\_max\_min\_transform\_block\_size** specifies the difference between the maximum and minimum transform size.

The variable Log2MaxTrafoSize is set equal to log2\_min\_transform\_block\_size\_minus 2 + 2 + log2\_diff\_max\_min\_transform\_block\_size.

The bitstream shall not contain data that result in Log2MaxTrafoSize greater than Log2MaxCUSize.

**log2\_min\_pcm\_coding\_block\_size\_minus3** + 3 specifies the minimum size of I\_PCM coding blocks.

The variable Log2MinIPCMCUSize is set equal to log2\_min\_pcm\_coding\_block\_size\_minus3 + 3. The variable Log2MinIPCMCUSize shall be equal or less than Min( Log2MaxCUSize, 5 ).

**log2\_diff\_max\_min\_pcm\_coding\_block\_size** specifies the difference between the maximum and minimum size of I\_PCM coding blocks.

The variable Log2MaxIPCMCUSize is set equal to log2\_min\_pcm\_coding\_block\_size\_minus3 + 3 + log2\_diff\_max\_min\_pcm\_coding\_block\_size. The variable Log2MaxIPCMCUSize shall be equal or less than Min( Log2MaxCUSize, 5 ).

**max\_transform\_hierarchy\_depth\_intra** specifies the maximum hierarchy depth for transform blocks of coding blocks coded in intra prediction mode.

**max\_transform\_hierarchy\_depth\_inter** specifies the maximum hierarchy depth for transform units of coding units coded in inter prediction mode.

**scaling\_list\_enable\_flag** equal to 1 specifies that scaling list is used for scaling process for transform coefficients in 8.5.2. scaling\_list\_enable\_flag equal to 0 specifies that scaling list is not used for scaling process for transform coefficients in 8.5.2.

**chroma\_pred\_from\_luma\_enabled\_flag** equal to 1 specifies the intra chroma prediction process using the reconstructed luma block is applied according to the intra chroma prediction mode.

**loop\_filter\_across\_slice\_flag** equal to 1 specifies the in-loop filtering operations are performed across slice boundary; otherwise, the in-loop operations are slice-independent and not across slice boundary. The in-loop filtering operations include deblocking filter, sample adaptive offset, and adaptive loop filter.

**sample\_adaptive\_offset\_enabled\_flag** equal to 1 specifies the sample adaptive offset process is applied to the reconstruced picture after the deblocking filter process.

**adaptive\_loop\_filter\_enabled\_flag** equal to 1 specifies the adaptive loop filter process is applied to the reconstructed picture after the deblocking filter process.

**pcm\_loop\_filter\_disable\_flag** specifies whether the loop filter process is disabled on reconstructed pixels of I\_PCM blocks. If the pcm\_loop\_filter\_disable\_flag value is equal to 1, both deblocking and adaptive loop filter processes on the reconstructed pixels of I\_PCM blocks are disabled; otherwise the pcm\_loop\_filter\_disable\_flag value is equal to 0, both deblocking and adaptive loop filter processes on the reconstructed pixels of I\_PCM blocks are not disabled.

[Ed. (WJ): select one expression – enabled\_flag or disable\_flag]

**temporal\_id\_nesting\_flag** specifies whether inter prediction is additionally restricted for the coded video sequence.

Dependent on temporal\_id\_nesting\_flag, the following applies.

– If temporal\_id\_nesting\_flag is equal to 0, additional constraints may not be obeyed.

– Otherwise (temporal\_id\_nesting\_flag is equal to 1), the following applies.

– For each access unit auA with temporal\_id equal to tIdA, an access unit auB with temporal\_id equal to tIdB and tIdB less than or equal to tIdA shall not be referenced by inter prediction when there exists an access unit auC with temporal\_id equal to tIdC and tIdC less than tIdB, which follows the access unit auB and precedes the access unit auA in decoding order.

NOTE – The syntax element temporal\_id\_nesting\_flag is used to indicate that temporal up-switching, i.e., switching from decoding of up to a specific temporal\_id tIdN to decoding up to a temporal\_id tIdM > tIdN, is always possible.

**inter\_4x4\_enabled\_flag** specifies whether inter prediction can be applied to blocks having the size of 4 by 4 luma samples.

**num\_tile\_columns\_minus1** plus 1 specifies the number of tile columns partitioning the picture.

**num\_tile\_rows\_minus1** plus 1 specifies the number of tile rows partitioning the picture.

**tile\_boundary\_independence\_flag** equal to 1 indicates that all neighbouring treeblocks not contained within the same tile as the treeblock being decoded are marked as not available and that the initialisation process for the entropy decoder is invoked when decoding the first treeblock in a tile. Otherwise, tile\_boundary\_independence\_flag equal to 0 indicates that availability of treeblocks is not affected by tile boundaries and that the synchronization process for the entropy decoder is invoked when decoding the first treeblock in a tile except to the first treeblock in a picture and that memorization process for the entropy decoder is invoked when decoding the first treeblock of the second row in a tile. [Ed. (WJ): is it ok to invoke some processes in semantics subclause?]

When not present, tile\_boundary\_independence\_flag is inferred to be 1.

**loop\_filter\_across\_tile\_flag** equal to 1 specifies the in-loop filtering operations are performed across tile boundary; otherwise, the in-loop filtering operations are tile-independent and not across tile boundary. The in-loop filtering operations include deblocking filter, sample adaptive offset, and adaptive loop filter. loop\_filter\_across\_tile\_flag is not coded if tile\_boundary\_independent\_flag is not equal to 1 or if num\_tile\_columns\_minus1 and num\_tile\_rows\_minus1 are both equal to 0; if loop\_filter\_across\_tile\_flag is not coded, loop\_filter\_across\_tile\_flag is set to 1.

**uniform\_spacing\_flag** equal to 1 indicates that column boundaries and likewise row boundaries are distributed uniformly across the picture. uniform\_spacing\_flag equal to 0 indicates that column boundaries and likewise row boundaries are not distributed uniformly across the picture but signaled explicitly using the syntax elements column\_width[ ] and row\_height[ ].

The vector columnWidth[ ], specifying the width for every of the num\_tile\_columns\_minus1 columns in units of treeblocks, is derived as follows.

for( i = 0; i < num\_tile\_columns\_minus1; i++ ) {  
 if( uniform\_spacing\_flag = = 1 )  
 columnWidth[i] = ((i+1) \* PicWidthInLCUs) / (num\_tile\_columns\_minus1+1) – (7‑8)  
 (i \* PicWidthInLCUs) / (num\_tile\_columns\_minus1+1)  
 else  
 columnWidth[i] = column\_width[i]   
}

The vector rowHeight[ ], specifying the height for every of the num\_tile\_rows\_minus1 rows in units of treeblocks, is derived as follows.

for( i = 0; i < num\_tile\_rows\_minus1; i++ ) {  
 if( uniform\_spacing\_flag = = 1 )  
 rowHeight[i] = ( (i + 1 ) \* PicHeightInLCUs ) / ( num\_tile\_rows\_minus1+1) – (7‑8)  
 ( i \* PicHeightInLCUs ) / ( num\_tile\_rows\_minus1+1)   
 else  
 rowHeight[i] = row\_height[i]   
}

The vector colBd[ ], specifying the location of the left column boundary for every column of the num\_tile\_columns\_minus1+1 columns in units of treeblocks, is derived as follows.

colBd[0] = 0  
colBd[num\_tile\_columns\_minus1+1] = picWidthInLCUs  
if( num\_tile\_columns\_minus1 > 0 ) {  
 colBd[1] = columnWidth[0] (7‑8)  
 for( i = 1; i < num\_tile\_columns\_minus1; i++ )  
 colBd[i+1] = colBd[i] + columnWidth[i]  
}

The vector rowBd[ ], specifying the location of the top row boundary for every row of the num\_tile\_columns\_minus1+1 rows in units of treeblocks, is derived as follows.

rowBd[0] = 0  
rowBd[num\_tile\_rows\_minus1+1] = picHeightInLCUs  
if( num\_tile\_rows\_minus1 > 0 ) {  
 rowBd [1] = rowHeight[0] (7‑8)  
 for( i = 1; i < num\_tile\_rows\_minus1; i++ )  
 rowBd[i+1] = rowBd[i] + rowHeight[i]   
}

**column\_width[**i**]** specifies the width of the i-th column in units of treeblocks. The value of column\_width[i] shall be limited such that colBd[i+1] is less than PicWidthInLCUs for i in the range of 0 to num\_tile\_columns\_minus1 – 1, inclusive, and num\_tile\_columns\_minus1 greater than 0.

**row\_height[**i**]** specifies the height of the i-th row in units of treeblocks. The value of row\_height[i] shall be limited such that rowBd[i+1] is less than PicHeightInLCUs for i in the range of 0 to num\_tile\_rows\_minus1 – 1, inclusive, and num\_tile\_rows\_minus1 greater than 0.

**sps\_extension\_flag** equal to 0 specifies that no sps\_extension\_data\_flag syntax elements are present in the sequence parameter set RBSP syntax structure. sps\_extension\_flag shall be equal to 0 in bitstreams conforming to this Recommendation | International Standard. The value of 1 for sps\_extension\_flag is reserved for future use by ITU-T | ISO/IEC. Decoders shall ignore all data that follow the value 1 for sps\_extension\_flag in a sequence parameter set NAL unit.

**sps\_extension\_data\_flag** may have any value. It shall not affect the conformance to profiles specified in this Recommendation | International Standard.

#### Picture parameter set RBSP semantics

**pic\_parameter\_set\_id** identifies the picture parameter set that is referred to in the slice header. The value of pic\_parameter\_set\_id shall be in the range of 0 to 255, inclusive.

**seq\_parameter\_set\_id** refers to the active sequence parameter set. The value of seq\_parameter\_set\_id shall be in the range of 0 to 31, inclusive.

**num\_short\_term\_ref\_pic\_sets** specifies the number of short-term reference picture sets that are specified in the picture parameter set. The value of num\_short\_term\_ref\_pic\_sets shall be in the range of 0 to 64, inclusive.

NOTE – A decoder must allocate space for a total number of num\_short\_term\_ref\_pic\_sets + 1 short-term reference picture sets since a coded video sequence may contain up to one short-term reference picture set explicitly signalled in the slice headers of a current picture. An explicitly signalled short-term reference picture set will always have an index equal to num\_short\_term\_ref\_pic\_sets in the list of short-term reference picture sets.

**long\_term\_ref\_pics\_present\_flag** equal to 0 specifies that no long-term reference picture is used for inter prediction of any coded picture referring to the picture parameter set. long\_term\_ref\_pics\_present\_flag equal to 1 specifies that long-term reference pictures may be used for inter prediction of one or more coded pictures referring to the picture parameter set.

**entropy\_coding\_synchro** specifies whether a specific synchronization process for context variables is invoked before decoding the first LCU of a line of LCUs in a tile. This syntax element also specifies whether a specific memorization process for context variables is invoked after decoding entropy\_coding\_synchro+1 LCUs of a line of LCUs in a tile. If entropy\_coding\_synchro is equal to 0, the specific memorization process and synchronization process for context variables are not invoked. If entropy\_coding\_synchro is different from 0, the specific memorization process is invoked after decoding entropy\_coding\_synchro+1 LCUs of a line of LCUs in a tile and the specific synchronization process is invoked before decoding the first LCU of a line of LCUs in a tile. The value of entropy\_coding\_synchro shall be in the range of 0 to tileWidthInLCUs−1, inclusive.

**cabac\_istate\_reset\_flag** specifies that a specific CABAC state re-initialization process of internal variables is invoked when a vertical independent tile boundary is reached.

**num\_substreams\_minus1** specifies the number of substream\_length\_mode and substream\_length syntax element to be decoded from the slice header.

When not present, num\_substreams\_minus1 is inferred to be −1.

**num\_temporal\_layer\_switching\_point\_flags** specifies how many temporal switching point flags are present. If temporal\_id\_nesting\_flag is equal to 1, num\_temporal\_layer\_switching\_point\_flags shall be equal to 0.

**temporal\_layer\_switching\_point\_flag**[ i ] specifies if the current access point is a temporal switching point allowing the decoding of higher temporal id layers following this access unit. If temporal\_id\_nesting\_flag is equal to 1, temporal\_layer\_switching\_point\_flag[ i ] shall be inferred to be equal to 1. If temporal\_id\_nesting\_flag is equal to 0 and num\_temporal\_layer\_switching\_point\_flags is less than i, temporal\_layer\_switching\_point\_flag[ i ] shall be inferred to be equal to 0 .

NOTE – When starting to decode a higher temporal layer i, availability of required reference pictures can be guaranteed immediately following an IDR, or a picture with the temporal\_id value j lower than i and temporal\_switching\_flag[ j ] equal to 1.

**num\_ref\_idx\_l0\_default\_active\_minus1** specifies how num\_ref\_idx\_l0\_active\_minus1 is inferred for P and B slices with num\_ref\_idx\_active\_override\_flag equal to 0. The value of num\_ref\_idx\_l0\_default\_active\_minus1 shall be in the range of 0 to 31, inclusive.

**num\_ref\_idx\_l1\_default\_active\_minus1** specifies how num\_ref\_idx\_l1\_active\_minus1 is inferred for B slices with num\_ref\_idx\_active\_override\_flag equal to 0. The value of num\_ref\_idx\_l1\_default\_active\_minus1 shall be in the range of 0 to 31, inclusive.

**pic\_init\_qp\_minus26** specifies the initial value minus 26 of SliceQPY for each slice. The initial value is modified at the slice layer when a non-zero value of slice\_qp\_delta is decoded, and is modified further when a non-zero value of cu\_qp\_delta is decoded at the coding unit layer. The value of pic\_init\_qp\_minus26 shall be in the range of −(26 + QpBdOffsetY ) to +25, inclusive.

**constrained\_intra\_pred\_flag** equal to 0 specifies that intra prediction allows usage of residual data and decoded samples of neighbouring macroblocks coded using Inter macroblock prediction modes for the prediction of macroblocks coded using Intra macroblock prediction modes. constrained\_intra\_pred\_flag equal to 1 specifies constrained intra prediction, in which case prediction of macroblocks coded using Intra macroblock prediction modes only uses residual data and decoded samples from I macroblock types.

**enable\_temporal\_mvp\_flag** specifies whether temporal motion vector predictor can be used or not. If enable\_temporal\_mvp\_flag is equal to 1, temporal motion vector predictors are used; otherwise (enable\_temporal\_mvp\_flag is equal to 0), the temporal motion vector predictors are not used. enable\_temporal\_mvp\_flag equal to 0 together with temporal\_id equal to 0 also indicates that the marking process will occur before decoding the first picture with an output order greater than the current non-TMVP picture.

**slice\_granularity** indicates the slice granularity within a picture. The value of slice\_granularity shall not be larger than Min( Log2MaxCUSize – 4, log2\_diff\_max\_min\_coding\_block\_size ). The variable SliceGranularity is set to the value of ( slice\_granularity << 1 ).

**shared\_pps\_info\_enabled\_flag** specifies the shared information in picture parameter set RBSP shall be used for the referred slices. If shared\_pps\_info\_enabled\_flag is equal to 1, the alf\_param() in picture parameter set RBSP shall be applied for the referred slices; otherwise, the alf\_param() in slice header(s) shall be applied.

**max\_cu\_qp\_delta\_depth** specifies the maximum allowed depth that is used for specifying QPY values for coding unit. The value of max\_cu\_qp\_delta\_depth shall be in the range of 0 to 15, inclusive.

[Ed. (BB): Why in the range of 0 to 15, inclusive? Is a Log2MaxCUSize of 15 supported?]

The variable log2MinCUDQPSize specifying the minimum coding unit size that can further modifies the value of QPY as follows:

log2MinCUDQPSize = Log2MaxCUSize - max\_cu\_qp\_delta\_depth (7‑7)

The value of log2MinCUDQPSize should be larger than or equal to the CU size specified by the value of slice\_granurality.

**chroma\_cb\_qp\_offset, chroma\_cr\_qp\_offset** are specifying the offset to the luma quantization parameter QP’Y used for deriving QP’Cb and QP’Cr , respectively.

**weighted\_pred\_flag** equal to 0 specifies that weighted prediction shall not be applied to P slices. weighted\_pred\_flag equal to 1 specifies that weighted prediction shall be applied to P slices.

**weighted\_bipred\_idc** equal to 0 specifies that the default weighted prediction shall be applied to B slices. weighted\_bipred\_idc equal to 1 specifies that explicit weighted prediction shall be applied to B slices. weighted\_bipred\_idc equal to 2 specifies that implicit weighted prediction shall be applied to B slices. The value of weighted\_bipred\_idc shall be in the range of 0 to 2, inclusive.

**tile\_info\_present\_flag** equal to 1 specifies that the syntax elements num\_tile\_columns\_minus1, num\_tile\_rows\_minus1 are present and the syntax elements uniform\_spacing\_flag, column\_width[i], and row\_height[i] may be present in the picture parameter set. tile\_info\_present\_flag equal to 0 specifies that syntax elements num\_tile\_columns\_minus1, num\_tile\_rows\_minus1, uniform\_spacing\_flag, column\_width[i], and row\_height[i] are not present in the picture parameter set.

**tile\_control\_present\_flag** equal to 1 specifies that the syntax elements tile\_boundary\_independece\_flag and loop\_filter\_across\_tile\_flag may be present in the picture parameter set. tile\_control\_present\_flag equal to 0 specifies that syntax elements tile\_boundary\_independece\_flag and loop\_filter\_across\_tile\_flag are not present in the picture parameter set.

**pps\_extension\_flag** equal to 0 specifies that no pps\_extension\_data\_flag syntax elements are present in the picture parameter set RBSP syntax structure. pps\_extension\_flag shall be equal to 0 in bitstreams conforming to this Recommendation | International Standard. The value of 1 for pps\_extension\_flag is reserved for future use by ITU‑T | ISO/IEC. Decoders shall ignore all data that follow the value 1 for pps\_extension\_flag in a picture parameter set NAL unit.

**pps\_extension\_data\_flag** may have any value. It shall not affect the conformance to profiles specified in this Recommendation | International Standard.

#### Scaling list data semantics

**seq\_scaling\_list\_present\_flag** equal to 0 specifies that the value of scaling list is not present and the values of all scaling lists are set equal to the default scaling list. seq\_scaling\_list\_present\_flag equal to 1 specifies that the values of scaling list are present. The default value of the scaling list is specified from Table 7‑2 to Table 7‑6.

[Ed. (WJ): Table size seems unnecessarily large. Do we need to keep this large table in the design?]

Table 7‑2 – Specification of default values of ScalingList[ 0 ][ MatrixID ][ 0 ][ i ] with i=0..15

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **i** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[0][0..2][0][i]** | 6 | 13 | 13 | 20 | 20 | 20 | 28 | 28 | 28 | 28 | 32 | 32 | 32 | 37 | 37 | 42 |
| **ScalingList[0][3..5][0][i]** | 10 | 14 | 14 | 20 | 20 | 20 | 24 | 24 | 24 | 24 | 27 | 27 | 27 | 30 | 30 | 34 |

Table 7‑2 – Specification of default values of ScalingList[ 1 ][ MatrixID ][ 0 ][ i ] with i=0..63

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **i** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[1][0..2][0][i]** | 6 | 10 | 10 | 13 | 11 | 13 | 16 | 16 | 16 | 16 | 18 | 18 | 18 | 18 | 18 | 23 |
| **ScalingList[1][3..5][0][i]** | 9 | 13 | 13 | 15 | 13 | 15 | 17 | 17 | 17 | 17 | 19 | 19 | 19 | 19 | 19 | 21 |
| **i – 16** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[1][0..2][0][i]** | 23 | 23 | 23 | 23 | 23 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 27 | 27 | 27 | 27 |
| **ScalingList[1][3..5][0][i]** | 21 | 21 | 21 | 21 | 21 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 24 | 24 | 24 | 24 |
| **i – 32** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[1][0..2][0][i]** | 27 | 27 | 27 | 27 | 29 | 29 | 29 | 29 | 29 | 29 | 29 | 31 | 31 | 31 | 31 | 31 |
| **ScalingList[1][3..5][0][i]** | 24 | 24 | 24 | 24 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 27 | 27 | 27 | 27 | 27 |
| **i – 48** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[1][0..2][0][i]** | 31 | 33 | 33 | 33 | 33 | 33 | 36 | 36 | 36 | 36 | 38 | 38 | 38 | 40 | 40 | 42 |
| **ScalingList[1][3..5][0][i]** | 27 | 28 | 28 | 28 | 28 | 28 | 30 | 30 | 30 | 30 | 32 | 32 | 32 | 33 | 33 | 35 |

Table 7‑2 – Specification of default values of ScalingList[ 2 ][ MatrixID ][ 0 ][ i ] with i=0..255

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **i** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[2][0..2][0][i]** | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| **ScalingList[2][3..5][0][i]** | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| **i – 16** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[2][0..2][0][i]** | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| **ScalingList[2][3..5][0][i]** | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| **i – 32** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[2][0..2][0][i]** | 12 | 12 | 12 | 12 | 13 | 12 | 12 | 12 | 13 | 12 | 12 | 12 | 13 | 13 | 13 | 12 |
| **ScalingList[2][3..5][0][i]** | 12 | 12 | 12 | 12 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 |
| **i – 48** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[2][0..2][0][i]** | 13 | 13 | 13 | 13 | 12 | 13 | 13 | 14 | 13 | 13 | 13 | 13 | 14 | 13 | 13 | 13 |
| **ScalingList[2][3..5][0][i]** | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 |
| **i – 64** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[2][0..2][0][i]** | 13 | 14 | 15 | 14 | 14 | 14 | 14 | 15 | 15 | 14 | 14 | 14 | 14 | 15 | 16 | 15 |
| **ScalingList[2][3..5][0][i]** | 14 | 14 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 16 | 16 |
| **i – 80** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[2][0..2][0][i]** | 14 | 14 | 15 | 15 | 16 | 15 | 15 | 14 | 14 | 15 | 16 | 17 | 16 | 15 | 15 | 15 |
| **ScalingList[2][3..5][0][i]** | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 17 | 17 | 17 | 17 | 17 |
| **i – 96** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[2][0..2][0][i]** | 16 | 17 | 17 | 16 | 15 | 15 | 15 | 16 | 17 | 18 | 17 | 16 | 16 | 16 | 17 | 18 |
| **ScalingList[2][3..5][0][i]** | 17 | 17 | 17 | 17 | 17 | 17 | 17 | 17 | 17 | 18 | 18 | 18 | 18 | 18 | 18 | 18 |
| **i – 112** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[2][0..2][0][i]** | 18 | 18 | 17 | 16 | 16 | 16 | 17 | 18 | 20 | 18 | 18 | 17 | 17 | 18 | 19 | 20 |
| **ScalingList[2][3..5][0][i]** | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 |
| **i – 128** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[2][0..2][0][i]** | 20 | 19 | 18 | 17 | 17 | 18 | 18 | 20 | 20 | 19 | 19 | 19 | 19 | 20 | 22 | 22 |
| **ScalingList[2][3..5][0][i]** | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 |
| **i – 144** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[2][0..2][0][i]** | 22 | 20 | 19 | 19 | 19 | 19 | 20 | 21 | 20 | 20 | 20 | 22 | 23 | 24 | 24 | 23 |
| **ScalingList[2][3..5][0][i]** | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 |
| **i – 160** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[2][0..2][0][i]** | 22 | 20 | 20 | 20 | 21 | 22 | 22 | 22 | 23 | 25 | 26 | 27 | 26 | 25 | 23 | 22 |
| **ScalingList[2][3..5][0][i]** | 21 | 21 | 21 | 21 | 21 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 |
| **i – 176** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[2][0..2][0][i]** | 22 | 22 | 23 | 24 | 25 | 26 | 28 | 30 | 30 | 28 | 26 | 25 | 24 | 23 | 26 | 27 |
| **ScalingList[2][3..5][0][i]** | 22 | 22 | 23 | 23 | 23 | 23 | 23 | 23 | 23 | 23 | 23 | 23 | 23 | 23 | 25 | 25 |
| **i – 192** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[2][0..2][0][i]** | 28 | 31 | 33 | 33 | 33 | 31 | 28 | 27 | 26 | 29 | 31 | 33 | 35 | 37 | 37 | 35 |
| **ScalingList[2][3..5][0][i]** | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 27 | 27 | 27 | 27 | 27 | 27 | 27 |
| **i – 208** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[2][0..2][0][i]** | 33 | 31 | 29 | 33 | 35 | 38 | 41 | 42 | 41 | 38 | 35 | 33 | 38 | 41 | 44 | 47 |
| **ScalingList[2][3..5][0][i]** | 27 | 27 | 27 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 34 | 34 | 34 | 34 |
| **i – 224** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[2][0..2][0][i]** | 47 | 44 | 41 | 38 | 45 | 48 | 51 | 53 | 51 | 48 | 45 | 53 | 57 | 59 | 59 | 57 |
| **ScalingList[2][3..5][0][i]** | 34 | 34 | 34 | 34 | 39 | 39 | 39 | 39 | 39 | 39 | 39 | 45 | 45 | 45 | 45 | 45 |
| **i – 240** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[2][0..2][0][i]** | 53 | 62 | 66 | 67 | 66 | 62 | 72 | 76 | 76 | 72 | 84 | 86 | 84 | 97 | 97 | 112 |
| **ScalingList[2][3..5][0][i]** | 45 | 52 | 52 | 52 | 52 | 52 | 59 | 59 | 59 | 59 | 68 | 68 | 68 | 78 | 78 | 89 |

Table 7‑5 – Specification of default values of ScalingList[ 3 ][ 0 ][ 0 ][ i ] with i=0..1023

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 13 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| 13 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 13 | 13 | 13 | 12 | 12 | 12 | 12 | 12 |
| 13 | 13 | 13 | 13 | 12 | 12 | 12 | 12 | 12 | 13 | 13 | 13 | 13 | 13 | 13 | 12 |
| 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 12 | 13 | 13 | 13 | 13 | 13 | 13 |
| 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 |
| 13 | 13 | 14 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 14 | 14 | 14 | 13 | 13 |
| 13 | 13 | 13 | 13 | 13 | 13 | 14 | 14 | 14 | 13 | 13 | 13 | 13 | 13 | 13 | 14 |
| 14 | 14 | 14 | 14 | 14 | 13 | 13 | 13 | 13 | 13 | 13 | 14 | 14 | 15 | 14 | 14 |
| 14 | 14 | 14 | 14 | 14 | 14 | 14 | 15 | 15 | 15 | 14 | 14 | 14 | 14 | 14 | 14 |
| 14 | 14 | 14 | 15 | 15 | 15 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 15 | 15 | 15 |
| 15 | 15 | 15 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 15 | 15 | 16 | 15 | 15 | 14 |
| 14 | 14 | 14 | 14 | 15 | 15 | 15 | 16 | 16 | 16 | 15 | 15 | 15 | 14 | 14 | 14 |
| 14 | 14 | 15 | 15 | 16 | 16 | 16 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 16 |
| 16 | 16 | 16 | 16 | 16 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 16 | 16 | 17 |
| 16 | 16 | 15 | 15 | 15 | 15 | 15 | 15 | 16 | 16 | 17 | 17 | 17 | 17 | 17 | 16 |
| 16 | 15 | 15 | 15 | 15 | 15 | 15 | 16 | 16 | 17 | 18 | 17 | 16 | 16 | 16 | 16 |
| 16 | 16 | 16 | 16 | 17 | 17 | 17 | 18 | 18 | 17 | 17 | 17 | 16 | 16 | 16 | 16 |
| 16 | 16 | 16 | 16 | 17 | 18 | 18 | 18 | 17 | 17 | 16 | 16 | 16 | 16 | 16 | 17 |
| 17 | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 17 | 17 | 16 | 16 | 16 | 16 | 16 | 17 |
| 17 | 18 | 18 | 19 | 18 | 18 | 17 | 17 | 17 | 17 | 17 | 17 | 17 | 18 | 18 | 19 |
| 19 | 19 | 19 | 19 | 19 | 18 | 18 | 17 | 17 | 17 | 17 | 17 | 17 | 17 | 18 | 18 |
| 19 | 20 | 19 | 18 | 18 | 18 | 17 | 17 | 17 | 17 | 18 | 18 | 19 | 19 | 20 | 20 |
| 20 | 20 | 20 | 19 | 19 | 18 | 18 | 17 | 17 | 17 | 17 | 18 | 18 | 18 | 19 | 20 |
| 21 | 20 | 19 | 19 | 18 | 18 | 18 | 18 | 18 | 18 | 19 | 19 | 20 | 20 | 21 | 21 |
| 21 | 21 | 20 | 20 | 19 | 19 | 18 | 18 | 18 | 18 | 18 | 18 | 19 | 19 | 20 | 21 |
| 21 | 20 | 19 | 19 | 19 | 19 | 18 | 19 | 19 | 19 | 20 | 20 | 21 | 22 | 22 | 22 |
| 22 | 22 | 21 | 20 | 20 | 19 | 19 | 19 | 18 | 19 | 19 | 19 | 19 | 20 | 21 | 21 |
| 20 | 20 | 19 | 19 | 19 | 19 | 19 | 20 | 20 | 21 | 22 | 22 | 23 | 23 | 23 | 23 |
| 22 | 22 | 21 | 20 | 20 | 19 | 19 | 19 | 19 | 19 | 20 | 20 | 21 | 21 | 21 | 20 |
| 20 | 20 | 20 | 20 | 20 | 21 | 22 | 22 | 23 | 24 | 24 | 24 | 24 | 24 | 23 | 22 |
| 22 | 21 | 20 | 20 | 20 | 20 | 20 | 20 | 21 | 21 | 22 | 21 | 21 | 21 | 21 | 21 |
| 21 | 22 | 22 | 23 | 24 | 25 | 25 | 26 | 26 | 25 | 25 | 24 | 23 | 22 | 22 | 21 |
| 21 | 21 | 21 | 21 | 21 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 23 | 24 | 25 |
| 26 | 26 | 27 | 27 | 27 | 26 | 26 | 25 | 24 | 23 | 22 | 22 | 22 | 22 | 22 | 22 |
| 22 | 23 | 23 | 22 | 23 | 23 | 23 | 24 | 25 | 26 | 26 | 27 | 28 | 28 | 28 | 28 |
| 27 | 26 | 26 | 25 | 24 | 23 | 23 | 23 | 22 | 23 | 23 | 24 | 23 | 24 | 24 | 24 |
| 25 | 26 | 26 | 27 | 28 | 29 | 30 | 30 | 30 | 29 | 28 | 27 | 26 | 26 | 25 | 24 |
| 24 | 24 | 23 | 24 | 25 | 25 | 25 | 25 | 26 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
| 32 | 31 | 30 | 29 | 28 | 27 | 26 | 26 | 25 | 25 | 25 | 25 | 26 | 26 | 26 | 27 |
| 27 | 28 | 29 | 31 | 32 | 33 | 33 | 33 | 33 | 33 | 32 | 31 | 29 | 28 | 27 | 27 |
| 26 | 26 | 26 | 27 | 27 | 28 | 28 | 29 | 30 | 32 | 33 | 34 | 35 | 35 | 35 | 35 |
| 34 | 33 | 32 | 30 | 29 | 28 | 28 | 27 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 |
| 35 | 36 | 37 | 37 | 37 | 36 | 35 | 34 | 33 | 32 | 31 | 30 | 29 | 28 | 30 | 31 |
| 32 | 33 | 34 | 35 | 37 | 38 | 39 | 39 | 39 | 39 | 38 | 37 | 35 | 34 | 33 | 32 |
| 31 | 30 | 32 | 33 | 34 | 35 | 37 | 38 | 40 | 41 | 41 | 42 | 41 | 41 | 40 | 38 |
| 37 | 35 | 34 | 33 | 32 | 34 | 35 | 37 | 38 | 40 | 41 | 43 | 44 | 44 | 44 | 44 |
| 43 | 41 | 40 | 38 | 37 | 35 | 34 | 37 | 38 | 40 | 41 | 43 | 44 | 46 | 47 | 47 |
| 47 | 46 | 44 | 43 | 41 | 40 | 38 | 37 | 40 | 41 | 43 | 45 | 46 | 48 | 49 | 50 |
| 50 | 49 | 48 | 46 | 45 | 43 | 41 | 40 | 43 | 45 | 47 | 48 | 50 | 51 | 52 | 53 |
| 52 | 51 | 50 | 48 | 47 | 45 | 43 | 46 | 48 | 50 | 52 | 54 | 55 | 56 | 56 | 55 |
| 54 | 52 | 50 | 48 | 46 | 50 | 53 | 55 | 57 | 58 | 59 | 59 | 59 | 58 | 57 | 55 |
| 53 | 50 | 55 | 57 | 59 | 61 | 62 | 63 | 63 | 62 | 61 | 59 | 57 | 55 | 59 | 62 |
| 64 | 66 | 67 | 67 | 67 | 66 | 64 | 62 | 59 | 65 | 67 | 69 | 71 | 71 | 71 | 71 |
| 69 | 67 | 65 | 70 | 72 | 74 | 76 | 76 | 76 | 74 | 72 | 70 | 76 | 78 | 80 | 81 |
| 81 | 80 | 78 | 76 | 82 | 84 | 86 | 86 | 86 | 84 | 82 | 89 | 91 | 92 | 92 | 91 |
| 89 | 96 | 97 | 98 | 97 | 96 | 103 | 104 | 104 | 103 | 111 | 112 | 111 | 119 | 119 | 127 |

Table 7‑6 – Specification of default values of ScalingList[ 3 ][ 1 ][ 0 ][ i ] with i=0..1023

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 |
| 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 |
| 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 |
| 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 |
| 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 |
| 13 | 13 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 |
| 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 |
| 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 15 | 15 | 15 |
| 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 |
| 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 |
| 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 16 | 16 | 16 | 16 |
| 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |
| 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |
| 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 17 |
| 17 | 17 | 17 | 17 | 17 | 17 | 17 | 17 | 17 | 17 | 17 | 17 | 17 | 17 | 17 | 17 |
| 17 | 17 | 17 | 17 | 17 | 17 | 17 | 17 | 17 | 17 | 18 | 18 | 18 | 18 | 18 | 18 |
| 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 |
| 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 |
| 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 | 18 |
| 18 | 18 | 18 | 19 | 19 | 19 | 19 | 19 | 19 | 19 | 19 | 19 | 19 | 19 | 19 | 19 |
| 19 | 19 | 19 | 19 | 19 | 19 | 19 | 19 | 19 | 19 | 19 | 19 | 19 | 19 | 19 | 19 |
| 19 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 |
| 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 |
| 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 |
| 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 |
| 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 |
| 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 |
| 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 |
| 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 |
| 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 |
| 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 22 | 22 | 22 | 22 | 22 | 22 |
| 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 |
| 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 |
| 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 |
| 22 | 23 | 23 | 23 | 23 | 23 | 23 | 23 | 23 | 23 | 23 | 23 | 23 | 23 | 23 | 23 |
| 23 | 23 | 23 | 23 | 23 | 23 | 23 | 23 | 23 | 23 | 23 | 24 | 24 | 24 | 24 | 24 |
| 24 | 24 | 24 | 24 | 24 | 24 | 24 | 24 | 24 | 24 | 24 | 24 | 24 | 24 | 24 | 24 |
| 24 | 24 | 24 | 24 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 |
| 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 26 | 26 | 26 | 26 |
| 26 | 26 | 26 | 26 | 26 | 26 | 26 | 26 | 26 | 26 | 26 | 26 | 26 | 26 | 26 | 26 |
| 26 | 26 | 26 | 27 | 27 | 27 | 27 | 27 | 27 | 27 | 27 | 27 | 27 | 27 | 27 | 27 |
| 27 | 27 | 27 | 27 | 27 | 27 | 27 | 27 | 27 | 28 | 28 | 28 | 28 | 28 | 28 | 28 |
| 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 29 | 29 |
| 29 | 29 | 29 | 29 | 29 | 29 | 29 | 29 | 29 | 29 | 29 | 29 | 29 | 29 | 29 | 29 |
| 29 | 29 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 |
| 30 | 30 | 30 | 30 | 30 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 |
| 31 | 31 | 31 | 31 | 31 | 31 | 31 | 33 | 33 | 33 | 33 | 33 | 33 | 33 | 33 | 33 |
| 33 | 33 | 33 | 33 | 33 | 33 | 33 | 33 | 35 | 35 | 35 | 35 | 35 | 35 | 35 | 35 |
| 35 | 35 | 35 | 35 | 35 | 35 | 35 | 35 | 37 | 37 | 37 | 37 | 37 | 37 | 37 | 37 |
| 37 | 37 | 37 | 37 | 37 | 37 | 37 | 39 | 39 | 39 | 39 | 39 | 39 | 39 | 39 | 39 |
| 39 | 39 | 39 | 39 | 39 | 42 | 42 | 42 | 42 | 42 | 42 | 42 | 42 | 42 | 42 | 42 |
| 42 | 42 | 46 | 46 | 46 | 46 | 46 | 46 | 46 | 46 | 46 | 46 | 46 | 46 | 49 | 49 |
| 49 | 49 | 49 | 49 | 49 | 49 | 49 | 49 | 49 | 53 | 53 | 53 | 53 | 53 | 53 | 53 |
| 53 | 53 | 53 | 57 | 57 | 57 | 57 | 57 | 57 | 57 | 57 | 57 | 61 | 61 | 61 | 61 |
| 61 | 61 | 61 | 61 | 65 | 65 | 65 | 65 | 65 | 65 | 65 | 70 | 70 | 70 | 70 | 70 |
| 70 | 75 | 75 | 75 | 75 | 75 | 80 | 80 | 80 | 80 | 86 | 86 | 86 | 92 | 92 | 98 |

**seq\_scaling\_list\_pred\_mode** equal to 0 specifies that the value of the scaling list is the same as the value of the scaling list already sent. The scaling list to copy the value is specified by seq\_scaling\_list\_pred\_matrix\_id\_delta. seq\_scaling\_list\_pred\_mode equal to 1 speficies that the value of the scaling list is encoded by DPCM and exp-Golomb code.

**seq\_scaling\_list\_pred\_matrix\_id\_delta** specifies the target reference matrix to copy the value of scaling list. When seq\_scaling\_list\_pred\_mode is equal to 0, seq\_scaling\_list\_pred\_matrix\_id\_delta specifies which matrix should be used in the current matrix by the following:

RefMatrixID = MatrixID – ( 1 + seq\_scaling\_list\_pred\_matrix\_id\_delta ) (7‑7)

where MatrixID is specified in Table 7‑2 and Table 7‑3.

Table 7‑2 – Specification of SizeID

|  |  |
| --- | --- |
| **Size of quantization matrix** | **SizeID** |
| 4x4 | 0 |
| 8x8 (16x4, 4x16) | 1 |
| 16x16 (32x8, 8x32) | 2 |
| 32x32 | 3 |

Table 7‑3 – Specification of MatrixID according to SizeID, prediction type and color component

|  |  |  |  |
| --- | --- | --- | --- |
| **SizeID** | **Prediction type** | **Color component** | **MatrixID** |
| 0, 1, 2 | Intra | Y | 0 |
| 0, 1, 2 | Intra | Cb | 1 |
| 0, 1, 2 | Intra | Cr | 2 |
| 0, 1, 2 | Inter | Y | 3 |
| 0, 1, 2 | Inter | Cb | 4 |
| 0, 1, 2 | Inter | Cr | 5 |
| 3 | Intra | Y | 0 |
| 3 | Inter | Y | 1 |

#### Scaling list semantics

**scaling\_list\_delta\_coef** specifies the difference of the matrix coefficient from the previous matrix, when pred\_mode is equal to 1.

The three-dimensional array ScalingList[ SizeID ][ MatrixID ][ trafoType ] specifies the array of values of scaling list according to the variables SizeID specified in Table 7‑2, MatrixID specified in Table 7‑3 and trafoType specified as follows:

* If the width of the quantization matrix is equal to the height of the quantization matrix, trafoType is set equal to 0,
* Otherwise, the width of the quantization matrix is greater than the height of the quantization matrix, trafoType is set equal to 1,
* Otherwise, trafoType is set equal to 2.

The elements of the quantization matrix of size 16x4, ScalingList[ 1 ][ MatrixID ][ 1 ][ ] is derived as follows:

ScalingList[ 1 ][ MatrixID ][ 1 ][ j\*16+i ] = ScalingList[ 2 ][ MatrixID ][ 0 ][ (j\*4)\*16+i ] (7‑7)  
with i=0..16, j=0..4 and MatrixID=0..5

The elements of the quantization matrix of size 4x16, ScalingList[ 1 ][ MatrixID ][ 2 ][ ] is derived as follows:

ScalingList[ 1 ][ MatrixID ][ 2 ][ j\*4+i ] = ScalingList[ 2 ][ MatrixID ][ 0 ][ j\*16+i\*4 ] (7‑7)  
with i=0..4, j=0..16 and MatrixID=0..5

The elements of the quantization matrix of size 32x8, ScalingList[ 2 ][ MatrixID ][ 1 ][ ] is derived as follows:

ScalingList[ 2 ][ MatrixID ][ 1 ][ j\*32+i ] = ScalingList[ 3 ][ MatrixID ][ 0 ][ (j\*4)\*32+i ] (7‑7)  
with i=0..32, j=0..8 and MatrixID=0..1

The elements of the quantization matrix of size 8x32, ScalingList[ 2 ][ MatrixID ][ 2 ][ ] is derived as follows:

ScalingList[ 2 ][ MatrixID ][ 2 ][ j\*8+i ] = ScalingList[ 3 ][ MatrixID ][ 0 ][ j\*32+i\*4 ] (7‑7)  
with i=0..8, j=0..32 and MatrixID=0..1

#### Adaptation parameter set RBSP semantics

**aps\_id** identifies the adaptation parameter set that is referred to in the slice header. The value of aps\_id shall be in the range of 0 to TBD, inclusive.

[Ed. (BB): Proponent suggests that this could be made level/profile dependend. ]

**aps\_scaling\_list\_data\_present\_flag** equal to 1 specifies that the scaling list parameters exist in this APS, equal to 0 specifies that scaling list parameters do not exist in this APS.

**aps\_sample\_adaptive\_offset\_flag** equal to 1 specifies that the SAO is on for slices referred to the current APS; equal to 0 specifies that the SAO is off for slices referred to the current APS. If there is no active APS, the aps\_sample\_adaptive\_offset\_flag value is inferred to be 0.

**aps\_adaptive\_loop\_filter\_flag** equal to 1 specifies that the ALF is on for slices referred to the current APS; equal to 0 specifies that the ALF is off for slices referred to the current APS. If there is no active APS, the aps\_adaptive\_loop\_filter\_flag value is inferred to be 0.

**aps\_extension\_flag** equal to 0 specifies that no aps\_extension\_data\_flag syntax elements are present in the picture parameter set RBSP syntax structure. aps\_extension\_flag shall be equal to 0 in bitstreams conforming to this Recommendation | International Standard. The value of 1 for aps\_extension\_flag is reserved for future use by ITU‑T | ISO/IEC. Decoders shall ignore all data that follow the value 1 for aps\_extension\_flag in a picture parameter set NAL unit.

**aps\_extension\_data\_flag** may have any value. It shall not affect the conformance to profiles specified in this Recommendation | International Standard.

#### Supplemental enhancement information RBSP semantics

Supplemental Enhancement Information (SEI) contains information that is not necessary to decode the samples of coded pictures from VCL NAL units.

##### Supplemental enhancement information message semantics

An SEI RBSP contains one or more SEI messages. Each SEI message consists of the variables specifying the type payloadType and size payloadSize of the SEI payload. SEI payloads are specified in Annex D. The derived SEI payload size payloadSize is specified in bytes and shall be equal to the number of RBSP bytes in the SEI payload.

NOTE – The NAL unit byte sequence containing the SEI message might include one or more emulation prevention bytes (represented by emulation\_prevention\_three\_byte syntax elements). Since the payload size of an SEI message is specified in RBSP bytes, the quantity of emulation prevention bytes is not included in the size payloadSize of an SEI payload.

**ff\_byte** is a byte equal to 0xFF identifying a need for a longer representation of the syntax structure that it is used within.

**last\_payload\_type\_byte** is the last byte of the payload type of an SEI message.

**last\_payload\_size\_byte** is the last byte of the payload size of an SEI message.

#### Access unit delimiter RBSP semantics

The access unit delimiter may be used to indicate the type of slices present in a primary coded picture and to simplify the detection of the boundary between access units. There is no normative decoding process associated with the access unit delimiter.

**primary\_pic\_type** indicates that the slice\_type values for all slices of the primary coded picture are members of the set listed in Table 7‑2 for the given value of primary\_pic\_type.

Table 7‑2 – Meaning of primary\_pic\_type

|  |  |
| --- | --- |
| **primary\_pic\_type** | **slice\_type values that may be present in the primary coded picture** |
| 0 | 2 |
| 1 | 0, 2 |
| 2 | 0, 1, 2 |

#### Filler data RBSP semantics

The filler data RBSP contains bytes whose value shall be equal to 0xFF. No normative decoding process is specified for a filler data RBSP.

**ff\_byte** is a byte equal to 0xFF.

#### Slice layer RBSP semantics

The slice layer RBSP consists of a slice header and slice data.

[Ed. (TW): insert text]

#### RBSP slice trailing bits semantics

**cabac\_zero\_word** is a byte-aligned sequence of two bytes equal to 0x0000.

Let NumBytesInVclNALunits be the sum of the values of NumBytesInNALunit for all VCL NAL units of a coded picture.

Let BinCountsInNALunits be the number of times that the parsing process function DecodeBin( ), specified in subclause 9.2.3.2, is invoked to decode the contents of all VCL NAL units of a coded picture. BinCountsInNALunits shall not exceed ( 32 ÷ 3 ) \* NumBytesInVclNALunits + ( RawMbBits \* PicSizeInMbs ) ÷ 32.

NOTE – The constraint on the maximum number of bins resulting from decoding the contents of the slice layer NAL units can be met by inserting a number of cabac\_zero\_word syntax elements to increase the value of NumBytesInVclNALunits. Each cabac\_zero\_word is represented in a NAL unit by the three-byte sequence 0x000003 (as a result of the constraints on NAL unit contents that result in requiring inclusion of an emulation\_prevention\_three\_byte for each cabac\_zero\_word).

#### RBSP trailing bits semantics

**rbsp\_stop\_one\_bit** shall be equal to 1.

**rbsp\_alignment\_zero\_bit** shall be equal to 0.

#### RBSP byte alignment semantics

**bit\_equal\_to\_one** specifies a bit equal to 1.

### Slice header semantics

When present, the value of the slice header syntax elements pic\_parameter\_set\_id, idr\_pic\_id, no\_output\_of\_prior\_pics\_flag, pic\_order\_cnt\_lsb, short\_term\_ref\_pic\_set\_pps\_flag, short\_term\_ref\_pic\_set\_idx and num\_long\_term\_pics shall be the same in all slice headers of a coded picture. When present, the value of the slice header syntax elements delta\_poc\_lsb\_lt\_minus1[ i ] and used\_by\_curr\_pic\_lt\_flag[ i ] shall be the same in all slice headers of a coded picture for each i in the range of 0 to num\_long\_term\_pics, inclusive.

**first\_slice\_in\_pic\_flag** indicates whether the slice is the first slice of the picture. If first\_slice\_in\_pic\_flag is equal to 1, the variables SliceAddress and LCUAddress are both set to 0 and the decoding starts with the first LCU in the picture.

**slice\_address** specifies the address in slice granularity resolution in which the slice starts and shall be represented by ( Ceil( Log2( NumLCUsInPicture ) ) + SliceGranularity ) bits in the bitstream where NumLCUsInPicture is the number of LCUs in a picture.

The variable LCUAddress is set to ( slice\_address >> SliceGranularity ) and represents the LCU part of the slice address in raster scan order. The variable GranularityAddress is set to ( slice\_address - ( LCUAddress << SliceGranularity ) ) and represents the sub-LCU part of the slice address expressed in z-scan order.

[Ed. (BB): “z-scan order” is not defined anywhere and should be defined, or the phrase herein is changed not to use the term.]

The variable SliceAddress is then set to ( LCUAddress << ( log2\_diff\_max\_min\_coding\_block\_size << 1 ) ) + ( GranularityAddress << ( ( log2\_diff\_max\_min\_coding\_block\_size << 1 ) – SliceGranularity ) and the slice decoding starts with the largest coding unit possible at the slice starting coordinate.

**slice\_type** specifies the coding type of the slice according to Table 7‑3.

Table 7‑3 – Name association to slice\_type

|  |  |
| --- | --- |
| slice\_type | Name of slice\_type |
| 0 | P (P slice) |
| 1 | B (B slice) |
| 2 | I (I slice) |

When nal\_unit\_type is equal to 5 (IDR picture), slice\_type shall be equal to 2.

When max\_num\_ref\_frames is equal to 0, slice\_type shall be equal to 2.

**entropy\_slice\_flag** equal to 1 specifies that the value of slice header syntax elements not present shall be inferred to be equal to the value of slice header syntax elements in a proceeding slice, where a proceeding slice is defined as the slice containing treeblock with location (LCUAddress – 1). entropy\_slice\_flag shall be equal to 0 when LCUAddress equal to 0.

**pic\_parameter\_set\_id** specifies the picture parameter set in use. The value of pic\_parameter\_set\_id shall be in the range of 0 to 255, inclusive.

**idr\_pic\_id** identifies an IDR picture. The values of idr\_pic\_id in all the slices of an IDR picture shall remain unchanged. When two consecutive access units in decoding order are both IDR access units, the value of idr\_pic\_id in the slices of the first such IDR access unit shall differ from the idr\_pic\_id in the second such IDR access unit. The value of idr\_pic\_id shall be in the range of 0 to 65535, inclusive.

**no\_output\_of\_prior\_pics\_flag** specifies how the previously-decoded pictures in the decoded picture buffer are treated after decoding of an IDR picture. See Annex C. When the IDR picture is the first IDR picture in the bitstream, the value of no\_output\_of\_prior\_pics\_flag has no effect on the decoding process. When the IDR picture is not the first IDR picture in the bitstream and the value of pic\_width\_in\_luma\_samples or pic\_height\_in\_luma\_samples or max\_dec\_frame\_buffering derived from the active sequence parameter set is different from the value of pic\_width\_in\_luma\_samples or pic\_height\_in\_luma\_samples or max\_dec\_frame\_buffering derived from the sequence parameter set active for the preceding picture, no\_output\_of\_prior\_pics\_flag equal to 1 may (but should not) be inferred by the decoder, regardless of the actual value of no\_output\_of\_prior\_pics\_flag.

**pic\_order\_cnt\_lsb** specifies the picture order count modulo MaxPicOrderCntLsb for the current picture. The length of the pic\_order\_cnt\_lsb syntax element is log2\_max\_pic\_order\_cnt\_lsb\_minus4 + 4 bits. The value of the pic\_order\_cnt\_lsb shall be in the range of 0 to MaxPicOrderCntLsb − 1, inclusive. When pic\_order\_cnt\_lsb is not present, pic\_order\_cnt\_lsb shall be inferred to be equal to 0.

**short\_term\_ref\_pic\_set\_pps\_flag** equal to 1 specifies that the short-term reference picture set of the current picture shall be created using syntax elements in the active picture parameter set. short\_term\_ref\_pic\_set\_pps\_flag equal to 0 specifies that the short-term reference picture set of the current picture shall be created using syntax elements in the short\_term\_ref\_pic\_set( ) syntax structure in the slice header.

**short\_term\_ref\_pic\_set\_idx** specifies the index to the list of the short-term reference picture sets specified in the active picture parameter set that shall be used for creation of the reference picture set of the current picture. The syntax element short\_term\_ref\_pic\_set\_idx shall be represented by ceil( log2( num\_short\_term\_ref\_pic\_sets ) ) bits. The value of short\_term\_ref\_pic\_set\_idx shall be in the range of 0 to num\_short\_term\_ref\_pic\_sets – 1, inclusive, where num\_short\_term\_ref\_pic\_sets is the syntax element from the active picture parameter set.

The variable StRpsIdx is derived as follows.

if( short\_term\_ref\_pic\_set\_pps\_flag )  
 StRpsIdx = short\_term\_ref\_pic\_set\_idx (7‑9)  
else  
 StRpsIdx = num\_short\_term\_ref\_pic\_sets

**num\_long\_term\_pics** specifies the number of the long-term reference pictures that are to be included in the long-term reference picture set of the current picture. The value of num\_long\_term\_pics shall be in the range of 0 to max\_num\_ref\_frames – NumNegativePics[ StRpsIdx ] – NumPositivePics[ StRpsIdx ] , inclusive. When not present, the value of num\_long\_term\_pics shall be inferred to be equal to 0.

**delta\_poc\_lsb\_lt\_minus1**[ i ] is used to determine the value of the least significant bits of the picture order count value of the i-th long-term reference picture that is included in the long-term reference picture set of the current picture. delta\_poc\_lsb\_lt\_minus1[ i ] shall be in the range of 0 to MaxPicOrderCntLsb − 1, inclusive.

The variable DeltaPocLt[ i ] is derived as follows.

if (i = = 0)   
 DeltaPocLt[ i ] = delta\_poc\_lsb\_lt\_minus1[ i ] + 1 (7‑9)  
else  
 DeltaPocLt[ i ] = delta\_poc\_lsb\_lt\_minus1[ i ] + 1 + DeltaPocLt[ i – 1 ]

The value of DeltaPocLt[ i ] shall be in the range of 0 to MaxPicOrderCntLsb, inclusive.

**used\_by\_curr\_pic\_lt\_flag**[ i ] equal to 0 specifies that the i-th long-term reference picture included in the long-term reference picture set of the current picture is not used for reference by the current picture.

**num\_ref\_idx\_active\_override\_flag** equal to 1 specifies that the syntax element num\_ref\_idx\_l0\_active\_minus1 is present for P and B slices and that the syntax element num\_ref\_idx\_l1\_active\_minus1 is present for B slices. num\_ref\_idx\_active\_override\_flag equal to 0 specifies that the syntax elements num\_ref\_idx\_l0\_active\_minus1 and num\_ref\_idx\_l1\_active\_minus1 are not present.

When the current slice is a P or B slice and field\_pic\_flag is equal to 0 and the value of num\_ref\_idx\_l0\_default\_active\_minus1 in the picture parameter set exceeds 15, num\_ref\_idx\_active\_override\_flag shall be equal to 1.

When the current slice is a B slice and field\_pic\_flag is equal to 0 and the value of num\_ref\_idx\_l1\_default\_active\_minus1 in the picture parameter set exceeds 15, num\_ref\_idx\_active\_override\_flag shall be equal to 1.

**num\_ref\_idx\_l0\_active\_minus1** specifies the maximum reference index for reference picture list 0 that shall be used to decode the slice.

When the current slice is a P or B slice and num\_ref\_idx\_l0\_active\_minus1 is not present, num\_ref\_idx\_l0\_active\_minus1 shall be inferred to be equal to num\_ref\_idx\_l0\_default\_active\_minus1.

The range of num\_ref\_idx\_l0\_active\_minus1 is specified as follows.p

– If field\_pic\_flag is equal to 0, num\_ref\_idx\_l0\_active\_minus1 shall be in the range of 0 to 15, inclusive. When MbaffFrameFlag is equal to 1, num\_ref\_idx\_l0\_active\_minus1 is the maximum index value for the decoding of frame macroblocks and 2 \* num\_ref\_idx\_l0\_active\_minus1 + 1 is the maximum index value for the decoding of field macroblocks.

– Otherwise (field\_pic\_flag is equal to 1), num\_ref\_idx\_l0\_active\_minus1 shall be in the range of 0 to 31, inclusive.

**num\_ref\_idx\_l1\_active\_minus1** specifies the maximum reference index for reference picture list 1 that shall be used to decode the slice.

When the current slice is a B slice and num\_ref\_idx\_l1\_active\_minus1 is not present, num\_ref\_idx\_l1\_active\_minus1 shall be inferred to be equal to num\_ref\_idx\_l1\_default\_active\_minus1.

The range of num\_ref\_idx\_l1\_active\_minus1 is constrained as specified in the semantics for num\_ref\_idx\_l0\_active\_minus1 with l0 and list 0 replaced by l1 and list 1, respectively.

**cabac\_init\_idc** specifies the index for determining the initialisation table used in the initialisation process for context variables. The value of cabac\_init\_idc shall be in the range of 0 to 2, inclusive.

**slice\_qp\_delta** specifies the initial value of QPY to be used for all the macroblocks in the slice until modified by the value of cu\_qp\_delta in the coding unit layer. The initial QPY quantisation parameter for the slice is computed as

SliceQPY = 26 + pic\_init\_qp\_minus26 + slice\_qp\_delta (7‑9)

The value of slice\_qp\_delta shall be limited such that SliceQPY is in the range of −QpBdOffsetY to +51, inclusive.

**disable\_deblocking\_filter\_flag** equal to 1 specifies that the operation of the deblocking filter shall be disabled and disable\_deblocking\_filter\_flag equal to 0 specifies that the operation of the deblocking filter shall be enabled.

**collocated\_from\_l0\_flag** equal to 1 specifies the picture that contains the co-located partition shall be derived from list 0, otherwise the picture shall be derived from list 1.

**5\_minus\_max\_num\_merge\_cand** specifies the maximum number of merging MVP candidates supported in the slice subtracted from 5. The maximum number of merging MVP candidates, MaxNumMergeCand is computed as

MaxNumMergeCand = 5 − 5\_minus\_max\_num\_merge\_cand (7‑9)

The value of 5\_minus\_max\_num\_merge\_cand shall be limited such that MaxNumMergeCand is in the range of 0 to 5, inclusive.

**substream\_length\_mode** specifies the number of bits to be used to decode syntax element substream\_length[ ], as defined in Table 7‑4.

Table ‑ – Number of bits used to decode substream\_length

|  |  |
| --- | --- |
| **substream\_length\_mode** | **Number of bits** |
| 0 | 8 |
| 1 | 16 |
| 2 | 24 |
| 3 | 32 |

**substream\_length[i]** specifies the number of bits to be used during the initialization process for the current bitstream pointer i.

#### Short-term reference picture set semantics

A short-term reference picture set may be present in a picture parameter set or in a slice header. If a short-term reference picture set is present in a slice header the content of the short-term reference picture set syntax structure shall be the same in all slice headers of a picture and the value of idx shall be equal to the syntax element num\_ref\_pic\_sets from the active picture parameter set.

**inter\_ref\_pic\_set\_prediction\_flag** equal to 1 specifies that the reference picture set of the current picture shall be predicted using another reference picture set in the active picture parameter set.

**delta\_idx\_minus1** plus 1 specifies the difference between the index of the reference picture set of the current picture and the index of the reference picture set used for inter reference picture set prediction.

The variable RIdx shall be derived as follows.

RIdx = idx ‑ (delta\_idx\_minus1+1) (7‑9)

**delta\_rps\_sign** specifies the sign of DiffPicOrderCnt( picA, picB ), where picA is associated with DeltaPoc[ RIdx ] and picB is associated with DeltaPoc[ idx ]. A value of 0 indicates that the sign is positive and a value of 1 indicates that the sign is negative.

**abs\_delta\_rps\_minus1** plus 1specifies the absolute value of DiffPicOrderCnt( picA, picB ), where picA is associated with DeltaPoc[ RIdx ] and picB is associated with DeltaPoc[ idx ].

The variable DeltaRPS is derived as follows.

DeltaRPS = (1 – (delta\_rps\_sign<<1)) \* (abs\_delta\_rps\_minus1 + 1) (7‑9)

**ref\_idc0[ j ]** and **ref\_idc1[ j ]** together specifies if DeltaPoc[ RIdx ][ j ] is used for prediction of DeltaPoc[ idx ][ i ]and if the i-th reference picture is used for reference by the current picture according to Table 7-5.

Table ‑ – ref\_idc0[ j ] and ref\_idc0[ j ] operations for the reference picture set

|  |  |  |
| --- | --- | --- |
| **ref\_idc0[ j ]** | **ref\_idc1[ j ]** | **Properties of i-th reference picture** |
| 1 |  | Picture is used for reference by the current picture DeltaPoc[ idx ][ i ] = DeltaPoc[ RIdx ][ j ] + DeltaRPS |
| 0 | 1 | Picture is not used for reference by the current picture (but used by future pictures) DeltaPoc[ idx ][ i ] = DeltaPoc[ RIdx ][ j ] + DeltaRPS |
| 0 | 0 | There is no corresponding i-th reference picture for DeltaPoc[ RIdx ][ j ]. |

When inter\_ref\_pic\_set\_prediction\_flagis equal to 1, the variables DeltaPocS0[ idx ][ i0 ], UsedByCurrPicS0[ idx ][ i0 ], DeltaPocS1[ idx ][ i1 ], UsedByCurrPicS1[ idx ][ i1], NumNegativePics[ idx ] and NumPositivePics[ idx ] are derived as follows.

for( j = 0, i0 = 0, i1 = 0; j <= NumDeltaPocs[ RIdx ]; j++ ) {  
 if( ref\_idc[ j ] = = “1” || ref\_idc[ j ] = = “01” ) {  
 DPoc = DeltaPoc[ RIdx ][ j ] + DeltaRPS  
 if( DPoc < 0 ) {  
 DeltaPocS0[ idx ][ i0 ] = DPoc (7‑9)  
 UsedByCurrPicS0[ idx ][ i0 ] = (ref\_idc[ j ] = = “1”) ? 1 : 0 (7‑9)  
 i0++  
 }  
 else {  
 DeltaPocS1[ idx ][ i1 ] = DPoc (7‑9)  
 UsedByCurrPicS1[ idx ][ i1 ] = (ref\_idc[ j ] = = “1”) ? 1 : 0 (7‑9)  
 i1++  
 }   
 }  
}

NumNegativePics[ idx ] = i0 (7‑9)

NumPositivePics[ idx ] = i1 (7‑9)

When DeltaPoc[ RIdx ][ j ] is unavaialable, it is set to 0.

DeltaPocS0[ idx ][ i0 ] and DeltaPocS1[ idx ][ i1 ] shall be sorted in decreasing order and in increasing order, respectively.

**num\_negative\_pics** specifies the number of the following delta\_poc\_s0\_minus1[ i ] and used\_by\_curr\_pic\_s0\_flag[ i ] syntax elements. The value of num\_negative\_pics shall be in the range of 0 to max\_num\_ref\_frames, inclusive.

The variable NumNegativePics[ idx ] is derived as follows.

NumNegativePics[ idx ] = num\_negative\_pics (7‑9)

**num\_positive\_pics** specifies the number of the following delta\_poc\_s1\_minus1[ i ] and used\_by\_curr\_pic\_s1\_flag1[ i ] syntax elements. The value of num\_positive\_pics shall be in the range of 0 to max\_num\_ref\_frames – num\_negative\_pics, inclusive.

The variable NumPositivePics[ idx ] is derived as follows.

NumPositivePics[ idx ] = num\_positive\_pics (7‑9)

**delta\_poc\_s0\_minus1**[ i ] plus 1 specifies an absolute difference between two picture order count values. The value of delta\_poc\_s0\_minus1[ i ] shall be in the range of 0 to 215 – 1, inclusive.

The variable DeltaPocS0[ idx ][ i ] is derived as follows.

if (i = = 0)   
 DeltaPocS0[ idx ][ i ] = – ( delta\_poc\_s0\_minus1[ i ] + 1 ) (7‑9)  
else  
 DeltaPocS0[ idx ][ i ] = DeltaPocS0[ idx ][ i – 1 ] – ( delta\_poc\_s0\_minus1[ i ] + 1 )

**used\_by\_curr\_pic\_s0\_flag**[ i ] equal to 0 specifies that the i-th reference picture that has picture order count less than that of the current picture is not used for reference by the current picture.

The variable UsedByCurrPicS0[ idx ][ i ] is derived as follows.

UsedByCurrPicS0[ idx ][ i ] = used\_by\_curr\_pic\_s0\_flag[ i ] (7‑9)

Let CurrPicOrderCntVal be the picture order count value of a current picture with RpsIdx equal to idx; If the reference picture with PicOrderCntVal equal to CurrPicOrderCntVal + DeltaPocS0[ idx ][ i ] has higher temporal\_id than the current picture, UsedByCurrPicS0[ idx ][ i ] must be equal to 0.

**delta\_poc\_s1\_minus1**[ i ] plus 1 specifies an absolute difference between two picture order count values. The value of delta\_poc\_s1\_minus1[ i ] shall be in the range of 0 to 215 – 1, inclusive.

The variable DeltaPocS1[ idx ][ i ] is derived as follows.

if (i = = 0)   
 DeltaPocS1[ idx ][ i ] = delta\_poc\_s1\_minus1[ i ] + 1 (7‑9)  
else  
 DeltaPocS1[ idx ][ i ] = DeltaPocS1[ idx ][ i – 1 ] + ( delta\_poc\_s1\_minus1[ i ] + 1 )

**used\_by\_curr\_pic\_s1\_flag**[ i ] equal to 0 specifies that the i-th reference picture that has picture order count greater than that of the current picture is not used for reference by the current picture.

The variable UsedByCurrPicS1[ idx ][ i ] is derived as follows.

UsedByCurrPicS1[ idx ][ i ] = used\_by\_curr\_pic\_s1\_flag[ i ] (7‑9)

Let CurrPicOrderCntVal be the picture order count value of a current picture with RpsIdx equal to idx; If the reference picture with PicOrderCntVal equal to CurrPicOrderCntVal + DeltaPocS1[ idx ][ i ] has higher temporal\_id than the current picture, UsedByCurrPicS1[ idx ][ i ] must be equal to 0.

The variables DeltaPoc[ idx ][ i ] and NumDeltaPocs[ idx ] are derived as follows.

for( i = 0; i <= NumNegativePics[ idx ]; i++ )  
 DeltaPoc[ idx ][ i ] = DeltaPocS0[ idx ][ i ] (7‑9)  
for( i = 0; i <= NumPositivePics[ idx ]; i++ )  
 DeltaPoc[ idx ][ i + NumNegativePics[ idx ] ] = DeltaPocS1[ idx ][ i ] (7‑9)

NumDeltaPocs[ idx ] = NumNegativePics[ idx ] + NumPositivePics[ idx ] (7‑9)

#### Reference picture list modification semantics

The syntax elements ref\_pic\_list\_modification\_idc and ref\_pic\_set\_idx specify the change from the initial reference picture lists to the reference picture lists to be used for decoding the slice.

**ref\_pic\_list\_modification\_flag\_l0** equal to 1 specifies that the syntax element ref\_pic\_list\_modification\_idc is present for specifying reference picture list 0. ref\_pic\_list\_modification\_flag\_l0 equal to 0 specifies that this syntax element is not present.

When ref\_pic\_list\_modification\_flag\_l0 is equal to 1, the number of times that ref\_pic\_list\_modification\_idc is not equal to 3 following ref\_pic\_list\_modification\_flag\_l0 shall not exceed num\_ref\_idx\_l0\_active\_minus1 + 1.

**ref\_pic\_list\_modification\_flag\_l1** equal to 1 specifies that the syntax element ref\_pic\_list\_modification\_idc is present for specifying reference picture list 1. ref\_pic\_list\_modification\_flag\_l1 equal to 0 specifies that this syntax element is not present.

When ref\_pic\_list\_modification\_flag\_l1 is equal to 1, the number of times that ref\_pic\_list\_modification\_idc is not equal to 3 following ref\_pic\_list\_modification\_flag\_l1 shall not exceed num\_ref\_idx\_l1\_active\_minus1 + 1.

**ref\_pic\_list\_modification\_idc** together with ref\_pic\_set\_idx specifies which of the reference pictures are re-mapped. The values of ref\_pic\_list\_modification\_idc are specified in Table 7‑4. The value of the first ref\_pic\_list\_modification\_idc that follows immediately after ref\_pic\_list\_modification\_flag\_l0 or ref\_pic\_list\_modification\_flag\_l1 shall not be equal to 3.

Table 7‑6 – ref\_pic\_list\_modification\_idc operations for modification of reference picture lists

|  |  |
| --- | --- |
| **ref\_pic\_list\_modification\_idc** | **modification specified** |
| 0 | For list 0: ref\_pic\_set\_idx is present and corresponds to an index to RefPicSetStCurr0; For list 1: ref\_pic\_set\_idx is present and corresponds to an index to RefPicSetStCurr1 |
| 1 | For list 0: ref\_pic\_set\_idx is present and corresponds to an index to RefPicSetStCurr1; For list 1: ref\_pic\_set\_idx is present and corresponds to an index to RefPicSetStCurr0 |
| 2 | ref\_pic\_set\_idx is present and corresponds to an index to RefPicSetLtCurr |
| 3 | End loop for modification of the initial reference picture list |

**ref\_pic\_set\_idx** specifies the index, to RefPicSetStCurr0, RefPicSetStCurr1 or RefPicSetLtCurr, of the reference picture being moved to the current index in the reference picture list. The value of ref\_pic\_set\_idx shall be in the range of 0 to max\_num\_ref\_frames, inclusive.

**abs\_diff\_pic\_num\_minus1** plus 1 specifies the absolute difference between the picture number of the picture being moved to the current index in the list and the picture number prediction value. abs\_diff\_pic\_num\_minus1 shall be in the range of 0 to MaxPicNum − 1. The allowed values of abs\_diff\_pic\_num\_minus1 are further restricted as specified in subclause 8.2.2.3.1.

[Ed. (TW): clarify the following paragraph]

**long\_term\_pic\_num** specifies the long-term picture number of the picture being moved to the current index in the list. When decoding a coded frame, long\_term\_pic\_num shall be equal to a LongTermPicNum assigned to one of the reference frames or complementary reference field pairs marked as "used for long-term reference". When decoding a coded field, long\_term\_pic\_num shall be equal to a LongTermPicNum assigned to one of the reference fields marked as "used for long-term reference".

#### Reference picture lists combination semantics

**ref\_pic\_list\_combination\_flag** equal to 1 indicates that the reference picture list 0 and the reference picture list 1 are combined to be an additional reference picture lists combination used for the prediction units being uni-directional predicted. This flag equal to 0 indicates that the reference picture list 0 and reference picture list 1 are identical thus reference picture list 0 is used as the reference picture lists combination. The reference picture lists combination is set to be empty at the start of the loop defined in this table.

**num\_ref\_idx lc\_active\_minus1**+1 specifies the number of reference pictures selected from reference picture list 0 or reference picture list 1 in the reference picture lists combination.

**ref\_pic\_list\_modification\_flag\_lc** equal to 1 specifies that the syntax elements pic\_from\_list\_0\_flag and ref\_idx\_list\_curr are present for specifying the mapping for the entries of the reference picture lists combination to the entries of reference picture list 0 and reference picture list 1. ref\_pic\_list\_modification\_flag\_lc equal to 0 specifies that these syntax elements are not present. The reference picture lists combination is initialized as specified in subclause X.X.X.X.

**pic\_from\_list\_0\_flag** indicates the current reference picture added into the reference picture lists combination is from reference picture list 0 or reference picture list 1. When this flag is equal to 1, the picture is from the reference picture list 0, and the CurrRefPicList is reference picture list 0; when this flag is equal to 0, the picture is from the reference picture list 1, and the CurrRefPicList is reference picture list 1.

**ref\_idx\_list\_curr** indicates the reference index of the picture in the reference picture list specified by pic\_from\_list\_0\_flag to be appended at the end of the reference picture lists combination.

#### Sample adaptive offset parameter semantics

**sao\_flag\_cb** equal to 1 denotes sample adaptive offset process for Cb shall be applied to the current picture.

**sao\_flag\_cr** equal to 1 denotes sample adaptive offset process for Cr shall be applied to the current picture.

**sao\_split\_flag**[ cIdx ][ saoDepth ][ rx ][ ry ] specifies whether a region is split into four sub regions with half horizontal and vertical number of LCU for the color component cIdx. The array indices rx and ry specify the region index and saoDepth specifies the split depth of the region. When sao\_split\_flag[ cIdx ][ saoDepth ][ rx ][ ry ] is not present, it shall be inferred to be equal to 0.

The maximum allowed depth for sample adaptive offset process SaoMaxDepth is derived as follows:

SaoMaxDepth = Min( 4, Min( Floor( Log2( PicWidthInLCUs ) ), Floor( Log2( PicHeightInLCUs ) ) ) ) (7‑10)

where

PicWidthInLCUs = Ceil( PicWidthInSamplesL ÷ ( 1 << Log2MaxCUSize ) ) (7‑10)  
PicHeightInLCUs = Ceil( PicHeightInSamplesL ÷ ( 1 << Log2MaxCUSize ) ) (7‑10)

[Ed.: (WJ) PicWidthInLCUs and PicHeightInLCUs could be defined elsewhere]

**sao\_type\_idx**[ cIdx ][ saoDepth ][ rx ][ ry ] indicates the offset type for the color component cIdx of the region specified by saoDepth, rx and ry.

**sao\_offset**[ cIdx ][ saoDepth ][ rx ][ ry ][ i ] indicates the offset value of i-th category for the color component cIdx of the region specified by saoDepth, rx and ry.

The variable bitDepth is derived as follows.

* If cIdx is equal to 0, bitDepth is set equal to BitDepthY..
* Otherwise (cIdx is equal to1 or 2), bitDepth is set equal to BitDepthC.

The offset value shall be in the range of [ -( 1 << ( SaoBitsRange – 1) ), ( 1<< ( SaoBitsRange – 1) ) – 1 ] where

SaoBitRange = Min( bitDepth, 10 ) – 4 (7‑10)

An array SaoOffsetVal is specified as

SaoOffsetVal[ cIdx ][ saoDepth ][ rx ][ ry ][ 0 ] = 0 (7‑10)

SaoOffsetVal[ cIdx ][ saoDepth ][ rx ][ ry ][ i + 1 ] =

sao\_offset[ cIdx ][ saoDepth ][ rx ][ ry ][ i ] << ( bitDepth – Min( bitDepth, 10 ) )

with i = 0..NumSaoCategory – 1 (7‑10)

The number of categories, NumSaoClass, is specified in Table 7‑7.

Table 7‑9 – Specification of NumSaoClass

|  |  |  |
| --- | --- | --- |
| **sao\_type\_idx[ cIdx ][ saoDepth ][ rx ][ ry ]** | **NumSaoCategory** | **Edge type (informative)** |
| 0 | 0 | Not applied |
| 1 | 4 | 1D 0-degree edge |
| 2 | 4 | 1D 90-degree edge |
| 3 | 4 | 1D 135-degree edge |
| 4 | 4 | 1D 45-degree edge |
| 5 | 16 | Central band |
| 6 | 16 | Side band |

#### Adaptive loop filter parameter semantics

**alf\_region\_adaptation\_flag** specifies the filter adaptation method shall be applied for the current slice. If alf\_region\_adaptation\_flag is equal to 1, the region-based filter adaptation shall be applied, otherwise, the block-based directional-activity filter adaptation shall be applied.

**alf\_length\_luma\_minus5\_div2** specifies the filter length in horizontal direction for the luma component alfTap used in the adaptive loop filter process as specified by

alfTap = ( alf\_length\_luma\_minus5\_div2 << 1 ) + 5 (7‑11)

The number of coded luma filter coefficients AlfCodedLengthLuma and the number of luma filter coefficients AlfLengthLuma are derived as follows:

* If alfTap is equal to 9, [Ed.: (WJ) truncated diamond shape]

AlfCodedLengthLuma = ( ( alfTap \* alfTap ) >> 2 ) + 1 (7‑11)

AlfLengthLuma = ( ( alfTap \* alfTap ) >> 1 ) (7‑12)

* Otherwise (alfTap is less than 9), [Ed.: (WJ) diamond shape]

AlfCodedLengthLuma = ( ( alfTap \* alfTap ) >> 2 ) + 2 (7‑13)

AlfLengthLuma = ( ( alfTap \* alfTap ) >> 1 ) + 2 (7‑14)

**alf\_no\_filters\_minus1** plus 1 specifies the number of filter sets for the current slice.

**alf\_start\_second\_filter** specifies the variance index of luma samples where the second filter is applied, when alf\_no\_filters\_minus1 is equal to 1.

**alf\_filter\_pattern**[ i ] specifies the filter index array corresponding to i-th variance index of luma samples, when alf\_no\_filters\_minus1 is greater than 1. The number of filter sets AlfNumFilters is derived as follows:

* If alf\_no\_filters\_minus1 is less than 2,

AlfNumFilters = alf\_no\_filters\_minus1 + 1 (7‑15)

* Otherwise (alf\_no\_filters\_minus1 is greater than 2)

AlfNumFilters = Σi alf\_filter\_pattern[ i ] with i = 0..15 (7‑16)

**alf\_pred\_method** specifies whether the filter coefficients are coded in a predictive way.

**alf\_nb\_pred\_luma**[ i ] specifies whether the filter coefficients of i-th filter is coded in a predictive way from spatially neighboring filter coefficients.

**alf\_coeff\_luma**[ i ][ j ] specifies the j-th filter coefficient of i-th filter for the adaptive loop filtering process for luma samples.

**alf\_chroma\_idc** specifies which chroma components are to be filtered.

Table 7‑10 – Specification of alf\_chroma\_idc

|  |  |
| --- | --- |
| **alf\_chroma\_idc** | **chroma component to be filtered** |
| 0 | None |
| 1 | Cr |
| 2 | Cb |
| 3 | Cb and Cr |

**alf\_length\_chroma\_minus5\_div2** specifies the filter length in horizontal direction for the luma component alfTapC used in the adaptive loop filter process as specified by

alfTapC = ( alf\_length\_chroma\_minus5\_div2 << 1 ) + 5 (7‑17)

The number of coded chroma filter coefficients AlfCodedLengthChroma and the number of chroma filter coefficients AlfLengthChroma are derived as follows:

AlfCodedLengthChroma = ( ( alfTapC \* alfTapC – 1 ) >> 1 ) + 2 (7‑18)

AlfLengthChroma = ( alfTapC \* alfTapC ) + 1 (7‑19)

**alf\_coeff\_chroma**[ i ] specifies the i-th filter coefficient for the adaptive loop filtering process for chroma samples.

#### Adaptive loop filter coding unit control parameter semantics

**alf\_cu\_control\_flag** specifies whether the adaptive loop filter process for luma component shall be applied adaptively according to the coding unit. If alf\_cu\_control\_flag is equal to 1, the filtering process shall be applied only when alf\_cu\_flag is equal to 1, otherwise, the filtering process shall be applied to all luma samples in the current slice.

**alf\_cu\_control\_max\_depth** specifies the maximum split depth from treeblock for transmitting the flag indicating the application of the adaptive loop filter process.

**alf\_length\_cu\_control\_info** specifies the information of the number of alf\_cu\_flag succeeding this information. When the variable specifying the number of alf\_cu\_flag shall be computed as

NumAlfCuFlag = alf\_length\_cu\_control\_info + NumTBsInPicture (7‑20)

**alf\_cu\_flag**[i] specifies the information whether a coding unit is filtered when the adaptive loop filter is applied.

#### Weighted prediction parameters semantics

**luma\_log2\_weight\_denom** is the base 2 logarithm of the denominator for all luma weighting factors. The value of luma\_log2\_weight\_denom shall be in the range of 0 to 7, inclusive.

**delta\_chroma\_log2\_weight\_denom** is the difference of the base 2 logarithm of the denominator for all chroma weighting factors.

The variable ChromaLog2WeightDenom is specified by luma\_log2\_weight\_denom + delta\_chroma\_log2\_weight\_denom and it shall be in the range of 0 to 7, inclusive.

**luma\_weight\_l0\_flag** equal to 1 specifies that weighting factors for the luma component of list 0 prediction are present. luma\_weight\_l0\_flag equal to 0 specifies that these weighting factors are not present.

**delta\_luma\_weight\_l0[** i **]** is the difference of the weighting factor applied to the luma prediction value for list 0 prediction using RefPicList0[ i ].

The variable LumaWeightL0[ i ] is specified by (1 << luma\_log2\_weight\_denom ) + delta\_luma\_weight\_l0[ i ]. When luma\_weight\_l0\_flag is equal to 1, the value of LumaWeightL0[ i ] shall be in the range of –128 to 127, inclusive. When luma\_weight\_l0\_flagis equal to 0, LumaWeightL0[ i ] shall be inferred to be equal to 2luma\_log2\_weight\_denom for RefPicList0[ i ].

**luma\_offset\_l0[** i **]** is the additive offset applied to the luma prediction value for list 0 prediction using RefPicList0[ i ]. The value of luma\_offset\_l0[ i ] shall be in the range of –128 to 127, inclusive. When luma\_weight\_l0\_flagis equal to 0, luma\_offset\_l0[ i ] shall be inferred as equal to 0 for RefPicList0[ i ].

**chroma\_weight\_l0\_flag** equal to 1 specifies that weighting factors for the chroma prediction values of list 0 prediction are present. chroma\_weight\_l0\_flag equal to 0 specifies that these weighting factors are not present.

**delta\_chroma\_weight\_l0[** i **][** j **]** is the difference of the weighting factor applied to the chroma prediction values for list 0 prediction using RefPicList0[ i ] with j equal to 0 for Cb and j equal to 1 for Cr.

The variable ChromaWeightL0[ i ][ j ] is specified by ( 1 << ChromaLog2WeightDenom ) + delta\_chroma\_weight\_l0[ i ][ j ]. When chroma\_weight\_l0\_flag is equal to 1, the value of ChromaWeightL0[ i ][ j ] shall be in the range of –128 to 127, inclusive. When chroma\_weight\_l0\_flag is equal to 0**,** ChromaWeightL0[ i ][ j ] shall be inferred to be equal to 2ChromaLog2WeightDenom for RefPicList0[ i ].

**delta\_chroma\_offset\_l0[** i **][** j **]** is the difference of the additive offset applied to the chroma prediction values for list 0 prediction using RefPicList0[ i ] with j equal to 0 for Cb and j equal to 1 for Cr.

The variable ChromaOffsetL0[ i ][ j ] is specified as follows:

shift = 1 << ( BitDepthC – 1 )

ChromaOffsetL0[ i ][ j ] = (delta\_chroma\_offset\_l0[i][j] –   
 ( (shift\*ChromaWeightL0[ i ][ j ]) >> ChromaLog2WeightDenom ) – shift ) (7‑21)

The variable ChromaOffsetL0[ i ][ j ] shall be in the range of -127 to 128, inclusive. When chroma\_weight\_l0\_flag is equal to 0**,** ChromaOffsetL0[ i ][ j ] shall be inferred to be equal to 0 for RefPicList0[ i ].

**luma\_weight\_l1\_flag, delta\_luma\_weight\_l1**, **luma\_offset\_l1**, **chroma\_weight\_l1\_flag**, **delta\_chroma\_weight\_l1**, **delta\_chroma\_offset\_l1** have the same semantics as luma\_weight\_l0\_flag, delta\_luma\_weight\_l0, luma\_offset\_l0, chroma\_weight\_l0\_flag, delta\_chroma\_weight\_l0, delta\_chroma\_offset\_l0, respectively, with l0, list 0, and List0 replaced by l1, list 1, and List1, respectively.

**luma\_weight\_lc\_flag, delta\_luma\_weight\_lc**, **luma\_offset\_lc**, **chroma\_weight\_lc\_flag**, **delta\_chroma\_weight\_lc**, **delta\_chroma\_offset\_lc** have the same semantics as luma\_weight\_l0\_flag, delta\_luma\_weight\_l0, luma\_offset\_l0, chroma\_weight\_l0\_flag, delta\_chroma\_weight\_l0, delta\_chroma\_offset\_l0, respectively, with l0, list 0, and List0 replaced by lc, list combination, and List combination, respectively. [Ed. (WJ): the use of terms list combination / List combination here may not be suitable]

### Slice data semantics

**end\_of\_slice\_flag** equal to 0 specifies that another macroblock is following in the slice. end\_of\_slice\_flag equal to 1 specifies the end of the slice and that no further macroblock follows.

### Coding tree semantics

**split\_coding\_unit\_flag**[ x0 ][ y0 ] specifies whether a coding unit is split into coding units with half horizontal and vertical size. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

When split\_coding\_unit\_flag[ x0 ][ y0 ] is not present, the following applies:

– If log2CUSize is greater than Log2MinCUSize, the value of split\_coding\_unit\_flag[ x0 ][ y0 ] is inferred to be equal to 1.

– Otherwise (log2CUSize is equal to Log2MinCUSize), the value of split\_coding\_unit\_flag[ x0 ][ y0 ] is inferred to be equal to 0.

### Coding unit semantics

**skip\_flag**[ x0 ][ y0 ] equal to 1 specifies that for the current coding unit, when decoding a P or B slice, no more syntax elements except the motion vector predictor indices are parsed after skip\_flag[ x0 ][ y0 ]. skip\_flag[ x0 ][ y0 ] equal to 0 specifies that the coding unit is not skipped. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

When skip\_flag[ x0 ][ y0 ] is not present, it shall be inferred to be equal to 0.

**pred\_mode\_flag** equal to 0 specifies that the current coding unit is coded in inter prediction mode. pred\_mode\_flag equal to 1 specifies that the current coding unit is coded in intra prediction mode. The variable PredMode is derived as follows.

* If pred\_mode\_flag is equal to 0,
  + PredMode is set to MODE\_INTER.
* Otherwise (pred\_mode\_flag is equal to 1),
  + PredMode is set to MODE\_INTRA.

When pred\_mode\_flag is not present, the variables PredMode is derived as follows.

* If slice\_type is equal to I,
  + PredMode is inferred to be equal to MODE\_INTRA
* Otherwise (slice\_type is equal to P or B), if skip\_flag[ x0 ][ y0 ] is equal to 1,
  + PredMode is inferred to be equal to MODE\_SKIP

**part\_mode** specifies partitioning mode of the current coding unit. The semantics of part\_mode depend on PredMode. The variables PartMode and IntraSplitFlag are derived from the value of part\_mode as defined in Table 7‑10.

The value of part\_mode is restricted as follows.

* If PredMode is equal to MODE\_INTRA, part\_mode shall be equal to 0 or 1.
* Otherwise (PredMode is equal to MODE\_INTER), the following applies
* If log2CUSize is greater than Log2MinCUSize, part\_mode shall be in the range of 0 to 2, inclusive and in the range of 4 to 7, inclusive.
* Otherwise if log2CUSize is equal to 3 and inter\_4x4\_enabled\_flag is equal to 0, ,the value of part\_mode shall be in the range of 0 to 2, inclusive.
* Otherwise (log2CUSize is greater than 3 or inter\_4x4\_enabled\_flag is equal to q), the value of part\_mode shall be in the range of 0 to 3, inclusive.

When part\_mode is not present, the variables PartMode and IntraSplitFlag are derived as follows.

* PartMode is inferred to be equal to PART\_2Nx2N,
* IntraSplitFlag is inferred to be equal to 0.

**pcm\_flag** specifies whether the associated coding unit with PART\_2Nx2N is coded by I\_PCM: If the pcm\_flag is equal to 1, the associated coding unit with PART\_2Nx2N is coded by I\_PCM. When the pcm\_flag is not present, it shall be infered to be equal to 0.

**pcm\_alignment\_zero\_bit** is a bit equal to 0.

**pcm\_sample\_luma**[ i ] represents a coded luma sample value in the raster scan within the coding unit. The number of bits used to represent each of these samples is PCMBitDepthY.

**pcm\_sample\_chroma**[ i ] represents a coded chroma sample value in the raster scan within the coding unit. The first half of the values represent coded Cb samples and the remaining half of the values represent coded Cr samples. The number of bits used to represent each of these samples is PCMBitDepthC.

Table 7‑11 ‑ Name association to prediction mode and partitioning type

|  |  |  |  |
| --- | --- | --- | --- |
| **PredMode** | **part\_mode** | **IntraSplitFlag** | **PartMode** |
| MODE\_INTRA | 0 | 0 | PART\_2Nx2N |
| 1 | 1 | PART\_NxN |
| MODE\_INTER | 0 | 0 | PART\_2Nx2N |
| 1 | 0 | PART\_2NxN |
| 2 | 0 | PART\_Nx2N |
| 3 | 0 | PART\_NxN |
| 4 | 0 | PART\_2NxnU |
| 5 | 0 | PART\_2NxnD |
| 6 | 0 | PART\_nLx2N |
| 7 | 0 | PART\_nRx2N |

### Prediction unit semantics

**mvp\_l0\_flag**[ x0 ][ y0 ] specifies the motion vector predictor index of list 0 where x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered prediction block relative to the top-left luma sample of the picture.

When mvp\_l0\_flag[ x0 ][ y0 ] is not present, it shall be inferred to be equal to 0.

**mvp\_l1\_flag**[ x0 ][ y0 ] has the same semantics as mvp\_l0\_flag, with l0 and list 0 replaced by l1 and list 1, respectively.

**mvp\_lc\_flag**[ x0 ][ y0 ] has the same semantics as mvp\_l0\_flag, with l0 and list 0 replaced by lc and list combination, respectively.

**prev\_intra\_luma\_pred\_flag**[ x0 ][ y0 ], **mpm\_flag[** x0 **][** y0 **]** and **rem\_intra\_luma\_pred\_mode**[ x0 ][ y0 ]specify the intra prediction mode for luma samples. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered prediction block relative to the top-left luma sample of the picture. When prev\_intra\_luma\_pred\_flag[ x0 ][ y0 ] is equal to 1, the intra prediction mode is inferred from a neighbouring intra-predicted prediction unit according to subclause 8.3.1.

**intra\_chroma\_pred\_mode**[ x0 ][ y0 ] specifies the intra prediction mode for chroma samples. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered prediction block relative to the top-left luma sample of the picture.

**merge\_flag**[ x0 ][ y0 ] specifies whether the inter prediction parameters for the current prediction unit are inferred from a neighbouring inter-predicted partition. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered prediction block relative to the top-left luma sample of the picture.

**merge\_idx**[ x0 ][ y0 ] specifies the merging candidate index of the merging candidate list where x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered prediction block relative to the top-left luma sample of the picture.

When merge\_idx[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

**inter\_pred\_flag**[ x0 ][ y0 ] specifies whether uni-prediction, or bi-prediction is used for the current prediction unit according to Table 7‑11. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered prediction block relative to the top-left luma sample of the picture.

Table 7‑12 – Name association to inter prediction mode

|  |  |  |
| --- | --- | --- |
| **slice\_type** | **inter\_pred\_flag** | **Name of inter\_pred\_flag** |
| P | inferred | Pred\_L0 |
| B | 0 | Pred\_LC |
| 1 | Pred\_BI |

When inter\_pred\_flag[ x0 ][ y0 ] is not present, it shall be inferred to be equal to Pred\_L0 when slice\_type is equal to P and Pred\_BI when slice\_type is equal to B.

**ref\_idx\_l0**[ x0 ][ y0 ] specifies the list 0 reference picture index for the current prediction unit. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered prediction block relative to the top-left luma sample of the picture.

When ref\_idx\_l0[ x0 ][ y0 ] is not present and combined\_inter\_pred\_ref\_idx is present, ref\_idx\_l0[ x0 ][ y0 ] shall be inferred to be equal to ( ( combined\_inter\_pred\_ref\_idx – NumPredRefLC ) / NumPredRefL1 ).

**ref\_idx\_l1**[ x0 ][ y0 ] has the same semantics as ref\_idx\_l0, with l0 and list 0 replaced by l1 and list 1, respectively.

**ref\_idx\_lc**[ x0 ][ y0 ] has the same semantics as ref\_idx\_l0, with l0 and list 0 replaced by lc and list combination, respectively.

**mvd\_l0**[ x0 ][ y0 ]**[** compIdx **]**, specifies the difference between a list 0 vector component to be used and its prediction. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered prediction block relative to the top-left luma sample of the picture. The horizontal motion vector component difference is assigned compIdx = 0 and the vertical motion vector component is assigned compIdx = 1.

When any of the two components is not present, the inferred value is 0.

**mvd\_l1**[ x0 ][ y0 ]**[** compIdx **]** has the same semantics as mvd\_l0, with l0 and list 0 replaced by l1 and list 1, respectively.

**mvd\_lc**[ x0 ][ y0 ]**[** compIdx **]** has the same semantics as mvd\_l0, with l0 and list 0 replaced by lc and list combination, respectively.

#### Motion vector difference coding semantics

**abs\_mvd\_greater0\_flag[** compIdx **]** specifies whether the absolute value of a motion vector component difference is greater than 0. The horizontal motion vector component difference is assigned compIdx = 0 and the vertical motion vector component is assigned compIdx = 1.

**abs\_mvd\_greater1\_flag[** compIdx **]** specifies whether the absolute value of a motion vector component difference is greater than 1. The horizontal motion vector component difference is assigned compIdx = 0 and the vertical motion vector component is assigned compIdx = 1.

When abs\_mvd\_greater1\_flag[ compIdx ] is not present, it is inferred to be equal to 0.

**abs\_mvd\_minus2[** compIdx **]** is the absolute value of a motion vector component difference minus 2. The horizontal motion vector component difference is assigned compIdx = 0 and the vertical motion vector component is assigned compIdx = 1.

When abs\_mvd\_minus2[ compIdx ] is not present, it is inferred as follows.

* If abs\_mvd\_greater1\_flag[ compIdx ] is equal to 0, abs\_mvd\_minus2[ compIdx ] is inferred to be equal to −1.
* Otherwise (abs\_mvd\_greater1\_flag[ compIdx ] is equal to 1), abs\_mvd\_minus2[ compIdx ] is inferred to be equal to 0.

**mvd\_sign\_flag[** compIdx **]** specifies the sign of a motion vector component difference as follows.

* If mvd\_sign\_flag[ compIdx ] is equal to 0, the corresponding motion vector component difference has a positive value.
* Otherwise (mvd\_sign\_flag[ compIdx ] is equal to 1), the corresponding motion vector component difference has a negative value.

The horizontal motion vector component difference is assigned compIdx = 0 and the vertical motion vector component is assigned compIdx = 1.

When mvd\_sign\_flag[ compIdx ] is not present, it is inferred to be equal to 0.

### Transform tree semantics

The variable InterTUSplitDirection defines how a transform block is split into four blocks with smaller horizontal or vertical size for the purpose of transform coding. The blocks are half horizontal and vertical size when PredMode is equal to MODE\_INTRA or InterTUSplitDirection is equal to 2, full horizontal and quarter vertical size when InterTUSplitDirection is equal to 0, quarter horizontal and full vertical size when InterTUSplitDirection is equal to 1. InterTUSplitDirection is specified as follows.

if( ( ( log2TrafoSize = = Log2MaxTrafoSize  
 | | ( log2TrafoSize < Log2MaxTrafoSize && trafoDepth = = 0 ) )   
 && log2TrafoSize > ( Log2MinTrafoSize + 1 )  
 && ( PartMode = = PART\_2NxN  
 | | PartMode = = PART\_2NxnU | | PartMode = = PART\_2NxnD ) )  
 | | ( log2TrafoSize = = ( Log2MinTrafoSize + 1 )  
 && log2TrafoWidth < log2TrafoHeight ) )   
{  
 InterTUSplitDirection = 0  
}  
else if( ( ( log2TrafoSize = = Log2MaxTrafoSize  
 | | ( log2TrafoSize < Log2MaxTrafoSize && trafoDepth = = 0 ) )   
 && log2TrafoSize > ( Log2MinTrafoSize + 1 )  
 && ( PartMode = = PART\_Nx2N  
 | | PartMode = = PART\_nLx2N | | PartMode = = PART\_nRx2N ) )  
 | | ( log2TrafoSize = = ( Log2MinTrafoSize + 1 )  
 && log2TrafoWidth > log2TrafoHeight ) )   
{  
 InterTUSplitDirection = 1  
}  
else   
{  
 InterTUSplitDirection = 2  
}

**no\_residual\_data\_flag** equal to 1 specifies that no residual data are present for the current coding unit. no\_residual\_data\_flag equal to 0 specifies that residual data are present for the current coding unit.

When no\_residual\_data\_flag is not present, its value shall be inferred to be equal to 0.

**split\_transform\_flag**[ x0 ][ y0 ][ trafoDepth ] specifies whether a block is split into four blocks with smaller horizontal or vertical size for the purpose of transform coding. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered block relative to the top-left luma sample of the picture. The array index trafoDepth specifies the current subdivision level of a coding unit into blocks for the purpose of transform coding. trafoDepth is equal to 0 for blocks that correspond to coding units.

When split\_transform\_flag[ x0 ][ y0 ][ trafoDepth ] is not present, it is inferred as follows:

* If log2TrafoSize is greater than Log2MaxTrafoSize or intraSplitFlag is equal to 1 or interSplitFlag is equal to 1, the value of split\_transform\_flag[ x0 ][ y0 ][ trafoDepth ] is inferred to be equal to 1.
* Otherwise (log2TrafoSize is less than or equal to Log2MaxTrafoSize, intraSplitFlag is equal to 0 and interSplitFlag is equal to 0), the value of split\_transform\_flag[ x0 ][ y0 ][ trafoDepth ] is inferred to be equal to 0.

**cbf\_luma**[ x0 ][ y0 ][ trafoDepth ] equal to 1 specifies that the luma transform block contains one or more transform coefficient levels not equal to 0. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered transform block relative to the top-left luma sample of the picture. The array index trafoDepth specifies the current subdivision level of a coding unit into blocks for the purpose of transform coding. trafoDepth is equal to 0 for blocks that correspond to coding units.

When cbf\_luma[ x0 ][ y0 ][ trafoDepth ] is not present, it is inferred to be equal to 1.

**cbf\_cb**[ x0 ][ y0 ][ trafoDepth ] equal to 1 specifies that the Cb transform block contains one or more transform coefficient levels not equal to 0. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered transform block relative to the top-left luma sample of the picture. The array index trafoDepth specifies the current subdivision level of a coding unit into blocks for the purpose of transform coding. trafoDepth is equal to 0 for blocks that correspond to coding units.

When cbf\_cb[ x0 ][ y0 ][ trafoDepth ] is not present and PredMode is not equal to MODE\_INTRA, the value of cbf\_cb[ x0 ][ y0 ][ trafoDepth ] is inferred to be equal to 0.

**cbf\_cr**[ x0 ][ y0 ][ trafoDepth ] equal to 1 specifies that the Cr transform block contains one or more transform coefficient levels not equal to 0. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered transform block relative to the top-left luma sample of the picture. The array index trafoDepth specifies the current subdivision level of a coding unit into blocks for the purpose of transform coding. trafoDepth is equal to 0 for blocks that correspond to coding units.

When cbf\_cr[ x0 ][ y0 ][ trafoDepth ] is not present and PredMode is not equal to MODE\_INTRA, the value of cbf\_cr[ x0 ][ y0 ][ trafoDepth ] is inferred to be equal to 0.

### Transform coefficient semantics

The transform coefficient levels are parsed into the arrays transCoeffLevel[ x0 ][ y0 ][ cIdx ][ xC ][ yC ]. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered transform block relative to the top-left luma sample of the picture. The array index cIdx specifies an indicator for the colour component; it is equal to 0 for luma, equal to 1 for Cb, and equal to 2 for Cr. The array indices xC, yC specify the transform coefficient location ( xC, yC ) within the current transform block.

When PredMode is equal to MODE\_INTRA, different scanning orders are used. The array ScanType[ log2TrafoSize − 2 ][ IntraPredMode ], specifying the scanning order for various luma transform block sizes and intra prediction modes, is derived as specified in Table 7‑16.

Table 7‑13 – Specification of ScanType[ log2TrafoSize − 2 ][ IntraPredMode ]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **IntraPredMode** | **log2TrafoSize − 2** | | | |
| **0** | **1** | **2** | **3** |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 2 | 2 | 0 | 0 |
| 2-3 | 0 | 0 | 0 | 0 |
| 4-5 | 1 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 |
| 7-8 | 2 | 2 | 0 | 0 |
| 9-10 | 0 | 0 | 0 | 0 |
| 11-12 | 1 | 1 | 0 | 0 |
| 13-14 | 0 | 0 | 0 | 0 |
| 15-16 | 2 | 2 | 0 | 0 |
| 17-19 | 0 | 0 | 0 | 0 |
| 20-23 | 1 | 1 | 0 | 0 |
| 24-27 | 0 | 0 | 0 | 0 |
| 28-31 | 2 | 2 | 0 | 0 |
| 32-33 | 0 | 0 | 0 | 0 |
| 34 | 0 | 0 | 0 | 0 |
| 35 | 0 | 0 | 0 | 0 |

**cu\_qp\_delta** can change the value of QPY for a quantization group of coding units where the quantization group of coding units is specified as follows:

* If a coding unit with the split\_coding\_unit\_flag[ x0 ][ y0 ] equal to 0 and the log2CUSize is greater than or equal to log2MinCUDQPSize, the quantization group includes this coding unit only.
* Otherwise, if a coding unit with the split\_coding\_unit\_flag[ x0 ][ y0 ] equal to 1 and the log2CUSize is equal to log2MinCUDQPSize, the quantization group includes all coding units split from this coding unit.

The decoded value of cu\_qp\_delta shall be in the range of –( 26+ QpBdOffsetY / 2 ) to +( 25+ QpBdOffsetY / 2 ), inclusive. cu\_qp\_delta shall be inferred to be equal to 0 when it is not present for any quantization group of coding units.

The value of QPY is derived as

QPY = ( ( ( QPY,PREV + cu\_qp\_delta +52+ 2\*QpBdOffsetY )%( 52 + QpBdOffsetY ) ) - QpBdOffsetY (7‑21)

where QPY,PREV is the luma quantization parameter, QPY, of the left neighbor quantization group of coding units in the current slice. If the left neighbor quantization group in the current slice is not available, QPY,PREV is the luma quantization parameter, of the previous quantization group in decoding order that is available in the current slice. For the first quantization group of coding units in the slice QPY,PREV is initially set equal to SliceQPY at the start of each slice. In addition, for the first set of quantization group of coding units in a tile QPY,PREV is initially set equal to SliceQPY

The value of QP’Y is derived as

QP’Y = QPY + QpBdOffsetY (7‑22)

The values of QP’Cb and QP’Cr are derived as

QP’Cb = Clip3(0, 51, QP’Y + chroma\_cb\_qp\_offset) (7‑22)

QP’Cr = Clip3(0, 51, QP’Y + chroma\_cr\_qp\_offset) (7‑22)

### Residual coding semantics

The array ScanOrder[ log2BlockWidth ][ log2BlockHeight ][ scanIdx ][ pos ][ comp ] specifies the mapping of the scan position pos, ranging from 0 to ( ( 1 << log2BlockWidth ) \* ( 1 << log2BlockHeight ) ) – 1, to horizontal and vertical components of the scan-order matrix. The array index scanIdx equal to 0 specifies an up-right diagonal scan, scanIdx equal to 1 specifies a horizontal scan, and scanIdx equal to 2 specifies a vertical scan. The array index comp equal to 0 specifies the horizontal component and the array index comp equal to 1 specifies the vertical component. The array ScanOrder is derived as follows.

The scanning order array initialisation process as specified in 6.6 is invoked with 1 << log2TrafoWidth and 1 << log2TrafoHeight as input and the output is assigned to ScanOrder[ log2TrafoWidth ][ log2TrafoHeight ].

**last\_significant\_coeff\_x\_prefix** specify the prefix of the column position of the last significant coefficient in scanning order within a transform block. The values of last\_significant\_coeff\_x\_prefix shall be in the range from 0 to ( log2TrafoWidth << 1 ) − 1, inclusive.

**last\_significant\_coeff\_y\_prefix** specify the prefix of the row position of the last significant coefficient in scanning order within a transform block. The values of last\_significant\_coeff\_y\_prefix shall be in the range from 0 to ( log2TrafoHeight << 1 ) − 1, inclusive.

**last\_significant\_coeff\_x\_suffix** specify the suffix of the column position of the last significant coefficient in scanning order within a transform block. The values of last\_significant\_coeff\_x\_suffix shall be in the range from 0 to ( 1 << ( ( last\_significant\_coeff\_x\_prefix >> 1 ) − 1 ) ) − 1, inclusive.

The column position of the last significant coefficient in scanning order within a transform block LastSignificantCoeffX is derived as follows.

* If last\_significant\_coeff\_x\_suffix is not present, the following applies.

LastSignificantCoeffX = last\_significant\_coeff\_x\_prefix (7‑37)

* Otherwise (last\_significant\_coeff\_x\_suffix is present), the following applies.

LastSignificantCoeffX = ( (1 << ((last\_significant\_coeff\_x\_prefix >> 1) − 1)) \*   
 (2 + (last\_significant\_coeff\_x\_prefix & 1)) + (7‑38)  
 last\_significant\_coeff\_x\_suffix

**last\_significant\_coeff\_y\_suffix** specify the suffix of the row position of the last significant coefficient in scanning order within a transform block. The values of last\_significant\_coeff\_y\_suffix shall be in the range from 0 to ( 1 << (  ( last\_significant\_coeff\_y\_prefix >> 1 ) − 1 ) ) − 1, inclusive.

The row position of the last significant coefficient in scanning order within a transform block LastSignificantCoeffY is derived as follows.

* If last\_significant\_coeff\_y\_suffix is not present, the following applies.

LastSignificantCoeffY = last\_significant\_coeff\_y\_prefix (7‑39)

* Otherwise (last\_significant\_coeff\_y\_suffix is present), the following applies.

LastSignificantCoeffY = ( (1 << ((last\_significant\_coeff\_y\_prefix >> 1) − 1)) \*   
 (2 + (last\_significant\_coeff\_y\_prefix & 1)) + (7‑40)   
 last\_significant\_coeff\_y\_suffix

When scanIdx is equal to 2, the coordinates are swapped as follows.

temp = LastSignificantCoeffX  
LastSignificantCoeffX = LastSignificantCoeffY (7‑41)  
LastSignificantCoeffY = temp

**significant\_coeff\_group\_flag**[ xCG ][ yCG ] specifies for the coefficient group position ( xCG, yCG ) within the current transform block whether the corresponding coefficient group at location ( xCG, yCG ) has non-zero transform coefficient level. A coefficient group at location ( xCG, yCG ) is a 4x4 array of 16 transform coefficient level at locations ( xC, yC ).

* If significant\_coeff\_group\_flag[ xCG ][ yCG ] is equal to 0, the 16 transform coefficient levels of the coefficient group at location ( xCG, yCG ) are inferred to be equal to 0;
* Otherwise (significant\_coeff\_group\_flag[ xCG ][ yCG ] is equal to 1), the following applies.
* If significant\_coeff\_group\_flag[ xCG ][ yCG ] is present or ( xCG << 2, yCG << 2 ) is the last significant position ( LastSignificantCoeffX, LastSignificantCoeffY ), at least one of the 16 transform coefficient levels of the coefficient group at location ( xCG, yCG ) has a non zero value.
* Otherwise (significant\_coeff\_group\_flag[ xCG ][ yCG ] is not present and ( xCG << 2, yCG << 2 ) is not the last significant position ( LastSignificantCoeffX, LastSignificantCoeffY )), at least one of the 16 significant\_coeff\_flag syntax elements is present for the coefficient group at location ( xCG, yCG )

When significant\_coeff\_group\_flag[ xCG ][ yCG ] is not present, it is inferred as follows.

* If one of the following conditions is true, significant\_coeff\_group\_flag[ xCG ][ yCG ] is inferred to be equal to 1.
* ( xCG, yCG ) is equal to ( LastSignificantCoeffX >> 2, LastSignificantCoeffY >> 2 )
* ( xCG, yCG ) is equal to ( 0, 0 )
* ( xCG < ( 1 << log2BlockWidth ) − 1 ) && ( yCG < ( 1 << log2BlockHeight ) − 1 ) &&  
  significant\_coeff\_group\_flag[ xCG+1 ][ yCG ] = = 1 && significant\_coeff\_group\_flag[ xCG ][ yCG+1 ] = = 1
* Otherwise (none of the above conditions is true), significant\_coeff\_group\_flag[ xCG ][ yCG ] is inferred to be equal to 0.

**significant\_coeff\_flag[** xC **][** yC **]** specifies for the transform coefficient position ( xC, yC ) within the current transform block whether the corresponding transform coefficient level at location ( xC, yC ) is non-zero as follows.

* If significant\_coeff\_flag[ xC ][ yC ] is equal to 0, the transform coefficient level at location ( xC, yC ) is set equal to 0.
* Otherwise (significant\_coeff\_flag[ xC ][ yC ] is equal to 1), the transform coefficient level at location ( xC, yC ) has a non‑zero value.

When significant\_coeff\_flag[ xC ][ yC ] is not present, it is inferred as follows.

* If ( xC, yC ) is the last significant location ( LastSignificantCoeffX, LastSignificantCoeffY ) in scan order, significant\_coeff\_flag[ xC ][ yC ] is inferred to be equal to 1
* Otherwise (( xC, yC ) is not the last significant location ( LastSignificantCoeffX, LastSignificantCoeffY ) in scan order), significant\_coeff\_flag[ xC ][ yC ] is inferred to be equal to 0.

**coeff\_abs\_level\_greater1\_flag[** n **]** specifies for the scanning position n whether there are transform coefficient levels greater than 1.

When coeff\_abs\_level\_greater1\_flag[ n ] is not present, it is inferred to be equal to 0.

**coeff\_abs\_level\_greater2\_flag[** n **]** specifies for the scanning position n whether there are transform coefficient levels greater than 2.

When coeff\_abs\_level\_greater2\_flag[ n ] is not present, it is inferred to be equal to 0.

**coeff\_abs\_level\_minus3[** n **]** is the absolute value of a transform coefficient level minus 3 at the scanning position n. The value of coeff\_abs\_level\_minus3 is constrained by the limits in subclause XX.

When coeff\_abs\_level\_minus3[ n ] is not present, it is inferred as follows.

* If coeff\_abs\_level\_greater1\_flag[ n ] is equal to 0, coeff\_abs\_level\_minus3[ n ] is inferred to be equal to −2.
* Otherwise (coeff\_abs\_level\_greater1\_flag[ n ] is equal to 1), coeff\_abs\_level\_minus3[ n ] is inferred to be equal to −1.

**coeff\_sign\_flag[** n **]** specifies the sign of a transform coefficient level for the scanning position n as follows.

* If coeff\_sign\_flag[ n ] is equal to 0, the corresponding transform coefficient level has a positive value.
* Otherwise (coeff\_sign\_flag[ n ] is equal to 1), the corresponding transform coefficient level has a negative value.

When coeff\_sign\_flag[ n ] is not present, it is inferred to be equal to 0.

# Decoding process

Outputs of this process are decoded samples of the current picture (sometimes referred to by the variable CurrPic).

Depending on the value of chroma\_format\_idc, the number of sample arrays of the current picture is as follows.

– If chroma\_format\_idc is equal to 0, the current picture consists of 1 sample array SL.

– Otherwise (chroma\_format\_idc is not equal to 0), the current picture consists of 3 sample arrays SL, SCb, SCr.

This clause describes the decoding process, given syntax elements and upper-case variables from clause 7.

The decoding process is specified such that all decoders shall produce numerically identical results. Any decoding process that produces identical results to the process described here conforms to the decoding process requirements of this Recommendation | International Standard.

Each picture referred to in this clause is a complete primary coded picture or part of a primary coded picture. Each slice referred to in this clause is a slice of a primary coded picture. Each slice data partition referred to in this clause is a slice data partition of a primary coded picture.

Depending on the value of separate\_colour\_plane\_flag, the decoding process is structured as follows.

– If separate\_colour\_plane\_flag is equal to 0, the decoding process is invoked a single time with the current picture being the output.

– Otherwise (separate\_colour\_plane\_flag is equal to 1), the decoding process is invoked three times. Inputs to the decoding process are all NAL units of the primary coded picture with identical value of colour\_plane\_id. The decoding process of NAL units with a particular value of colour\_plane\_id is specified as if only a coded video sequence with monochrome colour format with that particular value of colour\_plane\_id would be present in the bitstream. The output of each of the three decoding processes is assigned to the 3 sample arrays of the current picture with the NAL units with colour\_plane\_id equal to 0 being assigned to SL, the NAL units with colour\_plane\_id equal to 1 being assigned to SCb, and the NAL units with colour\_plane\_id equal to 2 being assigned to SCr.

NOTE – The variable ChromaArrayType is derived as 0 when separate\_colour\_plane\_flag is equal to 1 and chroma\_format\_idc is equal to 3. In the decoding process, the value of this variable is evaluated resulting in operations identical to that of monochrome pictures with chroma\_format\_idc being equal to 0.

An overview of the decoding process is given as follows:

1. The decoding of NAL units is specified in subclause 8.1.
2. The processes in subclause 8.2 specify decoding processes using syntax elements in the slice layer and above:

– Variables and functions relating to picture order count are derived in subclause 8.2.1 (only needed to be invoked for one slice of a picture).

– The decoding process for reference picture set in subclause 8.2.2 is invoked for marking of pictures as "unused for reference" (only needed to be invoked for one slice of a picture).

– If the first coded picture in the bitsream is a CRA picture, and the current picture is a leading picture of the first coded picture in the bitstream, the output\_flag of the current coded picture is set to be equal to 0 (regardless of the value of the output\_flag in the NAL unit header of the VCL NAL units of the coded picture), and the decoding process for generating non-existing reference pictures in subclause 8.2.3 is invoked (only needed to be invoked for one slice of a picture).

– At the beginning of the decoding process for each P, SP, or B slice, the decoding process for reference picture lists construction specified in subclause 8.2.3 is invoked for derivation of reference picture list 0 (RefPicList0), and when decoding a B slice, reference picture list 1 (RefPicList1).

– After all slices of the current picture have been decoded, if it is a reference picture the decoded reference picture is marked as "used for short-term reference" [Ed.Note (YK): Missing “used for long-term reference”.], otherwise (it is a non-reference picture) the decoded picture is marked as "unused for reference", as specified in subclause 8.2.4.

1. The processes in subclauses 8.3, 8.4, 8.5, and 8.6 specify decoding processes using syntax elements in the treeblock layer and above.

## NAL unit decoding process

Inputs to this process are NAL units.

Outputs of this process are the RBSP syntax structures encapsulated within the NAL units.

The decoding process for each NAL unit extracts the RBSP syntax structure from the NAL unit and then operates the decoding processes specified for the RBSP syntax structure in the NAL unit as follows.

Subclause 8.2 describes the decoding process for NAL units with nal\_unit\_type equal to 1 and 5.

NAL units with nal\_unit\_type equal to 7 and 8 contain sequence parameter sets and picture parameter sets, respectively. Picture parameter sets are used in the decoding processes of other NAL units as determined by reference to a picture parameter set within the slice headers of each picture. Sequence parameter sets are used in the decoding processes of other NAL units as determined by reference to a sequence parameter set within the picture parameter sets of each sequence.

## Slice decoding process

### Decoding process for picture order count

Output of this process is PicOrderCntVal, the picture order count of the current picture.

Picture order counts are used to identify pictures, for deriving motion parameters in temporal or spatial direct mode, to represent picture order differences between frames for motion vector derivation, for implicit mode weighted prediction in B slices (see subclause 8.4.2.2.3), and for decoder conformance checking (see subclause C.4).

Each coded picture is associated with one picture order count, called PicOrderCntVal. PicOrderCntVal indicates the picture order of the corresponding picture relative to the previous IDR picture in decoding order.

The variables prevPicOrderCntLsb and prevPicOrderCntMsb are derived as follows.

* If the current picture is an IDR picture, both prevPicOrderCntLsb and prevPicOrderCntMsb are set equal to 0.

– Otherwise (the current picture is not an IDR picture), the following applies.

– If the first coded picture in the bitstream is a CRA picture and the current picture is the first coded picture in the bitstream, both prevPicOrderCntLsb and prevPicOrderCntMsb are set equal to 0.

– Otherwise (the first coded picture in the bitstream is not a CRA picture, or the first coded picture in the bitstream is a CRA picture while the current picture is the not first coded picture in the bitstream), let prevRefPic be the previous reference picture in decoding order that has temporal\_id equal to 0. The variable prevPicOrderCntLsb is set equal to pic\_order\_cnt\_lsb of prevRefPic, and the variable prevPicOrderCntMsb is set equal to PicOrderCntMsb of prevRefPic.

PicOrderCntMsb of the current picture is derived as specified by the following pseudo-code:

if( ( pic\_order\_cnt\_lsb < prevPicOrderCntLsb ) &&  
 ( ( prevPicOrderCntLsb − pic\_order\_cnt\_lsb ) >= ( MaxPicOrderCntLsb / 2 ) ) )  
 PicOrderCntMsb = prevPicOrderCntMsb + MaxPicOrderCntLsb (8‑9)  
else if( (pic\_order\_cnt\_lsb > prevPicOrderCntLsb ) &&  
 ( (pic\_order\_cnt\_lsb − prevPicOrderCntLsb ) > ( MaxPicOrderCntLsb / 2 ) ) )  
 PicOrderCntMsb = prevPicOrderCntMsb − MaxPicOrderCntLsb  
else  
 PicOrderCntMsb = prevPicOrderCntMsb

PicOrderCntVal is derived as

PicOrderCntVal = PicOrderCntMsb + pic\_order\_cnt\_lsb (8‑9)

NOTE 1 – All IDR pictures will have PicOrderCntVal equal to 0 since pic\_order\_lsb is inferred to be 0 for IDR pictures and prevPicOrderCntLsb and prevPicOrderCntMsb are both set equal to 0.

The value of PicOrderCntVal shall be in the range of −231 to 231 − 1, inclusive. In one coded video sequence, the PicOrderCntVal values for any two coded pictures shall be different.

The function PicOrderCnt( picX ) is specified as follows:

PicOrderCnt( picX ) = PicOrderCntVal of the picture picX (8‑9)

The function DiffPicOrderCnt( picA, picB ) is specified as follows:

DiffPicOrderCnt( picA, picB ) = PicOrderCnt( picA ) − PicOrderCnt( picB ) (8‑9)

The bitstream shall not contain data that result in values of DiffPicOrderCnt( picA, picB ) used in the decoding process that exceed the range of −215 to 215 − 1, inclusive.

NOTE 2 – Let X be the current picture and Y and Z be two other pictures in the same sequence, Y and Z are considered to be in the same output order direction from X when both DiffPicOrderCnt( X, Y ) and DiffPicOrderCnt( X, Z ) are positive or both are negative.

NOTE 3 – Many encoders assign PicOrderCntVal proportional to the sampling time of the corresponding picture relative to the sampling time of the previous IDR picture.

### Decoding process for reference picture set

This process is invoked once per picture, after decoding of a slice header but prior to the decoding of any coding unit and prior to the decoding process for reference picture list construction of the slice as specified in subclause 8.2.3. The process may result in marking one or more reference pictures as "unused for reference".

NOTE 1 – The reference picture set is an absolute description of the reference pictures used in the decoding process of the current and future coded pictures. The reference picture set signaling is explicit in the sense that all reference pictures included in the reference picture set are listed explicitly and there is no default reference picture set construction process in the decoder that depends on the status of the decoded picture buffer.

Short-term reference pictures are identified by their PicOrderCntVal values. Long-term reference pictures are identified by their pic\_order\_cnt\_lsb values.

Five lists of picture order count values are constructed to derive the reference picture set; PocStCurr0, PocStCurr1, PocStFoll, PocLtCurr, and PocLtFoll with NumPocStCurr0, NumPocStCurr1, NumPocStFoll, NumPocLtCurr, and NumPocLtFoll number of elements, respectively.

If the current picture is an IDR picture, PocStCurr0, PocStCurr1, PocStFoll, PocLtCurr, and PocLtFoll are all set to empty, and NumPocStCurr0, NumPocStCurr1, NumPocStFoll, NumPocLtCurr, and NumPocLtFoll are all set to 0.

Otherwise, the following applies for derivation of the five lists of picture order count values and the numbers of entries.

for( i = 0, j = 0, k = 0; i < NumNegativePics[ StRpsIdx ] ; i++ ) {  
 if( UsedByCurrPicS0[ StRpsIdx ][ i ] )  
 PocStCurr0[ j++ ] = PicOrderCntVal + DeltaPocS0[ StRpsIdx ][ i ]  
 else  
 PocStFoll[ k++ ] = PicOrderCntVal + DeltaPocS0[ StRpsIdx ][ i ]  
}  
NumPocStCurr0 = j  
  
for( i = 0, j = 0; i < NumPositivePics[ StRpsIdx ]; i++ ) {  
 if( UsedByCurrPicS1[ StRpsIdx ][ i ] )  
 PocStCurr1[ j++ ] = PicOrderCntVal + DeltaPocS1[ StRpsIdx ][ i ]  
 else  
 PocStFoll[ k++ ] = PicOrderCntVal + DeltaPocS1[ StRpsIdx ][ i ]  
}  
NumPocStCurr1 = j  
NumPocStFoll = k (8‑9)

for( i = 0, j = 0, k = 0; i < num\_long\_term\_pics; i++ ) {  
 if( used\_by\_curr\_pic\_lt\_flag[ i ] )  
 PocLtCurr[ j++ ] = ( PicOrderCntVal − DeltaPocLt[ i ] + MaxPicOrderCntLsb ) %   
 MaxPicOrderCntLsb  
 else  
 PocLtFoll[ k++ ] = ( PicOrderCntVal − DeltaPocLt[ i ] + MaxPicOrderCntLsb ) %  
 MaxPicOrderCntLsb  
}  
NumPocLtCurr = j  
NumPocLtFoll = k

where PicOrderCntVal is the picture order count of the current picture as specified in subclause 8.2.1.

NOTE 2 – A value of StRpsIdx in the range from 0 to num\_short\_term\_ref\_pic\_sets – 1, inclusive, indicates that a short-term reference picture set from the active parameter set is being used, where StRpsIdx is the index of the short-term reference picture set to the list of short-term reference picture sets in the order in which they are signalled in the picture parameter set. StRpsIdx equal to num\_short\_term\_ref\_pic\_sets indicates that a short-term reference picture set explicitly signalled in the slice header is being used.

The reference picture set consists of five lists of reference pictures; RefPicSetStCurr0, RefPicSetStCurr1, RefPicSetStFoll, RefPicSetLtCurr and RefPicSetLtFoll.

NOTE 3 –RefPicSetStCurr0, RefPicSetStCurr1 and RefPicSetLtCurr contains all reference pictures that may be used in inter prediction of the current picture and that may be used in inter prediction of one or more of the pictures following the current picture in decoding order. RefPicSetStFoll and RefPicSetLtFoll consists of all reference pictures that are *not* used in inter prediction of the current picture but may be used in inter prediction of one or more of the pictures following the current picture in decoding order.

The marking of a reference picture can be "unused for reference", "used for short-term reference", or "used for long-term reference", but only one among these three. When a reference picture is referred to as being marked as "used for reference", this collectively refers to the picture being marked as "used for short-term reference" or "used for long-term reference" (but not both). A reference picture that is marked as "used for short-term reference" is referred to as a short‑term reference picture. A reference picture that is marked as "used for long-term reference" is referred to as a long‑term reference picture*.*

Derivation process for the reference picture set and picture marking shall be performed according to the following ordered steps, where DPB refers to the decoded picture buffer as described in Annex C:

1. The following applies:

for( i = 0; i < NumPocLtCurr; i++ ) {  
 if( there is a long-term reference picture picX in the DPB  
 with pic\_order\_cnt\_lsb equal to PocLtCurr[ i ] )  
 RefPicSetLtCurr[ i ] = picX  
 else if( there is a short-term reference picture picY in the DPB  
 with pic\_order\_cnt\_lsb equal to PocLtCurr[ i ] )  
 RefPicSetLtCurr[ i ] = picY  
 else   
 RefPicSetLtCurr[ i ] = "no reference picture"  
} (8‑9)

for( i = 0; i < NumPocLtFoll; i++ ) {  
 if( there is a long-term reference picture picX in the DPB  
 with pic\_order\_cnt\_lsb equal to PocLtFoll[ i ] )  
 RefPicSetLtFoll[ i ] = picX  
 else if( there is a short-term reference picture picY in the DPB  
 with pic\_order\_cnt\_lsb equal to PocLtFoll[ i ] )  
 RefPicSetLtFoll[ i ] = picY  
 else   
 RefPicSetLtFoll[ i ] = "no reference picture"  
}

1. All reference pictures included in RefPicSetLtCurr and RefPicSetLtFoll are marked as "used for long-term reference"
2. The following applies:

for( i = 0; i < NumPocStCurr0; i++ ) {  
 if( there is a short-term reference picture picX in the DPB  
 with PicOrderCntVal equal to PocStCurr0[ i ])  
 RefPicSetStCurr0[ i ] = picX  
 else  
 RefPicSetStCurr0[ i ] = "no reference picture"  
}

for( i = 0; i < NumPocStCurr1; i++ ) {  
 if( there is a short-term reference picture picX in the DPB  
 with PicOrderCntVal equal to PocStCurr1[ i ])  
 RefPicSetStCurr1[ i ] = picX  
 else  
 RefPicSetStCurr1[ i ] = "no reference picture"  
} (8‑9)

for( i = 0; i < NumPocStFoll; i++ ) {  
 if( there is a short-term reference picture picX in the DPB  
 with PicOrderCntVal equal to PocStFoll[ i ])  
 RefPicSetStFoll[ i ] = picX  
 else  
 RefPicSetStFoll[ i ] = "no reference picture"  
}

1. All reference pictures included in RefPicSetStCurr0, RefPicSetStCurr1 and RefPicSetStFoll are marked as "used for short-term reference".
2. All reference pictures in the decoded picture buffer that are not included in RefPicSetLtCurr, RefPicSetLtFoll, RefPicSetStCurr0, RefPicSetStCurr1 or RefPicSetStFoll are marked as "unused for reference".

NOTE 4 – There may be one or more reference pictures that are included in the reference picture set but that are not present in the decoded picture buffer. Entries in RefPicSetStFoll or RefPicSetLtFoll equal to "no reference picture" should be ignored if the first coded picture in the bitstream is an IDR picture, or if the first coded picture in the bitstream is a CRA picture and the current coded picture is not a leading picture of the first coded picture in the bitstream. An unintentional picture loss should be inferred for each entry in RefPicSetStCurr0, RefPicSetStCurr1 and RefPicSetLtCurr equal to "no reference picture".

The reference picture set is restricted as follows:

* There shall be no reference picture with temporal\_id greater than that of the current picture included in RefPicSetStCurr0, RefPicSetStCurr1 and RefPicSetLtCurr.
* There shall be no reference picture included in the reference picture set which precedes, in output order, any CRA picture that precedes the current picture both in decoding order and output order.
* If the first coded picture in the bitstream is an IDR picture, or if the first coded picture in the bitstream is a CRA picture and the current coded picture is not a leading picture of the first coded picture in the bitstream, there shall be no entry in RefPicSetStCurr0, RefPicSetStCurr1 or RefPicSetLtCurr equal to "no reference picture".

NOTE 5 – A reference picture can not be included in more than one of the five reference picture set lists.

### Decoding process for generating non-existing reference pictures

This process is invoked once per coded picture, after the invocation of the decoding process for reference picture set as specified in subclause 8.2.2.

When the first coded picture in the bitstream is a CRA picture and the current coded picture is a leading picture of the first coded picture in the bitstream, the following applies.

* For each RefPicSetStCurr0[ i ], with i in the range of 0 to NumPocStCurr0 – 1, inclusive, that is equal to "no reference picture", a reference picture is generated as specified in subclause 8.2.3.1, and the following applies.
* The value of PicOrderCntVal for the generated reference picture is set to PocStCurr0[ i ].
* The value of output\_flag for the generated reference picture is set to 0.
* The generated reference picture is marked as “used for short-term reference”.
* RefPicSetStCurr0[ i ] is set to be the generated reference picture.
* For each RefPicSetStCurr1[ i ], with i in the range of 0 to NumPocStCurr1 – 1, inclusive, that is equal to "no reference picture", a reference picture is generated as specified in subclause 8.2.3.1, and the following applies.
* The value of PicOrderCntVal for the generated reference picture is set to PocStCurr1[ i ].
* The value of output\_flag for the generated reference picture is set to 0.
* The generated reference picture is marked as “used for short-term reference”.
* RefPicSetStCurr1[ i ] is set to be the generated reference picture.
* For each RefPicSetLtCurr[ i ], with i in the range of 0 to NumPocLtCurr – 1, inclusive, that is equal to "no reference picture", a reference picture is generated as specified in subclause 8.2.3.1, and the following applies.
* The value of pic\_order\_cnt\_lsb for the generated reference picture is set to PocLtCurr[ i ].
* The value of output\_flag for the generated reference picture is set to 0.
* The generated reference picture is marked as “used for long-term reference”.
* RefPicSetLtCurr[ i ] is set to be the generated reference picture.

#### Generating of one non-existing reference picture

A (non-existingmissing) reference picture is generated as follows:

* The value of each element in the sample array SL for the is set to 1<< ( BitDepthY – 1 ).
* The value of each element in the sample arrays SCb and SCr is set to 1<< ( BitDepthC – 1 ).
* The prediction mode PredMode for each minimum coded block is set to MODE\_INTRA.

### Decoding process for reference picture lists construction

This process is invoked at the beginning of the decoding process for each P, SP, or B slice.

Decoded reference pictures are marked as "used for short-term reference" or "used for long-term reference" as specified by the bitstream and specified in subclause 8.2.2.4. Short-term reference pictures are identified by the value of frame\_num. Long-term reference pictures are assigned a long-term picture index as specified by the bitstream and specified in subclause 8.2.2.4.

Subclause 8.2.2.1 is invoked to specify

– the assignment of variables FrameNum, FrameNumWrap, and PicNum to each of the short-term reference pictures, and

– the assignment of variable LongTermPicNum to each of the long-term reference pictures.

Reference pictures are addressed through reference indices as specified in subclause **Error! Reference source not found.**. A reference index is an index into a reference picture list. When decoding a P or SP slice, there is a single reference picture list RefPicList0. When decoding a B slice, there is a second independent reference picture list RefPicList1 in addition to RefPicList0.

At the beginning of the decoding process for each slice, reference picture list RefPicList0, and for B slices RefPicList1, are derived as specified by the following ordered steps:

1. An initial reference picture list RefPicList0 and for B slices RefPicList1 are derived as specified in subclause 8.2.2.2.
2. When ref\_pic\_list\_modification\_flag\_l0 is equal to 1 or, when decoding a B slice, ref\_pic\_list\_modification\_flag\_l1 is equal to 1, the initial reference picture list RefPicList0 and, for B slices, RefPicList1 are modified as specified in subclause 8.2.2.3.

NOTE – The modification process for reference picture lists specified in subclause 8.2.2.3 allows the contents of RefPicList0 and for B slices RefPicList1 to be modified in a flexible fashion. In particular, it is possible for a picture that is currently marked "used for reference" to be inserted into RefPicList0 and for B slices RefPicList1 even when the picture is not in the initial reference picture list derived as specified in subclause 8.2.2.2.

The number of entries in the modified reference picture list RefPicList0 is num\_ref\_idx\_l0\_active\_minus1 + 1, and for B slices the number of entries in the modified reference picture list RefPicList1 is num\_ref\_idx\_l1\_active\_minus1 + 1. A reference picture may appear at more than one index in the modified reference picture lists RefPicList0 or RefPicList1.

#### Initialisation process for reference picture lists

This initialisation process is invoked when decoding a P or B slice header.

When decoding a P or B slice, there shall be at least one reference picture in RefPicSetStCurr0, RefPicSetStCurr1 or RefPicSetLtCurr.

The following procedure is conducted to construct the initial RefPicList0:

cIdx = 0  
while( cIdx <= num\_ref\_idx\_l0\_active\_minus1 )  
{  
 for( i=0; i < NumPocStCurr0 && cIdx <= num\_ref\_idx\_l0\_active\_minus1; cIdx++, i++ )  
 RefPicList0[ cIdx ] = RefPicSetStCurr0[ i ]  
 for( i=0; i < NumPocStCurr1 && cIdx <= num\_ref\_idx\_l0\_active\_minus1; cIdx++, i++ ) (8‑9)  
 RefPicList0[ cIdx ] = RefPicSetStCurr1[ i ]  
 for( i=0; i < NumPocLtCurr && cIdx <= num\_ref\_idx\_l0\_active\_minus1; cIdx++, i++ )  
 RefPicList0[ cIdx ] = RefPicSetLtCurr[ i ]  
}

The following procedure is conducted to construct the initial RefPicList1:

cIdx = 0  
while( cIdx <= num\_ref\_idx\_l1\_active\_minus1 ) {  
 for( i=0; i < NumPocStCurr1 && cIdx <= num\_ref\_idx\_l1\_active\_minus1; cIdx++, i++ )  
 RefPicList1[ cIdx ] = RefPicSetStCurr1[ i ]   
 for( i=0; i < NumPocStCurr0 && cIdx <= num\_ref\_idx\_l1\_active\_minus1; cIdx++, i++ ) (8‑9)  
 RefPicList1[ cIdx ] = RefPicSetStCurr0[ i ]  
 for( i=0; i < NumPocLtCurr && cIdx <= num\_ref\_idx\_l1\_active\_minus1; cIdx++, i++ )  
 RefPicList1[ cIdx ] = RefPicSetLtCurr[ i ]  
}

#### Modification process for reference picture lists

After the invocation of this process, there shall be no reference pictures with greater temporal\_id than the current slice included in the output RefPicList0 or RefPicList1.

When ref\_pic\_list\_modification\_flag\_l0 is equal to 1, the following applies:

1. Let refIdxL0 be an index into the reference picture list RefPicList0. It is initially set equal to 0.
2. The corresponding syntax elements modification\_of\_pic\_nums\_idc are processed in the order they occur in the bitstream. For each of these syntax elements, the following applies.

– If modification\_of\_pic\_nums\_idc is equal to 0 or equal to 1, the process specified in subclause 8.2.2.3.1 is invoked with refIdxL0 as input, and the output is assigned to refIdxL0.

– Otherwise, if modification\_of\_pic\_nums\_idc is equal to 2, the process specified in subclause 8.2.2.3.2 is invoked with refIdxL0 as input, and the output is assigned to refIdxL0.

– Otherwise (modification\_of\_pic\_nums\_idc is equal to 3), the modification process for reference picture list RefPicList0 is finished.

When the current slice is a B slice and ref\_pic\_list\_modification\_flag\_l1 is equal to 1, the following applies:

1. Let refIdxL1 be an index into the reference picture list RefPicList1. It is initially set equal to 0.
2. The corresponding syntax elements modification\_of\_pic\_nums\_idc are processed in the order they occur in the bitstream. For each of these syntax elements, the following applies.

– If modification\_of\_pic\_nums\_idc is equal to 0 or equal to 1, the process specified in subclause 8.2.2.3.1 is invoked with refIdxL1 as input, and the output is assigned to refIdxL1.

– Otherwise, if modification\_of\_pic\_nums\_idc is equal to 2, the process specified in subclause 8.2.2.3.2 is invoked with refIdxL1 as input, and the output is assigned to refIdxL1.

– Otherwise (modification\_of\_pic\_nums\_idc is equal to 3), the modification process for reference picture list RefPicList1 is finished.

##### Modification process of reference picture lists for short-term reference pictures

Input to this process is an index refIdxLX (with X being 0 or 1).

Output of this process is an incremented index refIdxLX.

The variable picNumLXNoWrap is derived as follows.

If ref\_pic\_list\_modification\_idc is equal to 0, the following applies.

– If the current reference picture list is RefPicList0, curRefPicSet is set to RefPicSetStCurr0.  
– Otherwise (the current reference picture list is RefPicList1), curRefPicSet is set to RefPicSetStCurr1.

– Otherwise, if ref\_pic\_list\_modification\_idc is equal to 1, the following applies.

– If the current reference picture list is RefPicList0, curRefPicSet is set to RefPicSetStCurr1.  
– Otherwise (the current reference picture list is RefPicList1), curRefPicSet is set to RefPicSetStCurr0.

– Otherwise, if ref\_pic\_list\_modification\_idc is equal to 2, curRefPicSet is set to RefPicSetLtCurr.

The variable pocLX is derived as follows.

pocLX = curRefPicSet[ ref\_pic\_set\_idx ] (8‑9)

The following procedure is conducted to place the picture picR with PicOrderCnt( picR ) equal to pocLX into the index position refIdxLX, shift the position of any other remaining pictures to later in the list, and increment the value of refIdxLX.

for( cIdx = num\_ref\_idx\_lX\_active\_minus1 + 1; cIdx > refIdxLX; cIdx− − )  
 RefPicListX[ cIdx ] = RefPicListX[ cIdx − 1]  
RefPicListX[ refIdxLX++ ] = pocLX  
nIdx = refIdxLX (8‑9)  
for( cIdx = refIdxLX; cIdx <= num\_ref\_idx\_lX\_active\_minus1 + 1; cIdx++ )   
 if( PicOrderCnt( RefPicListX[ cIdx ] ) != pocLX )  
 RefPicListX[ nIdx++ ] = RefPicListX[ cIdx ]

NOTE 2 – Within this pseudo-code procedure, the length of the list RefPicListX is temporarily made one element longer than the length needed for the final list. After the execution of this procedure, only elements 0 through num\_ref\_idx\_lX\_active\_minus1 of the list need to be retained.

#### Mapping process for reference picture lists combination in B slices

[Ed.: (WJ) needs to be checked once again. Try to find better way to represent]

This initialisation process is invoked when decoding a B slice header.

Input to this process are the reference picture list RefPicListX and num\_ref\_idx\_lX\_active\_minus1 with X being 0 or 1.

Outputs of this process are arrays PredLCToPredLx and RefIdxLCToRefIdxLx.

When the current slice is a B slice and ref\_pic\_list\_modification\_flag\_lc is equal to 0, the following ordered steps apply:

Let refIdxL0 and refIdxL1 be indices into the reference picture lists RefPicListL0 and RefPicListL1. They are initially set equal to 0.

Let refIdxLC be an index into PredLCToPredLx and RefIdxLCToRefIdxLx. It is initially set equal to 0.

The following process is repeated until refIdxL0 and refIdxL1 are both greater than num\_ref\_idx\_l0\_active\_minus1 and num\_ref\_idx\_l1\_active\_minus1, respectively:

– If refIdxL0 is less than or equal to num\_ref\_idx\_l0\_active\_minus1,

– If the entry RefPicListL0[ refIdxL0 ] is the first occurance of the reference picture,

PredLCToPredLx[ refIdxLC ] = Pred\_L0, (8‑9)  
 RefIdxLCToRefIdxLx[ refIdxLC++ ] = refIdxL0.

– refIdxL0++.

– If refIdxL1 is less than or equal to num\_ref\_idx\_l1\_active\_minus1 and ref\_pic\_list\_combination\_flag equal to 1,

– If the entry RefPicListL1[ refIdxL1 ] is the first occurance of the reference picture,

PredLCToPredLx[ refIdxLC ] = Pred\_L1, (8‑10) RefIdxLCToRefIdxLx[ refIdxLC++ ] = refIdxL1.

– refIdxL1++.

When the current slice is a B slice and ref\_pic\_list\_modification\_flag\_lc is equal to 1, the following ordered steps apply:

Let refIdxLC be an index into the reference picture list PredLCToPredLx and RefIdxLCToRefIdxLx. It is initially set equal to 0.

The corresponding syntax elements pic\_from\_list\_0\_flag and ref\_idx\_list\_curr are processed in the order they occur in the bitstream. For each of these syntax elements pairs, the following applies.

– If pic\_from\_list\_0\_flag is equal to 1,

PredLCToPredLx[ refIdxLC ] = Pred\_L0, (8‑11)

– Otherwise,

PredLCToPredLx[ refIdxLC ] = Pred\_L1 (8‑12)

– RefIdxLCToRefIdxLx[ refIdxLC++ ] = ref\_idx\_list\_curr

When refIdxLC is greater than num\_com\_ref\_list\_active\_minus1+ 1, the extra entries past position num\_com\_ref\_list\_active\_minus1 are discarded from PredLCToPredLx and RefIdxLCToRefIdxLx.

When refIdxLC is less than num\_com\_ref\_list\_active\_minus1 + 1, the remaining entries in PredLCToPredLx and RefIdxLCToRefIdxLx are set equal to Pred\_L0 and 0, respectively.

### Marking of reference pictures before decoding

This process is invoked once per picture, after decoding of a slice header and the decoding process for reference picture set as specified in 8.2.2 but prior to the decoding of any coding unit and prior to the decoding process for reference picture list construction of the slice as specified in subclause 8.2.3.

When enable\_temporal\_mvp\_flag of the current picture is equal to 0 and temporal\_id of the current picture is equal to 0, all the reference pictures in the DPB are marked as “unused for temporal motion vector prediction”.

### Marking of the current picture after decoding

This process is invoked after all slices of the current picture have been decoded.

– If nal\_ref\_idc of the current picture equals 0, the current picture is marked as "unused for reference".

– Otherwise, the current picture is marked as "used for short-term reference".

## Decoding process for coding units coded in intra prediction mode

Inputs to this process are:

– a luma location ( xB, yB ) specifying the top-left luma sample of the current coding unit relative to the top‑left luma sample of the current picture,

– a variable log2CUSize specifying the size of the current coding unit.

Output of this process is:

– a modified reconstructed picture before deblocking filtering.

A variable nS is set equal to ( 1 << log2CUSize ).

Depending on pcm\_flag and IntraSplitFlag, the decoding process for luma samples is specified as follows.

– If pcm\_flag is equal to 1, the reconstucted picture is modified as follows:

recSamplesL[ xB + i, yB + j ] =   
 pcm\_sample\_luma[ ( nS \* j ) + i ] << ( BitDepthY – PCMBitDepthY ), with i, j = 0..nS-1 (8‑13)

– Otherwise (pcm\_flag is equal to 0), if IntraSplitFlag is equal to 0, the following ordered steps apply:

1. The derivation process for the intra prediction mode as specified in subclause 8.3.1 is invoked with the luma location ( xB, yB ) and the variable log2PUSize set equal to log2CUSize as well as IntraPredMode that is previously (in decoding order) derived for adjacent coding units as the input and the output is the variable IntraPredMode[ xB ][ yB ].
2. The decoding process for intra blocks as specified in subclause 8.3.3 is invoked with the luma location ( xB, yB ), the variable log2TrafoSize set equal to log2CUSize, the variable trafoDepth set equal to 0, the luma intra prediction mode IntraPredMode[ xB ][ yB ] and the variable cIdx set equal to 0 as the inputs and the output is a modified reconstructed picture before deblocking filtering.

– Otherwise (pcm\_flag is equal to 0 and IntraSplitFlag is equal to 1), for the variable blkIdx proceeding over the values 0..3, the following ordered steps apply:

1. The variable xBS is set equal to xB + ( nS >> 1 ) \* ( blkIdx % 2 ).
2. The variable yBS is set equal to yB + ( nS >> 1 ) \* ( blkIdx / 2 ).
3. The derivation process for the intra prediction mode as specified in subclause 8.3.1 is invoked with the luma location ( xBS, yBS ) and the variable log2PUSize set equal to log2CUSize – 1 as well as IntraPredMode that is previously (in decoding order) derived for adjacent coding units as the input and the output is the variable IntraPredMode[ xBS ][ yBS ].
4. The decoding process for intra blocks as specified in subclause 8.3.3 is invoked with the luma location ( xBS, yBS ), the variable log2TrafoSize set equal to log2CUSize − 1, the variable trafoDepth set equal to 1, the luma intra prediction mode IntraPredMode[ xBS ][ yBS ] and the variable cIdx set equal to 0 as the inputs and the output is a modified reconstructed picture before deblocking filtering.

Depending on pcm\_flag, the decoding process for chroma samples is specified as follows:

– If pcm\_flag is equal to 1, the reconstucted picture is modified as follows:

recSamplesCb[ xB/2 + i, yB/2 + j ] =   
 pcm\_sample\_chroma[ ( nS/2 \* j ) + i ] << ( BitDepthC – PCMBitDepthC ) with i, j = 0..nS/2-1 (8‑13)

recSamplesCr[ xB/2 + i, yB/2 + j ] =   
 pcm\_sample\_chroma[ ( nS/2 \* ( j + nS ) ) + i ] << ( BitDepthC – PCMBitDepthC ) with i, j = 0..nS/2-1 (8‑13)

– Otherwise (pcm\_flag is equal to 0), the following ordered steps apply:

1. The derivation process for the chroma intra prediction mode as specified in 8.3.2 is invoked with the luma location ( xB, yB ) as input and the output is the variable IntraPredModeC.
2. The decoding process for intra blocks as specified in subclause 8.3.3 is invoked with the chroma location ( xB/2, yB/2 ), the variable log2TrafoSize set equal to log2CUSize-1, the variable trafoDepth set equal to 0, the chroma intra prediction mode IntraPredModeC, and the variable cIdx set equal to 1 as the inputs and the output is a modified reconstructed picture before deblocking filtering.
3. The decoding process for intra blocks as specified in subclause 8.3.3 is invoked with the chroma location ( xB/2, yB/2 ), the variable log2TrafoSize set equal to log2CUSize-1, the variable trafoDepth set equal to 0, the chroma intra prediction mode IntraPredModeC, and the variable cIdx set equal to 2 as the inputs and the output is a modified reconstructed picture before deblocking filtering.

### Derivation process for luma intra prediction mode

Inputs to this process are:

– a luma location ( xB, yB ) specifying the top-left luma sample of the current block relative to the top‑left luma sample of the current picture,

– a variable log2PUSize specifying the size of the current prediction unit,

– variable arrays IntraPredMode (If available) that are previously (in decoding order) derived for adjacent coding units.

Output of this process is the variable IntraPredMode[ xB ][ yB ].

Table 8‑5 specifies the value for the intra prediction mode and the associated names.

Table 8‑1 – Specification of intra prediction mode and associated names

|  |  |
| --- | --- |
| **Intra prediction mode** | **Associated names** |
| 0 | Intra\_Planar |
| 1 | Intra\_Vertical |
| 2 | Intra\_Horizontal |
| 3 | Intra\_DC |
| Otherwise (4..34) | Intra\_Angular |
| 35 | Intra\_FromLuma (used only for chroma) |

Table 8‑1 specifies the number of luma intra prediction modes intraPredModeNum depending on log2PUSize.

Table 8‑2 – Specification of intraPredModeNum

|  |  |
| --- | --- |
| **log2PUSize** | **intraPredModeNum** |
| 2 | 18 |
| 3 | 35 |
| 4 | 35 |
| 5 | 35 |
| 6 | 35 |

Table 8‑2 specifies the mapping table used for converting the number of intra prediction modes.

IntraPredMode[ xB ][ yB ] labelled 0, 1, 2, .., 34 represents directions of predictions as illustrated in Figure 8‑1.



Figure 8‑1 – Intra prediction mode directions (informative)

IntraPredMode[ xB ][ yB ] is derived as the following ordered steps. [Ed. (WJ): proponent suggests to move this part to the syntax since the other syntax elements utilize IntraPredMode. But it seems too complex to move all the following process to the syntax table. Maybe it’s better to move this part to the semantics section or simply avoid the use of IntraPredMode to parse the syntax item]

1. The derivation process for neighbouring treeblocks specified in subclause XXX with ( xB,  yB ) given as input and the output is assigned to tbAddrA and tbAddrB specifying the treeblock addresses of treeblocks covering ( xBA,  yBA ) and ( xBB, yBB ) respectively where ( xBA,  yBA ) is set equal to ( xB-1,  yB ) and ( xBB,  yBB ) is set equal to ( xB,  yB-1 ).
2. For N being either replaced A or B, the variables intraPredModeN are derived as follows.

* If the treeblock with address tbAddrN is not available, intraPredModeN is set equal to Intra\_Planar.
* Otherwise, if the coding unit covering ( xBN,  yBN ) is not coded as intra mode, intraPredModeN is set equal to Intra\_Planar,
* Otherwise, if yB-1 is smaller than YLCU, intraPredModeA is set equal to IntraPredMode[ xBA ][ yBA ] and intraPredModeB is set equal to Intra\_Planar.
* Otherwise, intraPredModeN is set equal to IntraPredMode[ xBN ][ yBN ], where IntraPredMode is the variable array assigned to the coding unit covering the luma location ( xBN, yBN ).

1. For N being either replaced A or B, the variables candIntraPredModeN are derived as follows.

* If intraPredModeN is greater than or equal to intraPredModeNum, candIntraPredModeN is set equal to Intra\_Planar.
* Otherwise, candIntraPredModeN is set equal to intraPredModeN

1. If candIntraPredModeA is equal to candIntraPredModeB, the candIntraPredModeA is modified as follows:

* If candIntraPredModeA is not equal to Intra\_Planar, candIntraPredModeA is set equal to Intra\_Planar
* Otherwise, candIntraPredModeA is set equal to Intra\_DC

1. The candModeList[x] is derived as follows:

candModeList[0] = Min( candIntraPredModeA, candIntraPredModeB ) (8‑13)  
candModeList[1] = Max( candIntraPredModeA, candIntraPredModeB ) (8‑13)

1. IntraPredMode[ xB ][ yB ] is derived by applying the following procedure:

* If prev\_intra\_pred\_flag[ xB ][ yB ] is true, the IntraPredMode[ xB ][ yB ] is set equal to candModeList[ mpm\_flag ][ xB ][ yB ]]
* Otherwise IntraPredMode[ xB ][ yB ] is derived by applying the following ordered steps:
  1. IntraPredMode[ xB ][ yB ] = rem\_intra\_luma\_pred\_mode
  2. When IntraPredMode[ xB ][ yB ] is equal or greater than candModeList[ 0 ], the value of IntraPredMode[ xB ][ yB ] is increased by one
  3. When IntraPredMode[ xB ][ yB ] is equal or greater than candModeList[ 1 ], the value of IntraPredMode[ xB ][ yB ] is increased by one

### Derivation process for chroma intra prediction mode

[Ed.: (WJ) this subclause may be moved to the semantics of intra\_chroma\_pred\_mode syntax]

Input to this process is a luma location ( xB, yB ) specifying the top-left luma sample of the current block relative to the top‑left luma sample of the current picture.

Output of this process is the variable IntraPredModeC.

The chroma intra prediction mode IntraPredModeC is derived as specifed in Table 8‑3 or Table 8‑4 with intra\_chroma\_pred\_mode, IntraPredMode[ xB ][ yB ] and chroma\_pred\_from\_luma\_enabled\_flag as inputs.

Table 8‑3 – Specification of IntraPredModeC according to the values of intra\_chroma\_pred\_mode and IntraPredMode[ xB ][ yB ] when chroma\_pred\_from\_luma\_enabled\_flag is equal to 1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **intra\_chroma\_pred\_mode** | **IntraPredMode[ xB ][ yB ]** | | | | |
| **0** | **1** | **2** | **3** | **X ( 0 <= X < 35 )** |
| 0 | 7 | 0 | 0 | 0 | 0 |
| 1 | 1 | 7 | 0 | 0 | 1 |
| 2 | 2 | 2 | 7 | 1 | 2 |
| 3 | 3 | 3 | 3 | 7 | 3 |
| 4 | LM | LM | LM | LM | LM |
| 5 | 0 | 1 | 2 | 3 | X |

Table 8‑4 – Specification of IntraPredModeC according to the values of intra\_chroma\_pred\_mode and IntraPredMode[ xB ][ yB ] when chroma\_pred\_from\_luma\_enabled\_flag is equal to 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **intra\_chroma\_pred\_mode** | **IntraPredMode[ xB ][ yB ]** | | | | |
| **0** | **1** | **2** | **3** | **X ( 0 <= X < 35 )** |
| 0 | 7 | 0 | 0 | 0 | 0 |
| 1 | 1 | 7 | 1 | 1 | 1 |
| 2 | 2 | 2 | 7 | 2 | 2 |
| 3 | 3 | 3 | 3 | 7 | 3 |
| 4 | 0 | 1 | 2 | 3 | X |

### Decoding process for intra blocks

Inputs to this process are:

– a sample location ( xB, yB ) specifying the top-left sample of the current block relative to the top‑left sample of the current picture,

– a variable log2TrafoSize specifying the size of the current block,

– a variable trafoDepth specifying the hierarchy depth of the current block relative to the coding unit,

– a variable intraPredMode specifying the intra prediction mode.

– a variable cIdx specifying the chroma component of the current block,

Output of this process is:

– a modified reconstructed picture before deblocking filtering.

Depending split\_transform\_flag[ xB ][ yB ][ trafoDepth ], the following applies:

– If split\_transform\_flag[ xB ][ yB ][ trafoDepth ] is equal to 1, the following ordered steps apply:

1. The variable xB1 is set equal to xB + ( ( 1 << log2TrafoSize ) >> 1 ).
2. The variable yB1 is set equal to yB + ( ( 1 << log2TrafoSize ) >> 1 ).
3. The decoding process for intra blocks as specified in this subclause is invoked with the location ( xB, yB ), the variable log2TrafoSize set equal to log2TrafoSize − 1, the variable trafoDepth set equal to trafoDepth + 1, the intra prediction mode intraPredMode, and the variable cIdx as the inputs and the output is a modified reconstructed picture before deblocking filtering.
4. The decoding process for intra blocks as specified in this subclause is invoked with the location ( xB1, yB ), the variable log2TrafoSize set equal to log2TrafoSize − 1, the variable trafoDepth set equal to trafoDepth + 1, the intra prediction mode intraPredMode, and the variable cIdx as the inputs and the output is a modified reconstructed picture before deblocking filtering.
5. The decoding process for intra blocks as specified in this subclause is invoked with the location ( xB, yB1 ), the variable log2TrafoSize set equal to log2TrafoSize − 1, the variable trafoDepth set equal to trafoDepth + 1, the intra prediction mode intraPredMode, and the variable cIdx as the inputs and the output is a modified reconstructed picture before deblocking filtering.
6. The decoding process for intra blocks as specified in this subclause is invoked with the location ( xB1, yB1 ), the variable log2TrafoSize set equal to log2TrafoSize − 1, the variable trafoDepth set equal to trafoDepth + 1, the intra prediction mode intraPredMode, and the variable cIdx as the inputs and the output is a modified reconstructed picture before deblocking filtering.

– Otherwise (split\_transform\_flag[ xB ][ yB ][ trafoDepth ] is equal to 0), the following ordered steps apply:

1. The variable nS is set equal to 1 << log2TrafoSize.
2. The intra sample prediction process as specified in subclause 8.3.3.1 is invoked with the location ( xB, yB ), the intra prediction mode intraPredMode, the prediction size nS and the variable cIdx as the inputs and the output is a (nS)x(nS) array predSamples.
3. The scaling and transformation process as specified in subclause 8.5.1 is invoked with the location ( xB, yB ), the variable trafoDepth, the variable cIdx, and the transform size trafoSize set equal to nS as the inputs and the output is a (nS)x(nS) array resSamples.
4. The residual signal accumulation process as specified in subclause XXX is invoked with the variable arraySize set equal to nS, the (nS)x(nS) array predSamples, and the (nS)x(nS) array resSamples as the inputs and the output is a (nS)x(nS) array recSamples.
5. The picture reconstruction process for a component before deblocking filtering as specified in subclause XXX is invoked with the location ( xB, yB ), the variable arraySize set equal to nS, the variable cIdx set equal to 0, and the (nS)x(nS) array recSamples as the inputs and the output is a modified reconstructed picture before deblocking filtering.

#### Intra sample prediction

Inputs to this process are:

– a sample location ( xB, yB ) specifying the top-left sample of the current block relative to the top‑left sample of the current picture,

– a variable intraPredMode specifying the intra prediction mode,

– a variable nS specifying the prediction size.

– a variable cIdx specifying the chroma component of the current block,

Output of this process is:

– the predicted samples predSamples[ x, y ], with x, y =0..nS-1.

The nS\*4+1 neighbouring samples p[ x, y ] that are constructed samples prior to the deblocking filter process, with x = -1, y = -1..nS\*2-1 and x = 0..nS\*2-1, y=-1, are derived as follows.

– The luma location (xBN, yBN ) is specified by

xBN = xB + x  (8‑14)

yBN = yB +y  (8‑15)

– Each sample p[ x, y ] with x = -1, y= -1..nS\*2-1  and x = 0..nS\*2-1, y = -1 is derived as follows

* If any of the following condition is true, the sample p[ x, y ] is marked as “not available for intra prediction”
  1. the coding unit covering ( xBN, yBN ) is not available
  2. the coding unit covering ( xBN, yBN ) is not coded as intra mode and constrained\_intra\_pred\_flag is equal to 1
* Otherwise, the sample p[ x, y ] is marked as “available for intra prediction” and the sample at the location ( xBN, yBN ) inside the treeblock tbAddrN is assigned to p[ x, y ].

When chroma\_pred\_from\_luma\_enabled\_flag is equal to 1 and cIdx is equal to 0, the nS\*4+1 neighbouring samples PLM[ x, y ] that are constructed luma samples for Intra\_FromLuma prediction mode, with x = -1, y = -1..nS\*2-1 and x = 0..nS\*2-1, y=-1, are derived as following ordered steps:

1. If the sample p[ x, y ] is marked as “not available for intra prediction”, the sample PLM[ x, y ] is marked as “not available for intra prediction”, otherwise, the sample PLM[ x, y ] is marked as “available for intra prediction”, with x = -1, y = -1..nS\*2-1 and x = 0..nS\*2-1, y=-1.
2. For x = -1, y= 0..nS\*2-1 , if the sample PLM[ x, y ] is marked as “available for intra prediction”, the sample location at the location ( xB + x – 1, yB + y ) inside the treeblock tbAddrN is assigned to PLM[ x, y ].
3. For x = -1..nS\*2-1, y=-1 , if the sample PLM[ x, y ] is marked as “available for intra prediction”, the sample location at the location ( xB + x , yB + y ) inside the treeblock tbAddrN is assigned to PLM[ x, y ].

When at least one sample p[ x, y ] with x = -1, y = -1..nS\*2 1 and x = 0..nS\*2 1, y = -1 is marked as “not available for intra prediction,” the reference sample substitution process for intra sample prediction in subclause 8.3.3.1.1 is invoked with the samples p[ x, y ] with x = -1, y = -1..nS\*2 1 and x = 0..nS\*2 1, y = -1 as input and the modified samples p[ x, y ] with x = -1, y = -1..nS\*2 1 and x = 0..nS\*2 1, y = -1 as output.

When chroma\_pred\_from\_luma\_enabled\_flag is equal to 1, cIdx is equal to 0, and at least one sample PLM[ x, y ] with x = -1, y = -1..nS\*2 1 and x = 0..nS\*2 1, y = -1 is marked as “not available for intra prediction,” the reference sample substitution process for intra sample prediction in subclause 8.3.3.1.1 is invoked with the samples PLM[ x, y ] with x = -1, y = -1..nS\*2 1 and x = 0..nS\*2 1, y = -1 as input and the modified samples PLM[ x, y ] with x = -1, y = -1..nS\*2 1 and x = 0..nS\*2 1, y = -1 as output.

Depending on intraPredMode, the following ordered steps apply:

1. When cIdx is equal to 0, filtering process of neighbouring samples specified in 8.3.3.1.2 is invoked with the sample array p and the prediction size nS as the inputs and the output is reassigned to the sample array p.
2. Intra sample prediction process according to intraPredMode applies as follows:
   1. One of the intra prediction modes specified in subclause 8.3.3.1.3 to 8.3.3.1.8 is invoked with the sample location ( xB, yB ), the sample array p, the prediction size nS and the chroma component index cIdx as the inputs and the output is the predicted sample array predSamples according to intraPredMode.

[Ed. (WJ): some functions do not use some input parameters. Above sentence should be improved]

##### Reference sample substitution process for intra sample prediction

Inputs to this process are the reference samples p[ x, y ] with x = ‑1, y = ‑1..nS\*2‑1 and x = 0..nS\*2‑1, y = ‑1 for intra sample prediction.

Outputs of this process are the modified reference samples p[ x, y ] with x = ‑1, y = ‑1..nS\*2‑1 and x = 0..nS\*2‑1, y = ‑1 for intra sample prediction.

The values of the samples p[ x, y ] with x = ‑1, y = ‑1..nS\*2‑1 and x = 0..nS\*2‑1, y = ‑1 are modified as follows:

* If all samples p[ x, y ] with x = ‑1, y = ‑1..nS\*2‑1 and x = 0..nS\*2‑1, y = ‑1 are marked as “not available for intra prediction,” the value ( 1 << ( BitDepthY ‑ 1 ) ) is substituted for the values of all samples p[ x, y ].
* Otherwise (at least one but not all samples p[ x, y ] are marked as “not available for intra prediction”), the following ordered steps are performed:

1. If p[ -1, nS\*2-1 ] is marked as “not available for intra prediction”, searching sequentially starting from x = -1, y = nS\*2-1 to x = -1, y = -1, then from x = 0,y = -1 to x = nS\*2-1 ,y = -1. As soon as a sample p[ x, y ] marked as “available for intra prediction” is found, the search is terminated and the value of p[ x, y ] is assigned to p[ -1, nS\*2-1 ].
2. For x = -1, y = nS\*2-2...-1, if p[ x, y ] is marked as “not available for intra prediction”, the value of p[ x, y+1 ] is substituted for the value of p[ x, y].
3. For x = 0..nS\*2-1, y = -1, if p[ x, y ] is marked as “not available for intra prediction”, the value of p[ x-1, y ] is substituted for the value of p[ x, y ].

All samples p[ x, y ] with x = ‑1, y = ‑1..nS\*2‑1 and x = 0..nS\*2‑1, y = ‑1 are marked as “available for intra prediction.”

##### Filtering process of neighbouring samples

Inputs to this process are:

– neighbouring samples p[ x, y ], with x, y = -1..2\*nS-1,

– a variable nS specifying the prediction size.

Output of this process is:

– filtered samples pF[ x, y ],. with x, y = -1..2\*nS-1.

Table 8‑5 – Specification of intraFilterType[ nS ][ IntraPredMode ] for various prediction unit sizes

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **IntraPredMode** | **intraFilterType**  **for nS = 4** | **intraFilterType**  **for nS = 8** | **intraFilterType**  **for nS = 16** | **intraFilterType**  **for nS = 32** | **intraFilterType**  **for nS = 64** |
| 0 | 0 | 1 | 1 | 1 | 0 |
| 1-2 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 1 | 1 | 1 | 0 |
| 4, 5 | 0 | 0 | 1 | 1 | 0 |
| 6 | 0 | 1 | 1 | 1 | 0 |
| 7, 8 | 0 | 0 | 1 | 1 | 0 |
| 9 | 0 | 1 | 1 | 1 | 0 |
| 10-20 | 0 | 0 | 1 | 1 | 0 |
| 21, 22 | 0 | 0 | 0 | 1 | 0 |
| 23-28 | 0 | 0 | 1 | 1 | 0 |
| 29, 30 | 0 | 0 | 0 | 1 | 0 |
| 31-33 | 0 | 0 | 1 | 1 | 0 |
| 34 | n/a | n/a | n/a | n/a | n/a |

Filtered sample array pF[ x, y ] with x = -1..nS\*2-1 and y = -1..nS\*2-1  are derived as follows:

– When intraFilterType[ nS ][ IntraPredMode ] is equal to 1, the following applies:

pF[ -1, nS\*2-1 ] = p[ -1, nS\*2-1 ] (8‑17)

pF[ nS\*2-1, -1 ] = p[ nS\*2-1, -1 ] (8‑18)

pF[ -1, y ] = ( p[ -1, y+1 ] + 2\*p[ -1, y ] + p[ -1, y-1 ] + 2 ) >> 2 for y = nS\*2-2..0 (8‑19)

pF[ -1, -1] = ( p[ -1, 0 ] + 2\*p[ -1, -1] + p[ 0, -1 ] + 2) >> 2 (8‑20)

pF[ x, -1 ] = ( p[ x-1, -1 ] + 2\*p[ x, -1 ] + p[ x+1, -1 ] + 2 ) >> 2 for x = 0..nS\*2-2 (8‑21)

##### Specification of Intra\_Vertical prediction mode

Inputs to this process are:

– neighbouring samples p[ x, y ], with x, y = -1..2\*nS-1,

– a variable nS specifying the prediction size

– a variable cIdx specifying the chroma component of the current block

Output of this process is:

– predicted samples predSamples[ x, y ], with x, y =0..nS-1.

This intra prediction mode is invoked when intraPredMode is equal to 0.

The values of the prediction samples predSamples[ x, y ], with x, y = 0..nS-1, are derived as follows:

– If cIdx is equal to 0,

predSamples[ x, y ] = p[ x, -1 ], with x = 1..nS-1, y = 0..nS-1 (8‑30)

predSamples[ x, y ] = Clip1Y( p[ x, -1 ] + ( ( d[ y ] + ( d[ y ] < 0 ? 1 : 0 ) )>> 1 ) ), with x = 0, y = 0..nS-1

where d[ y ] = p[ -1, y ] – p[ -1, -1 ] (8‑30)

– Otherwise,

predSamples[ x, y ] = p[ x, -1 ], with x, y = 0..nS-1 (8‑30)

##### Specification of Intra\_Horizontal prediction mode

Inputs to this process are:

– neighbouring samples p[ x, y ], with x, y = -1..2\*nS-1,

– a variable nS specifying the prediction block size.

– a variable cIdx specifying the chroma component of the current block

Output of this process is:

– predicted samples predSamples[ x, y ], with x, y =0..nS-1.

This intra prediction mode is invoked when intraPredMode is equal to 1.

The values of the prediction samples predSamples[ x, y ], with x, y = 0..nS-1, are derived as follows:

– If cIdx is equal to 0,

predSamples[ x, y ] = p[ -1, y ], with x = 0..nS-1, y = 1..nS-1 (8‑31)

predSamples[ x, y ] = Clip1Y( p[ -1, y ] + ( ( d[ x ] + ( d[ x ] < 0 ? 1 : 0 ) ) >> 1 ), with x = 0..nS-1, y = 0

where d[ x ] = p[ x, -1 ] – p[ -1, -1 ] (8‑31)

– Otherwise,

predSamples[ x, y ] = p[ -1, y ], with x, y = 0..nS-1 (8‑31)

##### Specification of Intra\_DC prediction mode

Inputs to this process are:

– neighbouring samples p[ x, y ], with x, y = -1..2\*nS-1,

– a variable nS specifying the prediction block size.

– a variable cIdx specifying the chroma component of the current block.

Output of this process is:

– predicted samples predSamples[ x, y ], with x, y =0..nS-1.

This intra prediction mode is invoked when intraPredMode is equal to 2.

The values of the prediction samples predSamples[ x, y ], with x, y = 0..nS-1, are derived as the following ordered steps:

1. A variable DCVal is derived as:

DCVal = , with x, y = 0..nS-1 (8‑32)  
where k=log2(nS)

1. Depending on the chroma component index cIdx, the following applies.

* If cIdx is equal to 0, the following applies.

predSamples[ 0, 0 ] = ( 1\*p[ -1, 0 ] + 2\*DCVal + 1\*p[ 0, -1 ] + 2 ) >> 2 (8‑32)  
predSamples[ x, 0 ] = ( 1\*p[ x, -1 ] + 3\*DCVal + 2 ) >> 2, with x = 1..nS-1 (8‑32)  
predSamples[ 0, y ] = ( 1\*p[ -1, y ] + 3\*DCVal + 2 ) >> 2, with y = 1..nS-1 (8‑32)  
predSamples[ x, y ] = DCVal, with x, y = 1..nS-1 (8‑32)

* Otherwise, the prediction samples predSamples[ x, y ] are derived as

predSamples[ x, y ] = DCVal, with x, y = 0..nS-1 (8‑32)

##### Specification of Intra\_Angular prediction mode

Inputs to this process are:

– neighbouring samples p[ x, y ], with x, y = -1..2\*nS-1,

– a variable nS specifying the prediction size.

Output of this process is:

– predicted samples predSamples[ x, y ], with x, y =0..nS-1.

This intra prediction mode is invoked when intraPredMode is in the range of 3..33.

Table 8‑6 specifies the mapping table between intraPredMode and the rearranged intra prediction order intraPredOrder.

Table 8‑6 – Specification of intraPredOrder

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **intraPredMode** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** | **17** |
| **intraPredOrder** | - | - | - | - | 1 | 5 | 13 | 17 | 21 | 29 | 33 | 3 | 7 | 11 | 15 | 19 | 23 | 27 |
| **intraPredMode** | **18** | **19** | **20** | **21** | **22** | **23** | **24** | **25** | **26** | **27** | **28** | **29** | **30** | **31** | **32** | **33** | **34** |  |
| **intraPredOrder** | 31 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 | 22 | 24 | 26 | 28 | 30 | 32 |  |

Figure 8‑2 illustrates the total 34 intra angles and Table 8‑7 specifies the mapping table between intraPredOrder and the angle parameter intraPredAngle.



Figure 8‑2 – Intra prediction angle definition (informative)

Table 8‑7 – Specification of intraPredAngle

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **intraPredOrder** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** |
| **intraPredAngle** | - | -32 | -26 | -21 | -17 | -13 | -9 | -5 | -2 | - | 2 | 5 | 9 | 13 | 17 | 21 | 26 |
| **intraPredOrder** | **17** | **18** | **19** | **20** | **21** | **22** | **23** | **24** | **25** | **26** | **27** | **28** | **29** | **30** | **31** | **32** | **33** |
| **intraPredAngle** | 32 | -26 | -21 | -17 | -13 | -9 | -5 | -2 | - | 2 | 5 | 9 | 13 | 17 | 21 | 26 | 32 |

Table 8‑8 further specifies the mapping table between intraPredOrder and the inverse angle parameter invAngle.

Table 8‑8 – Specification of invAngle

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **intraPredOrder** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |
| **invAngle** | -256 | -315 | -390 | -482 | -630 | -910 | -1638 | -4096 |
| **intraPredOrder** | **18** | **19** | **20** | **21** | **22** | **23** | **24** | **25** |
| **invAngle** | -315 | -390 | -482 | -630 | -910 | -1638 | -4096 | - |

The reference pixel array refMain[ x ], with x=-nS..2\*nS is specified as follows.

– If intraPredOrder is less than 18,

refMain[ x ] = p[ -1+x, -1 ], with x=0..nS (8‑36)

* If intraPredAngle is less than 0,

refMain[ x ] = p[ -1, -1+( ( x\*invAngle+128 )>>8 ) ], with x=( nS\*intraPredAngle ) >>5..-1 (8‑37)

* Otherwise,

refMain[ x ] = p[ -1+x, -1 ], with x=nS+1..2\*nS (8‑38)

Otherwise,

refMain[ x ] = p[ -1, -1+x ], with x=0..nS (8‑39)

* If intraPredAngle is less than 0,

refMain[ x ] = p[ -1+( ( x\*invAngle+128 )>>8 ), -1 ], with x=( nS\*intraPredAngle ) >>5..-1 (8‑40)

* Otherwise,

refMain[ x ] = p[ -1, -1+x ], with x=nS+1..2\*nS (8‑41)

The values of the prediction samples predSamples[ x, y ], with x, y = 0..nS-1 are derived by the following procedures.

– The index variable iIdx and the multiplication factor iFact are derived by

iIdx = ( ( y + 1 )\*intraPredAngle ) >> 5 (8‑42)

iFact = ( ( y + 1 )\*intraPredAngle ) && 31 (8‑43)

– Depending on the value of iFact, the following applies.

* If iFact is not equal to 0, the value of the prediction samples predSamples[ x, y ] is derived by

predSamples[ x, y ] = ( ( 32 – iFact )\*refMain[ x+iIdx+1 ] + iFact\*refMain[ x+iIdx+2] + 16 ) >> 5 (8‑44)

* Otherwise, the value of the prediction samples predSamples[ x, y ] is derived by

predSamples[ x, y ] = refMain[ x+iIdx+1 ] (8‑45)

##### Specification of Intra\_Planar prediction mode

Inputs to this process are:

– neighbouring samples p[ x, y ], with x, y = -1..2\*nS-1,

– a variable nS specifying the prediction size,

Output of this process is:

– predicted samples predSamples[ x, y ], with x, y =0..nS-1.

This intra prediction mode is invoked when intraPredMode is equal to 34.

The values of the prediction samples predSamples[ x, y ], with x, y = 0..nS-1, are derived by

predSamples[ x, y ] = (  
 ( nS – 1 – x ) \* p[ -1, y ] + ( x + 1 ) \* p[ nS, -1 ] +   
 ( nS – 1 – y ) \* p[ x ,-1 ] + ( y + 1 ) \* p[ -1, nS ] + nS ) >> ( k + 1 ) (8‑45)  
with x, y = 0..nS-1 where k = log2( nS )

##### Specification of Intra\_FromLuma prediction mode

Inputs to this process are:

– a sample location ( xB, yB ) specifying the top-left sample of the current block relative to the top‑left sample of the current picture,

– neighbouring samples p[ x, y ], with x, y = -1..2\*nS-1,

– a variable nS specifying the prediction size.

Output of this process is:

– predicted samples predSamples[ x, y ], with x, y =0..nS-1.

This intra prediction mode is invoked when intraPredMode is equal to 35.

The values of the prediction samples predSamples[ x, y ], with x, y = 0..nS-1, are derived as the following ordered steps:

1. Variable k3 and the sample array pY’ are derived as:

k3 = Max( 0, BitDepthC + log2( nS ) – 14 ) (8‑30)

pY’[ x, -1 ] = ( PLM[ 2x-1, -1 ] + 2\*PLM[ 2x, -1 ] + PLM[ 2x+1, -1 ] + 2 ) >> 2, with x = 0..nS-1 (8‑30)

pY’[ -1, y ] = ( PLM[ -1, 2y ] + PLM[ -1, 2y+1 ] ) >> 1, with y = 0..nS-1 (8‑30)

pY’[ x, y ] = ( recSamplesL[ 2x, 2y ] + recSamplesL[ 2x, 2y+1 ] ) >> 1, with x, y = 0..nS-1 (8‑30)

1. Variables L, C, LL, LC and k2 are derived as follows:

L =  (8‑30)

C =  (8‑30)

LL =  (8‑30)

LC =  (8‑30)

k2 = log2( (2\*nS) >> k3 ) (8‑30)

1. Variables a, b and k are derived as:

a1 = ( LC << k2 ) – L\*C (8‑30)  
a2 = ( LL << k2 ) – L\*L (8‑30)  
k1 = Max( 0, log2( abs( a2 ) ) – 5 ) – Max( 0, log2( abs( a1 ) ) – 14 ) + 2 (8‑30)  
a1s = a1 >> Max(0, log2( abs( a1 ) ) – 14 ) (8‑30)  
a2s = abs( a2 >> Max(0, log2( abs( a2 ) ) – 5 ) ) (8‑30)  
a3 = a2s < 1 ? 0 : Clip3( -215, 215-1, a1s\*lmDiv + ( 1 << ( k1 – 1 ) ) >> k1 ) (8‑30)

a = a3 >> Max( 0, log2( abs( a3 ) ) – 6 ) (8‑30)  
k = 13 – Max( 0, log2( abs( a ) ) – 6 ) (8‑30)

b = ( L – ( ( a\*C ) >> k1 ) + ( 1 << ( k2 – 1 ) ) ) >> k2 (8‑30)

where lmDiv is specified in Table 8‑10 with the input a2s.

1. The values of the prediction samples predSamples[ x, y ] are derived as:

predSamples[ x, y ] = Clip1C( ( ( pY’[ x, y ] \* a ) >> k ) + b ), with x, y = 0..nS-1 (8‑30)

Table 8‑9 – Specification of lmDiv

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **a2s** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| **lmDiv** | 32768 | 16384 | 10923 | 8192 | 6554 | 5461 | 4681 | 4096 | 3641 | 3277 | 2979 | 2731 | 2521 |
| **a2s** | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 |
| **lmDiv** | 2341 | 2185 | 2048 | 1928 | 1820 | 1725 | 1638 | 1560 | 1489 | 1425 | 1365 | 1311 | 1260 |
| **a2s** | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
| **lmDiv** | 1214 | 1170 | 1130 | 1092 | 1057 | 1024 | 993 | 964 | 936 | 910 | 886 | 862 | 840 |
| **a2s** | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 |
| **lmDiv** | 819 | 799 | 780 | 762 | 745 | 728 | 712 | 697 | 683 | 669 | 655 | 643 | 630 |
| **a2s** | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 | 64 |  |
| **lmDiv** | 618 | 607 | 596 | 585 | 575 | 565 | 555 | 546 | 537 | 529 | 520 | 512 |  |

## Decoding process for coding units coded in inter prediction mode

Inputs to this process are:

– a luma location ( xC, yC ) specifying the top-left luma sample of the current coding unit relative to the top left luma sample of the current picture,

– a variable log2CUSize specifying the size of the current coding unit.

Output of this process is a modified reconstructed picture before deblocking filtering.

The variable nCSL is set equal to 1 << log2CUSize and the variable nCSC is set equal to ( 1 << log2CUSize ) >> 1.

Decoding process for coding units coded in inter prediction mode consists of following ordered steps:

1. The inter prediction process as specified in subclause 8.4.1 is invoked with the luma location ( xC, yC ), coding unit size log2CUSize, inter prediction mode PredMode, and prediction partition mode PartMode as the inputs and the outputs are 3 arrays predSamplesL, predSamplesCb, predSamplesCr.
2. The decoding process for the residual signal of coding units coded in inter prediction mode specified in subclause 8.4.3 is invoked with the luma location ( xC, yC ), the size of the current coding unit log2CUSize as inputs and the outputs are 3 arrays resSamplesL, resSamplesCb, resSamplesCr.
3. The residual signal accumulation process as specified in subclause XXXX is invoked with arrays predSamplesL, predSamplesCb, predSamplesCr, and the arrays resSamplesL, resSamplesCb, resSamplesCr as the inputs and the outputs are 3 arrays recSamplesL, recSamplesCb, recSamplesCr. [Ed: (WJ) it may be split per each color component. Revisit after writing residual signal accumulation process]
4. The picture reconstruction process for a component before deblocking filtering as specified in subclause XXXX is invoked with the luma location ( xB, yB ), and 3 arrays recSamplesL, recSamplesCb, recSamplesCr as the inputs and the output is a modified reconstructed picture before deblocking filtering. [Ed: (WJ) it may be split per each color component. Revisit after writing picture reconstruction process]

### Inter prediction process

This process is invoked when decoding coding unit whose PredMode is not equal to MODE\_INTRA.

Inputs to this process are:

– a luma location ( xC, yC ) specifying the top-left luma sample of the current coding unit relative to the top left luma sample of the current picture,

– a variable log2CUSize specifying the size of the current coding unit,

– a variable PredMode specifying prediction mode of current coding unit,

– a variable PartMode specifying prediction partition mode of current coding unit.

Outputs of this process are:

– a (nCSL)x(nCSL) array predSamplesL of luma prediction samples, where nCSL is derived as specified below,

– a (nCSC)x(nCSC) array predSamplesCb of chroma prediction samples for the component Cb, where nCSC is derived as specified below,

– a (nCSC)x(nCSC) array predSamplesCr of chroma prediction samples for the component Cr, where nCSC is derived as specified below.

The variable nCSL is set equal to 1 << log2CUSize and the variable nCSC is set equal to ( 1 << log2CUSize ) >> 1. [Ed: (WJ) revisit for supporting other chroma formats]

The variable nCS1L is set equal to nCSL >> 1.

Depending on PartMode, the following applies: [Ed: (WJ) is PUType better?]

– If PartMode is equal to PART\_2Nx2N, the following ordered steps apply:

1. The decoding process for prediction units in inter prediction mode as specified in subclause 8.4.2 is invoked with the luma location ( xC, yC ), the luma location ( xB, yB ) set equal to ( 0, 0 ), the size of the coding unit nCSL, the width of the luma prediction samples nPSW set equal to nCSL, the height of the luma prediction samples nPSH set equal to nCSL and a partition index PartIdx set equal to 0 as inputs, and the outputs are a (nCSL)x(nCSL) array predSamplesL and two (nCSC)x(nCSC) arrays predSamplesCb and predSamplesCr.

– Otherwise, if PartMode is equal to PART\_2NxN, the following ordered steps apply:

1. The decoding process for prediction units in inter prediction mode as specified in subclause 8.4.2 is invoked with the luma location ( xC, yC ), the luma location ( xB, yB ) set equal to ( 0, 0 ), the size of the coding unit nCSL, the width of the luma prediction samples nPSW set equal to nCSL, the height of the luma prediction samples nPSH set equal to nCS1L and a partition index PartIdx set equal to 0 as inputs, and the outputs are a (nCSL)x(nCSL) array predSamplesL and two (nCSC)x(nCSC) arrays predSamplesCb and predSamplesCr.
2. The decoding process for prediction units in inter prediction mode as specified in subclause 8.4.2 is invoked with the luma location ( xC, yC ), the luma location ( xB, yB ) set equal to ( 0, nCS1L ), the size of the coding unit nCSL, the width of the luma prediction samples nPSW set equal to nCSL, the height of the luma prediction samples nPSH set equal to nCS1L and a partition index PartIdx set equal to 1 as inputs, and the outputs are a (nCSL)x(nCSL) array predSamplesL and two (nCSC)x(nCSC) arrays predSamplesCb and predSamplesCr.

Otherwise, if PartMode is equal to PART\_Nx2N, the following ordered steps apply:

1. The decoding process for prediction units in inter prediction mode as specified in subclause 8.4.2 is invoked with the luma location ( xC, yC ), the luma location ( xB, yB ) set equal to ( 0, 0 ), the size of the coding unit nCSL, the width of the luma prediction samples nPSW set equal to nCS1L, the height of the luma prediction samples nPSH set equal to nCSL and a partition index PartIdx set equal to 0 as inputs, and the outputs are a (nCSL)x(nCSL) array predSamplesL and two (nCSC)x(nCSC) arrays predSamplesCb and predSamplesCr.
2. The decoding process for prediction units in inter prediction mode as specified in subclause 8.4.2 is invoked with the luma location ( xC, yC ), the luma location ( xB, yB ) set equal to ( nCS1L, 0 ), the size of the coding unit nCSL, the width of the luma prediction samples nPSW set equal to nCS1L, the height of the luma prediction samples nPSH set equal to nCSL and a partition index PartIdx set equal to 1 as inputs, and the outputs are a (nCSL)x(nCSL) array predSamplesL and two (nCSC)x(nCSC) arrays predSamplesCb and predSamplesCr.

Otherwise, if PartMode is equal to PART\_NxN, the following ordered steps apply:

1. The decoding process for prediction units in inter prediction mode as specified in subclause 8.4.2 is invoked with the luma location ( xC, yC ), the luma location ( xB, yB ) set equal to ( 0, 0 ), the size of the coding unit nCSL, the width of the luma prediction samples nPSW set equal to nCS1L, the height of the luma prediction samples nPSH set equal to nCS1L as and a partition index PartIdx set equal to 0 inputs, and the outputs are a (nCSL)x(nCSL) array predSamplesL and two (nCSC)x(nCSC) arrays predSamplesCb and predSamplesCr.
2. The decoding process for prediction units in inter prediction mode as specified in subclause 8.4.2 is invoked with the luma location ( xC, yC ), the luma location ( xB, yB ) set equal to ( nCS1L, 0 ), the size of the coding unit nCSL, the width of the luma prediction samples nPSW set equal to nCS1L, the height of the luma prediction samples nPSH set equal to nCS1L and a partition index PartIdx set equal to 1 as inputs, and the outputs are a (nCSL)x(nCSL) array predSamplesL and two (nCSC)x(nCSC) arrays predSamplesCb and predSamplesCr.
3. The decoding process for prediction units in inter prediction mode as specified in subclause 8.4.2 is invoked with the luma location ( xC, yC ), the luma location ( xB, yB ) set equal to ( 0, nCS1L ), the size of the coding unit nCSL, the width of the luma prediction samples nPSW set equal to nCS1L, the height of the luma prediction samples nPSH set equal to nCS1L and a partition index PartIdx set equal to 2 as inputs, and the outputs are a (nCSL)x(nCSL) array predSamplesL and two (nCSC)x(nCSC) arrays predSamplesCb and predSamplesCr.
4. The decoding process for inter prediction units as specified in subclause 8.4.2 is invoked with the luma location ( xC, yC ), the luma location ( xB, yB ) set equal to ( nCS1L, nCS1L ), the size of the coding unit nCSL, the width of the luma prediction samples nPSW set equal to nCS1L, the height of the luma prediction samples nPSH set equal to nCS1L and a partition index PartIdx set equal to 3 as inputs, and the outputs are a (nCSL)x(nCSL) array predSamplesL and two (nCSC)x(nCSC) arrays predSamplesCb and predSamplesCr.

### Decoding process for prediction units in inter prediction mode

Inputs to this process are:

– a luma location ( xC, yC ) specifying the top-left luma sample of the current coding unit relative to the top left luma sample of the current picture,

– a luma location ( xB, yB ) specifying the top-left luma sample of the current prediction unit relative to the top left luma sample of the current coding unit,

– a variable nCS specifying the size of the current coding unit,

– a variable nPSW specifying the width of the current prediction unit,

– a variable nPSH specifying the width of the current prediction unit,

* a variable PartIdx specifying the index of the current prediction unit within the current coding unit.

Outputs of this process are:

– a (nCSL)x(nCSL) array predSamplesL of luma prediction samples, where nCSL is derived as specified below,

– a (nCSC)x(nCSC) array predSamplesCb of chroma prediction samples for the component Cb, where nCSC is derived as specified below,

– a (nCSC)x(nCSC) array predSamplesCr of chroma prediction samples for the component Cr, where nCSC is derived as specified below.

The variable nCSL is set equal to nCS and the variable nCSC is set equal to nCS >> 1. [Ed: (WJ) revisit for supporting other chroma formats]

The decoding process for prediction units in inter prediction mode consists of the following ordered steps:

1. Derivation process for motion vector components and reference indices as specified in subclause 8.4.2.1.

Inputs to this process are

– a luma location ( xC, yC ) of the top-left luma sample of the current coding unit relative to the top-left luma sample of the current picture,

– a luma location ( xB, yB ) of the top-left luma sample of the current prediction unit relative to the top-left luma sample of the current coding unit,

– variables specifying the width and the height of the prediction unit for luma, nPSW and nPSH,

* + a variable PartIdx specifying the index of the current prediction unit within the current coding unit.

Outputs of this process are

– luma motion vectors mvL0 and mvL1, and chroma motion vectors mvCL0 and mvCL1,

– reference indices refIdxL0 and refIdxL1,

– prediction list utilization flags predFlagL0 and predFlagL1.

1. Decoding process for inter sample prediction as specified in subclause 8.4.2.2.

Inputs to this process are

– a luma location ( xC, yC ) of the top-left luma sample of the current coding unit relative to the top-left luma sample of the current picture,

– a luma location ( xB, yB ) of the top-left luma sample of the current prediction unit relative to the top-left luma sample of the current coding unit,

– a variable nCS specifying the size of the current coding unit,

– variables specifying the width and the height of the prediction unit for luma, nPSW and nPSH.

– luma motion vectors mvL0 and mvL1, and chroma motion vectors mvCL0 and mvCL1,

– reference indices refIdxL0 and refIdxL1,

– prediction list utilization flags predFlagL0 and prefFlagL1.

Outputs of this process are

– inter prediction samples (predSamples); which are a (nCSL)x(nCSL) array predSamplesL of prediction luma samples and two (nCSC)x(nCSC) arrays predSamplesCr, and predSamplesCr of prediction chroma samples, one for each of the chroma components Cb and Cr.

For use in derivation processes of variables invoked later in the decoding process, the following assignments are made:

MvL0[ xB, yB ] = mvL0 (8‑46)  
MvL1[ xB, yB ] = mvL1 (8‑47)

RefIdxL0[ xB, yB ] = refIdxL0 (8‑48)  
RefIdxL1[ xB, yB ] = refIdxL1 (8‑49)

PredFlagL0[ xB, yB ] = predFlagL0 (8‑50)  
PredFlagL1[ xB, yB ] = predFlagL1 (8‑51)

#### Derivation process for motion vector components and reference indices

Input to this process are

* a luma location ( xC, yC ) of the top-left luma sample of the current coding unit relative to the top-left luma sample of the current picture,
* a luma location ( xB, yB ) of the top-left luma sample of the current prediction unit relative to the top-left luma sample of the current coding unit,
* variables specifying the width and the height of the prediction unit for luma, nPSW and nPSH,
* a variable PartIdx specifying the index of the current prediction unit within the current coding unit.

Outputs of this process are

* luma motion vectors mvL0 and mvL1 and chroma motion vectors mvCL0 and mvCL1,
* reference indices refIdxL0 and refIdxL1,
* prediction list utilization flags predFlagL0 and predFlagL1.

Let ( xP, yP ) specify the top-left luma sample of the current prediction unit relative to the top-left luma sample of the current picture where xP = xC + xB and yP = yC + yB.

For the derivation of the variables mvL0 and mvL1, refIdxL0 and refIdxL1 as well as PredFlagL0 and PredFlagL1, the following applies.

* If PredMode is equal to MODE\_SKIP, the derivation process for luma motion vectors for merge mode as specified in subclause 8.4.2.1.3 is invoked with the luma location ( xP, yP ), variables nPSW, nPSH and the partition index PartIdx as inputs and the output being the luma motion vectors mvL0, mvL1, the reference indices refIdxL0, refIdxL1, and the prediction list utilization flags predFlagL0 and predFlagL1.
* Otherwise, if PredMode is equal to MODE\_INTER and merge\_flag[ xP ][ yP ] is equal to 1,, the derivation process for luma motion vectors for merge mode as specified in subclause 8.4.2.1.3 is invoked with the luma location ( xP, yP ), variables nPSW and nPSH and the partition index PartIdx as inputs and the outputs being the luma motion vectors mvL0 and mvL1, the reference indices refIdxL0 and refIdxL1, the prediction utilization flags predFlagL0 and predFlagL1.
* Otherwise, for X being replaced by either 0 or 1 in the variables predFlagLX, mvLX, refIdxLX and in Pred\_LX and in the syntax elements ref\_idx\_lX and mvd\_lX, the following applies.

1. The variables LcToLx, refIdxLX and predFlagLX are derived as follows.

* If inter\_pred\_flag[ xP ][ yP ] is equal to Pred\_LC and PredLCToPredLx[ ref\_idx\_lc[ xP ][ yP ] ] is equal to Pred\_LX,

refIdxLX = RefIdxLCToRefIdxLx[ ref\_idx\_lc[ xP ][ yP ] ] (8‑52) predFlagLX = 1 (8‑53) mvd\_lX[ xP ][ yP ][ 0 ] = mvd\_lc[ xP ][ yP ][ 0 ] (8‑54)  
 mvd\_lX[ xP ][ yP ][ 1 ] = mvd\_lc[ xP ][ yP ][ 1 ] (8‑55)  
 mvp\_lX\_flag[ xP ][ yP ][ 0 ] = mvp\_lc\_flag[ xP ][ yP ] (8‑56)  
 LcToLx = LX (8‑57)

* Otherwise, if inter\_pred\_flag[ xP ][ yP ] is equal to Pred\_LX or Pred\_BI,

refIdxLX = ref\_idx\_lX[ xP ][ yP ] (8‑58)  
 predFlagLX = 1 (8‑59)

* Otherwise, the variables refIdxLX and predFlagLX are specified by

refIdxLX = -1 (8‑60)  
 predFlagLX = 0 (8‑61)

1. The variable mvdLX is derived as follows.

mvdLX[ 0 ] = mvd\_lX[ xP ][ yP ][ 0 ] (8‑62)  
 mvdLX[ 1 ] = mvd\_lX[ xP ][ yP ][ 1 ] (8‑63)

1. When predFlagLX is equal to 1, the variable mvpLX is derived as follows.

* The derivation process for luma motion vector prediction in subclause 8.4.2.1.5 is invoked with the luma location ( xP, yP ), variables nPSW and nPSH and refIdxLX as the inputs and the output being mvpLX.

1. When predFlagLX is equal to 1, the luma motion vector mvLX is derived as

mvLX[ 0 ] = mvpLX[ 0 ] + mvdLX[ 0 ] (8‑64)  
 mvLX[ 1 ] = mvpLX[ 1 ] + mvdLX[ 1 ] (8‑65)

When ChromaArrayType is not equal to 0 and predFlagLX (with X being either 0 or 1) is equal to 1, the derivation process for chroma motion vectors in subclause 8.4.2.1.8 is invoked with mvLX and refIdxLX as inputs and the output being mvCLX.

##### Derivation process for luma motion vectors for merge mode

This process is only invoked when PredMode is equal to MODE\_SKIP or PredMode is equal to MODE\_ INTER and merge\_flag [ xP ][ yP ] is equal to 1, where ( xP, yP ) specify the top-left luma sample of the current prediction unit relative to the top-left luma sample of the current picture.

Inputs of this process are

* a luma location ( xP, yP ) of the top-left luma sample of the current prediction unit relative to the top-left luma sample of the current picture,
* variables specifying the width and the height of the prediction unit for luma, nPSW and nPSH,
* a variable PartIdx specifying the index of the current prediction unit within the current coding unit.

Outputs of this process are

– the luma motion vectors mvL0 and mvL1,

– the reference indices refIdxL0 and refIdxL1,

– the prediction list utilization flags predFlagL0 and predFlagL1.

The motion vectors mvL0 and mvL1, the reference indices refIdxL0 and refIdxL1, and the prediction utilization flags predFlagL0 and predFlagL1 are derived as specified by the following ordered steps:

1. The derivation process for merging candidates from neighboring prediction unit partitions in subclause 8.4.2.1.2 is invoked with luma location ( xP, yP ), the width and the height of the prediction unit nPSW and nPSH and the partition index PartIdx as inputs and the output is assigned to the availability flags availableFlagN, the reference indices refIdxL0N and refIdxL1N, the prediction list utilization flags predFlagL0N and predFlagL1N and the motion vectors mvL0N and mvL1N with N being replaced by A0, A1, B0, B1 or B2.
2. The reference index for temporal merging candidate is derived as follows..

* If the prediction unit covering luma location ( xP − 1, yP + nPSH − 1 ) is available and PredMode is not MODE\_INTRA, refIdxLX is set equal to refIdxLX[ xP − 1, yP + nPSH − 1 ].
* Otherwise, refIdxLX is set equal to 0.

1. The derivation process for temporal luma motion vector prediction in subclause 8.4.2.1.9 is invoked with luma location ( xP, yP ), refIdxLX as the inputs and with the output being the availability flag availableFlagLXCol and the temporal motion vector mvLXCol. The variables availableFlagCol and predFlagLXCol (with X being 0 or 1, respectively) are derived as specified below.

availableFlagCol = availableFlagL0Col || availableFlagL1Col (8‑71)

predFlagLXCol = availableFlagLXCol (8‑72)

1. The merging candidate list, mergeCandList, is constructed as follows.
   1. A1, if availableFlagA1 is equal to 1
   2. B1, if availableFlagB1 is equal to 1
   3. B0, if availableFlagB0 is equal to 1
   4. A0, if availableFlagA0 is equal to 1
   5. B2, if availableFlagB2 is equal to 1
   6. Col, if availableFlagCol is equal to 1
2. When merging candidates have the same motion vectors and the same reference indices, the merging candidates are removed from the list except the merging candidate which has the smallest order in the mergeCandList.
3. The variable numMergeCand and numOrigMergeCand are set to the number of merging candidates in the mergeCandList.
4. When slice\_type is equal to B, the following applies.
   * The derivation process for combined bi-predictive merging candidates specified in subclause 8.4.2.1.4 is invoked with mergeCandList, the reference indices refIdxL0N and refIdxL1N, the prediction list utilization flags predFlagL0N and predFlagL1N, the motion vectors mvL0N and mvL1N of every candidate N being in mergeCandList, numMergeCand and numOrigMergeCand given as input and the output is assigned to mergeCandList, numMergeCand, the reference indices refIdxL0combCandk and refIdxL1combCandk, the prediction list utilization flags predFlagL0combCandk and predFlagL1combCandk and the motion vectors mvL0combCandk and mvL1combCandk of every new candidate combCandk being added in mergeCandList. The number of candidates being added numCombMergeCand is set equal to ( numMergeCand – numOrigMergeCand ). When numCombMergeCand is greater than 0, k ranges from 0 to numCombMergeCand − 1, inclusive.
   * The derivation process for non-scaled bi-predictive merging candidates specified in subclause 8.4.2.1.5 is invoked with mergeCandList, the reference indices refIdxL0N and refIdxL1N, the prediction list utilization flags predFlagL0N and predFlagL1N, the motion vectors mvL0N and mvL1N of every candidate N being in mergeCandList, numMergeCand and numOrigMergeCand given as input and the output is assigned to mergeCandList, numMergeCand, the reference indices refIdxL0nscaleCandl and refIdxL1nscaleCandl, the prediction list utilization flags predFlagL0nscaleCandl and predFlagL1nscaleCandl, the motion vectors mvL0nscaleCandl and mvL1nscaleCandl of every new candidate nscaleCandl being added in mergeCandList. The number of candidates being added numNscaleMergeCand is set equal to ( numMergeCand – numOrigMergeCand – numCombMergeCand ). When numNscaleMergeCand is greater than 0, l ranges from 0 to numNscaleMergeCand – 1, inclusive.
5. The derivation process for zero motion vector merging candidates specified in subclause 8.4.2.1.6 is invoked with the mergeCandList, the reference indices refIdxL0N and refIdxL1N, the prediction list utilization flags predFlagL0N and predFlagL1N, the motion vectors mvL0N and mvL1N of every candidate N being in mergeCandList and the NumMergeCand as the inputs and the output is assigned to mergeCandList, numMergeCand, the reference indices refIdxL0zeroCandm and refIdxL1zeroCandm, the prediction list utilization flags predFlagL0zeroCandm and predFlagL1zeroCandm, the motion vectors mvL0zeroCandm and mvL1zeroCandm of every new candidate zeroCandm being added in mergeCandList. The number of candidates being added numZeroMergeCand is set equal to ( numMergeCand – numOrigMergeCand – numCombMergeCand – numNscaleMergeCand ). When numZeroMergeCand is greater than 0, m ranges from 0 to numZeroMergeCand – 1, inclusive.
6. The following assignments are made with N being the candidate at position merge\_idx[ xP][ yP ] in the merging candidate list mergeCandList ( N = mergeCandList[ merge\_idx[ xP][ yP ] ] ) and X being replaced by 0 or 1:

mvLX[ 0 ] = mvLXN[ 0 ] (8‑88)

mvLX[ 1 ] = mvLXN[ 1 ] (8‑89)

refIdxLX = refIdxLXN (8‑90)

predFlagLX = predFlagLXN (8‑91)

##### Derivation process for spatial merging candidates

Inputs to this process are

* a luma location ( xP, yP ) specifying the top-left luma sample of the current prediction unit relative to the top-left sample of the current picture,
* variables specifying the width and the height of the prediction unit for luma, nPSW and nPSH,
* a variable PartIdx specifying the index of the current prediction unit within the current coding unit.

Outputs of this process are (with N being replaced by A0, A1, B0, B1 or B2 and with X being replaced by 0 or 1)

* the availability flags availableFlagN of the neighbouring prediction units,
* the reference indices refIdxLXN of the neighbouring prediction units,
* the prediction list utilization flags predFlagLXN of the neighbouring prediction units,
* the motion vectors mvLXN of the neighbouring prediction units.

For the derivation of availableFlagN, with N being A0, A1, B0, B1 or B2 and ( xN, yN ) being ( xP – 1,  yP + nPSH ), ( xP − 1,  yP + nPSH − 1 ), ( xP + nPSW,  yP – 1 ), ( xP + nPSW − 1,  yP – 1 ) or ( xP – 1,  yP – 1 ), the following applies.

– If yP-1 is smaller than YLCU, the following applies.

xB0 = (xB0>>Log2MinCUSize)<<Log2MinCUSize) + ((xB0>>Log2MinCUSize)&1)\*(MinPuSize\*2-1) (8‑88)  
xB1 = (xB1>>Log2MinCUSize)<<Log2MinCUSize) + ((xB1>>Log2MinCUSize)&1)\*(MinPuSize\*2-1) (8‑88)  
xB2 = (xB2>>Log2MinCUSize)<<Log2MinCUSize) + ((xB2>>Log2MinCUSize)&1)\*(MinPuSize\*2-1) (8‑88)

where MinPuSize = 1 << ( Log2MinCUSize – 1)

– If one of the following conditions is true with X being replaced by 0 and 1, the availableFlagN is set equal to 0, both components mvLXN are set equal to 0, refIdxLXN and predFlagLX[ xN, yN ] of the prediction unit covering luma location ( xN, yN ) are assigned respectively to mvLXN, refIdxLXN and predFlagLXN.

* N is equal to B2 and availableFlagA0 + availableFlagA1 + availableFlagB0 + availableFlagB1 is equal to 4.
* The prediction unit covering luma location ( xN, yN ) is not available or PredMode is MODE\_INTRA.
* PartMode of the current prediction unit is PART\_2NxN or PART\_2NxnU or PART\_2NxnD and PartIdx is equal to 1 and N is equal to B1
* PartMode of the current prediction unit is PART\_Nx2N or PART\_nLx2N or PART\_nRx2N and PartIdx is equal to 1 and N is equal to A1

– Otherwise, availableFlagN is set equal to 1 and the variables mvLX[ xN, yN ], refIdxLX[ xN, yN ] and predFlagLX[ xN, yN ] of the prediction unit covering luma location ( xN, yN ) are assigned respectively to mvLXN, refIdxLXN and predFlagLXN.

##### Derivation process for combined bi-predictive merging candidates

Inputs of this process are

* a merging candidate list mergeCandList,
* reference indices refIdxL0N and refIdxL1N of every candidate N being in mergeCandList,
* prediction list utilization flags predFlagL0N and predFlagL1N of every candidate N being in mergeCandList,
* motion vectors mvL0N and mvL1N of every candidate N being in mergeCandList,
* the number of elements numMergeCand within mergeCandList,
* the number of elements numOrigMergeCand within the mergeCandList after the spatial and temporal merge candidate derivation process,

Outputs of this process are

* the merging candidate list mergeCandList,
* the number of elements numMergeCand within mergeCandList.
* reference indices refIdxL0combCandk and refIdxL1combCandk of every new candidate combCandk being added in mergeCandList during the invokation of this process,
* prediction list utilization flags predFlagL0combCandk and predFlagL1combCandk of every new candidate combCandk being added in mergeCandList during the invokation of this process,
* motion vectors mvL0combCandk and mvL1combCandk of every new candidate combCandk being added in mergeCandList during the invokation of this process,

When numOrigMergeCand is greater than 1 and less than MaxNumMergeCand, the variable numInputMergeCand is set to numMergeCand, the variables combIdx and combCnt are set to 0, the variable combStop is set to false and the following steps are repeated until combStop is equal to true.

1. The variables l0CandIdx and l1CandIdx are derived using combIdx as specified in Table 8‑11.
2. The following assignments are made with l0Cand being the candidate at position l0CandIdx and l1Cand being the candidate at position l1CandIdx in the merging candidate list mergeCandList ( l0Cand = mergeCandList[ l0CandIdx ] , l1Cand = mergeCandList[ l1CandIdx ] ).
3. When all of the following conditions are true,
   * 1. predFlagL0l0Cand = = 1
     2. predFlagL1l1Cand = = 1
     3. PicOrderCnt( RefPicListL0( refIdxL0l0Cand ) ) != PicOrderCnt( RefPicListL1( refIdxL1l1Cand ) ) | | mvL0l0Cand != mvL1l1Cand

the following applies.

* + 1. The candidate combCandk with k equal to ( numMergeCand − numInputMergeCand ) is added at the end of mergeCandList ( mergeCandList[ numMergeCand ] = combCandk ) and the reference indices, the prediction list utilization flags and the motion vectors of combCandk are dervied as follows and numMergeCand is incremented by 1.

refIdxL0combCandk = refIdxL0l0Cand (8‑92)

refIdxL1combCandk = refIdxL1l1Cand (8‑93)

predFlagL0combCandk = 1 (8‑94)

predFlagL1combCandk = 1 (8‑95)

mvL0combCandk[ 0 ] = mvL0l0Cand[ 0 ] (8‑96)

mvL0combCandk[ 1 ] = mvL0l0Cand[ 1 ] (8‑97)

mvL1combCandk[ 0 ] = mvL1l1Cand[ 0 ] (8‑98)

mvL1combCandk[ 1 ] = mvL1l1Cand[ 1 ] (8‑99)

numMergeCand = numMergeCand + 1 (8‑100)

* + 1. The variable combCnt is incremented by 1.

1. The variable combIdx is incremented by 1.
2. When combIdx is equal to ( numOrigMergeCand \* ( numOrigMergeCand − 1 ) ) or numMergeCand is equal to MaxNumMergeCand or combCnt is equal to 5, combStop is set to true.

Table 8‑10 – Specification of l0CandIdx and l1CandIdx

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **combIdx** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** |
| **l0CandIdx** | 0 | 1 | 0 | 2 | 1 | 2 | 0 | 3 | 1 | 3 | 2 | 3 |
| **l1CandIdx** | 1 | 0 | 2 | 0 | 2 | 1 | 3 | 0 | 3 | 1 | 3 | 2 |

##### Derivation process for non-scaled bi-predictive merging candidates

Inputs of this process are

* a merging candidate list mergeCandList,
* reference indices refIdxL0N and refIdxL1N of every candidate N being in mergeCandList,
* prediction list utilization flags predFlagL0N and predFlagL1N of every candidate N being in mergeCandList,
* motion vectors mvL0N and mvL1N of every candidate N being in mergeCandList,
* the number of elements numMergeCand within mergeCandList,
* the number of elements numOrigMergeCand within the mergeCandList after the spatial and temporal merge candidate derivation process,

Outputs of this process are

* the merging candidate list mergeCandList,
* the number of elements numMergeCand within mergeCandList.
* reference indices refIdxL0nscaleCandl and refIdxL10nscaleCandl of every new candidate nscaleCandl being added in mergeCandList during the invokation of this process,
* prediction list utilization flags predFlagL0nscaleCandl and predFlagL10nscaleCandl of every new candidate nscaleCandl being added in mergeCandList during the invokation of this process,
* motion vectors mvL0nscaleCandl and mvL10nscaleCandl of every new candidate nscaleCandl being added in mergeCandList during the invokation of this process,

When numOrigMergeCand is greater than 0 and numMergeCand is less than MaxNumMergeCand, the variable numInputMergeCand is set to numMergeCand, the variables origIdx and nscaleCnt are set to 0, the variable nscaleStop is set to false and the following steps are repeated until nscaleStop is equal to true.

1. The following assignments are made with origCand being the candidate at position origIdx in the merging candidate list mergeCandList ( origCand = mergeCandList[ origIdx ] ).
2. The following is applied one time with X being 0 and Y being 1. When nscaleCnt is less than 1 and numMergeCand is less than MaxNumMergeCand, it is applied a second time with X being 1 and Y being 0.
   * 1. When all of the following conditions are true,
        + refIdxLXorigCand >= 0
        + PicOrderCnt( RefPicListLX( refIdxLXorigCand ) ) !=   
          PicOrderCnt( RefPicListLY( refIdxLXorigCand ) )
        + abs( PicOrderCntVal − PicOrderCnt( RefPicListLX( refIdxLXorigCand ) ) ) = =   
          abs( PicOrderCntVal − PicOrderCnt( RefPicListLY( refIdxLXorigCand ) ) )
        + refIdxLXorigCand < num\_ref\_idx\_lY\_active\_minus1 + 1

the following applies.

* + - * The candidate nscaleCandl with l equal to ( numMergeCand − numInputMergeCand )is added at the end of mergeCandList ( mergeCandList[ numMergeCand ] = nscaleCandl ) and the reference indices, the prediction list utilization flags and the motion vectors of nscaleCandl are dervied as follows and numMergeCand is incremented by 1.

refIdxL0nscaleCandl = refIdxLXorigCand (8‑92)

refIdxL1nscaleCandl = refIdxLXorigCand (8‑93)

predFlagL0nscaleCandl = 1 (8‑94)

predFlagL1nscaleCandl = 1 (8‑95)

mvL0nscaleCandl[ 0 ] = mvLXorigCand[ 0 ] (8‑96)

mvL0nscaleCandl[ 1 ] = mvLXorigCand[ 1 ] (8‑97)

mvL1nscaleCandl[ 0 ] = − mvLXorigCand[ 0 ] (8‑98)

mvL1nscaleCandl[ 1 ] = − mvLXorigCand[ 1 ] (8‑99)

numMergeCand = numMergeCand + 1 (8‑100)

* + - * The variable nscaleCnt is incremented by 1.

1. The variable origIdx is incremented by 1.
2. When origIdx is equal to numOrigMergeCand or numMergeCand is equal to MaxNumMergeCand or nscaleCnt is equal to 1, nscaleStop is set to true.

##### Derivation process for zero motion vector merging candidates

Inputs of this process are

* a merging candidate list mergeCandList,
* reference indices refIdxL0N and refIdxL1N of every candidate N being in mergeCandList,
* prediction list utilization flags predFlagL0N and predFlagL1N of every candidate N being in mergeCandList,
* motion vectors mvL0N and mvL1N of every candidate N being in mergeCandList,
* the number of elements numMergeCand within mergeCandList,

Outputs of this process are

* the merging candidate list mergeCandList,
* the number of elements numMergeCand within mergeCandList.
* reference indices refIdxL0zeroCandm and refIdxL10zeroCandm of every new candidate zeroCandm being added in mergeCandList during the invokation of this process,
* prediction list utilization flags predFlagL0zeroCandm and predFlagL10zeroCandm of every new candidate zeroCandm being added in mergeCandList during the invokation of this process,
* motion vectors mvL0zeroCandm and mvL10zeroCandm of every new candidate zeroCandm being added in mergeCandList during the invokation of this process,

The variable numRefIdx is derived as follows.

* If slice\_type is equal to P, numRefIdx is set to num\_ref\_idx\_l0\_active\_minus1 + 1
* Otherwise (slice\_type is equal to B), numRefIdx is set to min(num\_ref\_idx\_l0\_active\_minus1 + 1, num\_ref\_idx\_l1\_active\_minus1 + 1)

When numMergeCand is less than MaxNumMergeCand, the variable numInputMergeCand is set to numMergeCand, the variable zeroIdx is set to 0, the variable zeroStop is set to false and the following steps are repeated until zeroStop is equal to true.

1. For the derivation of the reference indices, the prediction list utilization flags and the motion vectors of the zero motion vector merging candidate, the following applies.
   * 1. If (slice\_type is equal to P), the candidate zeroCandm with m equal to ( numMergeCand − numInputMergeCand ) is added at the end of mergeCandList ( mergeCandList[ numMergeCand ] = zeroCandm ) and the reference indices, the prediction list utilization flags and the motion vectors of zeroCandm are dervied as follows and numMergeCand is incremented by 1.

refIdxL0zeroCandm = zeroIdx (8‑110)

refIdxL1zeroCandm = −1 (8‑111)

predFlagL0zeroCandm = 1 (8‑112)

predFlagL1zeroCandm = 0 (8‑113)

mvL0zeroCandm[ 0 ] = 0 (8‑114)

mvL0zeroCandm[ 1 ] = 0 (8‑115)

mvL1zeroCandm[ 0 ] = 0 (8‑116)

mvL1zeroCandm[ 1 ] = 0 (8‑117)

numMergeCand = numMergeCand + 1 (8‑118)

* + 1. Otherwise (slice\_type is equal to B), the candidate zeroCandm with m equal to ( numMergeCand − numInputMergeCand ) is added at the end of mergeCandList ( mergeCandList[ numMergeCand ] = zeroCandm ) and the reference indices, the prediction list utilization flags and the motion vectors of zeroCandm are dervied as follows and numMergeCand is incremented by 1.

refIdxL0zeroCandm = zeroIdx (8‑119)

refIdxL1zeroCandm = zeroIdx (8‑120)

predFlagL0zeroCandm = 1 (8‑121)

predFlagL1zeroCandm = 1 (8‑122)

mvL0zeroCandm[ 0 ] = 0 (8‑123)

mvL0zeroCandm[ 1 ] = 0 (8‑124)

mvL1zeroCandm[ 0 ] = 0 (8‑125)

mvL1zeroCandm[ 1 ] = 0 (8‑126)

numMergeCand = numMergeCand + 1 (8‑127)

1. The variable zeroIdx is incremented by 1.
2. When numMergeCand is equal to MaxNumMergeCand or zeroIdx is equal to or greater than numRefIdx, zeroStop is set to true.

##### Derivation process for luma motion vector prediction

Inputs to this process are

* a luma location ( xP, yP ) specifying the top-left luma sample of the current prediction unit relative to the top-left sample of the current picture,
* variables specifying the width and the height of the prediction unit for luma, nPSW and nPSH.
* the reference index of the current prediction unit partition refIdxLX (with X being 0 or 1).

Output of this process is

* the prediction mvpLX of the motion vector mvLX (with X being 0 or 1).

The motion vector predictor mvpLX is derived in the following ordered steps.

1. The derivation process for motion vector predictor candidates from neighboring prediction unit partitions in subclause 8.4.2.1.6 is invoked with luma location ( xP, yP ), the width and the height of the prediction unit nPSW and nPSH, and refIdxLX (with X being 0 or 1, respectively) as inputs and the availability flags availableFlagLXN and the motion vectors mvLXN with N being replaced by A, B as the output.
2. If both availableFlagLXA and availableFlagLXB are equal to 1 and mvLXA is not equal to mvLXB, availableFlagLXCol is set equal to 0, otherwise, the derivation process for temporal luma motion vector prediction in subclause 8.4.2.1.7 is invoked with luma location ( xP, yP ) , the width and the height of the prediction unit nPSW and nPSH, and refIdxLX (with X being 0 or 1, respectively) as the inputs and with the output being the availability flag availableFlagLXCol and the temporal motion vector predictor mvLXCol.
3. The motion vector predictor candidate list, mvpListLX, is constructed as follows.
4. mvLXA, if availableFlagLXA is equal to 1
5. mvLXB, if availableFlagLXB is equal to 1
6. mvLXCol, if availableFlagLXCol is equal to 1
7. When motion vectors have the same value, the motion vectors are removed from the list except the motion vector which has the smallest order in the mvpListLX.
8. When mvpListLX is empty, a zero motion vector is added as follows.

mvpListLX[ 0 ][0] = 0 (8‑128)

mvpListLX[ 0 ][1] = 0 (8‑129)

1. The variable numMVPCandLX is set to the number of elements within the mvpListLX and maxNumMVPCand is set to 2.
2. The motion vector predictor list is modifed to contain exactly maxNumMVPCand motion vector predictor candidates as follows.
   * 1. If numMVPCandLX is less than maxNumMVPCand, the derivation process for zero motion vector predictor candidates specified in subclause 8.4.2.1.10 is invoked with mvpListLX and numMVPCandLX given as input and the output is assigned to mvpListLX and numMVPCandLX.
     2. Otherwise (numMVPCandLX is equal to or greater than maxNumMVPCand), all motion vector predictor candidates mvpListLX[ idx ] with idx greater than maxNumMVPCand − 1 are removed from the list.
3. The motion vector of mvpListLX[ mvp\_lX\_flag[ xP, yP ] ] is assigned to mvpLX.

##### Derivation process for motion vector predictor candidates

Inputs to this process are

* a luma location ( xP, yP ) specifying the top-left luma sample of the current prediction unit relative to the top-left sample of the current picture,
* variables specifying the width and the height of the prediction unit for luma, nPSW and nPSH,
* the reference index of the current prediction unit partition refIdxLX (with X being 0 or 1).

Outputs of this process are (with N being replaced by A, or B)

* the motion vectors mvLXN of the neighbouring prediction units,
* the availability flags availableFlagLXN of the neighbouring prediction units.



Figure 8‑3 – Spatial motion vector neighbours

The variable isScaledFlagLX with X being 0 or 1 is set equal to 0.

The motion vector mvLXA and the availability flag availableFlagLXA are derived in the following ordered steps:

1. Let a set of two sample locations be (xAk, yAk), with k = 0, 1, specifies sample locations with xAk = xP – 1, yA0 = yP + nPSH and yA1 = yA0 - MinPuSize. The set of sample locations ( xAk, yAk ) represent the sample locations immediately to the left side of the left partition boundary and it’s extended line.
2. Let the availability flag availableFlagLXA be initially set equal to 0 and the both components of mvLXA are set equal to 0.
3. If one of the following conditions is true, the variable isScaledFlagLX is set equal to 1.

* the prediction unit covering luma location ( xA0, yA0 ) is available and PredMode is not MODE\_INTRA.
* the prediction unit covering luma location ( xA1, yA1 ) is available and PredMode is not MODE\_INTRA.

1. For ( xAk, yAk ) from ( xA0, yA0 ) to ( xA1, yA1 ) where yA1 = yA0 - MinPuSize, the following applies repeatedly until availableFlagLXA is equal to 1:

* If the prediction unit covering luma location ( xAk,yAk ) is available, PredMode is not MODE\_INTRA, predFlagLX[ xAk ][ yAk ] is equal to 1 and the reference index refIdxLX[ xAk ][ yAk ] is equal to the reference index of the current prediction unit refIdxLX, availableFlagLXA is set equal to 1 and the motion vector mvLXA is set equal to the motion vector mvLX[ xAk ][ yAk ], refIdxA is set equal to refIdxLX[ xAk ][ yAk ] and ListA is set equal to LX.
* Otherwise if the prediction unit covering luma location ( xAk, yAk ) is available, PredMode is not MODE\_INTRA, predFlagLY[ xAk ][ yAk ] (with Y = !X) is equal to 1 and PicOrderCnt( RefPicListLY( refIdxLY[ xAk ][ yAk ] ) ) is equal to PicOrderCnt( RefPicListLX( refIdxLX ) ), availableFlagLXA is set equal to 1, the motion vector mvLXA is set equal to the motion vector mvLY[ xAk ][ yAk ], refIdxA is set equal to refIdxLY[ xAk ][ yAk ] ,  ListA is set equal to LY and mvLXA is set equal to mvLXA.

1. When availableFlagLXA is equal to 0, for ( xAk, yAk ) from ( xA0, yA0 ) to ( xA1, yA1 ) where yA1 = yA0 - MinPuSize, the following applies repeatedly until availableFlagLXA is equal to 1:

* If the prediction unit covering luma location ( xAk, yAk ) is available, PredMode is not MODE\_INTRA, predFlagLX[ xAk ][ yAk ] is equal to 1, availableFlagLXA is set equal to 1, the motion vector mvLXA is set equal to the motion vector mvLX[ xAk ][ yAk ], refIdxA is set equal to refIdxLX[ xAk ][ yAk ], ListA is set equal to LX.
* Otherwise if the prediction unit covering luma location ( xAk, yAk ) is available, PredMode is not MODE\_INTRA, predFlagLY[ xAk ][ yAk ] (with Y = !X) is equal to 1, availableFlagLXA is set equal to 1, the motion vector mvLXA is set equal to the motion vector mvLY[ xAk ][ yAk ], refIdxA is set equal to refIdxLY[ xAk ][ yAk ], ListA is set equal to LY.
* When availableFlagLXA is equal to 1, mvLXA is derived as specified below.

tx = ( 16384 + ( Abs( td ) >> 1 ) ) / td (8‑130)

DistScaleFactor = Clip3( −4096, 4095, ( tb \* tx + 32 ) >> 6 ) (8‑131)

mvLXA = Sign( DistScaleFactor \* mvLXA ) \*    
 ( (Abs( DistScaleFactor \* mvLXA ) + 127 ) >> 8 ) (8‑132)

where td and tb are derived as

td = Clip3( -128, 127, PicOrderCntVal – PicOrderCnt( RefPicListListA( refIdxA ) ) ) (8‑133)

tb = Clip3( -128, 127, PicOrderCntVal – PicOrderCnt( RefPicListLX( refIdxLX ) ) ) (8‑134)

The motion vector mvLXB and the availability flag availableFlagLXB are derived in the following ordered steps:

1. Let a set of three sample location (xBk, yBk), with k = 0,1,2, specifies sample locations with xB0 = xP + nPSW, xB1 = xB0­ MinPuSize , xB2 = xP - MinPuSize and yBk = yP – 1. The set of sample locations ( xBk, yBk ) represent the sample locations immediately to the upper side of the above partition boundary and it’s extended line. [Ed.: (WJ) MinPuSize should be defined somewhere]
2. If yP-1 is smaller than YLCU, the following applies.

xB0 = (xB0>>Log2MinCUSize)<<Log2MinCUSize) + ((xB0>>Log2MinCUSize)&1)\*(MinPuSize\*2-1) (8‑88)  
xB1 = (xB1>>Log2MinCUSize)<<Log2MinCUSize) + ((xB1>>Log2MinCUSize)&1)\*(MinPuSize\*2-1) (8‑88)  
xB2 = (xB2>>Log2MinCUSize)<<Log2MinCUSize) + ((xB2>>Log2MinCUSize)&1)\*(MinPuSize\*2-1) (8‑88)

where MinPuSize = 1 << ( Log2MinCUSize – 1)

1. Let the availability flag availableFlagLXB be initially set equal to 0 and the both components of mvLXB are set equal to 0.
2. For ( xBk, yBk ) from ( xB0, yB0 ) to ( xB2, yB2 ) where xB0 = xP +nPSW, xB1 = xB0 - MinPuSize , and xB2 =  xP - MinPuSize, the following applies repeatedly until availableFlagLXB is equal to 1:

* If the prediction unit covering luma location ( xBk, yBk ) is available, PredMode is not MODE\_INTRA, predFlagLX[ xBk ][ yBk ] is equal to 1, and the reference index refIdxLX[ xBk ][ yBk ] is equal to the reference index of the current prediction unit refIdxLX, availableFlagLXB is set equal to 1 and the motion vector mvLXB is set equal to the motion vector mvLX[ xBk ][ yBk ], refIdxB is set equal to refIdxLX[ xBk ][ yBk ] and ListB is set equal to LX.
* Otherwise if the prediction unit covering luma location ( xBk, yBk ) is available, PredMode is not MODE\_INTRA, predFlagLY[ xBk ][ yBk ] (with Y = !X) is equal to 1 and PicOrderCnt( RefPicListLY( refIdxLY[ xBk ][ yBk ] ) ) is equal to PicOrderCnt( RefPicListLX( refIdxLX ) ), availableFlagLXB is set equal to 1, the motion vector mvLXB is set equal to the motion vector mvLY[ xBk ][ yBk ], refIdxB is set equal to refIdxLY[ xBk ][ yBk ],  and ListB is set equal to LY.

1. When isScaledFlagLX is equal to 0, availableFlagLXB is set equal to 0 and for ( xBk, yBk ) from ( xB0, yB0 ) to ( xB2, yB2 ) where xB0 = xP +nPSW, xB1 = xB0 - MinPuSize , and xB2 =  xP - MinPuSize, the following applies repeatedly until availableFlagLXB is equal to 1:

* If the prediction unit covering luma location ( xBk, yBk ) is available, PredMode is not MODE\_INTRA, predFlagLX[ xBk ][ yBk ] is equal to 1, availableFlagLXB is set equal to 1, the motion vector mvLXB is set equal to the motion vector mvLX[ xBk ][ yBk ], refIdxB is set equal to refIdxLX[ xBk ][ yBk ], ListB is set equal to LX.
* Otherwise if the prediction unit covering luma location ( xBk, yBk ) is available, PredMode is not MODE\_INTRA, predFlagLY[ xBk ][ yBk ] (with Y = !X) is equal to 1, availableFlagLXB is set equal to 1, the motion vector mvLXB is set equal to the motion vector mvLY[ xBk ][ yBk ], refIdxB is set equal to refIdxLY[ xBk ][ yBk ], ListB is set equal to LY.
* When availableFlagLXB is equal to 1 and PicOrderCnt( RefPicListListB( refIdxB ) ) is not equal to PicOrderCnt( RefPicListLX( refIdxLX ) ), mvLXB is derived as specified below.

tx = ( 16384 + ( Abs( td ) >> 1 ) ) / td (8‑135)

DistScaleFactor = Clip3( −4096, 4095, ( tb \* tx + 32 ) >> 6 ) (8‑136)

mvLXB = Sign( DistScaleFactor \* mvLXA ) \*    
 ( (Abs( DistScaleFactor \* mvLXA ) + 127 ) >> 8 ) (8‑137)

where td and tb are derived as

td = Clip3( -128, 127, PicOrderCntVal – PicOrderCnt( RefPicListListB( refIdxB ) ) ) (8‑138)

tb = Clip3( -128, 127, PicOrderCntVal – PicOrderCnt( RefPicListLX( refIdxLX ) ) ) (8‑139)

##### Derivation process for temporal luma motion vector prediction

Inputs to this process are

* a luma location ( xP, yP ) specifying the top-left luma sample of the current prediction unit relative to the top-left sample of the current picture,
* variables specifying the width and the height of the prediction unit for luma, nPSW and nPSH,
* the reference index of the current prediction unit partition refIdxLX (with X being 0 or 1).

Outputs of this process are

* the motion vector prediction mvLXCol,
* the availability flag availableFlagLXCol.

The function RefPicOrderCnt( picX, refIdx, LX ) returns the picture order count PicOrderCntVal of the reference picture with index refIdx from reference picture list LX of the picture picX and is specified as follows.

RefPicOrderCnt( picX, refIdx, LX ) = PicOrderCnt(RefPicListLX( refIdx ) of the picture picX) (8 141)

Depending on the values of slice\_type and collocated\_from\_l0\_flag, the variable colPic, specifying the picture that contains the co-located partition, is derived as follows.

* If slice\_type is equal to B and collocated\_from\_l0\_flag is equal to 0, the variable colPic specifies the picture that contains the co-located partition as specified by RefPicList1[ 0 ].
* Otherwise (slice\_type is equal to B and collocated\_from\_l0\_flag is equal to 1 or slice\_type is equal to P) , the variable colPic specifies the picture that contains the co-located partition as specified by RefPicList0[ 0 ].

Variable colPu and its position ( xPCol, yPCol ) are derived in the following ordered steps:

1. The variable colPu is derived as follows

yPRb = yP + nPSH (8‑141)

* + If ( yP >> Log2MaxCuSize ) is equal to ( yPRb >> Log2MaxCuSize ), the horizontal component of the right-bottom luma position of the current prediction unit is defined by

xPRb = xP + nPSW (8‑140)

and the variable colPu is set as the prediction unit covering the modified position given by ( ( xPRb >> 4 ) << 4, ( yPRb >> 4 ) << 4 ) inside the colPic.

* + Otherwise ( ( yP >> Log2MaxCuSize ) is not equal to ( yPRb >> Log2MaxCuSize ) ), colPu is marked as unavailable.

1. When colPu is coded in an intra prediction mode or colPu is unavailable, the following applies.
   * Central luma position of the current prediction unit is defined by

xPCtr = ( xP + ( nPSW >> 1 ) (8‑142)

yPCtr = ( yP + ( nPSH >> 1 ) (8‑143)

* + The variable colPu is set as the prediction unit covering the modified position given by ( ( xPCtr >> 4 ) << 4, ( yPCtr >> 4 ) << 4 ) inside the colPic.

1. ( xPCol, yPCol ) is set equal to the top-left luma sample of the colPu relative to the top-left luma sample of the colPic.

The variables mvLXCol and availableFlagLXCol are derived as follows.

* If one of the following conditions is true, both components of mvLXCol are set equal to 0 and availableFlagLXCol is set equal to 0.
  + colPu is coded in an intra prediction mode.
  + colPu is unavailable.
  + colPic is marked as “unused for temporal motion vector prediction”.
  + enable\_temporal\_mvp\_flag is equal to 0.
* Otherwise, the motion vector mvCol, the reference index refIdxCol, and the reference list identifier listCol are derived as follows.
  + If PredFlagL0[ xPCol ][ yPCol ] is equal to 0, mvCol, refIdxCol, and listCol are set equal to MvL1[ xPCol ][ yPCol ], RefIdxL1[ xPCol ][ yPCol ], and L1, respectively.
  + Otherwise (PredFlagL0[ xPCol ][ yPCol ] is equal to 1), the following applies.
  + If PredFlagL1[ xPCol ][ yPCol ] is equal to 0, mvCol, refIdxCol, and listCol are set equal to MvL0[ xPCol ][ yPCol ], RefIdxL0[ xPCol ][ yPCol ], and L0, respectively.
  + Otherwise (PredFlagL1[ xPCol ][ yPCol ] is equal to 1), the following assignments are made.
    - * If PicOrderCnt( pic ) of every picture pic in every reference picture lists is less than or equal to PicOrderCntVal, mvCol, refIdxCol, and listCol are set equal to MvLX[ xPCol ][ yPCol ], RefIdxLX[ xPCol ][ yPCol ] and LX, respectively with X being the value of X this process is invoked for.
      * Otherwise (PicOrderCnt( pic ) of at least one picture pic in at least one reference picture list is greater than PicOrderCntVal, mvCol, refIdxCol and listCol are set equal to MvLN[ xPCol ][ yPCol ], RefIdxLN[ xPCol ][ yPCol ] and LN, respectively with N being the value of collocated\_from\_l0\_flag.

and the variable availableFlagLXCol is set equal to 1 and the following applies.

* + If PicOrderCnt( colPic ) – RefPicOrderCnt( colPic, refIdxCol, listCol ) is equal to PicOrderCntVal - PicOrderCnt( RefPicListLX ( refIdxLX ) ),

mvLXCol = mvCol (8‑144)

* + Otherwise, mvLXCol is derived as scaled version of the motion vector mvCol as specified below

tx = ( 16384 + ( Abs( td ) >>1 ) ) / td (8‑145)

DistScaleFactor = Clip3( −4096, 4095, ( tb \* tx + 32 ) >> 6 ) (8‑146)

mvLXCol =  Sign( DistScaleFactor \* mvCol ) \*    
 ( (Abs( DistScaleFactor \* mvCol ) + 127 ) >> 8 ) (8‑147)

where td and tb are derived as

td = Clip3( -128, 127, PicOrderCnt( colPic ) – RefPicOrderCnt( colPic, refIdxCol, listCol ) ) (8‑148)

tb = Clip3( -128, 127, PicOrderCntVal – PicOrderCnt( RefPicListLX ( refIdxLX ) ) ) (8‑149)

##### Derivation process for zero motion vector predictor candidates

Inputs of this process are

* a motion vector predictor list mvpListLX,
* the number of elements numMVPCandLX within mvpListLX.

Outputs of this process are

* the motion vector predictor list mvpListLX,
* the number of elements numMVPCandLX within mvpListLX.

When no motion vector in mvpListLX is equal to (0,0), the zero motion vector predictor candidate is added at the end of mvpListLX and numMVPCandLX is incremented by 1 as follows.

mvpListLX[ numMVPCandLX ][ 0 ] = 0 (8‑150)

mvpListLX[ numMVPCandLX ][ 1 ] = 0 (8‑151)

numMVPCandLX = numMVPCandLX + 1 (8‑152)

##### Derivation process for chroma motion vectors

[Ed.: (WJ) 4:2:0 assumption yet]

Inputs to this process are a luma motion vector mvLX and a reference index refIdLX.

Output of this process is a chroma motion vector mvCLX.

A chroma motion vector is derived from the corresponding luma motion vector.

For the derivation of the chroma motion vector mvCLX, the following applies.

mvCLX[ 0 ] = mvLX[ 0 ] (8‑153)

mvCLX[ 1 ] = mvLX[ 1 ] (8‑154)

#### Decoding process for inter prediction samples

Inputs to this process are:

– a luma location ( xC, yC ) specifying the top-left luma sample of the current coding unit relative to the top left luma sample of the current picture,

– a luma location ( xB, yB ) specifying the top-left luma sample of the current prediction unit relative to the top‑left luma sample of the current coding unit,

– a variable nCS specifying the size of the current coding unit,

– variables specifying the width and the height of the prediction unit, nPSW and nPSH,

– luma motion vectors mvL0 and mvL1, and chroma motion vectors mvCL0 and mvCL1,

– reference indices refIdxL0 and refIdxL1,

– prediction list utilization flags, predFlagL0 and predFlagL1.

Outputs of this process are:

– a (nCSL)x(nCSL) array predSamplesL of luma prediction samples, where nCSL is derived as specified below,

– a (nCSC)x(nCSC) array preSamplesCb of chroma prediction samples for the component Cb, where nCSC is derived as specified below,

– a (nCSC)x(nCSC) array predSamplesCr of chroma residual samples for the component Cr, where nCSC is derived as specified below.

The variable nCSL is set equal to nCS and the variable nCSC is set equal to nCS >> 1. [Ed: (WJ) revisit for supporting other chroma formats]

Let predSamplesL0L and predSamplesL1L be (nPSW)x(nPSH) arrays of predicted luma sample values and predSampleL0Cb, predSampleL1Cb, predSampleL0Cr, and predSampleL1Cr be (nPSW/2)x(nPSH/2) arrays of predicted chroma sample values.

For LX being replaced by either L0 or L1 in the variables predFlagLX, RefPicListX, refIdxLX, refPicLX, and predPartLX, the following is specified.

When predFlagLX is equal to 1, the following applies.

– The reference picture consisting of an ordered two-dimensional array refPicLXL of luma samples and two ordered two-dimensional arrays refPicLXCb and refPicLXCr of chroma samples is derived by invoking the process specified in subclause 8.4.2.2.1 with refIdxLX and RefPicListX given as input.

– The arrays predSamplesLXL, predSamplesLXCb, and predSamplesLXCr are derived by invoking the fractional sample interpolation process specified in subclause 8.4.2.2.2 with the luma locations ( xC, yC ), ( xB, yB ), the width an the height of the current prediction unit nPSW, nPSH, the motion vectors mvLX, mvCLX, and the reference arrays with refPicLXL, refPicLXCb and refPicLXCr given as input.

The array predSampleL of the prediction samples of luma component is derived by invoking the weighted sample prediction process specified in subclause 8.4.2.2.3 with the luma location ( xB, yB ), the width an the height of the current prediction unit nPSW, nPSH, and the sample arrays predSamplesL0L and predSamplesL1L as well as predFlagL0, predFlagL1, refIdxL0, refIdxL1, mvL0, mvL1 and BitDepthY given as input.

For C being replaced by Cb, or Cr, the array predSampleC of the prediction samples of component C is derived by invoking the weighted sample prediction process specified in subclause 8.4.2.2.3 with the chroma location ( xB/2, yB/2 ), the width an the height of the current prediction unit nPSWC set equal to nPSW/2, nPSHC set equal to nPSH/2, and the sample arrays predSamplesL0C and predSamplesL1C as well as predFlagL0, predFlagL1, refIdxL0, refIdxL1, mvL0, mvL1 and BitDepthC given as input.

##### Reference picture selection process

[Ed: (WJ) same as AVC]

##### Fractional sample interpolation process

Inputs to this process are:

– a luma location ( xC, yC ) specifying the top-left luma sample of the current coding unit relative to the top left luma sample of the current picture,

– a luma location ( xB, yB ) specifying the top-left luma sample of the current prediction unit relative to the top left luma sample of the current coding unit,

– the width and height of this prediction unit, nPSW and nPSH, in luma-sample units,

– a luma motion vector mvLX given in quarter-luma-sample units,

– a chroma motion vector mvCLX given in eighth-chroma-sample units,

– the selected reference picture sample arrays refPicLXL, refPicLXCb, and refPicLXCr.

Outputs of this process are:

– a (nPSW)x(nPSH) array predSampleLXL of prediction luma sample values,

– two (nPSW/2)x(nPSH/2) arrays predSampleLXCb, and predSampleLXCr of prediction chroma sample values.

The location ( xP, yP ) given in full-sample units of the upper-left luma samples of the current prediction unit relative to the upper-left luma sample location of the given reference sample arrays is derived by

xP = xC + xB (8‑87)  
yP = yC + yB (8‑88)

Let ( xIntL, yIntL ) be a luma location given in full-sample units and ( xFracL, yFracL ) be an offset given in quarter-sample units. These variables are used only inside this subclause for specifying general fractional-sample locations inside the reference sample arrays refPicLXL, refPicLXCb, and refPicLXCr.

For each luma sample location ( 0 <= xL < nPSW, 0 <= yL < nPSH ) inside the prediction luma sample array predSampleLXL, the corresponding prediction luma sample value predSampleLXL[xL, yL] is derived as follows:

– The variables xIntL, yIntL, xFracL, and yFracL are derived by

xIntL = xP + ( mvLX[ 0 ] >> 2 ) + xL (8‑89)  
yIntL = yP + ( mvLX[ 1 ] >> 2 ) + yL (8‑90)

xFracL = mvLX[ 0 ] & 3 (8‑91)  
yFracL = mvLX[ 1 ] & 3 (8‑92)

– The prediction luma sample value predSampleLXL[ xL, yL ] is derived by invoking the process specified in subclause 8.4.2.2.2.1 with ( xIntL, yIntL ), ( xFracL, yFracL ) and refPicLXL given as input.

Let ( xIntC, yIntC ) be a chroma location given in full-sample units and ( xFracC, yFracC ) be an offset given in one-eighth sample units. These variables are used only inside this subclause for specifying general fractional-sample locations inside the reference sample arrays refPicLXCb and refPicLXCr.

For each chroma sample location ( 0 <= xC < nPSW/2, 0 <= yC < nPSH/2) inside the prediction chroma sample arrays predSampleLXCb and predSampleLXCr, the corresponding prediction chroma sample values predSampleLXCb[ xC, yC ] and predSampleLXCr[ xC, yC ] are derived asfollows:

– The variables xIntC, yIntC, xFracC, and yFracC are derived by

xIntC = ( xP / 2 ) + ( mvCLX[ 0 ] >> 3 ) + xC (8‑93)  
yIntC = ( yP / 2 ) + ( mvCLX[ 1 ] >> 3 ) + yC (8‑94)

xFracC = mvLX[ 0 ] & 7 (8‑95)  
yFracC = mvLX[ 1 ] & 7 (8‑96)

– The prediction sample value predSampleLXCb[ xC, yC ] is derived by invoking the process specified in subclause 8.4.2.2.2.2 with ( xIntC, yIntC ), ( xFracC, yFracC ) and refPicLXCb given as input.

– The prediction sample value predSampleLXCr[ xC, yC ] is derived by invoking the process specified in subclause 8.4.2.2.2.2 with ( xIntC, yIntC ), ( xFracC, yFracC ) and refPicLXCr given as input.

###### Luma sample interpolation process

Inputs to this process are:

– a luma location in full-sample units ( xIntL, yIntL ),

– a luma location in fractional-sample units ( xFracL, yFracL ),

– the luma reference sample array refPicLXL.

Output of this process is a predicted luma sample value predSampleLXL[ xL, yL ]



Figure 8‑4 – Integer samples (shaded blocks with upper-case letters) and fractional sample positions (un-shaded blocks with lower-case letters) for quarter sample luma interpolation

In Figure 8‑6, the positions labelled with upper-case letters Ai, j within shaded blocks represent luma samples at full-sample locations inside the given two-dimensional array refPicLXL of luma samples. These samples may be used for generating the predicted luma sample value predSampleLXL[ xL, yL ]. The locations ( xAi, j, yAi, j ) for each of the corresponding luma samples Ai, j inside the given array refPicLXL of luma samples are derived as follows:

xAi, j = Clip3( 0, PicWidthInSamplesL – 1, xIntL +i ) (8‑97)  
yAi, j = Clip3( 0, PicHeightInSamplesL – 1, yIntL +j ) (8‑98)

Variables shift1, shift2 and shift3 are derived as follows.

– The variable shift1 is set equal to BitDepthY – 8, the variable shift2 is set equal to 6, and the variable shift3 is set equal to 14 – BitDepthY.

Given the luma samples Ai, j at full-sample locations ( xAi, j, yAi, j ), the luma samples ‘a0,0’ to ‘r0,0’ at fractional sample positions are derived by the following rules.

– The samples labelled a0,0, b0,0, c0,0, d0,0, h0,0, and n0,0 shall be derived by applying the 8-tap filter to the nearest integer position samples:

a0,0 = ( −A−3,0 + 4\*A−2,0 − 10\*A−1,0 + 58\*A0,0 +  17\*A1,0 − 5\*A2,0 + A3,0 ) >> shift1 (8‑99)

b0,0 = ( −A−3,0 + 4\*A−2,0 − 11\*A−1,0 + 40\*A0,0 +  40\*A1,0 − 11\*A2,0 + 4\*A3,0 − A4,0 ) >> shift1 (8‑100)

c0,0 = ( A−2,0 − 5\*A−1,0 + 17\*A0,0 +  58\*A1,0 − 10\*A2,0 + 4\*A3,0 − A4,0 ) >> shift1 (8‑101)

d0,0 = ( −A0,−3 + 4\*A0,−2 − 10\*A0,−1 + 58\*A0,0 +  17\*A0,1 − 5\*A0,2 + A0,3 ) >> shift1 (8‑102)

h0,0 = ( −A0,−3 + 4\*A0,−2 − 11\*A0,−1 + 40\*A0,0 +  40\*A0,1 − 11\*A0,2 + 4\*A0,3 − A0,4 ) >> shift1 (8‑103)

n0,0 = ( A0,−2 − 5\*A0,−1 + 17\*A0,0 +  58\*A0,1 − 10\*A0,2 + 4\*A0,3 − A0,4 ) >> shift1 (8‑104)

– The samples labelled e0,0, f0,0, g0,0, i0,0, j0,0, k0,0, p0,0, q0,0 and r0,0 shall be derived by applying the 8-tap filter to the samples a0,i, b0,i and c0,i where i = −3..4 in vertical direction:

e0,0 = ( −a0,−3 + 4\*a0,−2 − 10\*a0,−1 + 58\*a0,0 +  17\*a0,1 − 5\*a0,2 + a0,3 ) >> shift2 (8‑108)

f0,0 = ( −a0,−3 + 4\*a0,−2 − 11\*a0,−1 + 40\*a0,0 +  40\*a0,1 − 11\*a0,2 + 4\*a0,3 − a0,4 ) >> shift2 (8‑109)

g0,0 = ( a0,−2 − 5\*a0,−1 + 17\*a0,0 +  58\*a0,1 − 10\*a0,2 + 4\*a0,3 − a0,4 ) >> shift2 (8‑110)

i0,0 = ( −b0,−3 + 4\*b0,−2 − 10\*b0,−1 + 58\*b0,0 +  17\*b0,1 − 5\*b0,2 + b0,3 ) >> shift2 (8‑111)

j0,0 = ( −b0,−3 + 4\*b0,−2 − 11\*b0,−1 + 40\*b0,0 +  40\*b0,1 − 11\*b0,2 + 4\*b0,3 − b0,4 ) >> shift2 (8‑112)

k0,0 = ( b0,−2 − 5\*b0,−1 + 17\*b0,0 +  58\*b0,1 − 10\*b0,2 + 4\*b0,3 − b0,4 ) >> shift2 (8‑113)

p0,0 = ( −c0,−3 + 4\*c0,−2 − 10\*c0,−1 + 58\*c0,0 +  17\*c0,1 − 5\*c0,2 + c0,3 ) >> shift2 (8‑114)

q0,0 = ( −c0,−3 + 4\*c0,−2 − 11\*c0,−1 + 40\*c0,0 +  40\*c0,1 − 11\*c0,2 + 4\*c0,3 − c0,4 ) >> shift2 (8‑115)

r0,0 = ( c0,−2 − 5\*c0,−1 + 17\*c0,0 +  58\*c0,1 − 10\*c0,2 + 4\*c0,3 − c0,4 ) >> shift2 (8‑116)

The positions labelled with lower-case letters within un-shaded blocks represent luma samples at quarter-pel sample fractional locations. The luma location offset in fractional-sample units ( xFracL, yFracL ) specifies which of the generated luma samples at full-sample and fractional-sample locations is assigned to the predicted luma sample value predSampleLXL[ xL, yL ]. This assignment is done according to Table 8‑9. The value of predSampleLXL[ xL, yL ] shall be the output.

Table 8‑11 – Assignment of the luma prediction sample predSampleLXL[ xL, yL ]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **xFracL** | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 |
| **yFracL** | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 |
| **predSampleLXL[ xL, yL ]** | A << shift3 | d | h | n | a | e | i | p | b | f | j | q | c | g | k | r |

###### Chroma sample interpolation process

Inputs to this process are:

– a chroma location in full-sample units ( xIntC, yIntC ),

– a chroma location in fractional-sample units ( xFracC, yFracC ),

– the chroma reference sample array refPicLXC.

Output of this process is a predicted chroma sample value predSampleLXC[ xC, yC ]



Figure 8‑5 – Integer samples (shaded blocks with upper-case letters) and fractional sample positions (un-shaded blocks with lower-case letters) for eighth sample chroma interpolation

In Figure 8‑5, the positions labelled with upper-case letters Bi, j within shaded blocks represent chroma samples at full-sample locations inside the given two-dimensional array refPicLXC of chroma samples. These samples may be used for generating the predicted chroma sample value predSampleLXC[ xC, yC ]. The locations ( xBi, j, yBi, j ) for each of the corresponding chroma samples Bi, j inside the given array refPicLXC of chroma samples are derived as follows:

xBi, j = Clip3( 0, PicWidthInSamplesC – 1, xIntC +i ) (8‑117)  
yBi, j = Clip3( 0, PicHeightInSamplesC – 1, yIntC +j ) (8‑118)

Variables shift1, shift2 and shift3 are derived as follows.

– The variable shift1 is set equal to BitDepthC – 8, the variable shift2 is set equal to 6, and the variable shift3 is set equal to 14 – BitDepthC.

Given the chroma samples Bi, j at full-sample locations ( xBi, j, yBi, j ), the chroma samples ‘ab0,0’ to ‘hh0,0’ at fractional sample positions are derived by the following rules.

– The samples labelled ab0,0, ac0,0, ad0,0, ae0,0, af0,0, ag0,0, and ah0,0 shall be derived by applying the 4-tap filter to the nearest integer position samples:

ab0,0 = ( −2\*B−1,0 + 58\*B0,0 + 10\*B1,0 – 2\*B2,0 ) >> shift1 (8‑119)

ac0,0 = ( −4\*B−1,0 + 54\*B0,0 + 16\*B1,0 – 2\*B2,0 ) >> shift1 (8‑120)

ad0,0 = ( −6\*B−1,0 + 46\*B0,0 + 28\*B1,0 – 4\*B2,0 ) >> shift1 (8‑121)

ae0,0 = ( −4\*B−1,0 + 36\*B0,0 + 36\*B1,0 – 4\*B2,0 ) >> shift1 (8‑122)

af0,0 = ( −4\*B−1,0 + 28\*B0,0 + 46\*B1,0 – 6\*B2,0 ) >> shift1 (8‑123)

ag0,0 = ( −2\*B−1,0 + 16\*B0,0 + 54\*B1,0 – 4\*B2,0 ) >> shift1 (8‑124)

ah0,0 = ( −2\*B−1,0 + 10\*B0,0 + 58\*B1,0 – 2\*B2,0 ) >> shift1 (8‑125)

– The samples labelled ba0,0, ca0,0, da0,0, ea0,0, fa0,0, ga0,0, and ha0,0 shall be derived by applying the 4-tap filter to the nearest integer position samples:

ba0,0 = ( −2\*B0,−1 + 58\*B0,0 + 10\*B0,1 – 2\*B0,2 ) >> shift1 (8‑126)

ca0,0 = ( −4\*B0,−1 + 54\*B0,0 + 16\*B0,1 – 2\*B0,2 ) >> shift1 (8‑127)

da0,0 = ( −6\*B0,−1 + 46\*B0,0 + 28\*B0,1 – 4\*B0,2 ) >> shift1 (8‑128)

ea0,0 = ( −4\*B0,−1 + 36\*B0,0 + 36\*B0,1 – 4\*B0,2 ) >> shift1 (8‑129)

fa0,0 = ( −4\*B0,−1 + 28\*B0,0 + 46\*B0,1 – 6\*B0,2 ) >> shift1 (8‑130)

ga0,0 = ( −2\*B0,−1 + 16\*B0,0 + 54\*B0,1 – 4\*B0,2 ) >> shift1 (8‑131)

ha0,0 = ( −2\*B0,−1 + 10\*B0,0 + 58\*B0,1 – 2\*B0,2 ) >> shift1 (8‑132)

– The samples labelled bX0,0, cX0,0, dX0,0, eX0,0, fX0,0, gX0,0 and hX0,0 for X being replaced by b, c, d, e, f, g and h, respectively, shall be derived by applying the 4-tap filter to the intermediate values aX0,i where i = −1..2 in vertical direction:

bX0,0 = ( −2\*aX0,−1 + 58\*aX0,0 + 10\*aX0,1 – 2\*aX0,2 ) >> shift2 (8‑140)

cX0,0 = ( −4\*aX0,−1 + 54\*aX0,0 + 16\*aX0,1 – 2\*aX0,2 ) >> shift2 (8‑141)

dX0,0 = ( −6\*aX0,−1 + 46\*aX0,0 + 28\*aX0,1 – 4\*aX0,2 ) >> shift2 (8‑142)

eX0,0 = ( −4\*aX0,−1 + 36\*aX0,0 + 36\*aX0,1 – 4\*aX0,2 ) >> shift2 (8‑143)

fX0,0 = ( −4\*aX0,−1 + 28\*aX0,0 + 46\*aX0,1 – 6\*aX0,2 ) >> shift2 (8‑144)

gX0,0 = ( −2\*aX0,−1 + 16\*aX0,0 + 54\*aX0,1 – 4\*aX0,2 ) >> shift2 (8‑145)

hX0,0 = ( −2\*aX0,−1 + 10\*aX0,0 + 58\*aX0,1 – 2\*aX0,2 ) >> shift2 (8‑146)

The positions labelled with lower-case letters within un-shaded blocks represent chroma samples at eighth-pel sample fractional locations. The chroma location offset in fractional-sample units ( xFracC, yFracC ) specifies which of the generated chroma samples at full-sample and fractional-sample locations is assigned to the predicted chroma sample value predSampleLXC[ xC, yC ]. This assignment is done according to Table 8‑10. The value of predSampleLXC[ xC, yC ] shall be the output.

Table 8‑12 – Assignment of the chroma prediction sample predSampleLXC[ xC, yC ] for ( X, Y ) being replaced by ( 1, b ), ( 2, c ), ( 3, d ), ( 4, e ), ( 5, f ), ( 6, g ), and ( 7, h ), respectively

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **xFracC** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | X | X | X | X | X | X | X | X |
| **yFracC** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| **predSampleLXC[ xC, yC ]** | B << shift3 | ba | ca | da | ea | fa | ga | ha | aY | bY | cY | dY | eY | fY | gY | hY |

##### Weighted sample prediction process

Inputs to this process are:

– a location ( xB, yB ) specifying the top-left sample of the current prediction unit relative to the top left sample of the current coding unit,

– the width and height of this prediction unit, nPSW and nPSH,

– two (nPSW)x(nPSH) arrays predSamplesL0 and predSamplesL1,

– prediction list utilization flags, predFlagL0 and predFlagL1,

– reference indices, refIdxL0 and refIdxL1,

– motion vectors, mvL0 and mvL1,

– the bit-depth of the chroma component, bitDepth.

Outputs of this process are:

– the (nPSW)x(nPSH) array predSamples of prediction sample values.

Variables shift1, shift2, offset1 and offset2 are derived as follows.

– The variable shift1 is set equal to 14 – bitDepth and the variable shift2 is set equal to 15 – bitDepth,

– The variable offset1 is set equal to 1 << ( shift1 – 1 ) and the variable offset2 is set equal to 1 << ( shift2 – 1 ).

In P slices, if the value of predFlagL0 is equal to 1, the following applies.

– If weighted\_pred\_flag is equal to 0, the default weighted sample prediction process as described in subclause 8.4.2.2.3.1 is invoked with the same inputs and outputs as the process described in this subclause.

– Otherwise (weighted\_pred\_flag is equal to 1), the explicit weighted sample prediction process as described in subclause 8.4.2.2.3.2 is invoked with the same inputs and outputs as the process described in this subclause.

In B slices, if predFlagL0 or predFlagL1 is equal to 1, the following applies.

– If weighted\_bipred\_idc is equal to 0, the default weighted sample prediction process as described in subclause 8.4.2.2.3.1 is invoked with the same inputs and outputs as the process described in this subclause.

– Otherwise, if weighted\_bipred\_idc is equal to 1 and if predFlagL0 or predFlagL1 equal to 1, the explicit weighted sample prediction process as described in subclause 8.4.2.2.3.2 is invoked with the same inputs and outputs as the process described in this subclause.

– Otherwise (weighted\_bipred\_idc is equal to 2), the following applies.

– If predFlagL0 is equal to 1 and predFlagL1 is equal to 1, the implicit weighted sample prediction process as described in subclause 8.4.2.2.3.2 is invoked with the same inputs and outputs as the process described in this subclause.

– Otherwise (predFlagL0 or predFlagL1 are equal to 1 but not both), the default weighted sample prediction process as described in subclause 8.4.2.2.3.1 is invoked with the same inputs and outputs as the process described in this subclause.

###### Default weighted sample prediction process

Inputs to this process are:

– the same as specified in subclause 8.4.2.2.3.

Outputs of this process are:

– the same as specified in subclause 8.4.2.2.3.

Depending on the value of predFlagL0 and predFlagL1, the prediction samples predSamples[ x, y ] with x = 0..(nPSW)-1 and y = 0..(nPSH)-1 are derived as follows.

– If predFlagL0 is equal to 1 and predFlagL1 is equal to 0,

predSamples[ x, y ] = Clip3( 0, ( 1 << bitDepth ) – 1, ( predSamplesL0[ x, y ] + offset1 ) >> shift1 ) (8‑147)

– Otherwise, if predFlagL0 is equal to 0 and predFlagL1 is equal to 1,

predSamples[ x, y ] = Clip3( 0, ( 1 << bitDepth ) – 1, ( predSamplesL1[ x, y ] + offset1 ) >> shift1 ) (8‑148)

– Otherwise (both predFlagL0 and predFlagL1 are equal to 1), if RefPicOrderCnt( currPic, refIdxL0, L0) is equal to RefPicOrderCnt( currPic, refIdxL1, L1) and mvL0 is equal to mvL1,

predSamples[ x, y ] = Clip3( 0, ( 1 << bitDepth ) – 1, ( predSamplesL0[ x, y ] + offset1 ) >> shift1 ) (8‑147)

– Otherwise,

predSamples[ x, y ] = Clip3( 0, ( 1 << bitDepth ) – 1 ,   
 ( predSamplesL0[ x, y ] + predSamplesL1[ x, y ] + offset2 ) >> shift2 ) (8‑149)

###### Weighted sample prediction process

Inputs to this process are:

– the same as specified in subclause 8.4.2.2.3.

– variables for weighted prediction logWDC, logWDC, w0C, w1C, o0C, o1C with C being replaced by L for luma samples and, when chroma\_format\_idc is not equal to 0, Cb and Cr for chroma samples.

Outputs of this process are:

– the same as specified in subclause 8.4.2.2.3.

The prediction sample predSamples[ x, y ] with x = 0..(nPSW)-1 and y = 0..(nPSH)-1, with H being replaced by Y for luma samples and by C for chroma samples are derived as follows:

– If the predFlagL0 is equal to 1 and predFlagL1 is equal to 0, the prediction samples are derived as follows:

if( logWDC >= 1 )   
 predSamples[ x, y ] = Clip1H( ( ( predSamplesL0 [ x, y ] \* w0C + 2logWDC − 1 ) >> logWDC ) + o0C ) (8‑189)  
else  
 predSamples[ x, y ] = Clip1H( predSamplesL0 [ x, y ] \* w0C + o0C ) (8‑190)

– Otherwise, if the predFlagL0 is equal to 0 and predFlagL1 is equal to 1, the final predicted sample values predSamples [ x, y ] are derived by

if( logWDC >= 1 )  
 predSamples[ x, y ] = Clip1H( ( ( predSamplesL1 [ x, y ] \* w1C + 2logWDC − 1 ) >> logWDC ) + o1C ) (8‑191)  
else  
 predSamples[ x, y ] = Clip1H( predSamplesL1 [ x, y ] \* w1C + o1C ) (8‑192)

– Otherwise (both predFlagL0 and predFlagL1 are equal to 1), if RefPicOrderCnt( currPic, refIdxL0, L0) is equal to RefPicOrderCnt( currPic, refIdxL1, L1) and mvL0 is equal to mvL1, the final predicted sample values predSamples [ x, y ] are derived by

predSamples[ x, y ] = Clip1H( ( predSamplesL0 [ x, y ] \* ( w0C + w1C ) +   
 ( ( o0C + o1C + 1 ) << logWDC ) ) >> ( logWDC + 1 ) ) (8‑193)

– Otherwise, the final predicted sample values predSamples[ x, y ] are derived by

predSamples[ x, y ] = Clip1H( ( predSamplesL0 [ x, y ] \* w0C + predSamplesL1 [ x, y ] \* w1C +   
 ( ( o0C + o1C + 1 ) << logWDC ) ) >> ( logWDC + 1 ) ) (8‑193)

Where the variables logWDC, o0C, o1C, and w0C, w1C are derived as follows.

– If weighted\_bipred\_idc is equal to 2 in B-slices, implicit mode weighted prediction is used as follows:

logWDc = 5+shift1 (8‑194)

o0C = 0 (8‑195)

o1C = 0 (8‑196)

The variable WeightScaleFactor is derived from the values currPoc, refIdxL0 and refIdxL1 as follows:

tb = Clip3( −128, 127, PicOrderCntVal – PicOrderCnt( RefPicListL0( refIdxL0 ) ) ) (8‑197)

td = Clip3( −128, 127, PicOrderCnt( RefPicListL1( refIdxL1 ) )   
  – PicOrderCnt( RefPicListL0( refIdxL0 ) ) ) (8‑198)

tx = ( 16384 + ( Abs( td ) >> 1 ) ) / td (8‑199)

WeightScaleFactor = Clip3( −1024, 1023, ( tb \* tx + 32 ) >> 6 ) (8‑200)

The variables w0C and w1C are derived as follows.

– If PicOrderCnt( RefPicListL0( refIdxL0 ) ) is equal to PicOrderCnt( RefPicListL1( refIdxL1 ) ) or ( WeightScaleFactor >> 2 ) < −64 or ( WeightScaleFactor >> 2 ) > 128, the following applies.

w0C=32 (8‑199)

w1C=32 (8‑201)

– Otherwise;

w0C = 64 – (WeightScaleFactor >> 2) (8‑202)

w1C = WeightScaleFactor >> 2 (8‑203)

– Otherwise, if both weighted\_bipred\_idc and ref\_pic\_list\_combination\_flag are equal to 1 in B slice, combined explicit mode weighted prediction is used as follows:

– If C is equal to L for luma samples,

logWDc = luma\_log2\_weight\_denom+ shift1 (8‑204)

w0C = LumaWeightLC[ RefIdxLCToRefIdxLx[ refIdxL0 ] ] (8‑205)

w1C = LumaWeightLC[ RefIdxLCToRefIdxLx[ refIdxL1 ] ] (8‑206)

o0C = luma\_offset\_lc[ RefIdxLCToRefIdxLx[ refIdxL0 ] ] \* ( 1 << ( BitDepthY – 8 ) ) (8‑207)

o1C = luma\_offset\_lc[ RefIdxLCToRefIdxLx[ refIdxL1 ] ] \* ( 1 << ( BitDepthY – 8 ) ) (8‑208)

– Otherwise (C is equal to Cb or Cr for chroma samples, with iCbCr = 0 for Cb, iCbCr = 1 for Cr),

logWDc = ChromaLog2WeightDenom + shift1 (8‑209)

w0C = ChromaWeightLC[ RefIdxLCToRefIdxLx[ refIdxL0 ] ][ iCbCr ] (8‑210)

w1C = ChromaWeightLC[ RefIdxLCToRefIdxLx[ refIdxL1 ] ][ iCbCr ] (8‑211)

o0C = ChromaOffsetLC[ RefIdxLCToRefIdxLx[ refIdxL0 ] ][ iCbCr ] \* ( 1 << ( BitDepthC – 8 ) ) (8‑212)

o1C = ChromaOffsetLC[ RefIdxLCToRefIdxLx[ refIdxL1 ] ][ iCbCr ] \* ( 1 << ( BitDepthC – 8 ) ) (8‑213)

– Otherwise (weighted\_pred\_flag is equal to 1 in P slice or ref\_pic\_list\_combination\_flag is equal to 0 in B-slice) explicit mode weighted prediction is used as follows:

– If C is equal to L for luma samples,

logWDc = luma\_log2\_weight\_denom+ shift1 (8‑204)

w0C = LumaWeightL0[refIdxL0] (8‑205)

w1C = LumaWeightL1[refIdxL1] (8‑206)

o0C = luma\_offset\_l0[refIdxL0] \* ( 1 << ( BitDepthY – 8 ) ) (8‑207)

o1C = luma\_offset\_l1[refIdxL1] \* ( 1 << ( BitDepthY – 8 ) ) (8‑208)

– Otherwise (C is equal to Cb or Cr for chroma samples, with iCbCr = 0 for Cb, iCbCr = 1 for Cr),

logWDc = ChromaLog2WeightDenom + shift1 (8‑209)

w0C = ChromaWeightL0[refIdxL0][ iCbCr ] (8‑210)

w1C = ChromaWeightL1[refIdxL1][ iCbCr ] (8‑211)

o0C = ChromaWeightL0[refIdxL0][ iCbCr ] \* ( 1 << ( BitDepthC – 8 ) ) (8‑212)

o1C = ChromaOffsetL1[refIdxL1][ iCbCr ] \* ( 1 << ( BitDepthC – 8 ) ) (8‑213)

### Decoding process for the residual signal of coding units coded in inter prediction mode

Inputs to this process are:

– a luma location ( xC, yC ) specifying the top-left luma sample of the current coding unit relative to the top‑left luma sample of the current picture,

– a variable log2CUSize specifying the size of the current coding unit.

Outputs of this process are:

– a (nCSL)x(nCSL) array resSamplesL of luma residual samples, where nCSL is derived as specified below,

– a (nCSC)x(nCSC) array resSamplesCb of chroma residual samples for the component Cb, where nCSC is derived as specified below,

– a (nCSC)x(nCSC) array resSamplesCr of chroma residual samples for the component Cr, where nCSC is derived as specified below.

The variable nCSL is set equal to 1 << log2CUSize and the variable nCSC is set equal to ( 1 << log2CUSize ) >> 1.

Let resSamplesL be a (nCSL)x(nCSL) array of luma residual samples and let resSamplesCb and resSamplesCr be two (nCSC)x(nCSC) arrays of chroma residual samples.

Depending on no\_residual\_data\_flag, the following applies:

– If no\_residual\_data\_flag is equal to 1, all samples of the (nCSL)x(nCSL) array resSamplesL and all samples of the two (nCSC)x(nCSC) arrays resSamplesCb and resSamplesCr are set equal to 0.

– Otherwise (no\_residual\_data\_flag is equal to 0), the following ordered steps apply:

1. The decoding process for luma residual blocks as specified in subclause 8.4.3.1 below is invoked with the luma location ( xC, yC ), the luma location ( xB, yB ) set equal to ( 0, 0 ), the variable log2TrafoWidth set equal to log2CUSize, the variable log2TrafoHeight set equal to log2CUSize, the variable trafoDepth set equal to 0, the variable nCS set equal to nCSL, and the (nCSL)x(nCSL) array resSamplesL as the inputs and the output is a modified version of the (nCSL)x(nCSL) array resSamplesL.
2. The decoding process for chroma residual blocks as specified in subclause 8.4.3.2 below is invoked with the luma location ( xC, yC ), the luma location ( xB, yB ) set equal to ( 0, 0 ), the variable log2TrafoWidth set equal to log2CUSize, the variable log2TrafoHeight set equal to log2CUSize, the variable trafoDepth set equal to 0, the variable cIdx set equal to 1, the variable nCS set equal to nCSC, and the (nCSC)x(nCSC) array resSamplesCb as the inputs and the output is a modified version of the (nCSC)x(nCSC) array resSamplesCb.
3. The decoding process for chroma residual blocks as specified in subclause 8.4.3.2 below is invoked with the luma location ( xC, yC ), the luma location ( xB, yB ) set equal to ( 0, 0 ), the variable log2TrafoWidth set equal to log2CUSize, the variable log2TrafoHeight set equal to log2CUSize, the variable trafoDepth set equal to 0, the variable cIdx set equal to 2, the variable nCS set equal to nCSC, and the (nCSC)x(nCSC) array resSamplesCr as the inputs and the output is a modified version of the (nCSC)x(nCSC) array resSamplesCr.

#### Decoding process for luma residual blocks

Inputs to this process are:

– a luma location ( xC, yC ) specifying the top-left luma sample of the current coding unit relative to the top‑left luma sample of the current picture,

– a luma location ( xB, yB ) specifying the top-left luma sample of the current block relative to the top‑left luma sample of the current coding unit,

– a variable log2TrafoWidth specifying the width of the current block,

– a variable log2TrafoHeight specifying the height of the current block,

– a variable trafoDepth specifying the hierarchy depth of the current block relative to the coding unit,

– a variable nCS specifying the size, in luma samples, of the current coding unit,

– a (nCS)x(nCS) array resSamples of luma residual samples.

Output of this process is:

– a modified version of the (nCS)x(nCS) array of luma residual samples.

Depending split\_transform\_flag[ xB ][ yB ][ trafoDepth ], the following applies:

– If split\_transform\_flag[ xB ][ yB ][ trafoDepth ] is equal to 1, the following ordered steps apply:

1. The variables xB1, yB1, xB2, yB2, xB3 and yB3 are derived as follows.
   * If InterTUSplitDirection is equal to 2, the following applies.
   * The variable xB1 is set equal to xB + ( ( 1 << log2TrafoWidth ) >> 1 ).
   * The variable yB2 is set equal to yB + ( ( 1 << log2TrafoHeight ) >> 1 ).
   * The variable xB3 is set equal to xB1.
   * The variable yB3 is set equal to yB2.
   * The variable log2TrafoWidth1 is set equal to log2TrafoWidth − 1.
   * The variable log2TrafoHeight1 is set equal to log2TrafoHeight − 1.
   * Otherwise (InterTUSplitDirection is equal to 0 or 1), the following applies.
   * The variable xB1 is set equal to xB + ((1 << (log2TrafoWidth)) >> 2) \* InterTUSplitDirection.
   * The variable yB1 is set equal to yB + ((1 << (log2TrafoHeight)) >> 2) \* (1 − InterTUSplitDirection).
   * The variable xB2 is set equal to xB1 + ((1 << (log2TrafoWidth)) >> 2) \* InterTUSplitDirection.
   * The variable yB2 is set equal to yB1 + ((1 << (log2TrafoHeight)) >> 2) \* (1 − InterTUSplitDirection).
   * The variable xB3 is set equal to xB2 + ((1 << (log2TrafoWidth)) >> 2) \* InterTUSplitDirection.
   * The variable yB3 is set equal to yB2 + ((1 << (log2TrafoHeight)) >> 2) \* (1 − InterTUSplitDirection).
   * The variable log2TrafoWidth1 is set equal to (log2TrafoWidth – 2) \* InterTUSplitDirection.
   * The variable log2TrafoHeight1 is set equal to (log2TrafoHeight − 2) \* (1 − InterTUSplitDirection).
2. The decoding process for luma residual blocks as specified in this subclause is invoked with the luma location ( xC, yC ), the luma location ( xB, yB ), the variable log2TrafoWidth set equal to log2TrafoWidth1, the variable log2TrafoHeight set equal to log2TrafoHeight1, the variable trafoDepth set equal to trafoDepth + 1, the variable nCS, and the (nCS)x(nCS) array resSamples as the inputs and the output is a modified version of the (nCS)x(nCS) array resSamples.
3. The decoding process for luma residual blocks as specified in this subclause is invoked with the luma location ( xC, yC ), the luma location ( xB1, yB1 ), the variable log2TrafoWidth set equal to log2TrafoWidth1, the variable log2TrafoHeight set equal to log2TrafoHeight1, the variable trafoDepth set equal to trafoDepth + 1, the variable nCS, and the (nCS)x(nCS) array resSamples as the inputs and the output is a modified version of the (nCS)x(nCS) array resSamples.
4. The decoding process for luma residual blocks as specified in this subclause is invoked with the luma location ( xC, yC ), the luma location ( xB2, yB2 ), the variable log2TrafoWidth set equal to log2TrafoWidth1, the variable log2TrafoHeight set equal to log2TrafoHeight1, the variable trafoDepth set equal to trafoDepth + 1, the variable nCS, and the (nCS)x(nCS) array resSamples as the inputs and the output is a modified version of the (nCS)x(nCS) array resSamples.
5. The decoding process for luma residual blocks as specified in this subclause is invoked with the luma location ( xC, yC ), the luma location ( xB3, yB3 ), the variable log2TrafoWidth set equal to log2TrafoWidth1, the variable log2TrafoHeight set equal to log2TrafoHeight1, the variable trafoDepth set equal to trafoDepth + 1, the variable nCS, and the (nCS)x(nCS) array resSamples as the inputs and the output is a modified version of the (nCS)x(nCS) array resSamples.

– Otherwise (split\_transform\_flag[ xB ][ yB ][ trafoDepth ] is equal to 0), the following ordered steps apply:

1. The variable nW is set equal to 1 << log2TrafoWidth.
2. The variable nH is set equal to 1 << log2TrafoHeight.
3. The scaling and transformation process as specified in subclause 8.5.1 is invoked with the luma location ( xC + xB, yC +yB ), the variable trafoDepth, the variable cIdx set equal to 0, the transform width trafoWidth set equal to nW, and the transform height trafoHeight set equal to nH as the inputs and the output is a (nW)x(nH) array resSamplesBlock.
4. The array construction process as specified in subclause 8.5.5 is invoked with the luma location ( xB, yB ), the variable cIdx set equal to 0, the variable inputArrayWidth set equal to nW, the variable inputArrayHeight set equal to nH, the variable outputArraySize set equal to nCS, the (nW)x(nH) array resSamplesBlock, and the (nCS)x(nCS) array resSamples as the inputs and the output is a modified version of the (nCS)x(nCS) array resSamples.

#### Decoding process for chroma residual blocks

Inputs to this process are:

– a luma location ( xC, yC ) specifying the top-left luma sample of the current coding unit relative to the top‑left luma sample of the current picture,

– a luma location ( xB, yB ) specifying the top-left luma sample of the current block relative to the top‑left luma sample of the current coding unit,

– a variable log2TrafoWidth specifying the width of the current block,

– a variable log2TrafoHeight specifying the height of the current block,

– a variable trafoDepth specifying the hierarchy depth of the current block relative to the coding unit,

– a variable cIdx specifying the chroma component of the current block,

– a variable nCS specifying the size, in chroma samples, of the current coding unit,

– a (nCS)x(nCS) array resSamples of chroma residual samples.

Output of this process is:

– a modified version of the (nCS)x(nCS) array of chroma residual samples.

The variable splitChromaFlag is derived as follows:

– If split\_transform\_flag[ xB ][ yB ][ trafoDepth ] is equal to 1 and log2TrafoSize is greater than Log2MinTrafoSizeC + 1, splitChromaFlag is set equal to 1.

– Otherwise (split\_transform\_flag[ xB ][ yB ][ trafoDepth ] is equal to 0 or log2TrafoSize is equal to Log2MinTrafoSizeC + 1), splitChromaFlag is set equal to 0.

Depending splitChromaFlag, the following applies:

– If splitChromaFlag is equal to 1, the following ordered steps apply:

1. The variables xB1, yB1, xB2, yB2, xB3 and yB3 are derived as follows.
   * If InterTUSplitDirection is equal to 2, the following applies.
   * The variable xB1 is set equal to xB + ( ( 1 << log2TrafoWidth ) >> 1 ).
   * The variable yB1 is set equal to yB.
   * The variable xB2 is set equal to xB.
   * The variable yB2 is set equal to yB + ( ( 1 << log2TrafoHeight ) >> 1 ).
   * The variable xB3 is set equal to xB1.
   * The variable yB3 is set equal to yB2.
   * The variable log2TrafoWidth1 is set equal to log2TrafoWidth − 1.
   * The variable log2TrafoHeight1 is set equal to log2TrafoHeight − 1.
   * Otherwise (InterTUSplitDirection is equal to 0 or 1), the following applies.
   * The variable xB1 is set equal to xB + ((1 << (log2TrafoWidth)) >> 2) \* InterTUSplitDirection.
   * The variable yB1 is set equal to yB + ((1 << (log2TrafoHeight)) >> 2) \* (1 − InterTUSplitDirection).
   * The variable xB2 is set equal to xB1 + ((1 << (log2TrafoWidth)) >> 2) \* InterTUSplitDirection.
   * The variable yB2 is set equal to yB1 + ((1 << (log2TrafoHeight)) >> 2) \* (1 − InterTUSplitDirection).
   * The variable xB3 is set equal to xB2 + ((1 << (log2TrafoWidth)) >> 2) \* InterTUSplitDirection.
   * The variable yB3 is set equal to yB2 + ((1 << (log2TrafoHeight)) >> 2) \* (1 − InterTUSplitDirection).
   * The variable log2TrafoWidth1 is set equal to (log2TrafoWidth − 2) \* InterTUSplitDirection.
   * The variable log2TrafoHeight1 is set equal to (log2TrafoHeight − 2) \* (1 − InterTUSplitDirection).
2. The decoding process for residual chroma blocks as specified in this subclause is invoked with the luma location ( xC, yC ), the luma location ( xB, yB ), the variable log2TrafoWidth set equal to log2TrafoWidth1, the variable log2TrafoHeight set equal to log2TrafoHeight1, the variable trafoDepth set equal to trafoDepth + 1, the variable cIdx, the variable nCS, and the (nCS)x(nCS) array resSamples as the inputs and the output is a modified version of the (nCS)x(nCS) array resSamples.
3. The decoding process for residual chroma blocks as specified in this subclause is invoked with the luma location ( xC, yC ), the luma location ( xB1, yB1 ), the variable log2TrafoWidth set equal to log2TrafoWidth1, the variable log2TrafoHeight set equal to log2TrafoHeight1, the variable trafoDepth set equal to trafoDepth + 1, the variable cIdx, the variable nCS, and the (nCS)x(nCS) array resSamples as the inputs and the output is a modified version of the (nCS)x(nCS) array resSamples.
4. The decoding process for residual chroma blocks as specified in this subclause is invoked with the luma location ( xC, yC ), the luma location ( xB2, yB2 ), the variable log2TrafoWidth set equal to log2TrafoWidth1, the variable log2TrafoHeight set equal to log2TrafoHeight1, the variable trafoDepth set equal to trafoDepth + 1, the variable cIdx, the variable nCS, and the (nCS)x(nCS) array resSamples as the inputs and the output is a modified version of the (nCS)x(nCS) array resSamples.
5. The decoding process for residual chroma blocks as specified in this subclause is invoked with the luma location ( xC, yC ), the luma location ( xB3, yB3 ), the variable log2TrafoWidth set equal to log2TrafoWidth1, the variable log2TrafoHeight set equal to log2TrafoHeight1, the variable trafoDepth set equal to trafoDepth + 1, the variable cIdx, the variable nCS, and the (nCS)x(nCS) array resSamples as the inputs and the output is a modified version of the (nCS)x(nCS) array resSamples.

– Otherwise (splitChromaFlag is equal to 0), the following ordered steps apply:

1. The variable nW is set equal to ( 1 << log2TrafoWidth ) >> 1.
2. The variable nH is set equal to (1 << log2TrafoHeight ) >> 1.
3. The scaling and transformation process as specified in subclause 8.5.1 is invoked with the luma location ( xC + xB, yC +yB ), the variable trafoDepth, the variable cIdx, the transform width trafoWidth set equal to nW, and the transform height trafoHeight set equal to nH as the inputs and the output is a (nW)x(nH) array resSamplesBlock.
4. The array construction process as specified in subclause 8.5.5 is invoked with the luma location ( xB, yB ), the variable cIdx, the variable inputArrayWidth set equal to nW, the variable inputArrayHeight set equal to nH, the variable outputArraySize set equal to nCS, the (nW)x(nH) array resSamplesBlock, and the (nCS)x(nCS) array resSamples as the inputs and the output is a modified version of the (nCS)x(nCS) array resSamples.

## Scaling, transformation and array construction process prior to deblocking filter process

### Scaling and transformation process

Inputs to this process are:

– a luma location ( xT, yT ) specifying the top-left luma sample of the current transform unit relative to the top‑left luma sample of the current picture,

– a variable trafoDepth specifying the hierarchy depth of the current block relative to the coding unit,

– a variable cIdx specifying the chroma component of the current block,

– a variable nW specifying the width of the current transform block.

– a variable nH specifying the height of the current transform block.

Output of this process is:

– a modified version of the (nW)x(nH) array of residual samples r with elements rij.

The quantization parameter qP is derived as follows.

– If cIdx is equal to 0,

qP = QP’Y (8‑150)

– Otherwise if cIdx is equal to 1,

qP = QP’C b (8‑151)

– Otherwise (cIdx is equal to 2),

qP = QP’C r (8‑151)

The (nW)x(nH) array of residual samples r are derived as specified in the following ordered steps.

1. The scaling process for transform coefficients as specified in subclause 8.5.3 is invoked with the width of the transform unit nW, the height of the transform unit nH, the (nW)x(nH) array of transform coefficients transCoeffLevel[ xT ][ yT ][ cIdx ], the chroma component variable cIdx and the quantization parameter qP as the inputs and the output is a scaled transform coefficient (nW)x(nH) array d.
2. The transformation process for scaled transform coefficients as specified in subclause 8.5.4 is invoked with the width of the transform unit nW, the height of the transform unit nH, the scaled transform coefficient (nW)x(nH) array d and the chroma component variable cIdx as the inputs and the output is a residual samples (nW)x(nH) array r.

### Scaling process for transform coefficients

Inputs of this process are:

– a variable nW specifying the width of the current transform unit,

– a variable nH specifying the height of the current transform unit,

– a (nW)x(nH) array c of transform coefficients with elements cij,

– a variable cIdx specifying the chroma component of the current block,

– a variable qP specifying the quantization parameter.

Output of this process is scaled transform coefficients as a (nW)x(nH) array of d with elements dij.

The variable log2TrSize is derived as follows:

log2TrSize = ( log2( NW ) + log2( NH ) ) >> 1 (8‑158)

The variable shift is derived as follows:

– If cIdx is equal to 0,

shift = BitDepthY + log2TrSize – 9 (8‑150)

– Otherwise,

shift = BitDepthC + log2TrSize – 9 (8‑150)

The scaling array levelScale[·] is specified as levelScale[k] = { 40, 45, 51, 57, 64, 72 } with k=0..5.

The variable shiftScale is set equal to shift + 4 – QP/6.

The elements of array M[ i ][ j ] with i = 0..nW – 1, j = 0..nH – 1 are set equal to ScalingList[ SizeID ][ RefMatrixID ][ trafoType ][ i\*nW+j ], where SizeID and RefMatrixID are specified in Table 7‑2 and Table 7‑3, respectively, and trafoType is derived by

trafoType = ( ( nW = = nH ) ? 0 : ( ( nW > nH ) ? 1 : 2 ) ) (8‑158)

The scaled transform coefficient dij with i = 0..nW − 1, j = 0..nH − 1 is derived as follows.

– If scaling\_list\_present\_flag is equal to 1, [Ed. (WJ): should be replaced by better conditioning]

dij = Clip3( -32768, 32767,( ( cij \* levelScale[ qP%6 ] << ( qP/6 ) ) + ( 1 << ( shift – 1 ) ) ) >> shift ) (8‑158)

– Otherwise, if shiftScale is greater than 0,

dij = Clip3( -32768, 32767, ( cij \* M[ i ][ j ]\*levelScale[ qP%6 ] +  ( 1 << ( shiftScale – 1 ) ) ) >> shiftScale ) (8‑158)

– Otherwise,

dij = Clip3( -32768, 32767, ( cij \* M[ i ][ j ]\*levelScale[ qP%6 ] ) << ( -shiftScale ) ) (8‑158)

[Ed. (WJ): do we need to clip cij to 16b before computing dij? – related to G719 – maybe not]

### Transformation process for scaled transform coefficients

Inputs of this process are:

– a variable nW specifying the width of the current transform unit,

– a variable nH specifying the height of the current transform unit,

– a (nW)x(nH) array d of scaled transform coefficients with elements dij.

– a variable cIdx specifying the chroma component of the current block,

Output of this process is residual samples as a (nW)x(nH) array r with elements rij.

Depending on PredMode and IntraPredMode, the following applies:

– If PredMode is equal to MODE\_INTRA, log2(nW\*nH) is equal to 4, and cIdx is equal to 0, the variables horizTrType and vertTrType are specified as Table 8‑11 with IntraPredMode as input. [Ed. (WJ): DST is applied only for luma 4x4 block]

– Otherwise, the variables horizTrType and vertTrType are set equal to 0.

Table 8‑13 – Specification of horizTrType and vertTrType

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **IntraPredMode** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| **horizTrType** | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| **vertTrType** | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **IntraPredMode** | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 |
| **horizTrType** | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| **vertTrType** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |

The constructed residual samples are derived as specified in the following ordered steps.

1. Each (horizontal) row of scaled transform coefficients dij (i=0..nW−1, j=0..nH−1) is transformed to eij (i=0..nW−1, j=0..nH−1) by invoking the one-dimensional transformation process as specified in subclause 8.5.4.1 according to the width of the transform unit nW, with the (nW)x(nH) array d and the transform type variable horizTrType as the inputs and the output is the (nW)x(nH) array e.
2. The intermediate sample values gij (i=0..nW-1, j=0..nH-1) are derived by

gij = Clip3( -32768, 32767, ( eij + 64 ) >> 7 ) (8‑159)

1. Each (vertical) column of the resulting matrix gij (i=0..nW−1, j=0..nH−1) is transformed to fij (i=0..nW−1, j=0..nH−1) by invoking the one-dimensional transformation process as specified in subclause 8.5.4.1 according to the height of the transform unit nH, with the (nW)x(nH) array e and the transform type variable vertTrType as the inputs and the output is the (nW)x(nH) array f.
2. The variable shift is derived as follows:

– If cIdx is equal to 0,

shift = 20 – BitDepthY (8‑150)

– Otherwise,

shift = 20 – BitDepthC (8‑150)

1. The residual sample value rij with i=0..(nW)−1, j=0..(nH)−1 is derived as follows.

rij = ( fij + (1 << ( shift – 1) ) ) >> shift (8‑160)

#### Transformation process

Inputs of this process are:

– a variable nS specifying the sample size of scaled transform coefficients,

– an array of scaled transform coefficients x with elements xi, with i=0..nS-1.

– a transform type variable trType

Output of this process is an array of samples of the residual samples y with elements yi, with i = 0..nS-1.

Depending on trType, the following applies:

– If nS is equal to 4 and trType is equal to 1, the following ordered steps apply:

1. A set of intermediate values c0, c1, c2 and c3 is calculated as follows:

c0 = x0 + x2 (8‑159)  
c1 = x2 + x3 (8‑159)  
c2 = x0 – x3 (8‑159)  
c3 = 74 \* x1 (8‑159)

1. The output values yi with i = 0..3 are then specified as follows:

y0 = ( 29\*c0 + 55\*c1 + c3 + 64 ) >> 7 (8‑159)  
y1 = ( 55\*c2 – 29\*c1 + c3 + 64 ) >> 7 (8‑159)  
y2 = ( 74\*( x0 - x2 + x3 ) + 64 ) >> 7 (8‑159)  
y3 = ( 55\*c0 + 29 \* c2 – c3 + 64 ) >> 7 (8‑159)

– Otherwise (nS is not equal to 4 or trType is not equal to 1), the following applies:

yi = ∑j( ckj \* xj ) with i=0..nS-1, j=0..nS-1 (8‑159)

where k = 1 << ( 5 – log2( nS ) ) \* i and the transform coefficient matrix c is specified as:

c =

{

{64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64}

{90 90 88 85 82 78 73 67 61 54 46 38 31 22 13 4 -4-13-22-31-38-46-54-61-67-73-78-82-85-88-90-90}

{90 87 80 70 57 43 25 9 -9-25-43-57-70-80-87-90-90-87-80-70-57-43-25 -9 9 25 43 57 70 80 87 90}

{90 82 67 46 22 -4-31-54-73-85-90-88-78-61-38-13 13 38 61 78 88 90 85 73 54 31 4-22-46-67-82-90}

{89 75 50 18-18-50-75-89-89-75-50-18 18 50 75 89 89 75 50 18-18-50-75-89-89-75-50-18 18 50 75 89}

{88 67 31-13-54-82-90-78-46 -4 38 73 90 85 61 22-22-61-85-90-73-38 4 46 78 90 82 54 13-31-67-88}

{87 57 9-43-80-90-70-25 25 70 90 80 43 -9-57-87-87-57 -9 43 80 90 70 25-25-70-90-80-43 9 57 87}

{85 46-13-67-90-73-22 38 82 88 54 -4-61-90-78-31 31 78 90 61 4-54-88-82-38 22 73 90 67 13-46-85}

{83 36-36-83-83-36 36 83 83 36-36-83-83-36 36 83 83 36-36-83-83-36 36 83 83 36-36-83-83-36 36 83}

{82 22-54-90-61 13 78 85 31-46-90-67 4 73 88 38-38-88-73 -4 67 90 46-31-85-78-13 61 90 54-22-82}

{80 9-70-87-25 57 90 43-43-90-57 25 87 70 -9-80-80 -9 70 87 25-57-90-43 43 90 57-25-87-70 9 80}

{78 -4-82-73 13 85 67-22-88-61 31 90 54-38-90-46 46 90 38-54-90-31 61 88 22-67-85-13 73 82 4-78}

{75-18-89-50 50 89 18-75-75 18 89 50-50-89-18 75 75-18-89-50 50 89 18-75-75 18 89 50-50-89-18 75}

{73-31-90-22 78 67-38-90-13 82 61-46-88 -4 85 54-54-85 4 88 46-61-82 13 90 38-67-78 22 90 31-73}

{70-43-87 9 90 25-80-57 57 80-25-90 -9 87 43-70-70 43 87 -9-90-25 80 57-57-80 25 90 9-87-43 70}

{67-54-78 38 85-22-90 4 90 13-88-31 82 46-73-61 61 73-46-82 31 88-13-90 -4 90 22-85-38 78 54-67}

{64-64-64 64 64-64-64 64 64-64-64 64 64-64-64 64 64-64-64 64 64-64-64 64 64-64-64 64 64-64-64 64}

{61-73-46 82 31-88-13 90 -4-90 22 85-38-78 54 67-67-54 78 38-85-22 90 4-90 13 88-31-82 46 73-61}

{57-80-25 90 -9-87 43 70-70-43 87 9-90 25 80-57-57 80 25-90 9 87-43-70 70 43-87 -9 90-25-80 57}

{54-85 -4 88-46-61 82 13-90 38 67-78-22 90-31-73 73 31-90 22 78-67-38 90-13-82 61 46-88 4 85-54}

{50-89 18 75-75-18 89-50-50 89-18-75 75 18-89 50 50-89 18 75-75-18 89-50-50 89-18-75 75 18-89 50}

{46-90 38 54-90 31 61-88 22 67-85 13 73-82 4 78-78 -4 82-73-13 85-67-22 88-61-31 90-54-38 90-46}

{43-90 57 25-87 70 9-80 80 -9-70 87-25-57 90-43-43 90-57-25 87-70 -9 80-80 9 70-87 25 57-90 43}

{38-88 73 -4-67 90-46-31 85-78 13 61-90 54 22-82 82-22-54 90-61-13 78-85 31 46-90 67 4-73 88-38}

{36-83 83-36-36 83-83 36 36-83 83-36-36 83-83 36 36-83 83-36-36 83-83 36 36-83 83-36-36 83-83 36}

{31-78 90-61 4 54-88 82-38-22 73-90 67-13-46 85-85 46 13-67 90-73 22 38-82 88-54 -4 61-90 78-31}

{25-70 90-80 43 9-57 87-87 57 -9-43 80-90 70-25-25 70-90 80-43 -9 57-87 87-57 9 43-80 90-70 25}

{22-61 85-90 73-38 -4 46-78 90-82 54-13-31 67-88 88-67 31 13-54 82-90 78-46 4 38-73 90-85 61-22}

{18-50 75-89 89-75 50-18-18 50-75 89-89 75-50 18 18-50 75-89 89-75 50-18-18 50-75 89-89 75-50 18}

{13-38 61-78 88-90 85-73 54-31 4 22-46 67-82 90-90 82-67 46-22 -4 31-54 73-85 90-88 78-61 38-13}

{ 9-25 43-57 70-80 87-90 90-87 80-70 57-43 25 -9 -9 25-43 57-70 80-87 90-90 87-80 70-57 43-25 9}

{ 4-13 22-31 38-46 54-61 67-73 78-82 85-88 90-90 90-90 88-85 82-78 73-67 61-54 46-38 31-22 13 -4}

}

(8‑159)

[Ed. (WJ): better way?]

### Array construction process

Inputs of this process are:

– a luma location ( xB, yB ) specifying the top-left luma sample of the current block relative to the top‑left luma sample of the current coding unit,

– a variable cIdx specifying the chroma component of the current block,

– a variable nW specifying the width of the current block,

– a variable nH specifying the height of the current block,

– a variable nCS specifying the size of the current coding unit,

– a (nW)x(nH) array resSampleBlock specifying the current block samples,

– a (nCS)x(nCS) array resSample specifying the samples of the current coding unit.

Output of this process is a modified (nCS)x(nCS) array resSample specifying the samples of the current coding unit.

The modified array resSample is derived as follows.

resSample[ xB+i, yB+j ] += resSampleBlock[ i, j ], with i = 0..nW − 1, j = 0..nH − 1 (8‑427)

## In-loop filter process

[Ed: (WJ) overall process description need to be inserted]

### Deblocking filter process

A conditional filtering process shall be performed on a treeblock basis after the completion of the picture construction process prior to deblocking filter process for the entire decoded picture (as specified in subclauses XXX and YYY) [Ed.: (WJ) those subclauses seem not defined yet], with all treeblocks in a picture processed in order of increasing treeblock addresses.

Each treeblock is processed on a coding unit basis with the same order as decoding process. For each coding unit, vertical edges are filtered first, starting with the edge on the left-hand side of the coding unit proceeding through the edges towards the right-hand side of the coding unit in their geometrical order. Then, the horizontal edges are filtered starting with the edge on the top of the coding unit proceeding through the edges towards the bottom of the coding unit in their geometrical order.

Sample values above of the current coding unit that may have already been modified by the filtering of horizontal edges of deblocking filter process operation on previous coding unit shall be used as inputs to the deblocking filter process on the current coding unit and may be further modified during the filtering of the current coding unit. Sample values to the left of the current coding unit shall be used as inputs to the deblocking filter process on the current coding unit and may be further modified during the filtering of the current coding unit. Sample values to the left of the current coding unit may be modified by the filtering of vertical edge and may be further modified by the filtering of horizontal edges.

Sample values modified during filtering of vertical edges are used as input for the filtering of the horizontal edges. For sample values modified by both filtering of horizontal edges and filtering of vertical edges, filtering of horizontal edges is applied after filtering of vertical edges.

The deblocking filter process shall be applied to all prediction unit edges and transform unit edges of a picture, except edges at the boundary of the picture, any edges for which the deblocking filter process is disabled by disable\_deblocking\_filter\_flag, any edges coinside with tile boundaries when loop\_filter\_across\_tile\_flag is equal to 0, and any edges coinside with slice boundaries when loop\_filter\_across\_slice\_flag is equal to 0. For the transform units and prediction units with edges smaller than 8 samples in either vertical or horizontal direction, only the edges lying on the 8x8 sample grid are filtered.

When disable\_deblocking\_filter\_flag is not equal to 1, the deblocking filter process is invoked as the following ordered steps for each coding unit with the same order as decoding process.

1. The coding unit size nS is set equal to 1 << log2CUSize.
2. The variables FilterInternalEdgesFlag, FilterLeftCuEdgeFlag and FilterTopCuEdgeFlag are derived as follows.

* The variable FilterInternalEdges is set equal to 1.
* If the left boundary of current coding unit is the left boundary of the picture, or if the left boundary of current coding unit is the left boundary of the tile and loop\_filter\_across\_tile\_flag is equal to 0, or if the left boundary of current coding unit is the left boundary of the slice and loop\_filter\_across\_slice\_flag is equal to 0, the variable FilterLeftCuEdgeFlag is set equal to 0, otherwise set equal to 1.
* If the top boundary of current coding unit is the top boundary of the picture, or if the top boundary of current coding unit is the top boundary of the tile and loop\_filter\_across\_tile\_flag is equal to 0, or if the top boundary of current coding unit is the top boundary of the slice and loop\_filter\_across\_slice\_flag is equal to 0, the variable FilterTopCuEdgeFlag is set equal to 0, otherwise set equal to 1.

1. All elements of two-dimensional array of size (nS)x(nS), horEdgeFlags and verEdgeFlags are initialized to zero.
2. The derivation process of transform unit boundary specified in subclause 8.6.1.1 are invoked with the luma location ( xB, yB ) set equal to ( 0, 0 ), the transform unit width log2TrafoWidth set equal to log2CUSize, the transform unit height log2TrafoHeight set equal to log2CUSize and the variable trafoDepth set equal to 0 as the inputs and the modified horEdgeFlags and verEdgeFlags as outputs.
3. The derivation process of prediction unit boundary specified in subclause 8.6.1.2 are invoked with the coding unit size log2CUSize and the prediction partition mode PartMode as inputs, and the modified horEdgeFlags and verEdgeFlags as outputs.
4. The derivation process of the boundary filtering strength specified in subclause 8.6.1.3 is invoked with the luma location ( xC, yC ), the coding unit size log2CUSize, horEdgeFlags and verEdgeFlags as inputs and an array of size (2)x(nS)x(nS), bS as output.
5. The filtering process for coding unit specified in subclause 8.6.1.4 are invoked with the luma location ( xC, yC ) specifying the top-left luma sample of the current coding unit relative to the top left luma sample of the current picture, the coding unit size log2CUSize and the array bS as inputs and the modified reconstructued picture as output.

#### Derivation process of transform unit boundary

Inputs of this process are:

– a luma location ( xB, yB ) specifying the top-left luma sample of the current block relative to the top‑left luma sample of the current coding unit,

– a variable log2TrafoWidth specifying the width of the current block,

– a variable log2TrafoHeight specifying the height of the current block,

– a variable trafoDepth.

Outputs of this process are:

– two-dimensional arrays of (nS)x(nS), horEdgeFlags and verEdgeFlags.

Depending on split\_transform\_flag[ xB ][ yB ][ trafoDepth ], the following applies:

– If split\_transform\_flag[ xB ][ yB ][ trafoDepth ] is equal to 1, the following ordered steps apply:

1. The variables xB1, yB1, xB2, yB2, xB3 and yB3 are derived as follows.
   * If InterTUSplitDirection is equal to 2, the following applies.
   * The variable xB1 is set equal to xB + ( ( 1 << log2TrafoWidth ) >> 1 ).
   * The variable yB1 is set equal to yB.
   * The variable xB2 is set equal to xB.
   * The variable yB1 is set equal to yB + ( ( 1 << log2TrafoHeight ) >> 1 ).
   * The variable xB3 is set equal to xB1.
   * The variable yB3 is set equal to yB2.
   * The variable log2TrafoWidth1 is set equal to log2TrafoWidth − 1.
   * The variable log2TrafoHeight1 is set equal to log2TrafoHeight − 1.
   * Otherwise (InterTUSplitDirection is equal to 0 or 1), the following applies.
   * The variable xB1 is set equal to xB + ((1 << (log2TrafoWidth)) >> 2) \* InterTUSplitDirection.
   * The variable yB1 is set equal to yB + ((1 << (log2TrafoHeight)) >> 2) \* (1 − InterTUSplitDirection).
   * The variable xB2 is set equal to xB1 + ((1 << (log2TrafoWidth)) >> 2) \* InterTUSplitDirection.
   * The variable yB2 is set equal to yB1 + ((1 << (log2TrafoHeight)) >> 2) \* (1 − InterTUSplitDirection).
   * The variable xB3 is set equal to xB2 + ((1 << (log2TrafoWidth)) >> 2) \* InterTUSplitDirection.
   * The variable yB3 is set equal to yB2 + ((1 << (log2TrafoHeight)) >> 2) \* (1 − InterTUSplitDirection).
   * The variable log2TrafoWidth1 is set equal to (log2TrafoWidth − 2) \* InterTUSplitDirection.
   * The variable log2TrafoHeight1 is set equal to (log2TrafoHeight − 2) \* (1 − InterTUSplitDirection).
2. The deriviation process of transform unit boundary as specified in this subclause is invoked with the luma location ( xB, yB ), the variable log2TrafoWidth set equal to log2TrafoWidth1, the variable log2TrafoHeight set equal to log2TrafoHeight1 and the variable trafoDepth1 set equal to trafoDepth + 1 as inputs and the outputs are the modified versions of two arrays, horEdgeFlags and verEdgeFlags.
3. The deriviation process of transform unit boundary as specified in this subclause is invoked with the luma location ( xB1, yB1 ), the variable log2TrafoWidth set equal to log2TrafoSizeWidth1, the variable log2TrafoHeight set equal to log2TrafoHeight1 and the variable trafoDepth1 set equal to trafoDepth + 1 as inputs and the outputs are the modified versions of two arrays, horEdgeFlags and verEdgeFlags.
4. The deriviation process of transform unit boundary as specified in this subclause is invoked with the luma location ( xB2, yB2 ), the variable log2TrafoWidth set equal to log2TrafoSizeWidth1, the variable log2TrafoHeight set equal to log2TrafoHeight1 and the variable trafoDepth1 set equal to trafoDepth + 1 as inputs and the outputs are the modified versions of two arrays, horEdgeFlags and verEdgeFlags.
5. The deriviation process of transform unit boundary as specified in this subclause is invoked with the luma location ( xB3, yB3 ), the variable log2TrafoWidth1 set equal to log2TrafoSizeWidth1, the variable log2TrafoHeight set equal to log2TrafoHeight1 and the variable trafoDepth1 set equal to trafoDepth + 1 as inputs and the outputs are the modified versions of two arrays, horEdgeFlags and verEdgeFlags.

– Otherwise (split\_transform\_flag[ xB ][ yB ][ trafoDepth ] is equal to 0), the following applies:

* If yB is equal to zero, horEdgeFlags[ xB + k ][ yB ] is set equal to FilterTopCuEdgeFlag, otherwise horEdgeFlags[ xB + k ][ yB ] is set equal to FilterInternalEdgesFlag for k = 0.. ( 1 << log2TrafoWidth ) – 1.
* If xB is equal to zero, verEdgeFlags[ xB ][ yB + k ] is set equal to FilterLeftCuEdgeFlag, otherwise verEdgeFlags[ xB ][ yB + k ] is set equal to FilterInternalEdgesFlag for k = 0.. ( 1 << log2TrafoHeight ) – 1.

#### Derivation process of prediction unit boundary

Inputs of this process are:

– a variable log2CUSize specifying the coding unit size,

– a prediction partition mode PartMode.

Outputs of this process are:

– two-dimensional arrays of (nS)x(nS), horEdgeFlags and verEdgeFlags.

Depending on PartMode, the following applies:

– If PartMode is equal to PART\_2NxN or PART\_NxN, horEdgeFlags[ k ][ 1 << ( log2CUSize – 1 ) ] is set equal to FilterInternalEdgesFlag for k = 0.. ( 1 << log2CUSize ) – 1.

– If PartMode is equal to PART\_Nx2N or PART\_NxN, verEdgeFlags[ 1 << ( log2CUSize – 1 ) ][ k ] is set equal to FilterInternalEdgesFlag for k = 0.. ( 1 << log2CUSize ) – 1.

– If PartMode is equal to PART\_2NxnU, horEdgeFlags[ k ][ (1 << ( log2CUSize – 1 )) – (1 << ( log2CUSize – 2 )) ] is set equal to FilterInternalEdgesFlag for k = 0.. ( 1 << log2CUSize ) – 1.

– If PartMode is equal to PART\_2NxnD, horEdgeFlags[ k ][ (1 << ( log2CUSize – 1 )) + (1 << ( log2CUSize – 2 )) ] is set equal to FilterInternalEdgesFlag for k = 0.. ( 1 << log2CUSize ) – 1.

– If PartMode is equal to PART\_nLx2N, verEdgeFlags[ (1 << ( log2CUSize – 1 )) – (1 << ( log2CUSize – 2 )) ][ k ] is set equal to FilterInternalEdgesFlag for k = 0.. ( 1 << log2CUSize ) – 1.

– If PartMode is equal to PART\_nRx2N, verEdgeFlags[ (1 << ( log2CUSize – 1 )) + (1 << ( log2CUSize – 2 )) ][ k ] is set equal to FilterInternalEdgesFlag for k = 0.. ( 1 << log2CUSize ) – 1.

#### Derivation process of boundary filtering strength

Inputs of this process are:

– a luma location ( xC, yC ) specifying the top-left luma sample of the current coding unit relative to the top-left luma sample of the current picture,

– a variable log2CUSize specifying the size of the current coding unit,

– a two-dimensional arrays of size (nS)x(nS), horEdgeFlags and verEdgeFlags.

Output of this process is an array of size (2)x(nS)x(nS), bS specifying the boundary filtering strength.

Let ( xEk, yEj ) with k = 0..nE-1 and j = 0..nE-1 specify a set of edge sample locations where nE is set equal to ( ( 1 << log2CUSize ) >> 3 ), xE0 = 0, yE0 = 0, xEk+1 = xEk + 8 and yEj+1 = yEj + 8.

For ( xEk, yEj ) with k = 0..nE-1 and j = 0..nE-1, the following applies.

* If horEdgeFlags[ xEk ][ yEj ] is equal to 1, for ( xEk+j, yEj ) with j=0, 1, the following applies.
* Set sample p0 = recPicture[ xC + xEk ][ yC + yEj – 1 ] and q0 = recPicture[ xC + xEk ][ yC + yEj ].
* The variable filterDir is set equal to 1.
* Otherwise, if verEdgeFlags[ xEk ][ yEj ] is equal to 1, for ( xEk, yEj +j) with j=0, 1, the following applies.
* Set sample p0 = recPicture[ xC + xEk – 1 ][ yC + yEj ] and q0 = recPicture[ xC + xEk ][ yC + yEj ].
* The variable filterDir is set equal to 0.
* Depending on the value of filterDir, the variable bS[ filterDir ][ xEk ][ yEj ] is derived as follows.
* If the following condition is true, the variable bS[ filterDir ][ xEk ][ yEj ] is set equal to 2.
* The sample p0 or q0 is in a coding unit coded with intra prediction mode
* Otherwise, if the block edge is also a transform unit edge and the following condition is true, the variable bS[ filterDir ][ xEk ][ yEj ] is set equal to 1.
* The sample p0 or q0 is in a transform unit which contains non-zero transform coefficient level.
* Otherwise, if any of the following conditions are true, the variable bS[ filterDir ][ xEk ][ yEj ] is set equal to 1.
* The prediction unit containing sample p0 has different reference pictures or a different number of motion vectors with the prediction unit containing the sample q0.

NOTE – The determination of whether the reference pictures used for the two prediction are the same or different is based on which pictures are referenced, without regard to whether a prediction is formed using an index into list 0 or an index into list 1, and also without regard to whether or not the index position within a reference picture list is different or not.

* If horEdgeFlags[ xEk ][ yEj ] is equal to 1 and yC + yEj – 1 is smaller than YLCU, the following applies.
  + - One motion vector is used to predict the prediction unit containing sample recPicture[ xL ][ yC + yEj – 1 ], one motion vector is used to predict the prediction unit containing sample q0, and the absolute difference between the horizontal or vertical component of the motion vector used is greater than or equal to 4 in units of quarter luma samples, where xL is set equal to ( ( ( xC + xEk ) >> Log2MinCUSize ) << Log2MinCUSize ) + ( ( ( xC + xEk ) >> Log2MinCUSize ) & 1 ) \* ( MinPuSize \* 2 – 1) and MinPuSize is set equal to 1<<(Log2MinCUSize – 1).
  + Otherwise, the following applies.
    - One motion vector is used to predict the prediction unit containing sample p0, one motion vector is used to predict the prediction unit containing sample q0, and the absolute difference between the horizontal or vertical component of the motion vector used is greater than or equal to 4 in units of quarter luma samples.
* [Ed.: (WJ) needs to be checked again whether this condition covers all 2-motion cases] Two motion vectors are used to predict the prediction unit containing sample p0, two motion vectors are used to predict the prediction unit containing sample q0, and at least one of the motion vector pairs corresponding the same reference pictures and the different boundary samples p0 and q0 satisfies the following condition:
  + - The absolute difference between the horizontal or vertical component of a motion vector used in the prediction of the two prediction units is greater than or equal to 4 in units of quarter luma samples.
* Otherwise, the variable bS[ filterDir ][ xEk ][ yEj ] is set equal to 0.

#### Filtering process for coding unit

Inputs of this process are:

– a luma location ( xC, yC ) specifying the top-left luma sample of the current coding unit relative to the top left luma sample of the current picture,

– a variable log2CUSize specifying the coding unit size,

– an array bS specifying the boundary filtering strength.

Output of this process is:

– modified reconstruction of the picture.

The filtering process for luma edges in the current coding unit consists of the following ordered steps:

1. The variable nD is set equal to 1 << ( log2CUSize – 3 ).
2. All elements of the four-dimensional array of size (2)x(nD)x(nD), dEdge are initialized to zero.
3. All elements of the three-dimensional array of size (2)x(nD)x(nD), bStrength ae initialized to zero.
4. For xDk set equal to xC+( k << 3 ), k=0..nD – 1, the following applies:

* For yDm set equal to yC+( m << 3 ), m=0..nD – 1, the following ordered steps apply:

1. Boundary filtering strength bSVer is derived as follows:

bSVer = Max( bS[ 0 ][ xDk ][ yDm + i ] ) for i = 0..7 (8‑428)

1. bStrength[1][k][m] is set equal to bSVer.
2. When bSVer is greater than 0, decision process for luma block edge in subclause 8.6.1.4.1 is invoked with the luma location of the coding unit ( xC, yC ), the luma location of the block ( xDk, yDm ), a variable verticalEdgeFlag set equal to 1, and the boundary filtering strength bSVer as inputs, the decisions dEdge[0][1][k][m] and dEdge[1][1][k][m] as outputs.
3. When bSVer is greater than 0, filtering process for luma block edge in subclause 8.6.1.4.2 is invoked with the luma location of the coding unit ( xC, yC ), the luma location of the block ( xDk, yDm ), a variable verticalEdgeFlag set equal to 1, the boundary filtering strength bStrength[1][k][m], the decisions dEdge[0][1][k][m] and dEdge[1][1][k][m] as inputs and the modified luma picture buffer as outputs.
4. For xDk set equal to xC + ( k << 3 ), k = 0..nD - 1, the following applies:

* For yDm set equal to yC+( m << 3 ), m=0..nD – 1, the following ordered steps apply:

1. Boundary filtering strength bSHor is derived as follows:

bSHor = Max( bS[ 1 ][ xDk + i ][ yDm ] ) for i = 0..7 (8‑428)

1. bStrength[0][k][m] is set equal to bSHor.
2. When bSHor is greater than 0, decision process for luma block edge in subclause 8.6.1.4.1 is invoked with the luma location of the coding unit ( xC, yC ), the luma location of the block ( xDk, yDm ), a variable verticalEdgeFlag set equal to 0, and the boundary filtering strength bSHor as inputs and the decisions dEdge[0][0][k][m] and dEdge[1][0][k][m] as outputs. [Ed. (WJ): comment from proponent – the decisions for the vertical filtering of the horizontal boundaries at the right hand side of a CU are being calculated based on the reconstructed signal and not based on the vertically filtered signal]
3. For yDm set equal to yC+( m << 3 ), m=0..nD - 1, the following applies:

* For xDk set equal to xC+( k << 3 ), k=0..nD – 1, the following ordered steps apply:
  + - If xDk is equal to 0, the parameter xPOS is set equal to 1. If xDk is equal to xC+( ( nD – 1) << 3 ) xPOS is set equal to 2. Otherwise xPOS is set to 0. [Ed. (WJ): comment from proponent – are horizontal edges at the right hand side of the picture really filtered vertically?]
    - dS[i] is set equal to dSample[0][m][ (k << 3) + i ] for i = 0..7.
    - When bSHor is greater than 0, filtering process for luma block edge in subclause 8.6.1.4.2 is invoked with the luma location of the coding unit ( xC, yC ), the luma location of the block ( xDk, yDm ), a variable verticalEdgeFlag set equal to 0, the boundary filtering strength bStrength[0][k][m], the decisions dEdge[0][0][k][m] and dEdge[1][0][k][m], xPOS, dSL[m][], dEL[m], bSL[m], and tCL[m], as inputs and the modified luma picture buffer as output.
* The elements of the array of size (nD), dEL are set as follows. dEL[m] is set equal to dEdge[1][0][k][m].
* The elements of the array of size (nD), bSL are set as follows. bSL[m] is set equal to bStrength[0][k][m].
* The elements of the array of size (nD), tC are set as follows. tCL[m] is set equal to tc. [Ed. (WJ) – comment from proponent – dEp and dEq should be buffered / the stored signal with subscript “L” seems to be used at a wrong position due to CU scan order (z)]

The filtering process for chroma edges in the current coding unit consists of the following ordered steps:

1. The variable nD is set equal to 1 << ( Max( log2CUSize, 4 ) – 4 ).
2. For xDk set equal to ( xC / 2 )+( k << 3 ), k=0..nD – 1, the following applies:

* For yDm set equal to ( yC / 2)+( m << 2 ), m=0..nD\*2 – 1, the following ordered steps apply:

1. Boundary filtering strength bSVer is derived as follows:

bSVer = bS[ 0 ][ xDk\*2 ][ yDm\*2 ] (8‑430)

1. When bSVer is greater than 0, filtering process for chroma block edge in subclause 8.6.1.4.3 is invoked with the chroma location of the coding unit ( xC/2, yC/2 ), the chroma location of the block ( xDk, yDm ), a variable verticalEdgeFlag set equal to 1, a chroma component index cIdx set equal to 1 and the boundary filtering strength bSVer as inputs and the modified chroma picture buffer as output.
2. When bSVer is greater than 0, filtering process for chroma block edge in subclause 8.6.1.4.3 is invoked with the chroma location of the coding unit ( xC/2, yC/2 ), the chroma location of the block ( xDk, yDm ), a variable verticalEdgeFlag set equal to 1, a chroma component index cIdx set equal to 2 and the boundary filtering strength bSVer as inputs and the modified chroma picture buffer as output.
3. For yDm set equal to ( yC / 2 )+( m << 3 ), m=0..nD – 1, the following applies:

* For xDk set equal to ( xC / 2 )+( k << 2 ), k=0..nD\*2 – 1, the following ordered steps apply:

1. If xDk is equal to 0, the parameter xPOS is set equal to 1. If xDk is equal to xB+( ( nD\*2 – 1) << 2 ) xPOS is set equal to 2. Otherwise xPOS is set to 0.
2. Boundary filtering strength bSHor is derived as follows:

bSHor = bS[ 1 ][ xDk\*2 ][ yDm\*2 ] (8‑431)

1. When bSHor is greater than 0, filtering process for chroma block edge in subclause 8.6.1.4.3 is invoked with the chroma location of the coding unit ( xC/2, yC/2 ), the chroma location of the block ( xDk, yDm ), a variable verticalEdgeFlag set equal to 0, a chroma component index cIdx set equal to 1 and the boundary filtering strength bSHor, xPOS, bSL[m] and tCL[m] as inputs and the modified chroma picture buffer as output.
2. When bSHor is greater than 0, filtering process for chroma block edge in subclause 8.6.1.4.3 is invoked with the chroma location of the coding unit ( xC/2, yC/2 ), the chroma location of the block ( xDk, yDm ), a variable verticalEdgeFlag set equal to 0, a chroma component index cIdx set equal to 2 and the boundary filtering strength bSHor, xPOS, bSL[m] and tCL[m] as inputs and the modified chroma picture buffer as output.

##### Decision process for luma block edge

Inputs of this process are:

– a luma location ( xC, yC ) specifying the top-left luma sample of the current coding unit relative to the top left luma sample of the current picture,

– a luma location ( xB, yB ) specifying the top-left luma sample of the current block relative to the top left luma sample of the current coding unit,

– a variable verticalEdgeFlag,

– a variable bS specifying the boundary filtering strength,

Output of this process is:

– variables dE[0], dE[1], dEp[0], dEp[1], dEq[0], and dEq[1] containing decisions,

This process is invoked only when bS is not equal to 0.

Let s’ represent the luma sample array recPictureL of the current picture.

A variable qPL is derived as follows:

qPL = ( ( QPY + QPP + 1 ) >> 1 )

where QPP is specified by the QPY value of the coding unit containing the sample p0,0. (8‑447)

A variables β is specified as Table 8‑13 with luma quantization parameter Clip3( 0, 51, qPL + ( beta\_offset\_div2 << 1 ) ) as input.

A variable tC is specified as Table 8‑13 with luma quantization parameter Clip3( 0, 55, qPL + 2\*( bS – 1) + ( tc\_offset\_div2 << 1) ) as input.

All elements of a one-dimensional array of size (4), dL are initialized to dL[0] = 0, dL[1] = 3, dL[2] = 4 and dL[3] = 7.

Depending on verticalEdgeFlag, the following applies:

– If verticalEdgeFlag is equal to 1, the following ordered steps apply:

1. The sample values pi,k and qi,k with i = 0..3 and k = 0, 3, 4, 7 are derived as follows:

qi,k = s’[ xC + xB +i, yC + yB + k ] (8‑447)

pi,k = s’[ xC + xB – i – 1, yC + yB + k ] (8‑448)

1. The elements of one dimensional arrays of size (),dp, dq and dpq are derived as follows with i=0..3:

dp[i] = | p2,dL[i] – 2\*p1,dL[i] + p0,dL[i] | (8‑449)

dq[i] = | q2,dL[i] – 2\*q1,dL[i] + q0,dL[i] | (8‑449)

dpq[i] = dp[i] + dq[i] (8‑449)

1. The elements of the one dimensional array d are derived as follows with i=0..1:

d[i] = dpq[ 2\*i ] + dpq[ 2\*i + 1 ] (8‑449)

1. All elements of the one dimensional arrays dE, dEp and dEq are set equal to 0.
2. For each j=0..1, if d[j] is less than β, the following ordered steps apply:
3. For each sample location ( xC + xB, yC + yB + dL[k] ), k = 0..1, the decision process for a luma sample specified in subclause 8.6.1.4.4 is invoked with sample values pi,dL[k], qi,dL[k] with i = 0..3, the variables 2\*dpq[ 2\*j + k ], β and tC as inputs and a decision dSam[k] as output.
4. The variable dE[j] is set equal to 1.
5. If dSam[0] is equal to 1 and dSam[1] is equal to 1, the variable dE[j] is set equal to 2.
6. If dp[ 2\*j ] + dp[ 2\*j + 1 ] is less than ( β + ( β >> 1 ) ) >> 3, the variable dEp[j] is set equal to 1.
7. If dq[ 2\*j ] + dq[ 2\*j + 1 ] is less than ( β + ( β >> 1 ) ) >> 3, the variable dEq[j] is set equal to 1.

– Otherwise (verticalEdgeFlag is equal to 0), the following ordered steps apply:

1. The sample values pi,k and qi,k with i = 0..3 and k = 0, 3, 4, 7 are derived as follows:

qi,k = s’[ xC + xB +k, yC + yB + i ] (8‑435)

pi,k = s’[ xC + xB +k, yC + yB – i – 1 ] (8‑436)

1. The elements of one dimensional array of size (4), dp, dq and dpq are derived as follows with i=0..3:

dp[i] = | p2,dL[i] – 2\*p1,dL[i] + p0,dL[i] | (8‑437)

dq[i] = | q2,dL[i] – 2\*q1,dL[i] + q0,dL[i] | (8‑437)

dpq[i] = dp[i] + dq[i] (8‑437)

1. The elements of the one dimensional array d are derived as follows with i=0..1:

d[i] = dpq[ 2\*i ] + dpq[ 2\*i + 1 ] (8‑449)

1. All elements of the one dimensional arrays dE, dEp and dEq are set equal to 0.
2. For each j=0..1, if d[j] is less than β, the following ordered steps apply:
3. For each sample location ( xC + xB + dL[k] + 4 \* j, yC + yB ), k = 0..1, the decision process for a luma sample specified in subclause 8.6.1.4.4 is invoked with sample values pi,dL[k], qi,dL[k] with i = 0..3, the variables 2\*dpq[ 2\*j + k ], β and tC as inputs and a decision dSam[k] as output.
4. The variable dE[j] is set equal to 1.
5. If dSam[0] is equal to 1 and dSam[1] is equal to 1, the variable dE[j] is set equal to 2.
6. If dp[ 2\*j ] + dp[ 2\*j + 1 ] is less than ( β + ( β >> 1 ) ) >> 3, the variable dEp[j] is set equal to 1.
7. If dq[ 2\*j ] + dq[ 2\*j + 1 ] is less than ( β + ( β >> 1 ) ) >> 3, the variable dEq[j] is set equal to 1.

##### Filtering process for luma block edge

Inputs of this process are:

– a luma location ( xC, yC ) specifying the top-left luma sample of the current coding unit relative to the top left luma sample of the current picture,

– a luma location ( xB, yB ) specifying the top-left luma sample of the current block relative to the top left luma sample of the current coding unit,

– a variable verticalEdgeFlag,

– a variable bS specifying the boundary filtering strength,

– variables dE[0], dE[1], dEp[0], dEp[1], dEq[0] and dEq[1] containing decisions,

– a variable bSL,

– a variable tCL,

Output of this process is:

– modified reconstruction of the picture.

This process is invoked only when bS is not equal to 0.

Let s’ represent the luma sample array recPictureL of the current picture.

Depending on pcm\_flag, a variable β is specified as follows:

– If pcm\_flag is equal to 1, the variables β is specified as Table 8‑11 with luma quantization parameter 0 as input.

– Otherwise, the variables β is specified as Table 8‑11 with luma quantization parameter qPL as input.

A variable tC is specified as Table 8‑11 with luma quantization parameter Clip3( 0, 55, qPL + 2\*(bS – 1) + ( tc\_offset\_div2 << 1 ) ) as input.

Depending on verticalEdgeFlag, the following applies:

– If verticalEdgeFlag is equal to 1, the following ordered steps apply:

1. The sample values pi,k and qi,k with i = 0..3 and k = 0..7 are derived as follows:

qi,k = s’[ xC + xB +i, yC + yB + k ] (8‑432)

pi,k = s’[ xC + xB – i – 1, yC + yB + k ] (8‑433)

1. For each m=0..1, if dE[m] is not equal to 0, for each sample location ( xC + xB, yC + yB + k + 4\*m ), k = 0..3, the following ordered steps apply:
2. The filtering process for a luma sample specified in subclause 8.6.1.4.5 is invoked with sample values pi,k, qi,k with i = 0..3, the decision dE[m], variables dEp[m] and dEq[m], the variable tC as inputs and the number of filtered samples nDp and nDq from each side of the block boundary, and the filtered sample values pi’ and qj’ as outputs.
3. The filtered sample values pi’ and qj’ with i = 0..nDp – 1, j = 0..nDq – 1 replace the corresponding samples inside the sample array s’ as follows:

s’[ xC + xB +j, yC + yB + k + 4\*m ] = qj’ (8‑435)

s’[ xC + xB – i – 1, yC + yB + k + 4\*m ] = pi’ (8‑436)

– Otherwise (verticalEdgeFlag is equal to 0), the following ordered steps apply:

1. If xPOS is equal to 1, the parameters ks and ke are set to -3 and 4 respectively. If xPOS is equal to 2, the parameters ks and ke are set to 0 and 4 respectively. Otherwise ks and ke are set to 0 and 7 respectively.
2. The sample values pi,k and qi,k with i = 0..3 and k = ks..ke are derived as follows:

qi,k = s’[ xC + xB +k, yC + yB + i ] (8‑437)

pi,k = s’[ xC + xB +k, yC + yB – i – 1 ] (8‑438)

1. If xPOS is less than 0 [Ed. (WJ): maybe wrong condition] and dEL is not equal to 0, for each sample location ( xC + xB + k, yC + yB ), k = -3..-1, the following ordered steps apply:
2. The filtering process for a luma sample specified in subclause 8.6.1.4.5 is invoked with sample values pi,k, qi,k with i = 0..3, decision arrays dEL, dEpL and dEqL [Ed. (WJ): dEpL and dEqL do not exist. Needs to be stored], the variable tCL as inputs and the number of filtered samples nDp and nDq from each side of the block boundary and the filtered sample values pi’ and qj’ as outputs.
3. The filtered sample values pi’ and qj’ with i = 0..nDp – 1, j = 0..nDq – 1 replace the corresponding samples inside the sample array s’ as follows:

s’[ xC + xB +k, yC + yB + j ] = qj’ (8‑459)

s’[ xC + xB +k, yC + yB – i – 1 ] = pi’ (8‑460)

1. For each m=0..1, if dE[m] is not equal to 0, for each sample location ( xC + xB + k + 4\*m, yC + yB ), k = 0.. ke-4 , the following ordered steps apply:
2. The filtering process for a luma sample specified in subclause 8.6.1.4.5 is invoked with sample values pi,k, qi,k with i = 0..3, decision dE[m], variables dEp[m] and dEq[m], the variable tC as inputs and the number of filtered samples nDp and nDq from each side of the block boundary and the filtered sample values pi’ and qj’ as outputs.
3. The filtered sample values pi’ and qj’ with i = 0..nDp – 1, j = 0..nDq – 1 replace the corresponding samples inside the sample array s’ as follows:

s’[ xC + xB + k + 4\*m, yC + yB + j ] = qj’ (8‑461)

s’[ xC + xB + k + 4\*m, yC + yB – i – 1 ] = pi’ (8‑462)

##### Filtering process for chroma block edge

[Ed.: (WJ) cIdx cannot be 0 here]

Inputs of this process are:

– a luma location ( xC, yC ) specifying the top-left chroma sample of the current coding unit relative to the top left chroma sample of the current picture,

– a luma location ( xB, yB ) specifying the top-left chroma sample of the current block relative to the top left chroma sample of the current coding unit,

– a variable verticalEdgeFlag,

– a variable bS specifying the boundary filtering strength,

– a variable cIdx specifying the chroma component index.

– a variable xPOS,

– a variable bSL,

– a variable tCL

Output of this process is:

– modified reconstruction of the picture.

This process is invoked only when bS is not equal to 0.

Let s’ be a variable specifying chroma sample array which is derived as follows.

– If cIdx is equal to 1, s’ represents the chroma sample array recPictureCb of the current picture.

– Otherwise (cIdx is equal to 2), s’ represents the chroma sample array recPictureCr of the current picture.

A variable qPL is derived as follows:

qPL = ( ( QPY + QPP + 1 ) >> 1 )

where QPP is specified by the QPY value of the coding unit containing the sample p0,0. (8‑447)

A variable tC is specified as Table 8‑13 with luma quantization parameter Clip3( 0, 55, qPL + 2\*(bS – 1) + (tc\_offset\_div2 << 1 ) ) as input.

Depending on verticalEdgeFlag, the following applies:

– If verticalEdgeFlag is equal to 1, for each sample location ( xC + xB, yC + yB + k ), k = 0..3, the following ordered steps apply:

1. The sample values pi and qi with i = 0..1 are derived as follows:

qi = s’[ xC + xB +i, yC + yB + k ] (8‑442)

pi = s’[ xC + xB – i – 1, yC + yB + k ] (8‑443)

1. The following ordered steps apply:
2. The filtering process for a sample specified in subclause 8.6.1.4.6 is invoked with sample values pi, qi, with i = 0..1, the boundary filtering strength bS and the variable tC as inputs and the filtered sample values p0’ and q0’ as outputs.
3. The filtered sample values p0’ and q0’ replace the corresponding samples inside the sample array s’ as follows:

s’[ xC + xB , yC + yB + k ] = q0’ (8‑444)

s’[ xC + xB – 1, yC + yB + k ] = p0’ (8‑445)

– Otherwise (verticalEdgeFlag is equal to 0), the following ordered steps apply:

1. If xPOS is equal to 1, the parameters ks and ke are set to -1 and 2 respectively. If xD is equal to 2, the parameters ks and ke are set to 0 and 2 respectively. Otherwise ks and ke are set to 0 and 3 respectively.
2. The sample values pi and qi with i = 0..1 and k = ks..ke are derived as follows:

qi = s’[ xC + xB +k, yC + yB + i ] (8‑446)

pi = s’[ xC + xB +k, yC + yB – i – 1 ] (8‑447)

1. If xPOS is less than 0, and if bSL, is greater than 1, for each sample location ( xC + xB - 1, yC + yB ), the following ordered steps apply:
2. The filtering process for a sample specified in subclause 8.6.1.4.6 is invoked with sample values pi, qi, with i = 0..1, the boundary filtering strength bSL and the variable tCL as inputs and the filtered sample values p0’ and q0’ as outputs.
3. The filtered sample values p0’ and q0’ replace the corresponding samples inside the sample array s’ as follows:

s’[ xC + xB +k, yC + yB ] = q0’ (8‑452)

s’[ xC + xB +k, yC + yB – i – 1 ] = pi’ (8‑445)

1. If bS is greater than 1, for each sample location ( xC + xB + k, yC + yB ), k = 0.. ke, the following ordered steps apply:
2. The filtering process for a sample specified in subclause 8.6.1.4.6 is invoked with sample values pi, qi, with i = 0..1, the boundary filtering strength bS and the variable tC as inputs and the filtered sample values p0’ and q0’ as outputs.
3. The filtered sample values p0’ and q0’ replace the corresponding samples inside the sample array s’ as follows:

s’[ xC + xB +k, yC + yB ] = q0’ (8‑448)

s’[ xC + xB +k, yC + yB – 1 ] = p0’ (8‑449)

##### Decision process for a luma sample

[Ed: (WJ) no filtering when bS is equal to 0]

Inputs of this process are:

– sample values, pi and qi with i = 0..2,

– variables d, β and tC.

Output of this process is:

– a variable dSam containing a decision

The variable dSam is specified as follows:

– If d is less than ( β >> 2 ), | p3 – p0 | + | q0 – q3 | is less than ( β >> 3 ) and | p0 – q0 | is less than ( 5\*tC + 1 ) >> 1, dSam is set equal to 1.

– Otherwise, dSam is set equal to 0.

##### Filtering process for a luma sample

Inputs of this process are:

– sample values, pi and qi with i = 0..3,

– variables dEp1 and dEq1 containing decisions to filter pixels p1 and q1 respectively,

– a variable tC.

Output of this process is:

– number of filtered samples nDp and nDq,

– filtered sample values, pi’ and qj’ with i = 0..nDp – 1, j = 0..nDq – 1

Depending on dE, the following applies:

– When the variable dE is equal to 2, the following strong filtering applies while nDp and nDq are set equal to 3:

p0’ = ( p2 + 2\*p1 + 2\*p0 + 2\*q0 + q1 + 4 ) >> 3  (8‑450)

p1’ = ( p2 + p1 + p0 + q0 + 2 ) >> 2  (8‑451)

p2’ = ( 2\*p3 + 3\*p2 + p1 + p0 + q0 + 4 ) >> 3  (8‑452)

q0’ = ( p1 + 2\*p0 + 2\*q0 + 2\*q1 + q2 + 4 ) >> 3  (8‑453)

q1’ = ( p0 + q0 + q1 + q2 + 2 ) >> 2  (8‑454)

q2’ = ( p0 + q0 + q1 + 3\*q2 + 2\*q3 + 4 ) >> 3  (8‑455)

– Otherwise, the following weak filtering applies while nDp and nDq are set equal to 0:

Δ = ( 9 \* ( q0 –  p0 ) – 3 \* ( q1 – p1 ) + 8 ) >> 4 (8‑456)

* + When abs(Δ) is less than tc\*10, the following ordered steps apply:
    1. The filtered sample values p0’ and q0’ are specified as follows:

Δ = Clip3( -tc, tc, Δ ) (8‑456)

p0’ = Clip1Y( p0 + Δ ) (8‑457)

q0’ = Clip1Y( q0 - Δ ) (8‑458)

* + 1. If dEp1 is equal to 1, the filtered sample value pi’ is specified as follows:

Δp = Clip3( -(tc >> 1), tc >> 1, ( ( ( p2 + p0 + 1 ) >> 1 ) – p1 + Δ ) >>1 ) (8‑458)

pi’ = Clip1Y( p1 + Δp ) (8‑458)

* + 1. If dEq1 is equal to 1, the filtered sample value qi’ is specified as follows:

Δq = Clip3( -(tc >> 1), tc >> 1, ( ( ( q2 + q0 + 1 ) >> 1 ) – q1 – Δ ) >>1 ) (8‑458)

qi’ = Clip1Y( q1 + Δq ) (8‑458)

* + 1. nDp is set equal to dEp1+1 and nDq is set equal to dEq1+1.

Each of the filtered sample values, pi’ with i = 0..nDp-1, is substituted by the corresponding input sample value pi if all of the following conditions are true.

– pi is a sample of an I\_PCM block.

– pcm\_loop\_filter\_disable\_flag value is equal to 1.

Similarly, each of the filtered sample values, qj’ with j = 0..nDq-1, is substituted by the corresponding input sample value qj if all of the following conditions are true.

– qj is a sample of an I\_PCM block.

– pcm\_loop\_filter\_disable\_flag value is equal to 1.

[Ed. (WJ): for PCM case, deblocking filter applies first and the filtered pixels are restored. Rather than this, it’s better to skip the filtering itself for PCM samples since first filtering is actually not needed.]

Table 8‑14 – Derivation of threshold variables β and tC from input Q

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Q** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| **β** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6 | 7 | 8 |
| **tC** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| **Q** | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 |
| **β** | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 20 | 22 | 24 | 26 | 28 | 30 | 32 | 34 | 36 |
| **tC** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 4 | 4 | 4 |
| **Q** | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 |  |
| **β** | 38 | 40 | 42 | 44 | 46 | 48 | 50 | 52 | 54 | 56 | 58 | 60 | 62 | 64 | 64 | 64 | 64 | 64 |  |
| **tC** | 5 | 5 | 6 | 6 | 7 | 8 | 9 | 9 | 10 | 10 | 11 | 11 | 12 | 12 | 13 | 13 | 14 | 14 |  |

##### Filtering process for a chroma sample

[Ed: (WJ) no filtering when bS is equal or less than 1]

Inputs of this process are:

– sample values, pi and qi with i = 0..1,

– a variable tC.

Output of this process is:

– The filtered sample values, p0’ and q0’.

The filtered sample values p0’ and q0’ are derived by

Δ = Clip3( -tC, tC, ( ( ( ( q0 – p0 ) << 2 ) + p1 – q1 + 4 ) >> 3 ) ) (8‑461)

p0’ = Clip1C( p0 + Δ ) (8‑462)

q0’ = Clip1C( q0 - Δ ) (8‑463)

The filtered sample value, p0’ is substituted by the corresponding input sample value p0 if all of the following conditions are true.

– p0 is a sample of an I\_PCM block.

– pcm\_loop\_filter\_disable\_flag value is equal to 1.

Similarly, the filtered sample value, q0’ is substituted by the corresponding input sample value q0 if all of the following conditions are true.

– q0 is a sample of an I\_PCM block.

– pcm\_loop\_filter\_disable\_flag value is equal to 1.

[Ed. (WJ): for PCM case, deblocking filter applies first and the filtered pixels are restored. Rather than this, it’s better to skip the filtering itself for PCM samples since first filtering is actually not needed.]

### Sample adaptive offset process

A sample adaptive offset process shall be conditionally performed after the completion of the deblocking filter process for the decoded picture. This process is invoked when both sample\_adaptive\_offset\_enabled\_flag and aps\_sample\_adaptive\_offset\_flag are equal to 1.

This process is performed on a region basis which is aligned with the LCU boundaries after the completion of the picture construction process prior to sample adaptive offset process for the entire decoded process.

The sample adaptive offset process is invoked with the region equal to entire picture here.

Depending on aps\_sample\_adaptive\_offset\_flag, sao\_flag\_cb and sao\_flag\_cr, the following applies:

– When aps\_sample\_adaptive\_offset\_flag is equal to 1, the following applies:

* + The sample adaptive offset process for luma region as specified in subclause 8.6.2.1 is invoked with the component index set equal to 0, the region index ( 0, 0 ), the largest CU location ( 0, 0 ), the variable saoDepth set equal to 0, the variable saoWidthInLCUs set equal to PicWidthInLCUs, and the variable saoHeightInLCUs set equal to PicHeightInLCUs, and the modified luma picture buffer of current region as output.
  + When sao\_flag\_cb is equal to 1, the sample adaptive offset process for color component Cb as specified in subclause 8.6.2.1 is invoked with the component index set equal to 1, the region index ( 0, 0 ), the largest CU location ( 0, 0 ), the variable saoDepth set equal to 0, the variable saoWidthInLCUs set equal to PicWidthInLCUs, and the variable saoHeightInLCUs set equal to PicHeightInLCUs, and the modified chroma picture buffer for Cb of current region as output.
  + If sao\_flag\_cr is equal to 1, the sample adaptive offset process for color component Cr as specified in subclause 8.6.2.1 is invoked with the component index set equal to 2, the region index ( 0, 0 ), the largest CU location ( 0, 0 ), the variable saoDepth set equal to 0, the variable saoWidthInLCUs set equal to PicWidthInLCUs, and the variable saoHeightInLCUs set equal to PicHeightInLCUs, and the modified chroma picture buffer for Cr of current region as output.

#### Modification process for sample adaptive offset

Inputs to this process are:

– a variable cIdx specifying chroma component index,

– a pair of variables ( rx, ry ) specifying the region index,

– a sample location ( x0, y0 ) specifying the top-left sample of the current largest coding unit,

– a variable saoDepth specifying the split depth of the region,

– a variable saoWidthInLCUs specifying the number of largest CUs in width of the current region

– a variable saoHeightInLCUs specifying the number of largest CUs in height of the current region.

Output of this process is a modified picture buffer for the chroma component cIdx.

Depending on sao\_split\_flag[ cIdx ][ saoDepth ][ rx ][ ry ], the following applies:

– If sao\_split\_flag[cIdx][saoDepth][rx][ry] is equal to 1 and saoDepth is smaller than SaoMaxDepth, the following ordered stesps apply:

1. Variable nS is set equal to ( 1 << Log2MaxCUSize ).
2. Variables saoWidthInLCU1, saoWidthInLCU2, saoHeightInLCU1, saoHeightInLCU2, x1, y1, x2, and y2 are derived as follows:

saoWidthInLCU1 = saoWidthInLCUs >> 1 (8‑463)  
 saoWidthInLCU2 = saoWidthInLCUs – saoWidthInLCU1 (8‑463)  
 saoHeightInLCU1 = saoHeightInLCUs >> 1 (8‑463)  
 saoHeightInLCU2 = saoHeightInLCUs – saoHeightInLCU1 (8‑463)

x1 = x0 (8‑463)  
 y1 = y0 (8‑463)  
 x2 = x1 + saoWidthInLCU1 \* nS (8‑463)  
 y2 = y1 + saoHeightInLCU1 \* nS (8‑463)

1. The sample adaptive offset process for region as specified in this subclause is invoked with the component index cIdx, the region index ( 2\*rx, 2\*ry ), the top-left sample position of largest CU ( x0, y0 ), the variable saoDepth set equal to ( saoDepth+1 ), the variable saoWidthInLCUs set equal to saoWidthInLCU1, and the variable saoHeightInLCUs set equal to saoHeightInLCU1.
2. The sample adaptive offset process for region as specified in this subclause is invoked with the component index cIdx, the region index ( 2\*rx+1, 2\*ry ), the top-left sample position of largest CU ( x1, y0 ), the variable saoDepth set equal to ( saoDepth+1 ), the variable saoWidthInLCUs set equal to saoWidthInLCU2, and the variable saoHeightInLCUs set equal to saoHeightInLCU1.
3. The sample adaptive offset process for region as specified in this subclause is invoked with the component index cIdx, the region index ( 2\*rx, 2\*ry+1 ), the top-left sample position of largest CU ( x0, y1 ), the variable saoDepth set equal to ( saoDepth+1 ), the variable saoWidthInLCUs set equal to saoWidthInLCU1, and the variable saoHeightInLCUs set equal to saoHeightInLCU2.
4. The sample adaptive offset process for region as specified in this subclause is invoked with the component index cIdx, the region index ( 2\*rx+1, 2\*ry+1 ), the top-left sample position of largest CU ( x1, y1 ), the variable saoDepth set equal to ( saoDepth+1 ), the variable saoWidthInLCUs set equal to saoWidthInLCU2, and the variable saoHeightInLCUs set equal to saoHeightInLCU2.

* Otherwise, (sao\_split\_flag[ cIdx ][ saoDepth ][ rx ][ ry ] is equal to 0), the following ordered steps apply:

1. Variable nS is set equal to ( 1 << Log2MaxCUSize ).
2. Modification process for samples in a largest coding unit covering ( iC, jC ) with iC = x0 + k\*nS for k = 0..saoWidthInLCU – 1 and jC = y0 + j\*nS for j = 0..saoHeightInLCU – 1 specified in subclause 8.6.2.1.1 is invoked with the component index cIdx, the top-left sample of current largest coding unit ( iC, jC ), the region index ( rx, ry ), the split depth saoDepth, the offset array saoValueArray set equal to SaoOffsetVal[ cIdx ][ saoDepth ][ rx ][ ry ] and the largest coding unit size nS as inputs and the output is the modified picture buffer of chroma component cIdx.

##### Modification process for luma and chroma samples

Inputs to this process are:

– a variable cIdx specifying chroma component index,

– a sample position ( xC, yC ),

– a pair of variables ( rx, ry ) specifying the region index,

– a variable saoDepth specifying the split depth of the region,

– an array saoValueArray specifying offset values,

– a block size nS.

Output of this process is a modified picture buffer for the chroma component cIdx.

Let recSaoPicture represents the processed sample array of the current picture of chroma component cIdx and saoTypeIdx is set equal to sao\_type\_idx[ cIdx ][ saoDepth ][ rx ][ ry ].

Variable bitDepth is set equal to BitDepthY if cIdx is equal to 0, otherwise, set equal to BitDepthC.

Depending on the value of saoTypeIdx, the following applies:

– If saoTypeIdx is equal to one of the values of 1, 2, 3 or 4, the following ordered steps apply:

1. Arrays hPos[2] and vPos[2] are specified in Table 8‑16.
2. A variable edgeIdx is specified as

edgeIdx = 2 + ∑k( Sign( recPicture[ xC + i, yC + j ] –

recPicture[ xC + i + hPos[ k ], yC + j + vPos[ k ] ] ) ) with k = 0..1 (8‑463)

1. The reconstructed picture buffer is modified as

recSaoPicture[ xC + i, yC + j ] = recPicture[ xC + i, yC + j ] + saoValueArray[ edgeTable[ edgeIdx ] ] (8‑463)

with i = 0..nS-1 and j = 0..nS-1 where edgeTable[5] = { 1, 2, 0, 3, 4}.

– Otherwise, if saoTypeIdx is equal to one of the values of 5 or 6, the following ordered steps applies:

1. A variable bandShift is set equal to BitDepthY – 5 if cIdx is equal to 0, otherwise, set equal to BitDepthC – 5.
2. The reconstructed picture buffer is modified as

recSaoPicture[ xC + i, yC + j ] = recPicture[ xC + i, yC + j ] +

saoValueArray[ bandTable[ saoTypeIdx – 5 ][ bandIdx ] ] (8‑463)

with i = 0..nS-1 and j = 0..nS-1 where bandIdx is set equal to ( recPicture[ xC + i, yC + j ] >> bandShift ) and bandTable is specified in Table 8‑17.

– Otherwise (sao\_type\_idx[ cIdx ][ saoDepth ][ rx ][ ry ] is equal to 0), the following applies:

recSaoPicture[ xC + i, yC + j ] = recPicture[ xC + i, yC + j ] with i = 0..nS-1 and j = 0..nS-1 (8‑463)

[Ed. (WJ): copy operation is necessary to use recSaoPicture later.]

Table 8‑15 – Specification of hPos[2] and vPos[2] according to the type of sample adaptive offset process

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| sao\_type\_idx[ cIdx ][ saoDepth ][ rx ][ ry ] | 1 | 2 | 3 | 4 |
| hPos[0] | -1 | 0 | -1 | 1 |
| hPos[1] | 1 | 0 | 1 | -1 |
| vPos[0] | 0 | -1 | -1 | 1 |
| vPos[1] | 0 | 1 | 1 | -1 |

Table 8‑16 – Specification of array bandTable according to bandIdx

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| bandIdx | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| bandTable[0][bandIdx] | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| bandTable[1][bandIdx] | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| bandIdx | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| bandTable[0][bandIdx] | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| bandTable[1][bandIdx] | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |

### Adaptive loop filter process

An adaptive loop filtering process shall be conditionally performed on a treeblock basis after the completion of the sample adaptive offset process for the entire decoded picture, with all treeblocks in a picture processed in order of increasing treeblock addresses.

Each treeblock is processed on a coding unit basis with the same order as decoding process.

This process is invoked when both adaptive\_loop\_filter\_enabled\_flag and aps\_adaptive\_loop\_filter\_flag are equal to 1.

This process is performed on a coding unit basis after the completion of the slice construction process prior to adaptive loop filter process for the entire decoded slice, with all coding units in a slice processed in order of coding unit scan order.

Filter coefficients cL for luma samples and cC for chroma samples are derived by invoking the process specified in subclause 8.6.2.1.Filter index array fIdx for luma samples are derived by invoking the process specified in subclause 8.6.2.2 with the luma location ( xC, yC ) specifying the top-left luma sample of the current coding unit relative to the top left luma sample of the current picture and the coding unit size log2CUSize as inputs.

When interpreting the luma samples of the coding unit as to be filtered, depending on pcm\_flag, pcm\_loop\_filter\_disable\_flag and alf\_cu\_control\_flag, the following applies.

* If pcm\_loop\_filter\_disable\_flag is equal to 1 and pcm\_flag is equal to 1, the luma samples are not filtered.
* Otherwise, if alf\_cu\_control\_flag is equal to 0, the luma samples are filtered,
* Otherwise (alf\_cu\_control\_flag is equal to 1), if AlfCuFlag[ xC ][ yC ] is equal to 1 where xC and yC are the top-left luma sample of the current coding unit relative to the top left luma sample of the current picture, the luma samples are filtered.
* Otherwise, the luma samples are not filtered.

When interpreting the chroma samples of the coding unit as to be filtered, depending on pcm\_flag, pcm\_loop\_filter\_disable\_flag and alf\_chroma\_idc, the following applies.

* If pcm\_loop\_filter\_disable\_flag is equal to 1 and pcm\_flag is equal to 1, both chroma samples are not filtered.
* Otherwise, if alf\_chroma\_idc is equal to 1, the Cr samples are filtered.
* Otherwise, if alf\_chroma\_idc is equal to 2, the Cb samples are filtered.
* Otherwise, if alf\_chroma\_idc is equal to 3, both chroma samples are filtered.
* Otherwise (alf\_chroma\_idc is equal to 0), both chroma samples are not filtered.

For the luma samples of the coding unit interpreted as to be filtered, the filtering process for luma samples specified in subclause 8.6.2.3 is invoked with the luma location ( xC, yC ) specifying the top-left luma sample of the current coding unit relative to the top left luma sample of the current picture, the coding unit size log2CUSize, and the filter index array fIdx as inputs and the output is the modified filtered picture, recFiltPictureL.

For the chroma samples of the coding unit interpreted as to be filtered, the filtering process for chroma samples specified in subclause 8.6.2.4 is invoked with the chroma location ( xC/2, yC/2 ) specifying the top-left chroma sample of the current coding unit relative to the top left chroma sample of the current picture, the coding unit size log2CUSize-1 and the chroma component index cIdx equal to 1 as inputs and the output is the modified filtered picture, recFiltPictureCb.

For the chroma samples of the coding unit interpreted as to be filtered, the filtering process for chroma samples specified in subclause 8.6.2.4 is invoked with the chroma location ( xC/2, yC/2 ) specifying the top-left chroma sample of the current coding unit relative to the top left chroma sample of the current picture, the coding unit size log2CUSize-1 and the chroma component index cIdx equal to 2 as inputs and the output is the modified filtered picture, recFiltPictureCr.

[Ed.: (WJ) recPicture: deblocked/output picture and recFiltPicture: ALFed picture]

[Ed.: (WJ) depending aps\_adaptive\_loop\_filter\_flag, recFiltPicture should be copied to recPicture in subclause 8.6]

#### Boundary padding process

Inputs of this process are:

– a location ( xC, yC ) specifying the top-left sample of the current coding unit relative to the top left sample of the current picture,

– a variable log2CUSize specifying the size of the current coding unit.

– a variable cIdx specifying the chroma component index.

Output of this process is the padded sample array s’’.

A variable nExtPixels is set equal to 4. [Ed. (WJ): do we need to use a variable instead of constant 4?]

Depending on cIdx, the following applies:

* If cIdx is equal to 0, a luma sample array s’ is set equal to recPictureL, a variable nS is set equal to ( 1 << log2CUSize ), and a variable nB is set equal to Min( ( 1<< log2CUSize ), ( (1 << log2MaxCUSize ) >> slice\_granularity ) ).
* Otherwise, if cIdx is equal to 1, a chroma sample array s’ is set equal to recPictureCb, a variable nS is set equal to ( 1 << ( log2CUSize – 1 ) ), and a variable nB is set equal to Min( ( 1 << ( log2CUSize – 1 ) ), ( ( 1 << ( log2MaxCUSize – 1 ) ) >> slice\_granularity ) ).
* Otherwise (cIdx is equal to 2), a chroma sample array s’ is set equal to recPictureCr, a variable nS is set equal to ( 1 << ( log2CUSize – 1 ) ), and a variable nB is set equal to Min( ( 1 << ( log2CUSize – 1 ) ), ( ( 1 << ( log2MaxCUSize – 1 ) ) >> slice\_granularity ) ).

The boundary padding process for each granularity block specified in subclause 8.6.3.1.1 is invoked with locations ( xC + p\*nB, yC + q\*nB ) where p = 0..( ( nS/nB ) – 1 ) and q=0..( ( nS/nB ) – 1 ), the size of granularity block nB, the padding size nExtPixels and the output is assigned to the sample array s’’.

##### Boundary padding process for granularity block

Inputs of this process are:

– a location ( xC, yC ) specifying the top-left sample of the granularity block relative to the top left sample of the current picture,

– a variable nB specifying the size of the granularity block size,

– a variable nExtPixels specifying the padding size,

Output of this process is the padded sample array s’’.

The padded sample array s’’ is derived as follows.

* If both loop\_filter\_across\_slice\_flag and loop\_filter\_across\_tile\_flag are equal to 1, s’’[ x ][ y ] is set equal to s’[ x ][ y ] for x = (xC-nExtPixels)..(xC+nExtPixels-1) and y = (yC-nExtPixels)..(yC+nExtPixels-1).
* Otherwise (loop\_filter\_across\_slice\_flag is equal to 0 or loop\_filter\_across\_tile\_flag is equal to 0), s’’[ x ][ y ] is derived depending on the slice and tile relationship between neighbouring regions specified in Figure 8‑6 and Table 8‑16 where the function Rs(·) is specified for any two regions A and B as follows: [Ed. (WJ): large table does not seem to be the best way to specify the process. Traditional way (if-clause) may be better]
  + Rs(A) == Rs(B) indicates region A and region B are at the same slice and at the same tile.
  + Rs(A) != Rs(B) indicates region A and region B are not at the same slice and at the same tile.

설명: D:\Documents and Settings\mtk03093\Desktop\Picture3.emf

Figure 8‑6 – Region definition for granularity block in slice boundary padding process in adaptive loop filter including input block region (shaded block) and padded regions (unshaded blocks)

Table 8‑17 – Specification of padded sample array s’’[x][y] according to sample location

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Location (x,y) | Condition | | | Output derivation |
| G |  | | | s’’[x][y]= s’[x][y] |
| PL,1 | Rs(PL,1) == Rs(G) | | | s’’[x][y]= s’[x][y] |
| Rs(PL,1) != Rs(G) | | | s’’[x][y]= s’[xC][y] |
| PL,2 | Rs(PL,2 ) == Rs(G) | | | s’’[x][y]= s’[x][y] |
| Rs(PL,2 ) != Rs(G) | Rs(PBL) == Rs(G) | | s’’[x][y]= s’[x][yC+ nB] |
| Rs(PBL) != Rs(G) | | s’’[x][y]= s’[xC][y] |
| PR | Rs(PR) == Rs(G) | | | s’’[x][y]= s’[x][y] |
| Rs(PR) != Rs(G) | | | s’’[x][y]= s’[xC+ nB-1][y] |
| PT | Rs(PT ) == Rs(G) | | | s’’[x][y]= s’[x][y] |
| Rs(PT ) != Rs(G) | | | s’’[x][y]= s’[x][yC] |
| PB,1 | Rs(PB,1) == Rs(G) | | | s’’[x][y]= s’[x][y] |
| Rs(PB,1) != Rs(G) | Rs(PBL) == Rs(G) | | s’’[x][y]= s’[xC-1][y] |
| Rs(PBL) != Rs(G) | | s’’[x][y]= s’[x][yC+nB-1] |
| PB,2 | Rs(PB,2) == Rs(G) | | | s’’[x][y]= s’[x][y] |
| Rs(PB,2) != Rs(G) | | | s’’[x][y]= s’[x][yC+nB-1] |
| PTL | Rs(PTL) == Rs(G) | | | s’’[x][y]= s’[x][y] |
| Rs(PTL) != Rs(G) | Rs(PL,1) == Rs(G) | | s’’[x][y]= s’[x][yC] |
| Rs(PL,1) != Rs(G) | Rs(PT) == Rs(G) | s’’[x][y]= s’[xC][y] |
| Rs(PT) != Rs(G) | s’’[x][y]= s’[xC][yC] |
| PTR | Rs(PTR)== Rs(G) | | | s’’[x][y]= s’[x][y] |
| Rs(PTR)!= Rs(G) | Rs(PR) == Rs(G) | | s’’[x][y]= s’[x][yC] |
| Rs(PR) != Rs(G) | Rs(PT) == Rs(G) | s’’[x][y]= s’[xC+nB-1][y] |
| Rs(PT) != Rs(G) | s’’[x][y]= s’[xC+nB-1][yC] |
| PBL | Rs(PBL)== Rs(G) | | | s’’[x][y]= s’[x][y] |
| Rs(PBL)!= Rs(G) | Rs(PL,2) == Rs(G) | | s’’[x][y]= s’[x][yC+nB-1] |
| Rs(PL,2) != Rs(G) | Rs(PB,1) == Rs(G) | s’’[x][y]= s’[xC][y] |
| Rs(PB,1) != Rs(G) | s’’[x][y]= s’[xC][yC+nB-1] |
| PBR | Rs(PBR)== Rs(G) | | | s’’[x][y]= s’[x][y] |
| Rs(PBR)!= Rs(G) | Rs(PR) == Rs(G) | | s’’[x][y]= s’[x][yC+nB-1] |
| Rs(PR) != Rs(G) | Rs(PB,2) == Rs(G) | s’’[x][y]= s’[xC+nB-1][y] |
| Rs(PB,2) != Rs(G) | s’’[x][y]= s’[xC+nB-1][yC+nB-1] |

#### Derivation process for filter coefficients

Outputs of this process are filter coefficients cL for the luma samples and filter coefficients cC for the chroma samples.

The luma filter coefficients cL with elements cL[ i ][ j ], i = 0..AlfNumFilters–1, j = 0..AlfCodedLengthLuma–1 are derived as follows:

* If alf\_nb\_pred\_luma[ i ] is equal to 0,
* If alf\_pred\_method is equal to 0 or the value of i is equal to 0,

cL[ i ][ j ] = alf\_coeff\_luma[ i ][ j ] (8‑464)

* Otherwise (alf\_pred\_method is equal to 1 and the value of i is greater than 1),

cL[ i ][ j ] = alf\_coeff\_luma[ i ][ j ] + cL[ i – 1 ][ j ] (8‑465)

* Otherwise (alf\_nb\_pred\_luma[i] is equal to 1), the following ordered steps apply:

1. The luma filter coefficients cL with elements cL[ i ][ j ], i = 0..AlfNumFilters–1, j = 0..AlfCodedLengthLuma–3 are derived as follows:
   * 1. If alf\_pred\_method is equal to 0 or the value of i is equal to 0,

cL[ i ][ j ] = alf\_coeff\_luma[ i ][ j ] (8‑464)

* + 1. Otherwise (alf\_pred\_method is equal to 1 and the value of i is greater than 1),

cL[ i ][ j ] = alf\_coeff\_luma[ i ][ j ] + cL[ i – 1 ][ j ] (8‑465)

1. The luma filter coefficients cL with elements cL[ i ][ j ], i = 0..AlfNumFilters–1, j = AlfCodedLengthLuma–2 are derived as follows:
   * 1. If alf\_pred\_method\_is equal to 0 or the value of i is equal to 0,

cL[ i ][ j ] = alf\_coeff\_luma[ i ][ j ] + 255 – ( Σk2\*alf\_coeff\_luma[ i ][ k ] ) >> 2 (8‑465)

where k=0..j-1

* + 1. Otherwise,

alf\_coeff\_luma\_nb[ i ] = alf\_coeff\_luma[ i ][ j ] – ( Σk2\*alf\_coeff\_luma[ i ][ k ] ) >> 2 (8‑465)

cL[ i ][ j ] = alf\_coeff\_luma\_nb[ i ] + cL[ i – 1 ][ j ]  (8‑465)

where k=0..j-1

1. The luma filter coefficients cL with elements cL[ i ][ j ], i = 0..AlfNumFilters–1, j = AlfCodedLengthLuma–1 are derived as follows:
   * 1. If alf\_pred\_method\_is equal to 0 or the value of i is equal to 0,

cL[ i ][ j ] = alf\_coeff\_luma[ i ][ j ] + 255 – Σk(2\*alf\_coeff\_luma[ i ][ k ]) – 2\*cL[ i ][ j – 1 ] (8‑465)

where k=0..j-2

* + 1. Otherwise,

cL[ i ][ j ] = alf\_coeff\_luma[ i ][ j ] + cL[ i – 1 ][ j ] – Σk(2\*alf\_coeff\_luma[ i ][ k ])

 – 2\*alf\_coeff\_luma\_nb[ i ] (8‑465)

where k=0..j-2

Considering the symmetry of the filter, the luma filter coefficients cL with elements cL[ i ][ j ], i = 0..AlfNumFilters – 1 is derived as follows:

cL[ i ][ AlfLengthLuma – 1 ] = cL[ i ][ AlfCodedLengthLuma – 1 ] (8‑466)

cL[ i ][ AlfLengthLuma – 2 – j ] = cL[ i ][ j ] for j = 0..AlfCodedLengthLuma – 2 (8‑467)

The chroma filter coefficients cC with elements cC[ i ], i = 0..AlfCodedLengthChroma – 1 is derived as follows:

* If i is equal to AlfCodedLengthChroma – 1, the coefficient cC[i] is derived as

cC[ i ] = 255 – sum – alf\_coeff\_chroma[ i ] (8‑468)

where

sum = alf\_coeff\_chroma[ AlfCodedLengthChroma – 2 ] + Σj( alf\_coeff\_chroma[ j ] << 1 ) (8‑469)

with j = 0..AlfCodedLengthChroma – 3

* Otherwise (i is less than AlfCodedLengthChroma – 1),

cC[ i ] = alf\_coeff\_chroma[ i ] (8‑470)

Considering the symmetry of the filter, the chroma filter coefficients cC with elements cC[ i ], i = 0..AlfLengthChroma – 1 is derived as follows:

cC[ i ][ AlfLengthChroma – 1 ] = cC[ i ][ AlfCodedLengthChroma – 1 ] (8‑471)

cC[ i ][ AlfLengthChroma – 2 – j ] = cC[ i ][ j ] for j = 0..AlfCodedLengthChroma – 2 (8‑472)

#### Derivation process for filter index array for luma samples

Inputs of this process are:

– a luma location ( xC, yC ) specifying the top-left luma sample of the current coding unit relative to the top left luma sample of the current picture,

– a variable log2CUSize specifying the size of the current coding unit.

Output of this process is the two-dimensional filter index array of (nS)x(nS), fIdx.

A variable nS is set equal to ( 1 << log2CUSize ).

The boundary padding process specified in subclause 8.6.3.1 is invoked with the luma location ( xC, yC ), the size of coding unit log2CUSize and the chroma component index cIdx set equal to 0, and the output is assigned to the luma sample array s’’. [Ed. (WJ): s’’ is now a picture-size array, but actually CU size + appropriate margin is enough]

The filter index array fIdx is specified in the follows:

* When alf\_region\_adaptation\_flag is equal to 1, the following ordered steps apply.

1. The variables xIdx and yIdx are derived as

regionTab[16] = { 0, 1, 4, 5, 15, 2, 3, 6, 14, 11, 10, 7, 13, 12, 9, 8 } (8‑474)  
 offset = 1 << (Log2MaxCUSize – 1) (8‑474)  
 xInterval = ( ( ( PicWidthInSamplesL + offset ) >> Log2MaxCUSize ) + 1 ) >> 2 (8‑474)  
 yInterval = ( ( ( PicHeightInSamplesL + offset ) >> Log2MaxCUSize ) + 1 ) >> 2 (8‑474)  
 xIdx = Min( 3, Floor( ( xC + x ) / ( xInterval << Log2MaxCUSize ) ) ) (8‑474)  
 yIdx = Min( 3, Floor( ( yC + y ) / ( yInterval << Log2MaxCUSize ) ) ) (8‑474)

1. The filter index fIdx[ x, y ] with x, y = 0..(nS)-1 is derived as

fIdx[ x ][ y ] = regionTab[ ( yIdx << 2 ) + xIdx ] (8‑474)

* Otherwise (alf\_region\_flag is equal to 0), the following ordered steps apply.

1. The variables varTempH[ x ][ y ], varTempV[ x ][ y ] and varTemp1[ x ][ y ] with x, y = 0, 2, 4,...(nS) is derived as

varTempH[ x ][ y ] = | ( s’’[ xC+x, yC+y ] << 1 ) – s’’[ xC+x-1, yC+y ] – s’’[ xC+x+1, yC+y ] | (8‑474)  
 varTempV[ x ][ y ] = | ( s’’[ xC+x, yC+y ] << 1 ) – s’’[ xC+x, yC+y-1 ] – s’’[ xC+x, yC+y+1 ] | (8‑474)

1. The variables varTempH1[ x, y ], varTempV1[ x, y ], and varTemp3[ x, y ] with x, y = 0..( (nS) – 1 )>>2 are derived as

varTempH1[ x ][ y ] = ΣiΣj varTempH[ (x << 2 ) + i ][ (y << 2) + j ] with i, j = 1, 2 (8‑474)  
 varTempV1[ x ][ y ] = ΣiΣj varTempV[ (x << 2 ) + i ][ (y << 2) + j ] with i, j = 1, 2 (8‑474)  
 varTemp3[ x ][ y ] = ( varTempH1[ x ][ y ] + varTempV1[ x ][ y ] ) >> 2 (8‑474)

1. The variable direction is derived as

* If varTempV1[ x >> 2 ][ y >> 2 ] is greater than varTempH1[ x >> 2 ][ y >> 2 ] << 1,

direction = 1

* Otherwise, if varTempH1[ x >> 2 ][ y >> 2 ] is greater than varTempV1[ x >> 2 ][ y >> 2 ] << 1,

direction = 2

* Otherwise,

direction = 0

1. The variable avgvar is derived as

varTab[16] = { 0, 1, 2, 2, 2, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4 } (8‑475)  
 avgVar = Clip3( 0, 15, (varTemp3[ x >> 2 ][ y >> 2 ] \* 1024 ) >> (3 + BitDepthY) ) (8‑475)

1. The filter index fIdx[ x, y ] with x, y = 0..(nS)-1 is derived as

fIdx[ x ][ y ] = Clip3( 0, 4, varTab[avgVar] ) + 5 \* direction (8‑504)

#### Filtering process for luma samples

Inputs of this process are:

– a luma location ( xC, yC ) specifying the top-left luma sample of the current coding unit relative to the top left luma sample of the current picture,

– a variable log2CUSize specifying the size of the current coding unit,

– a filter index array of (nS)x(nS), fIdx.

Output of this process is the filtered reconstruction of luma picture.

The boundary padding process specified in subclause 8.6.3.1 is invoked with the luma location ( xC, yC ), the size of coding unit log2CUSize and the chroma component index cIdx set equal to 0, and the output is assigned to the luma sample array s’’. [Ed. (WJ): s’’ is now a picture-size array, but actually CU size + appropriate margin is enough]

A variable nS is set equal to ( 1 << log2CUSize ) and a variable alfFilterShape is set equal to alf\_filter\_shape\_flag.

A variable lcuHeight is set equal to ( 1 << Log2MaxCUSize ) and a variable vbLine is set equal to lcuHeight – 4.

Each sample of luma picture recFiltPictureL[ xC + x ][ yC + y ] with x, y = 0..(nS)-1, is derived as following ordered steps:

1. A variable dist2VB is derived as follows.

dist2VB = ( ( yC + y ) % lcuHeight – vbLine ) (8‑476)

1. A variable dist2VB is modified as follows.

* If dist2VB is less than –vbLine+2 and yC is larger than 2, dist2VB is set equal to dist2VB+lcuHeight,
* Otherwise, if yC+lcuHeight is equal or larger than pic\_height\_in\_luma\_samples, dist2VB is set equal to 5.

1. If alfFilterShape is equal to 0 and dist2VB is equal to 0 or -1, the following applies.

 (8‑476)

1. Otherwise, if alfFilterShape is equal to 0 and dist2VB is equal to 1 or -2, the following applies.

 (8‑476)

where N is set equal to AlfNumCoeffLuma-1 and horPos[i] and verPos[i] are specified in Table 8‑14 and Table 8‑15, respectively.

1. Otherwise, the following applies.

 (8‑476)

Table 8‑18 – Specification of horPos[ i ] according to alfFilterShape for adaptive loop filter process

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **i** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| alfFilterShape = 0 | -2 | 0 | 2 | -1 | 0 | 1 | -2 | -1 | 0 | 1 | 2 | -1 | 0 | 1 | -2 | 0 | 2 |
| alfFilterShape = 1 | 0 | 0 | 0 | 0 | -4 | -3 | -2 | -1 | 0 | 1 | 2 | 3 | 4 | 0 | 0 | 0 | 0 |

Table 8‑19 – Specification of verPos[ i ] according to alfFilterShape for adaptive loop filter process

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **i** | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| dist2VB < -2 or dist2VB > 1 | alfShape == 0 | -2 | -2 | -2 | -1 | -1 | -1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 |
| dist2VB == -2 | alfShape == 0 | -2 | -2 | -2 | -1 | -1 | -1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| dist2VB == 1 | alfShape == 0 | -1 | -1 | -1 | -1 | -1 | -1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 |
| dist2VB < -4 or dist2VB > 3 | alfShape == 1 | -4 | -3 | -2 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 4 |
| dist2VB == -4 or dist2VB == 3 | alfShape == 1 | 0 | -3 | -2 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 0 |
| dist2VB == -3 or dist2VB == 2 | alfShape == 1 | 0 | 0 | -2 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 0 | 0 |
| dist2VB == -2 or dist2VB == 1 | alfShape == 1 | 0 | 0 | 0 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| dist2VB == -1 or dist2VB == 0 | alfShape == 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |



(a) alfFilterShape == 0 (b) alfFilterShape == 1

Figure 8‑6 Mapping between geometric position and luma adaptive loop filter index according to alfFilterShape (informative)

#### Filtering process for chroma samples

Inputs of this process are:

– a chroma location ( xC, yC ) specifying the top-left luma sample of the current coding unit relative to the top left chroma sample of the current picture,

– a variable log2CUSize specifying the size of the current coding unit.

– a variable cIdx specifying the chroma component index.

Output of this process is the filtered reconstruction of chroma picture.

The boundary padding process specified in subclause 8.6.3.1 is invoked with the chroma location ( xC, yC ), the size of coding unit log2CUSize and the chroma component index cIdx, and the output is assigned to the luma sample array s’’. [Ed. (WJ): s’’ is now a picture-size array, but actually CU size + appropriate margin is enough]

A variable nS is set equal to ( 1 << log2CUSize ) and a variable alfFilterShape is set equal to alf\_filter\_shape\_chroma\_flag.

A variable lcuHeight is set equal to ( 1 << ( Log2MaxCUSize – 1 ) ) and a variable vbLine is set equal to lcuHeight – 2.

Filtered samples of chroma picture recFiltPicture[ xC + x ][ yC + y ] with x, y = 0..(nS)-1, are derived as the following ordered steps:

1. A variable dist2VB is derived as follows.

dist2VB = ( ( yC + y ) % lcuHeight – vbLine ) (8‑476)

1. A variable dist2VB is modified as follows.

* If dist2VB is less than –vbLine+2 and yC is larger than 2, dist2VB is set equal to dist2VB+lcuHeight,
* Otherwise, if yC+lcuHeight is equal or larger than pic\_height\_in\_luma\_samples >> 1, dist2VB is set equal to 5.

1. If alfFilterShape is equal to 0 and dist2VB is equal to 0 or -1, the following applies.

 (8‑476)

1. Otherwise, if alfFilterShape is equal to 0 and dist2VB is equal to 1 or -2, the following applies.

 (8‑476)

where N is set equal to AlfNumCoeffChroma-1 and horPos[i] and verPos[i] are specified in Table 8‑14 and Table 8‑15, respectively.

1. Otherwise, the following applies.

 (8‑476)

# Parsing process

Inputs to this process are bits from the RBSP.

Outputs of this process are syntax element values.

This process is invoked when the descriptor of a syntax element in the syntax tables in subclause 7.3 is equal to ue(v), me(v), se(v), te(v) (see subclause 9.1), or ae(v) (see subclause 9.2).

## Parsing process for Exp-Golomb codes

This process is invoked when the descriptor of a syntax element in the syntax tables in subclause 7.3 is equal to ue(v), me(v), se(v), ge(v) or te(v).

Inputs to this process are bits from the RBSP.

Outputs of this process are syntax element values.

Syntax elements coded as ue(v), me(v), or se(v) are Exp-Golomb-coded. Syntax elements coded as te(v) are truncated Exp-Golomb-coded. The parsing process for these syntax elements begins with reading the bits starting at the current location in the bitstream up to and including the first non-zero bit, and counting the number of leading bits that are equal to 0. This process is specified as follows:

leadingZeroBits = −1  
for( b = 0; !b; leadingZeroBits++ ) (9‑1)  
 b = read\_bits( 1 )

The variable codeNum is then assigned as follows:

codeNum = 2leadingZeroBits − 1 + read\_bits( leadingZeroBits ) (9‑2)

where the value returned from read\_bits( leadingZeroBits ) is interpreted as a binary representation of an unsigned integer with most significant bit written first.

Table 9‑1 illustrates the structure of the Exp-Golomb code by separating the bit string into "prefix" and "suffix" bits. The "prefix" bits are those bits that are parsed in the above pseudo-code for the computation of leadingZeroBits, and are shown as either 0 or 1 in the bit string column of Table 9‑1. The "suffix" bits are those bits that are parsed in the computation of codeNum and are shown as xi in Table 9‑1, with i being in the range 0 to leadingZeroBits − 1, inclusive. Each xi can take on values 0 or 1.

Table 9‑1 – Bit strings with "prefix" and "suffix" bits and assignment to codeNum ranges (informative)

|  |  |
| --- | --- |
| **Bit string form** | **Range of codeNum** |
| 1 | 0 |
| 0 1 x0 | 1..2 |
| 0 0 1 x1 x0 | 3..6 |
| 0 0 0 1 x2 x1 x0 | 7..14 |
| 0 0 0 0 1 x3 x2 x1 x0 | 15..30 |
| 0 0 0 0 0 1 x4 x3 x2 x1 x0 | 31..62 |
| … | … |

Table 9‑2 illustrates explicitly the assignment of bit strings to codeNum values.

Table 9‑2 – Exp-Golomb bit strings and codeNum in explicit form and used as ue(v) (informative)

|  |  |
| --- | --- |
| **Bit string** | **codeNum** |
| 1 | 0 |
| 0 1 0 | 1 |
| 0 1 1 | 2 |
| 0 0 1 0 0 | 3 |
| 0 0 1 0 1 | 4 |
| 0 0 1 1 0 | 5 |
| 0 0 1 1 1 | 6 |
| 0 0 0 1 0 0 0 | 7 |
| 0 0 0 1 0 0 1 | 8 |
| 0 0 0 1 0 1 0 | 9 |
| … | … |

Depending on the descriptor, the value of a syntax element is derived as follows.

– If the syntax element is coded as ue(v), the value of the syntax element is equal to codeNum.

– Otherwise, if the syntax element is coded as se(v), the value of the syntax element is derived by invoking the mapping process for signed Exp-Golomb codes as specified in subclause 9.1.1 with codeNum as the input.

– Otherwise, if the syntax element is coded as me(v) [Ed. (TW): insert text]

– Otherwise, if the syntax element is coded as ge(v), the value of the syntax element is derived by invoking the mapping process for k-th order Exp-Golomb code with codeNum and k as the input. [Ed. (WJ): see subclause 9.2.2.4 for k-th order Exp-Golomb code]

[Ed. (WJ): only syntax element alf\_coeff\_luma uses ge(v), and the value of k is specified according to filter coefficient index i and the variable AlfFilterShape as follows. This table should be moved elsewhere. Do we really need ge(v)?]

Table 9‑3 – Specification of k for syntax item alf\_coeff\_luma

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Filter coefficient index** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| **AlfFilterShape == 0** | 2 | 3 | 2 | 4 | 5 | 4 | 4 | 5 | 6 | 8 |
| **AlfFilterShape == 1** | 3 | 5 | 2 | 3 | 3 | 4 | 5 | 6 | 8 | - |

– Otherwise (the syntax element is coded as te(v)), the range of possible values for the syntax element is determined first. The range of this syntax element may be between 0 and x, with x being greater than or equal to 1 and the range is used in the derivation of the value of the syntax element value as follows.

– If x is greater than 1, codeNum and the value of the syntax element is derived in the same way as for syntax elements coded as ue(v).

– Otherwise (x is equal to 1), the parsing process for codeNum which is equal to the value of the syntax element is given by a process equivalent to:

b = read\_bits( 1 ) (9‑3)  
codeNum = !b

### Mapping process for signed Exp-Golomb codes

Input to this process is codeNum as specified in subclause 9.1.

Output of this process is a value of a syntax element coded as se(v).

The syntax element is assigned to the codeNum by ordering the syntax element by its absolute value in increasing order and representing the positive value for a given absolute value with the lower codeNum. Table 9‑3 provides the assignment rule.

Table 9‑4 – Assignment of syntax element to codeNum for signed Exp-Golomb coded syntax elements se(v)

|  |  |
| --- | --- |
| **codeNum** | **syntax element value** |
| 0 | 0 |
| 1 | 1 |
| 2 | −1 |
| 3 | 2 |
| 4 | −2 |
| 5 | 3 |
| 6 | −3 |
| k | (−1)k+1 Ceil( k÷2 ) |

## CABAC parsing process for slice data

This process is invoked when parsing syntax elements with descriptor ae(v) in subclauses 7.3.4 and 7.3.5.

Inputs to this process are a request for a value of a syntax element and values of prior parsed syntax elements.

Output of this process is the value of the syntax element.

When starting the parsing of the slice data of a slice in subclause 7.3.4, the initialisation process of the CABAC parsing process is invoked as specified in subclause 9.2.1 and a mapping table BitStreamTable with num\_substreams\_minus1 + 1 entries specifying a bitstream pointer table to use for later current bitstream pointer derivation is derived as follows.

– BitStreamTable[0] is initialized to contain the bitstream pointer.

– For all indices i greater than 0 and less than num\_substreams\_minus1+1, BitStreamTable[ i ] contains a bitstream pointer to substream\_length[ i ] bits after BitStreamTable[ i − 1 ].

The current bitstream pointer is set to BitStreamTable[ 0 ].

Current

LCU

entropy\_coding\_synchro+1 LCUs

Left tile

boundary

T

Figure ‑ – Spatial neighbour T that is used to invoke the treeblock availability derivation process relative to the current largest coding unit (informative)

The address of the LCU containing the spatial neighbor block T (Figure 9‑1), tbAddrT, is derived using the location ( x0, y0 ) of the top-left luma sample of the current LCU as follows.

tbAddrT = cuAddress( x0 + entropy\_coding\_synchro \* ( 1 << Log2MaxCUSize ) − 1, y0 − 1 ) (9‑4)

The variable availableFlagT is obtained by invoking the treeblock availability derivation process as specified in subclause 6.4.2 with tbAddrT as input.

When starting the parsing of a coding tree as specified in subclause 7.3.5 and entropy\_coding\_synchro is not equal to 0, the following applies.

– If ( tile\_boundary\_independence\_flag is equal to 1 and ( tbAddr − firstTbInTileAddr ) % tileWidthInLCUs is equal to 0 ) or ( tile\_boundary\_independence\_flag is equal to 0 and tbAddr % picWidthInLCUs is equal to 0, the following applies.

– When availableFlagT is equal to 1, the synchronization process of the CABAC parsing process is invoked as specified in subclause 9.2.1.3.

– When cabac\_istate\_reset\_flag is equal to 1, the decoding process for binary decisions before termination specified in subclause 9.2.3.2.4 is invoked, followed by the initialisation process for the arithmetic decoding engine specified in subclause 9.2.1.4.

– If tile\_boundary\_independence\_flag is equal to 1, the current bitstream pointer is set to indicate BitStreamTable[i] with the index i derived as follows.

num\_substreams\_per\_tile = ( num\_substreams\_minus1 +1 ) /   
 ( ( num\_tile\_rows\_minus1 + 1 ) \* ( num\_tile\_columns\_minus1 + 1 ) ) (9‑4)   
i = ( ( tbAddr − firstTbInTileAddr ) / tileWidthInLCUs ) % ( num\_substreams\_per\_tile ) +   
 num\_substreams\_per\_tile \* tileIdx

NOTE: the first part of the above formula (before % sign) is used to determine the number of LCU lines in the current tile since the current slice has started. This number, modulo the number of substreams per tile, indicates which substream pointer to use for that tile. To this is added the number of substreams present in tiles already processed.

– Otherwise (tile\_boundary\_independence\_flag is equal to 0), the current bitstream pointer is set to indicate BitStreamTable[i] with the index i derived as follows.

i = ( tbAddr / picWidthInLCUs ) % ( num\_substreams\_minus1 + 1 ) (9‑4)

– Otherwise, if (tile\_boundary\_independence\_flag is equal to 1 and ( tbAddr − firstTbInTileAddr ) % tileWidthInLCUs is equal to entropy\_coding\_synchro + 1) or (tile\_boundary\_independence\_flag is equal to 0 and tbAddr % picWidthInLCUs is equal to entropy\_coding\_synchro + 1), the memorization process of the CABAC parsing process is invoked as specified in subclause 9.2.1.2.

When starting the parsing of a coding tree as specified in subclause 7.3.5 and tile\_boundary\_independence\_idc is equal to 0, the following applies.

– The variables x0Tile and y0Tile specifiy the location (x0Tile,y0Tile) of the top-left luma sample of the treeblock with address firstTbInTileAddr and the variables x0 and y0 specifiy the location (x0,y0) of the top-left luma sample of the treeblock with address tbAddr.

– If tbAddr is equal to firstTbInTileAddr and tbAddr is not equal to zero the decoding process for binary decisions before termination specified in subclause ‎‎9.2.3.2.4 is invoked and followed by byte alignment. Then the synchronization process of the CABAC parsing process is invoked as specified in subclause ‎‎9.2.1.3.

– Otherwise, if x0 is equal to x0Tile and y0 is equal to y0Tile+(1<< log2MaxCUSize) the memorization process of the CABAC parsing process is invoked as specified in subclause ‎‎9.2.1.2.

The parsing of syntax elements proceeds as follows:

For each requested value of a syntax element a binarization is derived as described in subclause 9.2.2.

The binarization for the syntax element and the sequence of parsed bins determines the decoding process flow as described in subclause 9.2.2.9.

For each bin of the binarization of the syntax element, which is indexed by the variable binIdx, a context index ctxIdx is derived as specified in subclause 9.2.3.1.

For each ctxIdx the arithmetic decoding process is invoked as specified in subclause 9.2.3.2.

The resulting sequence ( b0..bbinIdx ) of parsed bins is compared to the set of bin strings given by the binarization process after decoding of each bin. When the sequence matches a bin string in the given set, the corresponding value is assigned to the syntax element.

In case the request for a value of a syntax element is processed for the syntax element mb\_type and the decoded value of mb\_type is equal to I\_PCM, the decoding engine is initialised after the decoding of any pcm\_alignment\_zero\_bit and all pcm\_sample\_luma and pcm\_sample\_chroma data as specified in subclause 9.2.1.2.

### Initialisation process

Outputs of this process are initialised CABAC internal variables.

The processes in subclauses 9.2.1.1.1 and 9.2.1.2 are invoked when starting the parsing of the slice data of a slice in subclause 7.3.4 or when starting the parsing of the data of a coding tree in subclause 7.3.5 and the coding tree is the first coding tree in a tile and tile\_boundary\_independence\_flag is not equal to zero.

#### Initialisation process for context variables

Outputs of this process are the initialised CABAC context variables indexed by ctxIdxTable and ctxIdx.

Table 9‑21 toTable 9‑45 contain the values of the 8 bit variable initValue used in the initialisation of context variables that are assigned to all syntax elements in subclauses 7.3.4 to 7.3.10 except for the end-of-slice flag.

For each context variable, the two variables pStateIdx and valMPS are initialised.

NOTE 1 – The variable pStateIdx corresponds to a probability state index and the variable valMPS corresponds to the value of the most probable symbol as further described in subclause 9.2.3.2.

From the 8 bit table entry initValue, the two 4 bit variables slopeIdx and n are derived according to the following pseudo-code process:

slopeIdx = initValue >> 4  
n = initValue & 15

For every slopeIdx, a slope m is specifed in Table 9‑4.

**Table 9‑5 – Mapping from variable m to variable slopeIdx**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **slopeIdx** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **m** | -239 | -143 | -85 | -51 | -31 | -19 | -11 | 0 | 11 | 19 | 31 | 51 | 85 | 143 | 239 | 399 |

**Table 9‑6 – Mapping from variable segmentIdx to variables segOffset[segmentIdx] and accumulatedSegOffset[segmentIdx]**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **segmentIdx** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| **segOffset[segmentIdx]** | 6 | 7 | 5 | 7 | 10 | 14 | 16 | 1 |
| **accumulatedSegOffset[segmentIdx]** | −3 | 3 | 10 | 15 | 22 | 32 | 46 | 62 |

The two values assigned to pStateIdx and valMPS for the initialisation are derived from SliceQPY, which is derived in Equation 7‑9. Given the variable m as derived from slopeIdx according to Table 9‑4 and variable n, the initialisation is specified by the following pseudo-code process using the two mappings as specified in Table 9‑5.

qpMinus26 = Clip3( 0, 51, SliceQPY ) − 26  
val = Clip3( 0, 3839, ( n << 8 ) + m \* qpMinus26 + 128 )  
val −= 1920  
valMps = ( val >= 0 ) (9‑4)  
val = ( val ^ ( val >> 11 ) ) + 128  
segmentIdx = val >> 8  
pStateIdx = accumulatedSegOffset[segmentIdx] + ( (val & 255 ) \* segOffset[segmentIdx] >> 8 )

In Table 9‑20, the ctxIdxTable and the ctxIdx for which initialisation is needed for each of the slice types are listed. The referenced context index tables ctxIdxTable include the values of initValue needed for the initialisation.

Table 9‑7 – Association of ctxIdx and syntax elements for each slice type in the initialisation process

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Syntax element** | **ctxIdxTable** | **Slice Type** | | |
| **I** | **P** | **B** |
| slice\_header() | alf\_cu\_flag | Table 9‑21 | 0 | 1 | 2 |
| coding\_tree() | split\_coding\_unit\_flag | Table 9‑22 | 0..2 | 3..5 | 6..8 |
| coding\_unit() | skip\_flag | Table 9‑23 |  | 0..2 | 3..5 |
| cu\_qp\_delta | Table 9‑24 | 0..3 | 4..7 | 8..11 |
| pred\_mode\_flag | Table 9‑25 |  | 0 | 1 |
| part\_mode | Table 9‑25 | 2 | 3..5 | 6..8 |
| prediction\_unit() | prev\_intra\_luma\_pred\_flag | Table 9‑26 | 0 | 1 | 2 |
| merge\_flag | Table 9‑29 |  | 0 | 1 |
| merge\_idx | Table 9‑30 |  | 0..3 | 4..7 |
| inter\_pred\_flag | Table 9‑31 |  |  | 0..3 |
| ref\_idx\_lc, ref\_idx\_l0, ref\_idx\_l1 | Table 9‑32 |  | 0..2 | 3..5 |
| abs\_mvd\_greater0\_flag | Table 9‑33 |  | 0 | 1 |
| abs\_mvd\_greater1\_flag | Table 9‑33 |  | 2 | 3 |
| mvp\_lc\_flag, mvp\_l0\_flag, mvp\_l1\_flag | Table 9‑34 |  | 0 | 1 |
| transform\_tree() | no\_residual\_data\_flag | Table 9‑35 |  | 0 | 1 |
| split\_transform\_flag | Table 9‑36 | 0..3 | 4..7 | 8..11 |
| cbf\_luma | Table 9‑37 | 0..1 | 2..3 | 4..5 |
| cbf\_cb, cbf\_cr | Table 9‑38 | 0..2 | 3..5 | 6..8 |
| residual\_coding() | last\_significant\_coeff\_x\_prefix | Table 9‑40 | 0..29 | 30..59 | 60..89 |
| last\_significant\_coeff\_y\_prefix | Table 9‑26 | 0..29 | 30..59 | 60..89 |
| significant\_coeff\_group\_flag | Table 9-26 | 0..3 | 4..7 | 8..11 |
| significant\_coeff\_flag | Table 9‑41 | 0..47 | 48..95 | 96..143 |
| coeff\_abs\_level\_greater1\_flag | Table 9‑44 | 0..31 | 32..63 | 64..95 |
| coeff\_abs\_level\_greater2\_flag | Table 9‑45 | 0..23 | 24..47 | 48..71 |

NOTE 2 – ctxIdxTable equal to 0 and ctxIdx equal to 0 are associated with the end\_of\_slice\_flag. The decoding process specified in subclause 9.2.3.2.4 applies to ctxIdxTable equal to 0 and ctxIdx equal to 0. This decoding process, however, may also be implemented by using the decoding process specified in subclause 9.2.3.2.1. In this case, the initial values associated with ctxIdxTable equal to 0 and ctxIdx equal to 0 are specified to be pStateIdx = 63 and valMPS = 0, where pStateIdx = 63 represents a non‑adapting probability state.

Table 9‑8 – Values of variable initValue for alf\_cu\_flag ctxIdx

|  |  |  |  |
| --- | --- | --- | --- |
| **Initialisation variable** | **alf\_cu\_flag ctxIdx** | | |
| **0** | **1** | **2** |
| **initValue** | 153 | 87 | 135 |

Table 9‑9 – Values of variable initValue for split\_coding\_unit\_flag ctxIdx

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Initialisation variable** | **split\_coding\_unit\_flag ctxIdx** | | | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |
| **initValue** | 87 | 74 | 107 | 84 | 103 | 105 | 84 | 103 | 105 |

Table 9‑10 – Values of variable initValue for skip\_flag ctxIdx

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Initialisation variable** | **skip\_flag ctxIdx** | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** |
| **initValue** | 165 | 168 | 154 | 165 | 168 | 154 |

Table 9‑11 – Values of variable initValue for cu\_qp\_delta ctxIdx

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Initialisation variable** | **cu\_qp\_delta ctxIdx** | | | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |
| **initValue** | 119 | 119 | 119 | 119 | 119 | 119 | 119 | 119 | 119 |

Table 9‑12 – Values of variable initValue for pred\_mode and part\_mode

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Initialisation variable** | **pred\_type ctxIdx** | | | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |
| **initValue** | 114 | 98 | 167 | 119 | 87 | 119 | 119 | 87 | 119 |

Table 9‑13 – Values of variable initValue for prev\_intra\_luma\_pred\_flag ctxIdx

|  |  |  |  |
| --- | --- | --- | --- |
| **Initialisation variable** | **prev\_intra\_luma\_pred\_flag ctxIdx** | | |
| **0** | **1** | **2** |
| **initValue** | 167 | 119 | 150 |

Table 9‑14 – Values of variable initValue for intra\_chroma\_pred\_mode ctxIdx

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Initialisation variable** | **intra\_chroma\_pred\_mode ctxIdx** | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** |
| **initValue** | 53 | 103 | 85 | 87 | 101 | 87 |

Table 9‑15 – Value of variable initValue for merge\_flag ctxIdx

|  |  |  |
| --- | --- | --- |
| **Initialisation variable** | **merge\_flag ctxIdx** | |
| **0** | **1** |
| **initValue** | 72 | 119 |

Table 9‑16 – Values of variable initValue for merge\_idx ctxIdx

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Initialisation variable** | **merge\_idx ctxIdx** | | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| **initValue** | 100 | 86 | 102 | 133 | 116 | 87 | 119 | 103 |

Table 9‑17 – Values of variable initValue for inter\_pred\_flag ctxIdx

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Initialisation variable** | **inter\_pred\_flag ctxIdx** | | | |
| **0** | **1** | **2** | **3** |
| **initValue** | 41 | 39 | 38 | 36 |

Table 9‑18 – Values of variable initValue for ref\_idx\_lc, ref\_idx\_l0, ref\_idx\_l1 ctxIdx

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Initialisation variable** | **ref\_idx\_lc, ref\_idx\_l0, ref\_idx\_l1 ctxIdx** | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** |
| **initValue** | 102 | 118 | 103 | 118 | 118 | 134 |

Table 9‑19 – Values of variable initValue for abs\_mvd\_greater0\_flag and abs\_mvd\_greater1\_flag ctxIdx

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Initialisation variable** | **abs\_mvd\_greater0\_flag, abs\_mvd\_greater1\_flag ctxIdx** | | | | |
| **0** | **1** | **2** | **3** |
| **initValue** | 120 | 166 | 135 | 166 |

Table 9‑20 – Values of variable initValue for mvp\_lc\_flag, mvp\_l0\_flag, mvp\_l1\_flag ctxIdx

|  |  |  |
| --- | --- | --- |
| **Initialisation variable** | **mvp\_lc\_flag, mvp\_l0\_flag, mvp\_l1\_flag ctxIdx** | |
| **0** | **1** |
| **initValue** | 134 | 134 |

Table 9‑21 – Values of variable initValue for no\_residual\_data\_flag ctxIdx

|  |  |  |
| --- | --- | --- |
| **Initialisation variable** | **no\_residual\_data\_flag ctxIdx** | |
| **0** | **1** |
| **initValue** | 39 | 39 |

Table 9‑22 – Values of variable initValue for split\_transform\_flag ctxIdx

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Initialisation variable** | **split\_transform\_flag ctxIdx** | | | | | | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** |
| **initValue** | 119 | 162 | 148 | 100 | 119 | 71 | 86 | 55 | 119 | 102 | 86 | 86 |

Table 9‑23 – Values of variable initValue for cbf\_luma ctxIdx

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Initialisation variable** | **cbf\_luma ctxIdx** | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** |
| **initValue** | 73 | 74 | 102 | 89 | 102 | 89 |

Table 9‑24 – Values of variable initValue for cbf\_cb and cbf\_cr ctxIdx

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Initialisation variable** | **cbf\_cb ctxIdx** | | | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |
| **initValue** | 55 | 86 | 133 | 114 | 84 | 117 | 114 | 68 | 117 |

Table 9‑25 – Values of variable initValue for last\_significant\_coeff\_x\_prefix ctxIdx

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Initialisation variable** | **last\_significant\_coefficient\_x\_prefix ctxIdx** | | | | | | | | | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** |
| **initValue** | 72 | 72 | 71 | 72 | 104 | 89 | 71 | 88 | 89 | 59 | 73 | 86 | 89 | 106 | 60 |
|  | **15** | **16** | **17** | **18** | **19** | **20** | **21** | **22** | **23** | **24** | **25** | **26** | **27** | **28** | **29** |
| **initValue** | 59 | 43 | 55 | 54 | 70 | 53 | 53 | 87 | 71 | 69 | 54 | 88 | 73 | 72 | 53 |
|  | **30** | **31** | **32** | **33** | **34** | **35** | **36** | **37** | **38** | **39** | **40** | **41** | **42** | **43** | **44** |
| **initValue** | 57 | 72 | 71 | 72 | 57 | 72 | 102 | 88 | 73 | 73 | 72 | 102 | 103 | 73 | 89 |
|  | **45** | **46** | **47** | **48** | **49** | **50** | **51** | **52** | **53** | **54** | **55** | **56** | **57** | **58** | **59** |
| **initValue** | 73 | 57 | 87 | 54 | 70 | 54 | 101 | 71 | 55 | 70 | 116 | 103 | 72 | 72 | 119 |
|  | **60** | **61** | **62** | **63** | **64** | **65** | **66** | **67** | **68** | **69** | **70** | **71** | **72** | **73** | **74** |
| **initValue** | 88 | 72 | 71 | 72 | 57 | 72 | 102 | 88 | 73 | 73 | 72 | 118 | 103 | 73 | 89 |
|  | **75** | **76** | **77** | **78** | **79** | **80** | **81** | **82** | **83** | **84** | **85** | **86** | **87** | **88** | **89** |
| **initValue** | 73 | 57 | 87 | 54 | 70 | 69 | 85 | 71 | 55 | 70 | 85 | 103 | 72 | 72 | 119 |

Table 9‑26 – Values of variable initValue for last\_significant\_coeff\_y\_prefix ctxIdx

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Initialisation variable** | **last\_significant\_coeff\_y\_prefix ctxIdx** | | | | | | | | | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** |
| **initValue** | 72 | 72 | 71 | 72 | 104 | 89 | 71 | 88 | 89 | 59 | 73 | 86 | 89 | 106 | 60 |
|  | **15** | **16** | **17** | **18** | **19** | **20** | **21** | **22** | **23** | **24** | **25** | **26** | **27** | **28** | **29** |
| **initValue** | 59 | 43 | 55 | 54 | 70 | 53 | 53 | 87 | 71 | 69 | 54 | 88 | 73 | 72 | 53 |
|  | **30** | **31** | **32** | **33** | **34** | **35** | **36** | **37** | **38** | **39** | **40** | **41** | **42** | **43** | **44** |
| **initValue** | 57 | 72 | 71 | 72 | 57 | 72 | 102 | 88 | 73 | 73 | 72 | 102 | 103 | 73 | 89 |
|  | **45** | **46** | **47** | **48** | **49** | **50** | **51** | **52** | **53** | **54** | **55** | **56** | **57** | **58** | **59** |
| **initValue** | 73 | 57 | 87 | 54 | 70 | 54 | 101 | 71 | 55 | 70 | 116 | 103 | 72 | 72 | 119 |
|  | **60** | **61** | **62** | **63** | **64** | **65** | **66** | **67** | **68** | **69** | **70** | **71** | **72** | **73** | **74** |
| **initValue** | 88 | 72 | 71 | 72 | 57 | 72 | 102 | 88 | 73 | 73 | 72 | 118 | 103 | 73 | 89 |
|  | **75** | **76** | **77** | **78** | **79** | **80** | **81** | **82** | **83** | **84** | **85** | **86** | **87** | **88** | **89** |
| **initValue** | 73 | 57 | 87 | 54 | 70 | 69 | 85 | 71 | 55 | 70 | 85 | 103 | 72 | 72 | 119 |

Table ‑ – Values of variable initValue for significant\_coeff\_group\_flag ctxIdx

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Initialisation variable** | **significant\_coeff\_group\_flag ctxIdx** | | | | | | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** |
| **initValue** | 83 | 122 | 98 | 121 | 99 | 120 | 67 | 119 | 99 | 120 | 67 | 119 |

Table 9‑28 – Values of variable initValue for significant\_coeff\_flag ctxIdx

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Initialisation variable** | **significant\_coeff\_flag ctxIdx** | | | | | | | | | | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **initValue** | 74 | 73 | 88 | 72 | 72 | 55 | 71 | 54 | 71 | 88 | 103 | 71 | 53 | 87 | 134 | 86 |
|  | **16** | **17** | **18** | **19** | **20** | **21** | **22** | **23** | **24** | **25** | **26** | **27** | **28** | **29** | **30** | **31** |
| **initValue** | 84 | 70 | 68 | 89 | 90 | 84 | 88 | 74 | 130 | 118 | 88 | 120 | 87 | 149 | 70 | 52 |
|  | **32** | **33** | **34** | **35** | **36** | **37** | **38** | **39** | **40** | **41** | **42** | **43** | **44** | **45** | **46** | **47** |
| **initValue** | 118 | 133 | 116 | 114 | 129 | 132 | 162 | 115 | 51 | 115 | 66 | 120 | 74 | 115 | 87 | 89 |
|  | **48** | **49** | **50** | **51** | **52** | **53** | **54** | **55** | **56** | **57** | **58** | **59** | **60** | **61** | **62** | **63** |
| **initValue** | 152 | 119 | 103 | 118 | 87 | 70 | 70 | 53 | 118 | 134 | 118 | 101 | 68 | 85 | 101 | 116 |
|  | **64** | **65** | **66** | **67** | **68** | **69** | **70** | **71** | **72** | **73** | **74** | **75** | **76** | **77** | **78** | **79** |
| **initValue** | 100 | 68 | 67 | 136 | 168 | 147 | 150 | 120 | 115 | 118 | 119 | 136 | 102 | 70 | 53 | 67 |
|  | **80** | **81** | **82** | **83** | **84** | **85** | **86** | **87** | **88** | **89** | **90** | **91** | **92** | **93** | **94** | **95** |
| **initValue** | 117 | 102 | 117 | 1158 | 114 | 84 | 115 | 99 | 100 | 83 | 114 | 152 | 168 | 131 | 150 | 120 |
|  | **96** | **97** | **98** | **99** | **100** | **101** | **102** | **103** | **104** | **105** | **106** | **107** | **108** | **109** | **110** | **111** |
| **initValue** | 152 | 119 | 103 | 118 | 87 | 70 | 70 | 53 | 71 | 103 | 118 | 101 | 68 | 85 | 101 | 116 |
|  | **112** | **113** | **114** | **115** | **116** | **117** | **118** | **119** | **120** | **121** | **122** | **123** | **124** | **125** | **126** | **127** |
| **initValue** | 116 | 68 | 67 | 152 | 168 | 147 | 150 | 120 | 115 | 118 | 119 | 136 | 102 | 86 | 84 | 67 |
|  | **128** | **129** | **130** | **131** | **132** | **133** | **134** | **135** | **136** | **137** | **138** | **139** | **140** | **141** | **142** | **143** |
| **initValue** | 117 | 102 | 117 | 115 | 99 | 100 | 115 | 99 | 100 | 83 | 114 | 152 | 152 | 131 | 150 | 120 |

Table 9‑29 – Values of variable initValue for coeff\_abs\_level\_greater1\_flag ctxIdx

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Initialisation variable** | **coeff\_abs\_level\_greater1\_flag ctxIdx** | | | | | | | | | | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **initValue** | 104 | 68 | 116 | 86 | 104 | 132 | 86 | 87 | 105 | 134 | 87 | 103 | 102 | 66 | 114 | 68 |
|  | **16** | **17** | **18** | **19** | **20** | **21** | **22** | **23** | **24** | **25** | **26** | **27** | **28** | **29** | **30** | **31** |
| **initValue** | 87 | 84 | 100 | 101 | 72 | 69 | 101 | 86 | 104130 | 130 | 147 | 149 | 104 | 196 | 100 | 165 |
|  | **32** | **33** | **34** | **35** | **36** | **37** | **38** | **39** | **40** | **41** | **42** | **43** | **44** | **45** | **46** | **47** |
| **initValue** | 119 | 179 | 179 | 164 | 119 | 85 | 117 | 149 | 136 | 103 | 103 | 103 | 133 | 98 | 114 | 115 |
|  | **48** | **49** | **50** | **51** | **52** | **53** | **54** | **55** | **56** | **57** | **58** | **59** | **60** | **61** | **62** | **63** |
| **initValue** | 118 | 99 | 115 | 116 | 87 | 100 | 85 | 117 | 135 | 146 | 147 | 164 | 119 | 148 | 116 | 133 |
|  | **64** | **65** | **66** | **67** | **68** | **69** | **70** | **71** | **72** | **73** | **74** | **75** | **76** | **77** | **78** | **79** |
| **initValue** | 119 | 179 | 148 | 164 | 119 | 85 | 117 | 149 | 136 | 87 | 103 | 103 | 133 | 98 | 114 | 115 |
|  | **80** | **81** | **82** | **83** | **84** | **85** | **86** | **87** | **88** | **89** | **90** | **91** | **92** | **93** | **94** | **95** |
| **initValue** | 118 | 99 | 115 | 100 | 87 | 84 | 85 | 85 | 135 | 177 | 147 | 164 | 119 | 132 | 148 | 149 |

Table 9‑30 – Values of variable initValue for coeff\_abs\_level\_greater2\_flag ctxIdx

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Initialisation variable** | **coeff\_abs\_level\_greater2\_flag ctxIdx** | | | | | | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** |
| **initValue** | 86 | 103 | 73 | 102 | 103 | 73 | 103 | 88 | 89 | 115 | 117 | 103 |
|  | **12** | **13** | **14** | **15** | **16** | **17** | **18** | **19** | **20** | **21** | **22** | **23** |
| **initValue** | 117 | 118 | 103 | 102 | 103 | 72 | 101 | 103 | 104 | 101 | 167 | 121 |
|  | **24** | **25** | **26** | **27** | **28** | **29** | **30** | **31** | **32** | **33** | **34** | **35** |
| **initValue** | 84 | 102 | 88 | 117 | 118 | 104 | 103 | 119 | 136 | 83 | 116 | 118 |
|  | **36** | **37** | **38** | **39** | **40** | **41** | **42** | **43** | **44** | **45** | **46** | **47** |
| **initValue** | 100 | 117 | 87 | 85 | 86 | 103 | 84 | 118 | 120 | 117 | 150 | 120 |
|  | **48** | **49** | **50** | **51** | **52** | **53** | **54** | **55** | **56** | **57** | **58** | **59** |
| **initValue** | 84 | 102 | 88 | 117 | 118 | 104 | 87 | 119 | 136 | 83 | 116 | 118 |
|  | **60** | **61** | **62** | **63** | **64** | **65** | **66** | **67** | **68** | **69** | **70** | **71** |
| **initValue** | 84 | 117 | 87 | 69 | 86 | 87 | 84 | 118 | 120 | 117 | 150 | 120 |

#### Memorization process for context variables

Inputs of this process are the CABAC context variables indexed by ctxIdx.

Output of this process are variables TableStateSync and TableMPSSync containing the values of the variables m and n used in the initialisation process of context variables that are assigned to all syntax elements in subclauses ‎7.3.4 and ‎7.3.5 except for the end-of-slice flag and those related to CAVLC.

For each context variable, the corresponding entries n and m of tables TableStateSync and TableMPSSync are initialized to the corresponding pStateIdx and valMPS.

#### Synchronization process for context variables

Inputs of this process are variables TableStateSync and TableMPSSync containing the values of the variables n and m used in the memorization process of context variables that are assigned to all syntax elements in subclauses ‎7.3.4 and ‎7.3.5 except for the end-of-slice flag and those related to CAVLC.

Outputs of this process are the CABAC context variables indexed by ctxIdx..

For each context variable, the corresponding context variables pStateIdx and valMPS are initialized to the corresponding entries n and m of tables TableStateSync and TableMPSSync.

#### Initialisation process for the arithmetic decoding engine

This process is invoked before decoding the first treeblock of a slice.

Outputs of this process are the initialised decoding engine registers codIRange and codIOffset both in 16 bit register precision.

The status of the arithmetic decoding engine is represented by the variables codIRange and codIOffset. In the initialisation procedure of the arithmetic decoding process, codIRange is set equal to 510 and codIOffset is set equal to the value returned from read\_bits( 9 ) interpreted as a 9 bit binary representation of an unsigned integer with most significant bit written first.

The bitstream shall not contain data that result in a value of codIOffset being equal to 510 or 511.

NOTE – The description of the arithmetic decoding engine in this Recommendation | International Standard utilizes 16 bit register precision. However, a minimum register precision of 9 bits is required for storing the values of the variables codIRange and codIOffset after invocation of the arithmetic decoding process (DecodeBin) as specified in subclause 9.2.3.2. The arithmetic decoding process for a binary decision (DecodeDecision) as specified in subclause 9.2.3.2.1 and the decoding process for a binary decision before termination (DecodeTerminate) as specified in subclause 9.2.3.2.4 require a minimum register precision of 9 bits for the variables codIRange and codIOffset. The bypass decoding process for binary decisions (DecodeBypass) as specified in subclause 9.2.3.2.3 requires a minimum register precision of 10 bits for the variable codIOffset and a minimum register precision of 9 bits for the variable codIRange.

### Binarization process

Input to this process is a request for a syntax element.

Output of this process is the binarization of the syntax element, maxBinIdxCtx, ctxIdxOffset, and bypassFlag.

Table 9‑46 specifies the type of binarization process, maxBinIdxCtx, ctxIdxTable, and ctxIdxOffset associated with each syntax element.

The specification of the unary (U) binarization process, the truncated unary (TU) binarization process, the truncated Rice (TR), the k-th order Exp-Golomb (EGk) binarization process, and the fixed-length (FL) binarization process are given in subclauses 0 to 9.2.2.5, respectively. Other binarizations are specified in subclauses 9.2.2.6 to 9.2.2.10.

The binarizations for the syntax element coeff\_abs\_level\_minus3 as specified in subclause 9.2.2.9 consist of bin strings given by a concatenation of prefix and suffix bit strings. For these binarization processes, the prefix and the suffix bit string are separately indexed using the binIdx variable as specified further in subclause 9.2.3. The two sets of prefix bit strings and suffix bit strings are referred to as the binarization prefix part and the binarization suffix part, respectively.

Associated with each binarization or binarization part of a syntax element is a specific value of the context index table (ctxIdxTable) variable and the related context index offset (ctxIdxOffset) variable and a specific value of the maxBinIdxCtx variable as given in Table 9‑46. When two values for each of these variables are specified for one syntax element in Table 9‑46, the value in the upper row is related to the prefix part while the value in the lower row is related to the suffix part of the binarization of the corresponding syntax element.

The use of the DecodeBypass process and the variable bypassFlag is derived as follows.

– If no value is assigned to ctxIdxOffset for the corresponding binarization or binarization part in Table 9‑46 labelled as "na", all bins of the bit strings of the corresponding binarization or of the binarization prefix/suffix part are decoded by invoking the DecodeBypass process as specified in subclause 9.3.3.2.3. In such a case, bypassFlag is set equal to 1, where bypassFlag is used to indicate that for parsing the value of the bin from the bitstream the DecodeBypass process is applied.

– Otherwise, for each possible value of binIdx up to the specified value of maxBinIdxCtx given in Table 9‑46, a specific value of the variable ctxIdx is further specified in subclause 9.3.3. bypassFlag is set equal to 0.

The possible values of the context index ctxIdx vary depending on the value of ctxIdxTable. The value assigned to ctxIdxOffset specifies the lower value of the range of ctxIdx assigned to the corresponding binarization or binarization part of a syntax element.

ctxIdxTable = 0 and ctxIdx = ctxIdxOffset = 0 are assigned to the syntax element end\_of\_slice\_flag as further specified in subclause 9.3.3.1. For parsing the value of the corresponding bin from the bitstream, the arithmetic decoding process for decisions before termination (DecodeTerminate) as specified in subclause 9.3.3.2.4 is applied.

| Table 9‑31 – Syntax elements and associated types of binarization, maxBinIdxCtx, ctxIdxTable, and ctxIdxOffset | | | | | |
| --- | --- | --- | --- | --- | --- |
| **Syntax element** |  | **Type of binarization** | **maxBinIdxCtx** | **ctxIdxTable** | **ctxIdxOffset** |
| alf\_cu\_flag | I | FL, cMax = 1 | 0 | Table 9‑21 | 0 |
| P | 0 | Table 9‑21 | 1 |
| B | 0 | Table 9‑21 | 2 |
| end\_of\_slice\_flag | all | FL, cMax = 1 | 0 | 0 | 0 |
| split\_coding\_unit\_flag | I | FL, cMax = 1 | 0 | Table 9‑22 | 0 |
| P | 0 | Table 9‑22 | 3 |
| B | 0 | Table 9‑22 | 6 |
| skip\_flag | P | FL, cMax = 1 | 0 | Table 9‑23 | 0 |
| B | 0 | Table 9‑23 | 3 |
| cu\_qp\_delta | I | as specified in subclause 9.3.2.6 | 2 | Table 9‑24 | 0 |
| P | 2 | Table 9‑24 | 3 |
| B | 2 | Table 9‑24 | 6 |
| pred\_mode\_flag | P | FL, cMax = 1 | 0 | Table 9‑25 | 0 |
| B | 0 | Table 9‑25 | 1 |
| part\_mode | I | as specified in subclause  9.3.2.7 | 0 | Table 9‑25 | 2 |
| P | 3 | Table 9‑25 | 3 |
| B | 3 | Table 9‑25 | 6 |
| pcm\_flag | all | FL, cMax = 1 | 0 | 0 | 0 |
| prev\_intra\_luma\_pred\_flag | I | FL, cMax = 1 | 0 | Table 9‑26 | 0 |
| P | 0 | Table 9‑26 | 1 |
| B | 0 | Table 9‑26 | 2 |
| mpm\_flag | all | FL, cMax = 1 | na | na | na, (uses Decode Bypass) |
| rem\_intra\_luma\_pred\_mode | all | as specified in subclause 9.3.2.8 | na | na | na, (uses Decode Bypass) |
| intra\_chroma\_pred\_mode ( IntraPredMode < 4 ) | I | TU, cMax = 3 | 1 | Table 9‑28 | 0 |
| P | 1 | Table 9‑28 | 2 |
| B | 1 | Table 9‑28 | 4 |
| intra\_chroma\_pred\_mode ( 4 <= IntraPredMode < 34 ) | I | TU, cMax = 4 | 1 | Table 9‑28 | 0 |
| P | 1 | Table 9‑28 | 2 |
| B | 1 | Table 9‑28 | 4 |
| merge\_flag | P | FL, cMax = 1 | 0 | Table 9‑29 | 0 |
| B | 0 | Table 9‑29 | 1 |
| merge\_idx | P | TU, cMax = MaxNumMergeCand − 1 | 3 | Table 9‑30 | 0 |
| B | 3 | Table 9‑30 | 4 |
| inter\_pred\_flag | B | FL, cMax = 1 | 0 | Table 9‑31 | 0 |
| ref\_idx\_lc | B | TU, cMax = num\_ref\_idx\_lC\_active\_minus1 | 2 | Table 9‑32 | 3 |
| ref\_idx\_l0 | P | TU, cMax = num\_ref\_idx\_l0\_active\_minus1 | 2 | Table 9‑32 | 0 |
| B | 2 | Table 9‑32 | 3 |
| ref\_idx\_l1 | B | TU, cMax = num\_ref\_idx\_l1\_active\_minus1 | 2 | Table 9‑32 | 3 |
| abs\_mvd\_greater0\_flag[ ] | P | FL, cMax = 1 | 0 | Table 9‑33 | 0 |
| B | 0 | Table 9‑33 | 1 |
| abs\_mvd\_greater1\_flag[ ] | P | FL, cMax = 1 | 0 | Table 9‑33 | 2 |
| B | 0 | Table 9‑33 | 3 |
| abs\_mvd\_minus2[ ] | P/B | EG1 | na | na | na, (uses Decode Bypass) |
| mvd\_sign\_flag[ ] | P/B | FL, cMax = 1 | na | na | na, (uses Decode Bypass) |
| mvp\_lc\_flag | B | FL, cMax = 1 | 0 | Table 9‑34 | 1 |
| mvp\_l0\_flag | P | FL, cMax = 1 | 0 | Table 9‑34 | 0 |
| B | 0 | Table 9‑34 | 1 |
| mvp\_l1\_flag | B | FL, cMax = 1 | 0 | Table 9‑34 | 1 |
| no\_residual\_data\_flag | P | FL, cMax = 1 | 0 | Table 9‑35 | 0 |
| B | 0 | Table 9‑35 | 1 |
| split\_transform\_flag | I | FL, cMax = 1 | 0 | Table 9‑36 | 0 |
| P | 0 | Table 9‑36 | 4 |
| B | 0 | Table 9‑36 | 8 |
| cbf\_luma | I | FL, cMax = 1 | 0 | Table 9‑37 | 0 |
| P | 0 | Table 9‑37 | 2 |
| B | 0 | Table 9‑37 | 4 |
| cbf\_cb, cbf\_cr | I | FL, cMax = 1 | 0 | Table 9‑38 | 0 |
| P | 0 | Table 9‑38 | 3 |
| B | 0 | Table 9‑38 | 6 |
| last\_significant\_coeff\_x\_prefix | I | TU,  cMax = ( log2TrafoWidth << 1 ) − 1 | 8 | Table 9‑40 | 0 |
| P | 8 | Table 9‑40 | 30 |
| B | 8 | Table 9‑40 | 60 |
| last\_significant\_coeff\_y\_prefix | I | TU,  cMax = ( log2TrafoHeight << 1 ) − 1 | 8 | Table 9‑26 | 0 |
| P | 8 | Table 9‑26 | 30 |
| B | 8 | Table 9‑26 | 60 |
| last\_significant\_coeff\_x\_suffix | all | FL, cMax =   (last\_significant\_coeff\_x\_prefix >> 1) − 1 | na | na | na, (uses Decode Bypass) |
| last\_significant\_coeff\_y\_suffix | all | FL, cMax =   (last\_significant\_coeff\_y\_prefix >> 1) − 1 | na | na | na, (uses Decode Bypass) |
| significant\_coeff\_group\_flag | I | FL, cMax = 1 | 0 | Table 9-26 | 0 |
| P | 0 | Table 9-26 | 4 |
| B | 0 | Table 9-26 | 8 |
| significant\_coeff\_flag | I | FL, cMax = 1 | 0 | Table 9‑41 | 0 |
| P | 0 | Table 9‑41 | 48 |
| B | 0 | Table 9‑41 | 96 |
| coeff\_abs\_level\_greater1\_flag | I | FL, cMax = 1 | 0 | Table 9‑44 | 0 |
| P | 0 | Table 9‑44 | 32 |
| B | 0 | Table 9‑44 | 64 |
| coeff\_abs\_level\_greater2\_flag | I | FL, cMax = 1 | 0 | Table 9‑45 | 0 |
| P | 0 | Table 9‑45 | 24 |
| B | 0 | Table 9‑45 | 48 |
| coeff\_abs\_level\_minus3 | all | prefix and suffix as specified in subclause 9.3.2.9 | prefix: na  suffix: na | prefix: na  suffix: na | prefix: na, (uses Decode Bypass) suffix: na, (uses Decode Bypass) |
| coeff\_sign\_flag | all | FL, cMax = 1 | na | na | na, (uses Decode Bypass) |

#### Unary (U) binarization process

Input to this process is a request for a U binarization for a syntax element.

Output of this process is the U binarization of the syntax element.

The bin string of a syntax element having (unsigned integer) value synElVal is a bit string of length synElVal + 1 indexed by binIdx. The bins for binIdx less than synElVal are equal to 1. The bin with binIdx equal to synElVal is equal to 0.

Table 9‑47 illustrates the bin strings of the unary binarization for a syntax element.

Table 9‑32 – Bin string of the unary binarization (informative)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Value of syntax element** | **Bin string** | | | | | |
| 0 (I\_NxN) | 0 |  |  |  |  |  |
| 1 | 1 | 0 |  |  |  |  |
| 2 | 1 | 1 | 0 |  |  |  |
| 3 | 1 | 1 | 1 | 0 |  |  |
| 4 | 1 | 1 | 1 | 1 | 0 |  |
| 5 | 1 | 1 | 1 | 1 | 1 | 0 |
| … |  |  |  |  |  |  |
| binIdx | 0 | 1 | 2 | 3 | 4 | 5 |

#### Truncated unary (TU) binarization process

Input to this process is a request for a TU binarization for a syntax element and cMax.

Output of this process is the TU binarization of the syntax element.

For syntax element (unsigned integer) values less than cMax, the U binarization process as specified in subclause 9.3.2.1 is invoked. For the syntax element value equal to cMax the bin string is a bit string of length cMax with all bins being equal to 1.

NOTE – TU binarization is always invoked with a cMax value equal to the largest possible value of the syntax element being decoded.

#### Truncated Rice (TR) binarization process

Input to this process is a request for a TR binarization for a syntax element, cRiceParam and cTRMax.

Output of this process is the TR binarization of the syntax element.

A TR bin string is a concatenation of a prefix bit string and a suffix bit string. The prefix of the binarization is specified by invoking the TU binarization process for the prefix part of the value specified by synElVal >> cRiceParam with cMax = cTRMax >> cRiceParam. The suffix of the TR bin string is the binary representation of synElVal – ( ( synElVal >> cRiceParam ) << cRiceParam ).

NOTE – For the input parameter cRiceParam = 0 the TR binarization is exactly the TU binarization.

#### k-th order Exp-Golomb (EGk) binarization process

Input to this process is a request for an EGk binarization for a syntax element.

Output of this process is the EGk binarization of the syntax element.

The bin string of the EGk binarization process of a syntax element synVal is specified by a process equivalent to the following pseudo-code:

absV = Abs( synVal )  
stopLoop = 0  
do {  
 if( absV >= ( 1 << k ) ) {  
 put( 1 )  
 absV = absV − ( 1 << k )  
 k++  
 } else {  
 put( 0 ) (9‑5)  
 while( k− − )   
 put( ( absV >> k ) & 1 )  
 stopLoop = 1  
 }  
} while( !stopLoop )

NOTE 1 – The specification for the k-th order Exp-Golomb (EGk) code uses 1's and 0's in reverse meaning for the unary part of the Exp-Golomb code of 0-th order as specified in subclause 9.1.

#### Fixed-length (FL) binarization process

Input to this process is a request for a FL binarization for a syntax element and cMax.

Output of this process is the FL binarization of the syntax element.

FL binarization is constructed by using a fixedLength‑bit unsigned integer bin string of the syntax element value, where fixedLength = Ceil( Log2( cMax + 1 ) ). The indexing of bins for the FL binarization is such that the binIdx = 0 relates to the least significant bit with increasing values of binIdx towards the most significant bit.

#### Binarization process for cu\_qp\_delta

Input to this process is a request for a binarization for the syntax element cu\_qp\_delta.

Output of this process is the binarization of the syntax element.

The bin string of this binarization process of a syntax element value synVal is specified by a process equivalent to the following pseudo-code.

absV = Abs( synVal )   
if( absV = = 0 ){   
 put( 0 )   
} else {  
 put( 1 )   
 signV = ( synVal > 0 ) ? 0 : 1  
 put( signV )   
 cMax = 24 + ( QpBdOffsetY >> 1 ) + signV  
 cNum = absV − 1  
 absVGreaterThan1Flag = ( absV = = 1 ) ? 0 : 1  
 put( absVGreaterThan1Flag )   
 if( absVGreaterThan1Flag ){   
 while( cNum−− )   
 put( 1 )   
 if( cMax > absV − 1 )  
 put( 0 )   
 }  
}

#### Binarization process for part\_mode

Input to this process is a request for a binarization for the syntax element part\_mode and a variable cLog2CUSize specifying the current CU size.

Output of this process is the binarization of the syntax element.

The binarization for part\_mode is given by Table 9‑32 depending on PredMode, cLog2CUSize and inter\_4x4\_enabled\_flag.

Table 9‑33 – Binarization for part\_mode

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **PredMode** | **Value of part\_mode** | **PartMode** | **Bin string** | | |
| cLog2CUSize >  Log2MinCUSize | cLog2CUSize = = Log2MinCUSize | |
| cLog2CUSize = = 3 &&  !inter\_4x4\_enabled\_flag | cLog2CUSize > 3 | |  inter\_4x4\_enabled\_flag |
| MODE\_INTRA | 0 | PART\_2Nx2N | - | 1 | 1 |
| 1 | PART\_NxN | - | 0 | 0 |
| MODE\_INTER | 0 | PART\_2Nx2N | 1 | 1 | 1 |
| 1 | PART\_2NxN | 011 | 01 | 01 |
| 2 | PART\_Nx2N | 001 | 00 | 001 |
| 3 | PART\_NxN | - | - | 000 |
| 4 | PART\_2NxnU | 0100 | - | - |
| 5 | PART\_2NxnD | 0101 | - | - |
| 6 | PART\_nLx2N | 0000 | - | - |
| 7 | PART\_nRx2N | 0001 | - | - |

#### Binarization process for rem\_intra\_luma\_pred\_mode

Input to this process is a request for a binarization for the syntax element rem\_intra\_luma\_pred\_mode and log2PUSize.

Output of this process is the binarization of the syntax element.

The binarization for rem\_intra\_luma\_pred\_mode is given by Table 9‑49.

Table 9‑34 – Binarization for rem\_intra\_luma\_pred\_mode

|  |  |  |
| --- | --- | --- |
| **rem\_intra\_luma\_pred\_mode** | **log2PUSize** | **Bin string** |
| less than 32 | 2 | FL, cMax = 4 |
| 3...6 | FL, cMax = 5 |
| 32 | 2...6 | 111110 |
| 33 | 2...6 | 111111 |

#### Binarization process for coeff\_abs\_level\_minus3

Input to this process is a request for a binarization for the syntax element coeff\_abs\_level\_minus3[ n ].

Output of this process is the binarization of the syntax element.

The variables cLastSE and cLastRiceParam are derived as follows.

* If n is equal to 0 or all previous syntex elements coeff\_abs\_level\_minus3[ pos ] with pos less than n are derived to be equal to 0 instead of being explicitly parsed, cLastSE and cLastRiceParam are set equal to 0.
* Otherwise ( n is not equal to 0 ), cLastSE is set equal to coeff\_abs\_level\_minus3[ n  − 1 ] and cLastRiceParam is set equal to the value of cRiceParam that has been derived during the invocation of this subclause for the syntax element coeff\_abs\_level\_minus2[ n – 1 ] of the same transform block.

The variable cRiceParam is derived from cLastSE and cLastRiceParam as given in Table 9‑34.

**Table 9‑35 – Specifcation of cRiceParam depending on cLastSE and cLastRiceParam**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **cLastRiceParam** | **cLastSE** | | | | | | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **>10** |
| **0** | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 3 |
| **1** | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 3 |
| **2** | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 3 |
| **3** | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |

The variable cTRMax is derived from cRiceParam as given in Table 9‑35.

Table 9‑36 – Specifcation of cTRMax depending on cRiceParam

|  |  |
| --- | --- |
| **cRiceParam** | **cTRMax** |
| 0 | 7 |
| 1 | 20 |
| 2 | 42 |
| 3 | 70 |

The binarization of coeff\_abs\_level\_minus3 consists of a prefix part and (when present) a suffix part. The prefix part of the binarization is derived by invoking the TR binarization process as specified in subclause 9.3.2.3 for the syntax element coeff\_abs\_level\_minus3[ n ] with the variables cRiceParam and cTRMax as the inputs.

When the prefix bit string is not equal to the bin string that consists of ( cTRMax >> cRiceParam ) + cRiceParam ones, the bin string consists of a prefix bin string and a suffix bin string. The suffix bin string is derived using the EG0 binarization as specified in subclause 9.3.2.4.

### Decoding process flow

Input to this process is a binarization of the requested syntax element, maxBinIdxCtx, bypassFlag, ctxIdxTable and ctxIdxOffset as specified in subclause 9.3.2.

Output of this process is the value of the syntax element.

This process specifies how each bit of a bit string is parsed for each syntax element.

After parsing each bit, the resulting bit string is compared to all bin strings of the binarization of the syntax element and the following applies.

– If the bit string is equal to one of the bin strings, the corresponding value of the syntax element is the output.

– Otherwise (the bit string is not equal to one of the bin strings), the next bit is parsed.

While parsing each bin, the variable binIdx is incremented by 1 starting with binIdx being set equal to 0 for the first bin.

When the binarization of the corresponding syntax element consists of a prefix and a suffix binarization part, the variable binIdx is set equal to 0 for the first bin of each part of the bin string (prefix part or suffix part). In this case, after parsing the prefix bit string, the parsing process of the suffix bit string related to the binarizations specified in subclauses 9.3.2.3 and 9.3.2.10 is invoked depending on the resulting prefix bit string as specified in subclauses 9.3.2.3 and 9.3.2.10.

Depending on the variable bypassFlag, the following applies.

– If bypassFlag is equal to 1, the bypass decoding process as specified in subclause 9.3.3.2.3 is applied for parsing the value of the bins from the bitstream.

– Otherwise (bypassFlag is equal to 0), the parsing of each bin is specified by the following two ordered steps:

1. Given binIdx, maxBinIdxCtx, ctxIdxTable, and ctxIdxOffset, ctxIdx is derived as specified in subclause 9.3.3.1.

2. Given ctxIdxTable and ctxIdx, the value of the bin from the bitstream as specified in subclause 9.3.3.2 is decoded.

#### Derivation process for ctxIdx

Inputs to this process are binIdx, maxBinIdxCtx, ctxIdxTable, and ctxIdxOffset.

Output of this process is ctxIdx.

Table 9‑51 shows the assignment of ctxIdx increments (ctxIdxInc) to binIdx for all ctxIdxTable and ctxIdxOffset.

The ctxIdx to be used with a specific binIdx is specified by first determining the ctxIdxTable and ctxIdxOffset associated with the given bin string or part thereof. The ctxIdxTable and the ctxIdxOffset are listed in Table 9‑51, the ctxIdx for a binIdx is the sum of ctxIdxOffset and ctxIdxInc, which is found in Table 9‑51. When more than one value is listed in Table 9‑51 for a binIdx, the assignment process for ctxIdxInc for that binIdx is further specified in the subclauses given in parenthesis of the corresponding table entry.

All bins with binIdx greater than maxBinIdxCtx are parsed using the value of ctxIdx being assigned to binIdx equal to maxBinIdxCtx.

All entries in Table 9‑51 labelled with "na" correspond to values of binIdx that do not occur for the corresponding ctxIdxOffset.

| Table 9‑37 – Assignment of ctxIdxInc to binIdx for all ctxIdxTable and ctxIdxOffset values | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Syntax element** | **ctxIdxTable,  ctxIdxOffset** | | **binIdx** | | | | | |
| **0** | **1** | **2** | **3** | | **>=4** |
| alf\_cu\_flag | Table 9‑21 | 0 | 0 | na | na | na | | na |
| 1 | 0 | na | na | na | | na |
| 2 | 0 | na | na | na | | na |
| split\_coding\_unit\_flag | Table 9‑22 | 0 | 0,1,2 (subclause 9.3.3.1.1.1) | na | na | na | | na |
| 3 | 0,1,2 (subclause 9.3.3.1.1.1) | na | na | na | | na |
| 6 | 0,1,2 (subclause 9.3.3.1.1.1) | na | na | na | | na |
| skip\_flag | Table 9‑23 | 0 | 0,1,2 (subclause 9.3.3.1.1.1) | na | na | na | | na |
| 3 | 0,1,2 (subclause 9.3.3.1.1.1) | na | na | na | | na |
| cu\_qp\_delta | Table 9‑24 | 0 | 0 | na (uses Decode Bypass) | 1 | 2 | | 2 |
| 3 | 0 | na (uses Decode Bypass) | 1 | 2 | | 2 |
| 6 | 0 | na (uses Decode Bypass) | 1 | 2 | | 2 |
| pred\_mode\_flag | Table 9‑25 | 0 | 0 | na | na | na | | na |
| 1 | 0 | na | na | na | | na |
| part\_mode | Table 9‑25 | 2 | 0 | na | na | na | | na |
| 3 | 0 | 1 | 0 | 1 | | na |
| 6 | 0 | 1 | 0 | 1 | | na |
| part\_mode (cLog2CUSize = = Log2MinCUSize &&  cLog2CUSize > 3 | | inter\_4x4\_enabled\_flag) | Table 9‑25 | 3 | 0 | 1 | 2 | na | | na |
| 6 | 0 | 1 | 2 | na | | na |
| prev\_intra\_luma\_pred\_flag | Table 9‑26 | 0 | 0 | na | na | na | | na |
| 1 | 0 | na | na | na | | na |
| 2 | 0 | na | na | na | | na |
| intra\_chroma\_pred\_mode | Table 9‑28 | 0 | 0 | 1 | 1 | 1 | | na |
| 2 | 0 | 1 | 1 | 1 | | na |
| 4 | 0 | 1 | 1 | 1 | | na |
| merge\_flag | Table 9‑29 | 0 | 0 | na | na | na | | na |
| 1 | 0 | na | na | na | | na |
| merge\_idx | Table 9‑30 | 0 | 0,1,2,3 (subclause 9.3.3.1.1.2) | 0,1,2,3 (subclause 9.3.3.1.1.2) | 0,2,3 (subclause 9.3.3.1.1.2) | 3 | | na |
| 4 | 0,1,2,3 (subclause 9.3.3.1.1.2) | 0,1,2,3 (subclause 9.3.3.1.1.2) | 0,2,3 (subclause 9.3.3.1.1.2) | 3 | | na |
| inter\_pred\_flag | Table 9‑31 | 0 | cuDepth | na | na | na | | na |
| ref\_idx\_l0 | Table 9‑32 | 0 | 0 | 1 | 2 | 2 | | 2 |
| ref\_idx\_l0, ref\_idx\_l1, ref\_idx\_lc | Table 9‑32 | 3 | 0 | 1 | 2 | 2 | | 2 |
| abs\_mvd\_greater0\_flag[ ] | Table 9‑33 | 0 | 0 | na | na | na | | na |
| 1 | 0 | na | na | na | | na |
| abs\_mvd\_greater1\_flag[ ] | Table 9‑33 | 2 | 0 | na | na | na | | na |
| 3 | 0 | na | na | na | | na |
| mvp\_l0\_flag | Table 9‑34 | 0 | 0 | na | na | na | | na |
| mvp\_l0\_flag, mvp\_l1\_flag, mvp\_lc\_flag | Table 9‑34 | 1 | 0 | na | na | na | | na |
| no\_residual\_data\_flag | Table 9‑35 | 0 | 0 | na | na | na | | na |
| 1 | 0 | na | na | na | | na |
| split\_transform\_flag | Table 9‑36 | 0 | cuDepth + trafoDepth | na | na | na | | na |
| 4 | cuDepth + trafoDepth | na | na | na | | na |
| 8 | cuDepth + trafoDepth | na | na | na | | na |
| cbf\_luma | Table 9‑37 | 0 | ( trafoDepth = = 0 ) | |  ( log2TrafoSize = =  Log2MaxTrafoSize ) ? 1 : 0 | na | na | na | | na |
| 2 | ( trafoDepth = = 0 ) | |  ( log2TrafoSize = =  Log2MaxTrafoSize ) ? 1 : 0 | na | na | na | | na |
| 4 | ( trafoDepth = = 0 ) | |  ( log2TrafoSize = =  Log2MaxTrafoSize ) ? 1 : 0 | na | na | na | | na |
| cbf\_cb, cbf\_cr | Table 9‑38 | 0 | trafoDepth | na | na | na | | na |
| 3 | trafoDepth | na | na | na | | na |
| 6 | trafoDepth | na | na | na | | na |
| last\_significant\_coeff\_x\_prefix | Table 9‑40 | 0 | 0..29 (subclause 9.3.3.1.1.3) | | | | | |
| 30 | 0..29 (subclause 9.3.3.1.1.3) | | | | | |
| 60 | 0..29 (subclause 9.3.3.1.1.3) | | | | | |
| last\_significant\_coeff\_y\_prefix | Table 9‑26 | 0 | 0..29 (subclause 9.3.3.1.1.3) | | | | | |
| 30 | 0..29 (subclause 9.3.3.1.1.3) | | | | | |
| 60 | 0..29 (subclause 9.3.3.1.1.3) | | | | | |
| significant\_coeff\_group\_flag | Table 9-26 | 0 | 0..3 (subclause 9.2.3.1.1.4) |  |  | |  |  |
| 4 | 0..3 (subclause 9.2.3.1.1.4) |  |  | |  |  |
| 8 | 0..3 (subclause 9.2.3.1.1.4) |  |  | |  |  |
| significant\_coeff\_flag | Table 9‑41 | 0 | 0..47 (subclause 9.3.3.1.1.4) | na | na | na | | na |
| 48 | 48..95 (subclause 9.3.3.1.1.4) | na | na | na | | na |
| 96 | 96..143 (subclause 9.3.3.1.1.4) | na | na | na | | na |
| coeff\_abs\_level\_greater1\_flag | Table 9‑44 | 0 | 0..31 (subclause 9.3.3.1.1.5) | na | na | na | | na |
| 32 | 0..63 (subclause 9.3.3.1.1.5) | na | na | na | | na |
| 64 | 0..95 (subclause 9.3.3.1.1.5) | na | na | na | | na |
| coeff\_abs\_level\_greater2\_flag | Table 9‑45 | 0 | 0..23 (subclause 9.3.3.1.1.6) | na | na | na | | na |
| 24 | 0..47 (subclause 9.3.3.1.1.6) | na | na | na | | na |
| 48 | 0..71 (subclause 9.3.3.1.1.6) | na | na | na | | na |

##### Assignment process of ctxIdxInc using neighbouring syntax elements

Subclause 9.3.3.1.1.1 specifies the derivation process of ctxIdxInc for the syntax elements split\_coding\_unit\_flag and skip\_flag.

###### Derivation process of ctxIdxInc using left and above syntax elements

Input to this process is the luma location ( xP, yP ) specifying the top-left luma sample of the current prediction unit relative to the top-left sample of the current picture.

Output of this process is ctxIdxInc.

Let the luma location ( xL, yL ) specify a location covered by the prediction unit to the left of the top-left luma sample of the current prediction unit with xL = xP − MinPuSize and yL = yP, the variable availableL is derived as follows. [Ed.: (BB) MinPuSize should be defined somewhere]

* If the prediction unit covering location ( xL, yL ) is available, availableL is set equal to 1.
* Otherwise (the prediction unit covering location ( xL, yL ) is not available), availableL is set equal to 0.

Let the luma location ( xA, yA ) specify a location covered by the prediction unit above the top-left luma sample of the current prediction unit with xA = xP and yA = yP − MinPuSize, the variable availableA is derived as follows. [Ed.: (BB) MinPuSize should be defined somewhere]

* If the prediction unit covering location ( xA, yA ) is available, availableA is set equal to 1.
* Otherwise (the prediction unit covering location ( xA, yA ) is not available), availableA is set equal to 0

The assignment of ctxIdxInc for the syntax elements alf\_cu\_flag and skip\_flag is specified in Table 9‑52.

Table 9‑38 – Specifcation of ctxIdxInc using left and above syntax elements

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax element** | **condL** | **condA** | **ctxIdxInc** |
| split\_coding\_unit\_flag | cuDepth[ xL ][ yL ]  >  cuDepth[ xP ][ yP ] | cuDepth[ xA ][ yA ]  >  cuDepth[ xP ][ yP ] | ( condL && availableL ) + ( condA && availableA ) |
| skip\_flag | skip\_flag[ xL ][ yL ] | skip\_flag[ xA ][ yA ] | ( condL && availableL ) + ( condA && availableA ) |

###### Derivation process of ctxIdxInc for the syntax element merge\_index

Input to this process is binIdx.

Output of this process is ctxIdxInc.

The variable ctxIdxInc is derived as follows.

* If binIdx is equal to 0, the following applies.
* If availableLeft is equal to 1and availableAbove is equal to 1, ctxIdxInc is set equal to 0.
* Otherwise if availableLeft is equal to 1 or availableAbove is equal to 1, ctxIdxInc is set equal to 2.
* Otherwise (availableLeft is equal to 0 and availableAbove is equal to 0), ctxIdxInc is set equal to 3.
* Otherwise if binIdx is equal to 1 and NumMergeCand is greater than 2, the following applies.
* If availableAbove is equal to 1, ctxIdxInc is set equal to 2.
* Otherwise (availableAbove is equal to 0), ctxIdxInc is set equal to 3.
* Otherwise if binIdx is equal to 2 and NumMergeCand is greater than 3, ctxIdxInc is set equal to 3.
* Otherwise, ctxIdxInc is set equal to 0.

When ctxIdxInc is not equal to 0, ctxIdxInc is derived as follows.

ctxIdxInc = ctxIdxInc − availableCollocated (9‑51)

[Ed. (BB): The flags availableLeft, availableAbove, availableCollocated need to be defined or a simpler context derivation should be used.]

###### Derivation process of ctxIdxInc for the syntax elements last\_significant\_coeff\_x\_prefix and last\_significant\_coeff\_y\_prefix

Inputs to this process are the binIdx, the color component index cIdx, the transform block width log2TrafoWidth and the transform block height log2TrafoHeight.

Output of this process is ctxIdxInc.

Table 9‑39 – Specifcation of lastCtx[ i ]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **i** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** |
| **lastCtx[ i ]** | 0 | 1 | 2 | 2 | 3 | 4 | 5 | 5 | 6 | 6 | 7 | 8 | 9 | 9 |
| **i** | **14** | **15** | **16** | **17** | **18** | **19** | **20** | **21** | **22** | **23** | **24** | **25** | **26** | **27** |
| **lastCtx[ i ]** | 10 | 10 | 11 | 11 | 12 | 13 | 14 | 14 | 15 | 15 | 16 | 16 | 17 | 17 |

The vector lastCtx[ i ] is specified in Table 9‑53 and ctxIdxInc is derived as follows.

– For last\_significant\_coeff\_x\_prefix, ctxIdxInc is derived as follows.

ctxIdxInc = lastCtx[( ( log2TrafoWidth – 2 )\*( log2TrafoWidth + 1 ) ) + binIdx ] (9‑52)

– For last\_significant\_coeff\_y\_prefix, ctxIdxInc is derived as follows.

ctxIdxInc = lastCtx[ ( ( log2TrafoHeight – 2 )\*( log2TrafoHeight + 1 ) ) + binIdx ] (9‑53)

When cIdx is greater than 0, ctxIdxInc is modified as follows.

ctxIdxInc = ctxIdxInc + 18 (9‑54)

[Ed. (BB): The context derivation assumes maximum transform sizes less than or equal to 32x32 for luma and 16x16 for chroma and minimum transform sizes greater than or equal to 4x4.]

###### Derivation process of ctxIdxInc for the syntax element significant\_coeff\_group\_flag

Inputs to this process are the color component index cIdx, the current coefficient group scan position ( xCG, yCG ), the previously decoded bins of the syntax element significant\_coeff\_group\_flag, the width of the current transform block, log2TrafoWidth and the height of the current transform block, log2TrafoHeight.

Output of this process is ctxIdxInc.

The variable ctxCG is derived using the current position ( xCG, yCG ), two previously decoded bins of the syntax element significant\_coeff\_group\_flag in scan order, and the width of the current transform block, log2TrafoWidth and the height of the current transform block, log2TrafoHeight as follows.

* ctxCG is initialized with 0 as follows.

ctxCG = 0 (9‑13)

* When xCG is less than ( 1 << ( log2TrafoWidth − 2 ) ) − 1, ctxCG is modified as follows.

ctxCG += significant\_coeff\_group\_flag[ xCG + 1 ][ yCG ] (9‑14)

* When yCG is less than ( 1 << ( log2TrafoHeight − 2 ) ) − 1, ctxCG is modified as follows.

ctxCG += significant\_coeff\_group\_flag[ xCG ][ yCG + 1 ] (9‑15)

The context index increment ctxIdxInc is derived using the color component index cIdx and ctxCG as follows.

* If cIdx is equal to 0, ctxIdxInc is derived as follows.

ctxIdxInc = ctxCG (9‑16)

* Otherwise (cIdx is greater than 0), ctxIdxInc is derived as follows.

ctxIdxInc = 2 + ctxCG (9‑17)

NOTE – The values of ctxCG are in the range of 0 to 2, inclusive. A ctxCG value of 3 does not occur because significant\_coeff\_group\_flag[ xCG ][ yCG ] is inferred when significant\_coeff\_group\_flag[ xCG + 1 ][ yCG ] and significant\_coeff\_group\_flag[ xCG ][ yCG + 1 ] are both equal to 1.

###### Derivation process of ctxIdxInc for the syntax element significant\_coeff\_flag

Inputs to this process are the color component index cIdx, the current coefficient scan position ( xC , yC ), the transform block width log2TrafoWidth and the transform block height log2TrafoHeight.

Output of this process is ctxIdxInc.

The variable sigCtx depends on the current position ( xC, yC ), the color component index cIdx, the transform block size and previsously decoded bins of the syntax element significant\_coeff\_flag. For the derivation of sigCtx, the following applies.

* If log2TrafoWidth is equal to log2TrafoHeight and log2TrafoWidth is equal to 2, sigCtx is derived using ctxIdxMap4x4[ ] specified in Table 9‑39 as follows..

sigCtx = ctxIdxMap4x4[ ((cIdx > 0) ? 15 : 0) + (yC << 2) + xC ] (9‑55)

* Otherwise if log2TrafoWidth is equal to log2TrafoHeight and log2TrafoWidth is equal to 3, sigCtx is derived using ctxIdxMap8x8[ ] specified in Table 9‑40 as follows.

sigCtx = ((xC + yC) = = 0) ? 10 : ctxIdxMap8x8[ ((yC >> 1 ) << 2) + (xC >> 1) ] (9‑56)

sigCtx += ( cIdx > 0) ? 6: 9 (9‑56)

* Otherwise if xC + yC is equal to 0, sigCtx is derived as follows.

sigCtx = ( cIdx > 0) ? 17: 20 (9‑57)

* Otherwise (xC + yC is greater than 0), sigCtx is derived using previously decoded bins of the syntax element significant\_coeff\_flag as follows.
* The variable sigCtx is initialized as follows.

sigCtx = 0 (9‑58)

* When xC is less than ( 1 << log2TrafoWidth ) − 1, the following applies.

sigCtx = sigCtx + significant\_coeff\_flag[ xC + 1 ][ yC ] (9‑59)

* When xC is less than ( 1 << log2TrafoWidth ) − 1 and yC is less than ( 1 << log2TrafoHeight ) − 1, the following applies.

sigCtx = sigCtx + significant\_coeff\_flag[ xC + 1 ][ yC + 1 ] (9‑60)

* When xC is less than ( 1 << log2Width ) − 2, the following applies.

sigCtx = sigCtx + significant\_coeff\_flag[ xC + 2 ][ yC ] (9‑61)

* When all of the following conditions are true,
  + yC is less than ( 1 << log2TrafoHeight ) − 1,
  + xC % 4 is not equal to 0 or yC % 4 is not equal to 0,
  + xC % 4 is not equal to 3 or yC % 4 is not equal to 2,

the following applies.

sigCtx = sigCtx + significant\_coeff\_flag[ xC ][ yC + 1 ] (9‑62)

* When yC is less than ( 1 << log2TrafoHeight ) − 2 and sigCtx is less than 4, the following applies.

sigCtx = sigCtx + significant\_coeff\_flag[ xC ][ yC + 2 ] (9‑63)

* The variable sigCtx is modified as follows.
  + If cIdx is equal to 0 and xC + yC are greater than (1 << (max(log2TrafoWidth, log2TrafoHeight) − 2)) − 1, the following applies.

sigCtx = ( (sigCtx + 1) >> 1 ) + 24 (9‑63)

* + Otherwise, the following applies.

sigCtx = ( (sigCtx + 1) >> 1 ) + ( (cIdx > 0) ? 18 : 21 ) (9‑63)

The context index increment ctxIdxInc is derived using the color component index cIdx and sigCtx as follows.

* If cIdx is equal to 0, ctxIdxInc is derived as follows.

ctxIdxInc = sigCtx (9‑64)

* Otherwise (cIdx is greater than 0), ctxIdxInc is derived as follows.

ctxIdxInc = 27 + sigCtx (9‑65)

Table ‑ – Specifcation of ctxIdxMap4x4[ i ]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **i** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** |
| **ctxIdxMap4x4[ i ]** | 0 | 1 | 4 | 5 | 2 | 3 | 4 | 5 | 6 | 6 | 8 | 8 | 7 | 7 | 8 |
| **i** | **15** | **16** | **17** | **18** | **19** | **20** | **21** | **22** | **23** | **24** | **25** | **26** | **27** | **28** | **29** |
| **ctxIdxMap4x4[ i ]** | 0 | 1 | 2 | 4 | 1 | 1 | 2 | 4 | 3 | 3 | 5 | 5 | 4 | 4 | 5 |

Table ‑ – Specifcation of ctxIdxMap8x8[ i ]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **i** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ctxIdxMap8x8[ i ]** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 3 | 8 | 6 | 6 | 7 | 9 | 9 | 7 | 7 |

[Ed. (BB): The context derivation assumes maximum transform sizes less than or equal to 32x32 for luma and 16x16 for chroma and minimum transform sizes greater than or equal to 4x4.]

###### Derivation process of ctxIdxInc for the syntax element coeff\_abs\_level\_greater1\_flag

Inputs to this process are the color component index cIdx, the 16 coefficient subset index i and the current coefficient scan index n within the current subset.

Output of this process is ctxIdxInc.

The variable ctxSet specifies the current context set and for its derivation the following applies.

* If n is equal to 15 or all previous syntax elements coeff\_abs\_level\_greater1\_flag[ pos ] with pos greater than n are derived to be equal to 0 instead of being explicitly parsed, the following applies.
* The variable ctxSet is initialized as follows.
* If the current subset index i is equal to 0 or cIdx is greater than 0, the following applies.

ctxSet = 0 (9‑66)

* Otherwise (i is greater than 0 and cIdx is equal to 0), the following applies.

ctxSet = 3 (9‑67)

* When the subset i is not the first one to be processed in this subclause, the following applies.
* The variable numGreater1 is set equal to the variable numGreater1 that has been derived during the last invocation of subclause 9.3.3.1.1.6 for the syntax element coeff\_abs\_level\_greater2\_flag for the subset i + 1.
* When ( numGreater1 >> 1 ) is greater than 0, ctxSet is incremented by one as follows.

ctxSet = ctxSet + 1 (9‑68)

* When ( numGreater1 >> 1 ) is greater than 3 and cIdx is equal to 0, ctxSet is incremented by one as follows.

ctxSet = ctxSet + 1 (9‑69)

* The variable greater1Ctx is set equal to 1.
* Otherwise (coeff\_abs\_level\_greater1\_flag[ n ] is not the first to be parsed within the current subset i),.for the derivation of ctxSet and greater1Ctx the following applies.
* The variable ctxSet is set equal to the variable ctxSet that has been derived during the last invocation of this subclause.
* The variable greater1Ctx is set equal to the variable greater1Ctx that has been derived during the last invocation of this subclause.
* When greater1Ctx is greater than 0, the variable lastGreater1Flag is set equal to the syntax element coeff\_abs\_level\_greater1\_flag that has been used during the last invocation of this subclause and greater1Ctx is modifed as follows.
* If lastGreater1Flag is equal to 1, greater1Ctx is set equal to 0.
* Otherwise (lastGreater1Flag is equal to 0), greater1Ctx is incremented by 1.

The context index increment ctxIdxInc is derived using the current context set ctxSet and the current context greater1Ctx as follows.

ctxIdxInc = ( ctxSet \* 4 ) + Min( 3, greater1Ctx ) (9‑70)

When cIdx is greater than 0, ctxIdxInc is modified as follows.

ctxIdxInc = ctxIdxInc + 24 (9‑71)

###### Derivation process of ctxIdxInc for the syntax element coeff\_abs\_level\_greater2\_flag

Inputs to this process are the color component index cIdx, the 16 coeffcient subset index i and the current coefficient scan index n within the current subset.

Output of this process is ctxIdxInc.

The variable ctxSet specifies the current context set and for its derivation the following applies.

* If n is equal to 15 or all previous syntex elements coeff\_abs\_level\_greater2\_flag[ pos ] with pos greater than n are derived to be equal to 0 instead of being explicitly parsed, the following applies.
* The variable ctxSet is initialized as follows.
* If the current subset index i is equal to 0 or cIdx is greater than 0, the following applies.

ctxSet = 0 (9‑72)

* Otherwise (i is greater than 0 and cIdx is equal to 0), the following applies.

ctxSet = 3 (9‑73)

* The variable numGreater1 is set equal to 0.
* When the subset i is not the first one to be processed in this subclause, the following applies.
* The variable numGreater1 is set equal to the variable numGreater1 that has been derived during the last invocation of this subclause for the subset i + 1 and is modifed as follows.

numGreater1 = numGreater1 >> 1 (9‑74)

* When numGreater1 is greater than 0, ctxSet is incremented by one as follows.

ctxSet = ctxSet + 1 (9‑75)

* When numGreater1 is greater than 3 and cIdx is equal to 0, ctxSet is incremented by one as follows.

ctxSet = ctxSet + 1 (9‑76)

* The variable greater2Ctx is set equal to 0.
* Otherwise (coeff\_abs\_level\_greater2\_flag[ n ] is not the first to be parsed within the current subset i), for the derivation of ctxSet and greater2Ctx the following applies.
* The variable ctxSet is set equal to the variable ctxSet that has been derived during the last invocation of this subclause.
* The variable greater2Ctx is set equal to the variable greater2Ctx that has been derived during the last invocation of this subclause incremented by 1.
* The variable numGreater1 is set equal to the variable numGreater1 that has been derived during the last invocation of this subclause incremented by 1.

The context index increment ctxIdxInc is derived using the current context set ctxSet and the current context greater2Ctx as follows.

ctxIdxInc = ( ctxSet \* 3 ) + Min( 2, greater2Ctx ) (9‑77)

When cIdx is greater than 0, ctxIdxInc is modified as follows.

ctxIdxInc = ctxIdxInc + 18 (9‑78)

#### Arithmetic decoding process

Inputs to this process are the bypassFlag, ctxIdxTable, and ctxIdx as derived in subclause 9.3.3.1, and the state variables codIRange and codIOffset of the arithmetic decoding engine.

Output of this process is the value of the bin.

Figure 9‑1 illustrates the whole arithmetic decoding process for a single bin. For decoding the value of a bin, the context index table ctxIdxTable and the ctxIdx is passed to the arithmetic decoding process DecodeBin(ctxIdxTable, ctxIdx), which is specified as follows.

– If bypassFlag is equal to 1, DecodeBypass( ) as specified in subclause 9.3.3.2.3 is invoked.

– Otherwise, if bypassFlag is equal to 0, ctxIdxTable is equal  0 and ctxIdx is equal to 0, DecodeTerminate( ) as specified in subclause 9.3.3.2.4 is invoked.

– Otherwise (bypassFlag is equal to 0, ctxIdxTable is not equal to 0 and ctxIdx is not equal to 0), DecodeDecision( ) as specified in subclause 9.3.3.2.1 is applied.



Figure 9‑2 – Overview of the arithmetic decoding process for a single bin (informative)

NOTE – Arithmetic coding is based on the principle of recursive interval subdivision. Given a probability estimation p( 0 ) and p( 1 ) = 1 − p( 0 ) of a binary decision ( 0, 1 ), an initially given code sub-interval with the range codIRange will be subdivided into two sub-intervals having range p( 0 ) \* codIRange and codIRange − p( 0 ) \* codIRange, respectively. Depending on the decision, which has been observed, the corresponding sub-interval will be chosen as the new code interval, and a binary code string pointing into that interval will represent the sequence of observed binary decisions. It is useful to distinguish between the most probable symbol(MPS) and the least probable symbol(LPS), so that binary decisions have to be identified as either MPS or LPS, rather than 0 or 1. Given this terminology, each context is specified by the probability pLPS of the LPS and the value of MPS (valMPS), which is either 0 or 1.

The arithmetic core engine in this Recommendation | International Standard has three distinct properties:

– The probability estimation is performed by means of a finite-state machine with a table-based transition process between 64 different representative probability states { pLPS(pStateIdx) | 0 <= pStateIdx < 64 } for the LPS probability pLPS. The numbering of the states is arranged in such a way that the probability state with indexpStateIdx = 0 corresponds to an LPS probability value of 0.5, with decreasing LPS probability towards higher state indices.

– The range codIRange representing the state of the coding engine is quantised to a small set {Q1,…,Q4} of pre-set quantisation values prior to the calculation of the new interval range. Storing a table containing all 64x4 pre-computed product values of Qi \* pLPS(pStateIdx) allows a multiplication-free approximation of the product codIRange \* pLPS(pStateIdx).

– For syntax elements or parts thereof for which an approximately uniform probability distribution is assumed to be given a separate simplified encoding and decoding bypass process is used.

##### Arithmetic decoding process for a binary decision

Inputs to this process are ctxIdxTable, ctxIdx, codIRange, and codIOffset.

Outputs of this process are the decoded value binVal, and the updated variables codIRange and codIOffset.

Figure 9‑2 shows the flowchart for decoding a single decision (DecodeDecision):

1. The value of the variable codIRangeLPS is derived as follows.

– Given the current value of codIRange, the variable qCodIRangeIdx is derived by

qCodIRangeIdx =( codIRange >> 6 ) & 3 (9‑29)

– Given qCodIRangeIdx and pStateIdx associated with ctxIdxTable and ctxIdx, the value of the variable rangeTabLPS as specified in Table 9‑40 is assigned to codIRangeLPS:

codIRangeLPS = rangeTabLPS[ pStateIdx ][ qCodIRangeIdx ] (9‑30)

1. The variable codIRange is set equal to codIRange − codIRangeLPS and the following applies.

– If codIOffset is greater than or equal to codIRange, the variable binVal is set equal to 1 − valMPS, codIOffset is decremented by codIRange, and codIRange is set equal to codIRangeLPS.

– Otherwise, the variable binVal is set equal to valMPS.

Given the value of binVal, the state transition isperformed as specified in subclause 9.3.3.2.1.1. Depending on the current value of codIRange, renormalization is performed as specified in subclause 9.3.3.2.2.

###### State transition process

Inputs to this process are the current pStateIdx, the decoded value binVal and valMPS values of the context variable associated with ctxIdxTable and ctxIdx.

Outputs of this process are the updated pStateIdx and valMPS of the context variable associated with ctxIdx.

Depending on the decoded value binVal, the update of the two variables pStateIdx and valMPS associated with ctxIdx is derived as specified by the following pseudo-code:

if( binVal = = valMPS )   
 pStateIdx = transIdxMPS( pStateIdx )  
else { (9‑31)  
 if( pStateIdx = = 0 )  
 valMPS = 1 − valMPS  
 pStateIdx = transIdxLPS( pStateIdx )  
}

Table 9‑41 specifies the transition rules transIdxMPS( ) and transIdxLPS( ) after decoding the value of valMPS and 1 − valMPS, respectively.



Figure 9‑3 – Flowchart for decoding a decision

Table 9‑42 – Specification of rangeTabLPS depending on pStateIdx and qCodIRangeIdx

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **pStateIdx** | **qCodIRangeIdx** | | | | **pStateIdx** | **qCodIRangeIdx** | | | |
| **0** | **1** | **2** | **3** | **0** | **1** | **2** | **3** |
| **0** | 128 | 176 | 208 | 240 | **32** | 27 | 33 | 39 | 45 |
| **1** | 128 | 167 | 197 | 227 | **33** | 26 | 31 | 37 | 43 |
| **2** | 128 | 158 | 187 | 216 | **34** | 24 | 30 | 35 | 41 |
| **3** | 123 | 150 | 178 | 205 | **35** | 23 | 28 | 33 | 39 |
| **4** | 116 | 142 | 169 | 195 | **36** | 22 | 27 | 32 | 37 |
| **5** | 111 | 135 | 160 | 185 | **37** | 21 | 26 | 30 | 35 |
| **6** | 105 | 128 | 152 | 175 | **38** | 20 | 24 | 29 | 33 |
| **7** | 100 | 122 | 144 | 166 | **39** | 19 | 23 | 27 | 31 |
| **8** | 95 | 116 | 137 | 158 | **40** | 18 | 22 | 26 | 30 |
| **9** | 90 | 110 | 130 | 150 | **41** | 17 | 21 | 25 | 28 |
| **10** | 85 | 104 | 123 | 142 | **42** | 16 | 20 | 23 | 27 |
| **11** | 81 | 99 | 117 | 135 | **43** | 15 | 19 | 22 | 25 |
| **12** | 77 | 94 | 111 | 128 | **44** | 14 | 18 | 21 | 24 |
| **13** | 73 | 89 | 105 | 122 | **45** | 14 | 17 | 20 | 23 |
| **14** | 69 | 85 | 100 | 116 | **46** | 13 | 16 | 19 | 22 |
| **15** | 66 | 80 | 95 | 110 | **47** | 12 | 15 | 18 | 21 |
| **16** | 62 | 76 | 90 | 104 | **48** | 12 | 14 | 17 | 20 |
| **17** | 59 | 72 | 86 | 99 | **49** | 11 | 14 | 16 | 19 |
| **18** | 56 | 69 | 81 | 94 | **50** | 11 | 13 | 15 | 18 |
| **19** | 53 | 65 | 77 | 89 | **51** | 10 | 12 | 15 | 17 |
| **20** | 51 | 62 | 73 | 85 | **52** | 10 | 12 | 14 | 16 |
| **21** | 48 | 59 | 69 | 80 | **53** | 9 | 11 | 13 | 15 |
| **22** | 46 | 56 | 66 | 76 | **54** | 9 | 11 | 12 | 14 |
| **23** | 43 | 53 | 63 | 72 | **55** | 8 | 10 | 12 | 14 |
| **24** | 41 | 50 | 59 | 69 | **56** | 8 | 9 | 11 | 13 |
| **25** | 39 | 48 | 56 | 65 | **57** | 7 | 9 | 11 | 12 |
| **26** | 37 | 45 | 54 | 62 | **58** | 7 | 9 | 10 | 12 |
| **27** | 35 | 43 | 51 | 59 | **59** | 7 | 8 | 10 | 11 |
| **28** | 33 | 41 | 48 | 56 | **60** | 6 | 8 | 9 | 11 |
| **29** | 32 | 39 | 46 | 53 | **61** | 6 | 7 | 9 | 10 |
| **30** | 30 | 37 | 43 | 50 | **62** | 6 | 7 | 8 | 9 |
| **31** | 29 | 35 | 41 | 48 | **63** | 2 | 2 | 2 | 2 |

Table 9‑43 – State transition table

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **pStateIdx** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **transIdxLPS** | 0 | 0 | 1 | 2 | 2 | 4 | 4 | 5 | 6 | 7 | 8 | 9 | 9 | 11 | 11 | 12 |
| **transIdxMPS** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| **pStateIdx** | **16** | **17** | **18** | **19** | **20** | **21** | **22** | **23** | **24** | **25** | **26** | **27** | **28** | **29** | **30** | **31** |
| **transIdxLPS** | 13 | 13 | 15 | 15 | 16 | 16 | 18 | 18 | 19 | 19 | 21 | 21 | 22 | 22 | 23 | 24 |
| **transIdxMPS** | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
| **pStateIdx** | **32** | **33** | **34** | **35** | **36** | **37** | **38** | **39** | **40** | **41** | **42** | **43** | **44** | **45** | **46** | **47** |
| **transIdxLPS** | 24 | 25 | 26 | 26 | 27 | 27 | 28 | 29 | 29 | 30 | 30 | 30 | 31 | 32 | 32 | 33 |
| **transIdxMPS** | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 |
| **pStateIdx** | **48** | **49** | **50** | **51** | **52** | **53** | **54** | **55** | **56** | **57** | **58** | **59** | **60** | **61** | **62** | **63** |
| **transIdxLPS** | 33 | 33 | 34 | 34 | 35 | 35 | 35 | 36 | 36 | 36 | 37 | 37 | 37 | 38 | 38 | 63 |
| **transIdxMPS** | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 62 | 63 |

##### Renormalization process in the arithmetic decoding engine

Inputs to this process are bits from slice data and the variables codIRange and codIOffset.

Outputs of this process are the updated variables codIRange and codIOffset.

A flowchart of the renormalization is shown in Figure 9‑3. The current value of codIRange is first compared to 256 and further steps are specified as follows.

– If codIRange is greater than or equal to 256, no renormalization is needed and the RenormD process is finished;

– Otherwise (codIRange is less than 256), the renormalization loop is entered. Within this loop, the value of codIRange is doubled, i.e., left-shifted by 1 and a single bit is shifted into codIOffset by using read\_bits( 1 ).

The bitstream shall not contain data that result in a value of codIOffset being greater than or equal to codIRange upon completion of this process.



Figure 9‑4 – Flowchart of renormalization

##### Bypass decoding process for binary decisions

Inputs to this process are bits from slice data and the variables codIRange and codIOffset.

Outputs of this process are the updated variable codIOffset and the decoded value binVal.

The bypass decoding process is invoked when bypassFlag is equal to 1. Figure 9‑4 shows a flowchart of the corresponding process.

First, the value of codIOffset is doubled, i.e., left-shifted by 1 and a single bit is shifted into codIOffset by using read\_bits( 1 ). Then, the value of codIOffset is compared to the value of codIRange and further steps are specified as follows.

– If codIOffset is greater than or equal to codIRange, the variable binVal is set equal to 1 and codIOffset is decremented by codIRange.

– Otherwise (codIOffset is less than codIRange), the variable binVal is set equal to 0*.*

The bitstream shall not contain data that result in a value of codIOffset being greater than or equal to codIRange upon completion of this process.



Figure 9‑5 – Flowchart of bypass decoding process

##### Decoding process for binary decisions before termination

Inputs to this process are bits from slice data and the variables codIRange and codIOffset.

Outputs of this process are the updated variables codIRange and codIOffset, and the decoded value binVal.

This special decoding routine applies to decoding of end\_of\_slice\_flag and pcm\_flag corresponding to ctxIdxTable equal to 0 and ctxIdx equal to 0. Figure 9‑5 shows the flowchart of the corresponding decoding process, which is specified as follows.

First, the value of codIRange is decremented by 2. Then, the value of codIOffset is compared to the value of codIRange and further steps are specified as follows.

– If codIOffset is greater than or equal to codIRange, the variable binVal is set equal to 1, no renormalization is carried out, and CABAC decoding is terminated. The last bit inserted in register codIOffset is equal to 1. When decoding end\_of\_slice\_flag, this last bit inserted in register codIOffset is interpreted as rbsp\_stop\_one\_bit.

– Otherwise (codIOffset is less than codIRange), the variable binVal is set equal to 0 and renormalization is performed as specified in subclause 9.3.3.2.2.

NOTE – This procedure may also be implemented using DecodeDecision(ctxIdxTable, ctxIdx) with ctxIdxTable = 0 and ctxIdx = 0. In the case where the decoded value is equal to 1, seven more bits would be read by DecodeDecision(ctxIdxTable, ctxIdx) and a decoding process would have to adjust its bitstream pointer accordingly to properly decode following syntax elements.



Figure 9‑6 – Flowchart of decoding a decision before termination

### Arithmetic encoding process (informative)

This subclause does not form an integral part of this Recommendation | International Standard.

Inputs to this process are decisions that are to be encoded and written.

Outputs of this process are bits that are written to the RBSP.

This informative subclause describes an arithmetic encoding engine that matches the arithmetic decoding engine described in subclause 9.3.3.2. The encoding engine is essentially symmetric with the decoding engine, i.e., procedures are called in the same order. The following procedures are described in this section: InitEncoder, EncodeDecision, EncodeBypass, EncodeTerminate, which correspond to InitDecoder, DecodeDecision, DecodeBypass, and DecodeTerminate, respectively. The state of the arithmetic encoding engine is represented by a value of the variable codILow pointing to the lower end of a sub-interval and a value of the variable codIRange specifying the corresponding range of that sub-interval.

#### Initialisation process for the arithmetic encoding engine (informative)

This subclause does not form an integral part of this Recommendation | International Standard.

This process is invoked before encoding the first macroblock of a slice, and after encoding any pcm\_alignment\_zero\_bit and all pcm\_sample\_luma and pcm\_sample\_chroma data for a coding unit of type I\_PCM.

Outputs of this process are the values codILow, codIRange, firstBitFlag, bitsOutstanding, and BinCountsInNALunits of the arithmetic encoding engine.

In the initialisation procedure of the encoder, codILow is set equal to 0, and codIRange is set equal to 510. Furthermore, firstBitFlag is set equal to 1 and the counter bitsOutstanding is set equal to 0.

Depending on whether the current slice is the first slice of a coded picture, the following applies.

– If the current slice is the first slice of a coded picture, the counter BinCountsInNALunits is set equal to 0.

– Otherwise (the current slice is not the first slice of a coded picture), the counter BinCountsInNALunits is not modified. The value of BinCountsInNALunits is the result of encoding all the slices of a coded picture that precede the current slice in decoding order. After initialising for the first slice of a coded picture as specified in this subclause, BinCountsInNALunits is incremented as specified in subclauses 9.3.4.2, 9.3.4.4, and 9.3.4.5.

NOTE – The minimum register precision required for storing the values of the variables codILow and codIRange after invocation of any of the arithmetic encoding processes specified in subclauses 9.3.4.2, 9.3.4.4, and 9.3.4.5 is 10 bits and 9 bits, respectively. The encoding process for a binary decision (EncodeDecision) as specified in subclause 9.3.4.2 and the encoding process for a binary decision before termination (EncodeTerminate) as specified in subclause 9.3.4.5 require a minimum register precision of 10 bits for the variable codILow and a minimum register precision of 9 bits for the variable codIRange. The bypass encoding process for binary decisions (EncodeBypass) as specified in subclause 9.3.4.4 requires a minimum register precision of 11 bits for the variable codILow and a minimum register precision of 9 bits for the variable codIRange. The precision required for the counters bitsOutstanding and BinCountsInNALunits should be sufficiently large to prevent overflow of the related registers. When maxBinCountInSlice denotes the maximum total number of binary decisions to encode in one slice and maxBinCountInPic denotes the maximum total number of binary decisions to encode a picture, the minimum register precision required for the variables bitsOutstanding and BinCountsInNALunits is given by Ceil( Log2( maxBinCountInSlice + 1 ) ) and Ceil( Log2( maxBinCountInPic + 1 ) ), respectively.

#### Encoding process for a binary decision (informative)

This subclause does not form an integral part of this Recommendation | International Standard.

Inputs to this process are the context index ctxIdx, the value of binVal to be encoded, and the variables codIRange, codILow and BinCountsInNALunits.

Outputs of this process are the variables codIRange, codILow, and BinCountsInNALunits.

Figure 9‑6 shows the flowchart for encoding a single decision. In a first step, the variable codIRangeLPS is derived as follows.

Given the current value of codIRange, codIRange is mapped to the index qCodIRangeIdx of a quantised value of codIRange by using Equation 9‑6. The value of qCodIRangeIdx and the value of pStateIdx associated with ctxIdx are used to determine the value of the variable rangeTabLPS as specified in Table 9‑8, which is assigned to codIRangeLPS. The value of codIRange − codIRangeLPS is assigned to codIRange.

In a second step, the value of binVal is compared to valMPS associated with ctxIdx. When binVal is different from valMPS, codIRange is added to codILow and codIRange is set equal to the value codIRangeLPS.Given the encoded decision, the state transition isperformed as specified in subclause 9.3.3.2.1.1. Depending on the current value of codIRange, renormalization is performed as specified in subclause 9.3.4.3. Finally, the variable BinCountsInNALunits is incremented by 1.



Figure 9‑7 – Flowchart for encoding a decision

#### Renormalization process in the arithmetic encoding engine (informative)

This subclause does not form an integral part of this Recommendation | International Standard.

Inputs to this process are the variables codIRange, codILow, firstBitFlag, and bitsOutstanding.

Outputs of this process are zero or more bits written to the RBSP and the updated variables codIRange, codILow, firstBitFlag, and bitsOutstanding.

Renormalization is illustrated in Figure 9‑7.



Figure 9‑8 – Flowchart of renormalization in the encoder

The PutBit( ) procedure described in Figure 9‑8 provides carry over control. It uses the function WriteBits( B, N ) that writes N bits with value B to the bitstream and advances the bitstream pointer by N bit positions. This function assumes the existence of a bitstream pointer with an indication of the position of the next bit to be written to the bitstream by the encoding process.



Figure 9‑9 – Flowchart of PutBit(B)

#### Bypass encoding process for binary decisions (informative)

This subclause does not form an integral part of this Recommendation | International Standard.

Inputs to this process are the variables binVal, codILow, codIRange, bitsOutstanding, and BinCountsInNALunits.

Output of this process is a bit written to the RBSP and the updated variables codILow, bitsOutstanding, and BinCountsInNALunits.

This encoding process applies to all binary decisions with bypassFlag equal to 1. Renormalization is included in the specification of this process as given in Figure 9‑9.



Figure 9‑10 – Flowchart of encoding bypass

#### Encoding process for a binary decision before termination (informative)

This subclause does not form an integral part of this Recommendation | International Standard.

Inputs to this process are the variables binVal, codIRange, codILow, bitsOutstanding, and BinCountsInNALunits.

Outputs of this process are zero or more bits written to the RBSP and the updated variables codILow, codIRange, bitsOutstanding, and BinCountsInNALunits.

This encoding routine shown in Figure 9‑10 applies to encoding of the end\_of\_slice\_flag and pcm\_flag indicating the I\_PCM mode both associated with ctxIdx equal to 276.



Figure 9‑11 – Flowchart of encoding a decision before termination

When the value of binVal to encode is equal to 1, CABAC encoding is terminated and the flushing procedure shown in Figure 9‑11 is applied. In this flushing procedure, the last bit written by WriteBits( B, N )  is equal to 1. When encoding end\_of\_slice\_flag, this last bit is interpreted as the rbsp\_stop\_one\_bit.



Figure 9‑12 – Flowchart of flushing at termination

#### Byte stuffing process (informative)

This subclause does not form an integral part of this Recommendation | International Standard.

This process is invoked after encoding the last macroblock of the last slice of a picture and after encapsulation.

Inputs to this process are the number of bytes NumBytesInVclNALunits of all VCL NAL units of a picture, the number of macroblocks PicSizeInMbs in the picture, and the number of binary symbols BinCountsInNALunits resulting from encoding the contents of all VCL NAL units of the picture.

NOTE – The value of BinCountsInNALunits is the result of encoding all slices of a coded picture. After initialising for the first slice of a coded picture as specified in subclause 9.3.4.1, BinCountsInNALunits is incremented as specified in subclauses 9.3.4.2, 9.3.4.4, and 9.3.4.5.

Outputs of this process are zero or more bytes appended to the NAL unit.

Let the variable k be set equal to Ceil( ( Ceil( 3 \* ( 32 \* BinCountsInNALunits − RawMbBits \* PicSizeInMbs ) ÷ 1024 ) − NumBytesInVclNALunits ) ÷ 3 ). Depending on the variable k the following applies.

– If k is less than or equal to 0, no cabac\_zero\_word is appended to the NAL unit.

– Otherwise (k is greater than 0), the 3-byte sequence 0x000003 is appended k times to the NAL unit after encapsulation, where the first two bytes 0x0000 represent a cabac\_zero\_word and the third byte 0x03 represents an emulation\_prevention\_three\_byte.

1. Annex A  
     
   Profiles and levels

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

Profiles and levels specify restrictions on bitstreams and hence limits on the capabilities needed to decode the bitstreams. Profiles and levels may also be used to indicate interoperability points between individual decoder implementations.

NOTE 1 – This Recommendation | International Standard does not include individually selectable "options" at the decoder, as this would increase interoperability difficulties.

Each profile specifies a subset of algorithmic features and limits that shall be supported by all decoders conforming to that profile.

NOTE 2 – Encoders are not required to make use of any particular subset of features supported in a profile.

Each level specifies a set of limits on the values that may be taken by the syntax elements of this Recommendation | International Standard. The same set of level definitions is used with all profiles, but individual implementations may support a different level for each supported profile. For any given profile, levels generally correspond to decoder processing load and memory capability.

The profiles that are specified in subclause A.2 are also referred to as the profiles specified in Annex A.

* 1. Requirements on video decoder capability

Capabilities of video decoders conforming to this Recommendation | International Standard are specified in terms of the ability to decode video streams conforming to the constraints of profiles and levels specified in this annex. For each such profile, the level supported for that profile shall also be expressed.

Specific values are specified in this annex for the syntax elements profile\_idc and level\_idc. All other values of profile\_idc and level\_idc are reserved for future use by ITU-T | ISO/IEC.

NOTE – Decoders should not infer that when a reserved value of profile\_idc or level\_idc falls between the values specified in this Recommendation | International Standard that this indicates intermediate capabilities between the specified profiles or levels, as there are no restrictions on the method to be chosen by ITU-T | ISO/IEC for the use of such future reserved values.

* 1. Profiles

All constraints for picture parameter sets that are specified are constraints for picture parameter sets that are activated in the bitstream. All constraints for sequence parameter sets that are specified in subclause A.2.1 are constraints for sequence parameter sets that are activated in the bitstream.

* + 1. ABC profile
  1. Levels

Table A‑9‑44 – Level limits [Ed. (TW) Kept for convenience]

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Level number** | **Max macroblock processing rate MaxMBPS (MB/s)** | **Max picture size MaxFS (MBs)** | **Max decoded picture buffer size MaxDpbMbs (MBs)** | **Max  video bit rate MaxBR  (1000 bits/s, 1200 bits/s, cpbBrVclFactor bits/s, or cpbBrNalFactor bits/s)** | **Max CPB size MaxCPB (1000 bits, 1200 bits, cpbBrVclFactor bits, or cpbBrNalFactor bits)** | **Vertical MV component range  MaxVmvR (luma picture samples)** | **Min compression ratio MinCR** | **Max number of motion vectors per two consecutive MBs  MaxMvsPer2Mb** |
| **1** | 1 485 | 99 | 396 | 64 | 175 | [−64,+63.75] | 2 | - |
| **1b** | 1 485 | 99 | 396 | 128 | 350 | [−64,+63.75] | 2 | - |
| **1.1** | 3 000 | 396 | 900 | 192 | 500 | [−128,+127.75] | 2 | - |
| **1.2** | 6 000 | 396 | 2 376 | 384 | 1 000 | [−128,+127.75] | 2 | - |
| **1.3** | 11 880 | 396 | 2 376 | 768 | 2 000 | [−128,+127.75] | 2 | - |
| **2** | 11 880 | 396 | 2 376 | 2 000 | 2 000 | [−128,+127.75] | 2 | - |
| **2.1** | 19 800 | 792 | 4 752 | 4 000 | 4 000 | [−256,+255.75] | 2 | - |
| **2.2** | 20 250 | 1 620 | 8 100 | 4 000 | 4 000 | [−256,+255.75] | 2 | - |
| **3** | 40 500 | 1 620 | 8 100 | 10 000 | 10 000 | [−256,+255.75] | 2 | 32 |
| **3.1** | 108 000 | 3 600 | 18 000 | 14 000 | 14 000 | [−512,+511.75] | 4 | 16 |
| **3.2** | 216 000 | 5 120 | 20 480 | 20 000 | 20 000 | [−512,+511.75] | 4 | 16 |
| **4** | 245 760 | 8 192 | 32 768 | 20 000 | 25 000 | [−512,+511.75] | 4 | 16 |
| **4.1** | 245 760 | 8 192 | 32 768 | 50 000 | 62 500 | [−512,+511.75] | 2 | 16 |
| **4.2** | 522 240 | 8 704 | 34 816 | 50 000 | 62 500 | [−512,+511.75] | 2 | 16 |
| **5** | 589 824 | 22 080 | 110 400 | 135 000 | 135 000 | [−512,+511.75] | 2 | 16 |
| **5.1** | 983 040 | 36 864 | 184 320 | 240 000 | 240 000 | [−512,+511.75] | 2 | 16 |

Levels with non-integer level numbers in Table A‑9‑8 are referred to as "intermediate levels".

NOTE 4 – All levels have the same status, but some applications may choose to use only the integer-numbered levels.

1. Annex B  
     
   Byte stream format

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

This annex specifies syntax and semantics of a byte stream format specified for use by applications that deliver some or all of the NAL unit stream as an ordered stream of bytes or bits within which the locations of NAL unit boundaries need to be identifiable from patterns in the data, such as ITU-T Rec. H.222.0 | ISO/IEC 13818-1 systems or ITU‑T Rec. H.320 systems. For bit-oriented delivery, the bit order for the byte stream format is specified to start with the MSB of the first byte, proceed to the LSB of the first byte, followed by the MSB of the second byte, etc.

The byte stream format consists of a sequence of byte stream NAL unit syntax structures. Each byte stream NAL unit syntax structure contains one start code prefix followed by one nal\_unit( NumBytesInNALunit ) syntax structure. It may (and under some circumstances, it shall) also contain an additional zero\_byte syntax element. It may also contain one or more additional trailing\_zero\_8bits syntax elements. When it is the first byte stream NAL unit in the bitstream, it may also contain one or more additional leading\_zero\_8bits syntax elements.

* 1. Byte stream NAL unit syntax and semantics
     1. Byte stream NAL unit syntax

|  |  |  |
| --- | --- | --- |
| byte\_stream\_nal\_unit( NumBytesInNALunit ) { | C | Descriptor |
| while( next\_bits( 24 ) != 0x000001 &&  next\_bits( 32 ) != 0x00000001 ) |  |  |
| **leading\_zero\_8bits** /\* equal to 0x00 \*/ |  | f(8) |
| if( next\_bits( 24 ) != 0x000001 ) |  |  |
| **zero\_byte** /\* equal to 0x00 \*/ |  | f(8) |
| **start\_code\_prefix\_one\_3bytes** /\* equal to 0x000001 \*/ |  | f(24) |
| nal\_unit( NumBytesInNALunit ) |  |  |
| while( more\_data\_in\_byte\_stream( ) &&  next\_bits( 24 ) != 0x000001 &&  next\_bits( 32 ) != 0x00000001 ) |  |  |
| **trailing\_zero\_8bits** /\* equal to 0x00 \*/ |  | f(8) |
| } |  |  |

* + 1. Byte stream NAL unit semantics

The order of byte stream NAL units in the byte stream shall follow the decoding order of the NAL units contained in the byte stream NAL units (see subclause 7.4.1.2). The content of each byte stream NAL unit is associated with the same access unit as the NAL unit contained in the byte stream NAL unit (see subclause 7.4.1.2.3).

**leading\_zero\_8bits** is a byte equal to 0x00.

NOTE – The leading\_zero\_8bits syntax element can only be present in the first byte stream NAL unit of the bitstream, because (as shown in the syntax diagram of subclause B.1.1) any bytes equal to 0x00 that follow a NAL unit syntax structure and precede the four-byte sequence 0x00000001 (which is to be interpreted as a zero\_byte followed by a start\_code\_prefix\_one\_3bytes) will be considered to be trailing\_zero\_8bits syntax elements that are part of the preceding byte stream NAL unit.

**zero\_byte** is a single byte equal to 0x00.

When any of the following conditions are fulfilled, the zero\_byte syntax element shall be present:

– the nal\_unit\_type within the nal\_unit( ) is equal to 7 (sequence parameter set) or 8 (picture parameter set),

– the byte stream NAL unit syntax structure contains the first NAL unit of an access unit in decoding order, as specified by subclause 7.4.1.2.3.

**start\_code\_prefix\_one\_3bytes** is a fixed-value sequence of 3 bytes equal to 0x000001. This syntax element is called a start code prefix.

**trailing\_zero\_8bits** is a byte equal to 0x00.

* 1. Byte stream NAL unit decoding process

Input to this process consists of an ordered stream of bytes consisting of a sequence of byte stream NAL unit syntax structures.

Output of this process consists of a sequence of NAL unit syntax structures.

At the beginning of the decoding process, the decoder initialises its current position in the byte stream to the beginning of the byte stream. It then extracts and discards each leading\_zero\_8bits syntax element (if present), moving the current position in the byte stream forward one byte at a time, until the current position in the byte stream is such that the next four bytes in the bitstream form the four-byte sequence 0x00000001.

The decoder then performs the following step-wise process repeatedly to extract and decode each NAL unit syntax structure in the byte stream until the end of the byte stream has been encountered (as determined by unspecified means) and the last NAL unit in the byte stream has been decoded:

1. When the next four bytes in the bitstream form the four-byte sequence 0x00000001, the next byte in the byte stream (which is a zero\_byte syntax element) is extracted and discarded and the current position in the byte stream is set equal to the position of the byte following this discarded byte.
2. The next three-byte sequence in the byte stream (which is a start\_code\_prefix\_one\_3bytes) is extracted and discarded and the current position in the byte stream is set equal to the position of the byte following this three‑byte sequence.
3. NumBytesInNALunit is set equal to the number of bytes starting with the byte at the current position in the byte stream up to and including the last byte that precedes the location of any of the following conditions:

– A subsequent byte-aligned three-byte sequence equal to 0x000000,

– A subsequent byte-aligned three-byte sequence equal to 0x000001,

– The end of the byte stream, as determined by unspecified means.

1. NumBytesInNALunit bytes are removed from the bitstream and the current position in the byte stream is advanced by NumBytesInNALunit bytes. This sequence of bytes is nal\_unit( NumBytesInNALunit ) and is decoded using the NAL unit decoding process.
2. When the current position in the byte stream is not at the end of the byte stream (as determined by unspecified means) and the next bytes in the byte stream do not start with a three-byte sequence equal to 0x000001 and the next bytes in the byte stream do not start with a four byte sequence equal to 0x00000001, the decoder extracts and discards each trailing\_zero\_8bits syntax element, moving the current position in the byte stream forward one byte at a time, until the current position in the byte stream is such that the next bytes in the byte stream form the four-byte sequence 0x00000001 or the end of the byte stream has been encountered (as determined by unspecified means).
   1. Decoder byte-alignment recovery (informative)

This subclause does not form an integral part of this Recommendation | International Standard.

Many applications provide data to a decoder in a manner that is inherently byte aligned, and thus have no need for the bit-oriented byte alignment detection procedure described in this subclause.

A decoder is said to have byte-alignment with a bitstream when the decoder is able to determine whether or not the positions of data in the bitstream are byte-aligned. When a decoder does not have byte alignment with the encoder's byte stream, the decoder may examine the incoming bitstream for the binary pattern '00000000 00000000 00000000 00000001' (31 consecutive bits equal to 0 followed by a bit equal to 1). The bit immediately following this pattern is the first bit of an aligned byte following a start code prefix. Upon detecting this pattern, the decoder will be byte aligned with the encoder and positioned at the start of a NAL unit in the byte stream.

Once byte aligned with the encoder, the decoder can examine the incoming byte stream for subsequent three-byte sequences 0x000001 and 0x000003.

When the three-byte sequence 0x000001 is detected, this is a start code prefix.

When the three-byte sequence 0x000003 is detected, the third byte (0x03) is an emulation\_prevention\_three\_byte to be discarded as specified in subclause 7.4.1.

When an error in the bitstream syntax is detected (e.g., a non-zero value of the forbidden\_zero\_bit or one of the three‑byte or four-byte sequences that are prohibited in subclause 7.4.1), the decoder may consider the detected condition as an indication that byte alignment may have been lost and may discard all bitstream data until the detection of byte alignment at a later position in the bitstream as described in this subclause.

1. Annex C  
     
   Hypothetical reference decoder

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

This annex specifies the hypothetical reference decoder (HRD) and its use to check bitstream and decoder conformance.

Two types of bitstreams are subject to HRD conformance checking for this Recommendation | International Standard. The first such type of bitstream, called Type I bitstream, is a NAL unit stream containing only the VCL NAL units and filler data NAL units for all access units in the bitstream. The second type of bitstream, called a Type II bitstream, contains, in addition to the VCL NAL units and filler data NAL units for all access units in the bitstream, at least one of the following:

– additional non-VCL NAL units other than filler data NAL units,

– all leading\_zero\_8bits, zero\_byte, start\_code\_prefix\_one\_3bytes, and trailing\_zero\_8bits syntax elements that form a byte stream from the NAL unit stream (as specified in Annex B).

Figure C‑9‑12 shows the types of bitstream conformance points checked by the HRD.



Figure C‑9‑13 – Structure of byte streams and NAL unit streams for HRD conformance checks

The syntax elements of non-VCL NAL units (or their default values for some of the syntax elements), required for the HRD, are specified in the semantic subclauses of clause 7, Annexes D and E.

The HRD contains a coded picture buffer (CPB), an instantaneous decoding process, a decoded picture buffer (DPB), and output cropping as shown in Figure C‑9‑13.



Figure C‑9‑14 – HRD buffer model

* 1. Operation of coded picture buffer (CPB)

The specifications in this subclause apply independently to each set of CPB parameters that is present and to both the Type I and Type II conformance points shown in Figure C‑9‑12.

* + 1. Timing of bitstream arrival
    2. Timing of coded picture removal
  1. Operation of the decoded picture buffer (DPB)

The decoded picture buffer contains frame buffers. Each of the frame buffers may contain a decoded picture that is marked as "used for reference" or is held for future output. Prior to initialisation, the DPB is empty (the DPB fullness is set to zero). The following steps of the subclauses of this subclause all happen instantaneously at tr( n ) and in the sequence listed.

* + 1. **Removal of pictures from the DPB**

The removal of pictures from the DPB before decoding of the current picture (but after parsing the slice header of the first slice of the current picture) proceeds as follows.

The decoding process for reference picture set as specified in subclause 8.2.2 is invoked.

If the current picture is an IDR picture, the following applies:

1. When the IDR picture is not the first IDR picture decoded and the value of pic\_width\_in\_luma\_samples or pic\_height\_in\_luma\_samples or max\_dec\_frame\_buffering derived from the active sequence parameter set is different from the value of pic\_width\_in\_luma\_samples or pic\_height\_in\_luma\_samples or max\_dec\_frame\_buffering derived from the sequence parameter set that was active for the preceding picture, respectively, no\_output\_of\_prior\_pics\_flag is inferred to be equal to 1 by the HRD, regardless of the actual value of no\_output\_of\_prior\_pics\_flag.

NOTE – Decoder implementations should try to handle frame or DPB size changes more gracefully than the HRD in regard to changes in pic\_width\_in\_luma\_samples or pic\_height\_in\_luma\_samples.

1. When no\_output\_of\_prior\_pics\_flag is equal to 1 or is inferred to be equal to 1, all frame buffers in the DPB are emptied without output of the pictures they contain, and DPB fullness is set to 0.

All pictures m in the DPB, for which all of the following conditions are true, are removed from the DPB:

– picture m is marked as "unused for reference",

– picture m has output\_flag equal to 0 or its DPB output time is less than or equal to the CPB removal time of the current picture n; i.e., to,dpb( m ) <= tr( n )

When a picture is removed from the DPB, the DPB fullness is decremented by one.

* + 1. **Picture decoding and output**

Picture n is decoded. When it has output\_flag equal to 1, its DPB output time to,dpb( n ) is derived by

to,dpb( n ) = tr( n ) + tc \* dpb\_output\_delay( n ) (C-x)

where dpb\_output\_delay( n ) is the value of dpb\_output\_delay specified in the picture timing SEI message associated with access unit n.

The output of the current picture is specified as follows.

– If output\_flag is equal to 1 and to,dpb( n ) = tr( n ), the current picture is output.

– Otherwise, if output\_flag is equal to 0, the current picture is not output, but will be stored in the DPB as specified in subclause C.2.2.

– Otherwise (output\_flag is equal to 1 and to,dpb( n ) > tr( n ) ), the current picture is output later and will be stored in the DPB (as specified in subclause C.2.2) and is output at time to,dpb( n ) unless indicated not to be output by the decoding or inference of no\_output\_of\_prior\_pics\_flag equal to 1 at a time that precedes to,dpb( n ).

When output, the picture shall be cropped, using the cropping rectangle specified in the active sequence parameter set.

When picture n is a picture that is output and is not the last picture of the bitstream that is output, the value of Δto,dpb( n ) is defined as:

Δto,dpb( n ) = to,dpb( nn ) − to,dpb( n ) (C-x)

where nn indicates the picture that follows after picture n in output order and has output\_flag equal to 1.

* + 1. **Current decoded picture marking and storage**

The current decoded picture is stored in the DPB in an empty frame buffer, and the DPB fullness is incremented by one. If the current picture is a reference picture, it is marked as "used for reference", otherwise it is marked as "unused for reference".

* 1. Bitstream conformance

The first coded picture in a bitstream shall be either an IDR picture or a CRA picture.[Ed.Note (YK): Add other bitstream conformance requirements similar as in AVC.]

* 1. Decoder conformance
     1. Operation of the output order DPB

The decoded picture buffer contains frame buffers. Each of the frame buffers contains a decoded picture that is marked as "used for reference" or is held for future output. At HRD initialisation, the DPB fullness, measured in frames, is set to 0. The following steps all happen instantaneously when an access unit is removed from the CPB, and in the order listed.

* + 1. Removal of pictures from the DPB

The removal of pictures from the DPB before decoding of the current picture (but after parsing the slice header of the first slice of the current picture) proceeds as follows.

The decoding process for reference picture set as specified in subclause 8.2.2 is invoked.

– If the current picture is an IDR picture, the following applies.

1. When the IDR picture is not the first IDR picture decoded and the value of pic\_width\_in\_luma\_samples or pic\_height\_in\_luma\_samples or max\_dec\_frame\_buffering derived from the active sequence parameter set is different from the value of pic\_width\_in\_luma\_samples or pic\_height\_in\_luma\_samples or max\_dec\_frame\_buffering derived from the sequence parameter set that was active for the preceding picture, respectively, no\_output\_of\_prior\_pics\_flag is inferred to be equal to 1 by the HRD, regardless of the actual value of no\_output\_of\_prior\_pics\_flag.

NOTE – Decoder implementations should try to handle changes in the value of pic\_width\_in\_luma\_samples or pic\_height\_in\_luma\_samples or max\_dec\_frame\_buffering more gracefully than the HRD.

1. When no\_output\_of\_prior\_pics\_flag is equal to 1 or is inferred to be equal to 1, all frame buffers in the DPB are emptied without output of the pictures they contain, and DPB fullness is set to 0.

– Otherwise (the current picture is not an IDR picture), frame buffers containing a picture which are marked as "not needed for output" and "unused for reference" are emptied (without output), and the DPB fullness is decremented by the number of frame buffers emptied. When there is no empty frame buffer (i.e., DPB fullness is equal to DPB size), the "bumping" process specified in subclause C.4.2.1 is invoked repeatedly until there is an empty frame buffer to store the current decoded picture.

When the current picture is an IDR picture for which no\_output\_of\_prior\_pics\_flag is not equal to 1 and is not inferred to be equal to 1, the following two steps are performed.

1. Frame buffers containing a picture that is marked as "not needed for output" and "unused for reference" are emptied (without output), and the DPB fullness is decremented by the number of frame buffers emptied.

2. All non-empty frame buffers in the DPB are emptied by repeatedly invoking the "bumping" process specified in subclause C.4.2.1, and the DPB fullness is set to 0.

* + - 1. "Bumping" process

The "bumping" process is invoked in the following cases.

– The current picture is an IDR picture and no\_output\_of\_prior\_pics\_flag is not equal to 1 and is not inferred to be equal to 1, as specified in subclause C.4.2.

– There is no empty frame buffer (i.e., DPB fullness is equal to DPB size) and an empty frame buffer is needed for storage of a decoded (non-IDR) picture, as specified in subclause C.4.2.

The "bumping" process consists of the following ordered steps:

1. The picture that is first for output is selected as the one having the smallest value of PicOrderCntVal of all pictures in the DPB marked as "needed for output".
2. The picture is cropped, using the cropping rectangle specified in the active sequence parameter set for the picture, the cropped picture is output, and the picture is marked as "not needed for output".
3. If the frame buffer that included the picture that was cropped and output contains a picture marked as "unused for reference", the frame buffer is emptied and the DPB fullness is decremented by 1.
   * 1. Picture decoding, marking and storage

The current picture is decoded and stored in an empty frame buffer in the DPB. The DPB fullness is incremented by one, and the following applies.

– If the current decoded picture has output\_flag equal to 1, it is marked as "needed for output".

– Otherwise (the current decoded picture has output\_flag equal to 0), it is marked as "not needed for output".

If the current decoded picture is a reference picture, it is marked as "used for reference", otherwise (the current decoded picture is a non-reference picture), it is marked as "unused for reference".

1. Annex D  
     
   Supplemental enhancement information

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

This annex specifies syntax and semantics for SEI message payloads.

SEI messages assist in processes related to decoding, display or other purposes. However, SEI messages are not required for constructing the luma or chroma samples by the decoding process. Conforming decoders are not required to process this information for output order conformance to this Recommendation | International Standard (see Annex 0 for the specification of conformance). Some SEI message information is required to check bitstream conformance and for output timing decoder conformance.

In Annex D, specification for presence of SEI messages are also satisfied when those messages (or some subset of them) are conveyed to decoders (or to the HRD) by other means not specified by this Recommendation | International Standard. When present in the bitstream, SEI messages shall obey the syntax and semantics specified in subclauses 7.3.2.3 and 0 and this annex. When the content of an SEI message is conveyed for the application by some means other than presence within the bitstream, the representation of the content of the SEI message is not required to use the same syntax specified in this annex. For the purpose of counting bits, only the appropriate bits that are actually present in the bitstream are counted.

* 1. SEI payload syntax

[Ed. (TW): insert table]

* 1. SEI payload semantics

1. Annex E  
     
   Video usability information

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

This annex specifies syntax and semantics of the VUI parameters of the sequence parameter sets.

VUI parameters are not required for constructing the luma or chroma samples by the decoding process. Conforming decoders are not required to process this information for output order conformance to this Recommendation | International Standard (see Annex 0 for the specification of conformance). Some VUI parameters are required to check bitstream conformance and for output timing decoder conformance.

In Annex E, specification for presence of VUI parameters is also satisfied when those parameters (or some subset of them) are conveyed to decoders (or to the HRD) by other means not specified by this Recommendation | International Standard. When present in the bitstream, VUI parameters shall follow the syntax and semantics specified in subclauses 7.3.2.1 and 7.4.2.1 and this annex. When the content of VUI parameters is conveyed for the application by some means other than presence within the bitstream, the representation of the content of the VUI parameters is not required to use the same syntax specified in this annex. For the purpose of counting bits, only the appropriate bits that are actually present in the bitstream are counted.

* 1. VUI syntax
     1. VUI parameters syntax

[Ed. (TW): insert table]

* 1. VUI semantics